UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

"DESARROLLO FULL-STACK DE SISTEMA DE SIMULACIÓN SOLAR Y VISUALIZACIÓN DE RESULTADOS MULTIUSUARIO"

TRABAJO FIN DE GRADO

Julio - 2025

AUTOR: Carlos Ortuño Domínguez
DIRECTOR/ES: Xavier Moreno-Vassart Martinez

Índice

Agradecimientos

Me gustaría hacer un agradecimiento especial al tutor de este proyecto, Xavier Moreno-Vassart Martinez. Sin él, este proyecto no se podría haber llevado a cabo. Me ha acompañado desde el primer momento, guiándome e instruyéndome en cada etapa del desarrollo. Además, sus avanzados conocimientos en programación y energía fotovoltaica, han facilitado mucho el desarrollo e implementación técnica que veremos en este documento.

También me gustaría agradecerle enormemente a mi madre que haya seguido el proyecto de cerca de principio a fin, apoyándome a lo largo de estos meses de desarrollo. Además, ha tenido la paciencia e interés de leer este documento completo para intentar comprender en profundidad todos los detalles del proyecto.

Gracias.

1. Introducción

En este primer apartado introductorio se expondrán los motivos por los que se ha decidido desarrollar este proyecto, además se explicarán los objetivos iniciales que se establecieron en etapas tempranas del mismo y se aportará una visión global sobre energías renovables, en concreto, la energía solar.

1.1 Motivación

En el contexto actual de transición energética y crisis climática, el aprovechamiento de fuentes de energía renovables se ha vuelto una prioridad a nivel global. Entre estas, la energía solar destaca por su abundancia, bajo impacto ambiental y facilidad de integración tanto en entornos domésticos como industriales. Especialmente en España, la energía solar es una gran aliada, ya que este país tiene un elevado número de días soleados al año, aproximadamente unos 300 días o 2500 horas de sol anuales.

Para poder aprovechar al máximo esta energía, es imprescindible contar con herramientas que nos permitan realizar simulaciones solares para evaluar el rendimiento de los equipos que permiten captar esta energía y convertirla en energía útil, como la energía eléctrica. Un ejemplo muy conocido de este tipo de equipos son los paneles solares. Cada vez abundan más en viviendas, y hay una creciente tendencia de creación de parques solares, en los que el principal componente son los paneles fotovoltaicos.[1]

Las herramientas disponibles actualmente para poder medir el rendimiento de los equipos mencionados anteriormente, facilitan la planificación y optimización de sistemas solares, permitiendo evaluar diferentes condiciones y escenarios sin necesidad de instalaciones físicas previas, lo que ahorra costes, tiempo y recursos. Sin embargo, no son fáciles de utilizar, por lo que requieren altos conocimientos y una curva de aprendizaje lenta para poder hacer un uso eficaz de ellos. Es por eso mismo que este proyecto se ha realizado, para poder facilitar el uso y reducir los tiempos y dificultad de aprendizaje del uso de estos equipos mediante el desarrollo de una plataforma web interactiva y sencilla, que permita a cualquier usuario realizar simulaciones solares precisas y adaptadas a diversas condiciones geográficas y atmosféricas.

1.2 Objetivos

1.2.1 Objetivo general

El principal objetivo de este proyecto es crear una plataforma web intuitiva que permita configurar simulaciones solares de forma sencilla, sin requerir el uso de comandos complejos ni acciones complicadas.

1.2.2 Objetivos específicos:

- Integración de dos modos de simulación, uno haciendo uso de espectros de referencia y otro con escenarios personalizados según el modelo BIRD. Inclusión de un mapa que permita seleccionar coordenadas de forma sencilla para escenarios personalizados.

- Sistema de usuarios, cada usuario con información independiente e inaccesible por el resto de usuarios.

- Visualización clara de los resultados de las simulaciones, incluyendo gráfica de potencia obtenida.

- Posibilidad de descargar resultados en formato CSV.

- Historial de simulaciones, permitirá al usuario ver sus simulaciones y acceder a la información completa de cada una de ellas.

2. Energía solar y simulación

2.1 Conceptos básicos

Cuando hablamos de energía solar, nos referimos a aquella energía que se obtiene directamente del sol en forma de radiación electromagnética. Es una energía renovable y limpia. Esta energía solar puede ser aprovechada al convertirse en energía eléctrica mediante paneles fotovoltaicos y, además, se puede almacenar en baterías para usarse cuando no haya sol.

Para optimizar la instalación de sistemas solares, es muy recomendable utilizar simulaciones solares, ya que estas permiten estimar con precisión cómo reaccionará un sistema solar [2](por ejemplo, un conjunto de paneles solares sobre el techo de una casa) ante diferentes condiciones ambientales y diversos parámetros técnicos, sin tener que realizar una instalación física en el lugar deseado. Realizando una simulación, obtendremos el rendimiento, es decir, la cantidad de electricidad que podemos obtener, de un panel solar. En un ejemplo práctico, sabiendo el consumo medio de una vivienda, y sabiendo cuanta electricidad produce un panel fotovoltaico bajo ciertas condiciones geográficas y atmosféricas, podríamos saber cuántos paneles son necesarios para satisfacer las necesidades energéticas de dicha vivienda.

Es por esto mismo, que las simulaciones solares facilitan la planificación y toma de decisiones estratégicas, permitiendo a ingenieros, investigadores y empresas optimizar al máximo el potencial energético solar disponible, minimizando así los costes y riesgos asociados a la implementación de estos sistemas.

2.2 Proceso de simulación

Para poder realizar una simulación, necesitamos el panel fotovoltaico que queremos evaluar, un simulador solar y un Source Measure Unit (SMU), que permite generar y medir tanto voltaje como corriente. De forma básica, se le proporcionan unos parámetros al simulador que representan las condiciones tanto geográficas como atmosféricas del lugar donde se instalaría el panel. El simulador irradia luz sobre el panel, y con el SMU medimos los resultados que obtenemos de la simulación.

2.3 Resultados de una simulación

Para entender bien secciones posteriores de este documento, es necesario comprender una serie de conceptos clave respecto a que es lo que obtenemos después de realizar una simulación solar sobre un panel fotovoltaico. Como desarrolladores de una plataforma web, es necesario entender mínimamente el campo de aplicación sobre el que estamos trabajando y, si bien no es necesario comprender todos los parámetros y detalles a la perfección, debemos disponer de un conocimiento básico.

En esta sección se van a explicar los parámetros y conceptos mínimos necesarios que, como desarrollador, me ha hecho falta entender.

Conforme está configurado el software existente del simulador solar que se ha aprovechado para la realización de este proyecto, cuando realizamos una simulación, obtendremos una matriz de 2 columnas con pares de valores de voltaje V (Voltios) e intensidad I (Amperios). La cantidad de puntos que obtendremos vendrá especificada por el valor de Sample Count, el cual se configura antes de lanzar una simulación. Es decir, si establecemos el Sample Count en 50, obtendremos 50 pares de valores V e I como resultado.

Hay ciertos valores que son más importantes que otros, es a lo que nos referimos como puntos característicos o “remarkables”, así nos referiremos a ellos de aquí en adelante. Estos puntos se suelen obtener indirectamente de la curva IV, aplicando a los resultados el modelo de diodo único, que veremos en apartados posteriores.

2.3.1 Remarkables

Los puntos de interés, llamados remarkables, son 4 y son los siguientes:

- Voc (V): Este parámetro se refiere a la tensión en circuito abierto, es decir, la tensión máxima que genera el panel cuando no tiene ninguna carga conectada (no hay intensidad).

- Isc (A): Es la máxima corriente que puede entregar el panel cuando los polos negativo y positivo están cortocircuitados (no hay voltaje).

- Vmpp (V): Se refiere a la tensión en el punto de máxima potencia. La potencia se obtiene multiplicando la V por la I. Para obtener este punto, tendremos que multiplicar todos los pares de valores V e I (obtendremos potencia en W), buscar el valor máximo y ver cuál es el voltaje en ese punto.

- Impp (A): Es muy parecido al Vmpp, pero esta vez, nos tenemos que fijar en la corriente A en el punto de máxima potencia.

2.3.2 Algoritmos Single Diode Model (SDM)

El modelo SDM es un modelo físico que representa el comportamiento eléctrico de una celda solar. Para ello, usa:

- Una fuente de corriente (Iph): corriente generada por la luz.
- Un diodo (D): que representa el comportamiento no lineal de la celda. Este diodo cuenta con 2 parámetros: el factor de idealidad y la corriente de saturación inversa.
- Una resistencia serie (Rs).
- Una resistencia en paralelo o shunt (Rsh).

Con el fin de poder obtener los parámetros que necesita el modelo SDM para calcular una curva IV se necesitan algoritmos SDM que permiten encontrar los valores óptimos de estos parámetros para que el modelo SDM reproduzca bien la curva IV.

Los algoritmos que usaremos para encontrar los valores óptimos de estos parámetros y para que el modelo SDM funcione bien son los siguientes: TSLLS, y TSLLS-refined [3]. Para el propósito de este documento, y para entender el proyecto correctamente, no es necesario saber cómo trabajan estos algoritmos. Es importante recordar estos conceptos para entender explicaciones que se verán en apartados posteriores.

3. Tecnologías utilizadas

3.1 Hardware

3.1.1 Raspberry PI 4

Como servidor para alojar la plataforma web, se ha utilizado una Raspberry PI 4 model B. Este dispositivo es un ordenador de tamaño muy reducido. Cuenta con un procesador de 64 bits, 8 GB de memoria RAM, un puerto gigabit Ethernet y varios puertos de salida de vídeo microHDMI.

Este dispositivo contendrá todo el código necesario para hacer funcionar la plataforma, y se utilizará como servidor para atender todas las peticiones que se hagan a la web.

Hay que tener en cuenta que tanto el simulador solar como la SMU se controlan mediante comunicaciones serie. Para poder realizar estas comunicaciones, se reutiliza una base de código preexistente en Python, que se integra en el backend de la plataforma para facilitar la interacción directa con los dispositivos

3.1.2 Simulador Solar G2V Sunbrick

El simulador solar utilizado para irradiar energía electromagnética en el panel que se está evaluando es el G2V Sunbrick. Es un simulador bastante avanzado y con una tecnología muy competente capaz de superar con creces los estándares de calidad actuales. Utiliza una tecnología LED de estado sólido, es decir, no hay bombillas ni filtros, permite un encendido rápido y no ocurre la degradación típica de la tecnología xenón. Este simulador proporciona un desajuste espectral menor al 5%, lo que supera ampliamente los estándares IEC 60904-9:2020 y JIS C8912 (Japón) y lo clasifica como un simulador de Clase AAA.

3.1.3 Keithley 2461

El Keithley 2461 es un Source Measure Unit (SMU) o fuente-medidor de precisión diseñado para generar y medir tanto voltaje como corriente. Permite medir una alta capacidad de corriente y potencia: 10A a 1000W en modo pulsado, y hasta 7A a 100W en modo de corriente continua DC, soportando tensiones de hasta 105V aproximadamente. Además, cuenta con una resolución de 6½ dígitos (6 dígitos completos y un dígito parcial, o es 1 o es 0) y una exactitud del 0.012% en DCV, por lo que es un dispositivo con una alta precisión. El SMU se utilizará para medir la corriente que produce el panel al irradiarle energía electromagnética con el simulador solar.

3.2 Tecnologías utilizadas del lado del servidor

3.2.1 Python

Python es un lenguaje de programación de alto nivel, muy conocido, cuyo uso ha crecido enormemente en los últimos años. Se caracteriza por su sintaxis clara y legible. La gran ventaja de este lenguaje es su ecosistema de bibliotecas y frameworks, que aumentan enormemente las posibilidades de desarrollo de este lenguaje, en la mayoría de los casos estas librerías promueven y facilitan el desarrollo rápido y eficiente.

Gracias a esta facilidad, es posible centrarse en desarrollar la lógica del programa sin perder tiempo implementando funciones o algoritmos básicos que en otros lenguajes deberíamos implementar para hacer una función que en una sola línea de código podemos hacer con Python. Además, muchos frameworks modernos utilizan Python como base. Al haber elegido el framework FastAPI, Python es el lenguaje que debemos utilizar. [10]

3.2.2 FastAPI

FastAPI es un framework web asíncrono basado en el estándar Asynchronous Server Gateway Interface (ASGI) y su principal ventaja es su gran rendimiento, permitiendo una experiencia de desarrollo basada en Python.

FastAPI aprovecha las librerías typing y pydantic para poder validar datos y autogenerar documentación. Esto es útil porque para este proyecto es importante utilizar un lenguaje tipado que no permita introducir valores de un tipo que no corresponde, ya que el simulador solar no podría hacer uso de ellos. Además, al ser asíncrono de forma nativa (basado en async/await), permite manejar un gran número de peticiones concurrentes sin bloquear el hilo principal; este es uno de los motivos por los que FastAPI consigue un rendimiento muy bueno. Una característica adicional relevante es la seguridad integrada, ya que FastAPI cuenta con helpers para OAuth2, JWT, CORS, entre otros. Esto es útil ya que nos permite manejar tokens de autenticación y configurar la seguridad del sitio web de una forma razonablemente sencilla.

Existen otros frameworks que se podrían haber usado para este proyecto, como por ejemplo Flask, que tiene una comunidad muy extensa y una gran cantidad de plugins, o Django, que ofrece una solución muy completa con herramientas integradas para administración, autenticación y ORM, pero se ha optado por FastAPI, ya que su curva de aprendizaje no es excesivamente pronunciada si se conoce el lenguaje Python, genera documentación y valida los tipos automáticamente, con librerías como FastAPI Users, permite una gestión de usuarios y sesiones sencilla, y posibilita concurrencia de usuarios transparente al desarrollador. En general, es una opción moderna y eficiente que permite desarrollar código que no es específico para una tarea, sino que se desenvuelve en proyectos completos facilitando y acortando el tiempo el desarrollo.[8]

3.2.3 Sqlite3

Sqlite3 es un gestor de bases de datos relacional. Su principal ventaja es su ligereza, junto con que no necesita servidor, y que guarda todos los datos en un único archivo local. A diferencia de PostgreSQL o MySQL, SQLite no requiere una instalación en un servidor de bases de datos. Esto simplifica el desarrollo del proyecto, y es ideal para aplicaciones de tamaño pequeño o mediano como la expuesta aquí.

Otro aspecto a favor de Sqlite3 es que se puede integrar perfectamente con SQLAlchemy (explicado en el siguiente apartado), pudiendo así mantener el código ORM que ya tengamos sin importar el motor de la base de datos, lo que permitiría cambiar este motor en un futuro sin modificar el código de la aplicación.

Una duda razonable que podría surgir al usar Sqlite3 es: ¿es suficiente para el proyecto, o la base de datos puede crecer tanto que SQLite no pueda almacenar todo lo necesario? Para el volumen de datos que va a manejar esta aplicación,

SQLite cumple perfectamente con estas necesidades funcionales. Ninguna de las tablas de la base de datos presentará limitaciones prácticas en cuanto al número de entradas se refiere, ya que SQLite3 permite almacenar millones de tuplas sin problemas. [4][7][9]

3.2.4 SQLAlchemy

SQLAlchemy es una biblioteca de Python, hace la funcion de Object Relational Mapper (ORM), lo que nos permite interactuar con bases de datos relacionales de forma práctica y sencilla, orientada a objetos. SQLAlchemy nos permite manipular los datos mediante clases y objetos de Python, esto mejora la claridad y mantenibilidad del código. De todas formas, siempre podemos realizar consultas SQL manualmente si fuera necesario en algún caso.

Como se ha mencionado anteriormente, SQLAlchemy es totalmente compatible con SQLite3, por tanto, es una opción perfecta para utilizar en el proyecto. Además, está nativamente integrado con FastAPI, esto hace que sean completamente compatibles y permite, por ejemplo, la creación de endpoints que interactúan directamente con la base de datos de forma segura y eficiente.

Para aumentar el nivel de robustez de la aplicación, SQLAlchemy gestiona automáticamente las conexiones a la base de datos, reduciendo así posibles errores, y asegurando la integridad de los datos en casos de alta concurrencia o en operaciones complejas.

Por último, cabe destacar que si en un futuro, por algún motivo se decidiera cambiar el motor de la base de datos, SQLAlchemy es muy fácil de adaptar a otros motores de bases de datos porque los cambios necesarios en el código para hacer esto serían muy pocos. Esto aumenta la escalabilidad del proyecto. [11]

3.2.5 FastAPI Users

FastAPI Users es una extensión para FastAPI, proporciona herramientas para facilitar la implementación de los sistemas de gestión de usuarios y autenticación de forma segura y rápida. Dispone de funcionalidades ya hechas para hacer operaciones como por ejemplo registrar usuarios, hacer login/logout, crear rutas protegidas o gestionar tokens de autenticación JWT.

Gracias a esta herramienta, ha sido posible implementar un sistema de autenticación y gestión de usuarios robusto en poco tiempo, ya que no ha habido que diseñar desde cero todas las estrategias de protección y autenticación, que hubieran alargado mucho el tiempo de desarrollo del proyecto. [12]

3.2.6 Uvicorn

Uvicorn es una implementación de un servidor web ASGI para Python. En nuestro caso, lo hemos usado para acceder a la aplicación web desde otro dispositivo. Para ello tenemos que usar un comando de consola que convierte a

la Raspberry en el propio servidor. Se puede acceder a la web desde cualquier dispositivo que conozca la IP de la raspberry. [13]

3.3 Tecnologías del Lado del Cliente

3.3.1 HTML, JavaScript

Como cualquier web, es necesario el uso de HTML para poder mostrar contenido en un navegador. Se ha utilizado para construir la interfaz principal mostrando el contenido necesario en cada una de las páginas.

JavaScript es un lenguaje de programación interpretado y orientado a eventos que se ejecuta en el navegador y permite que las páginas web sean interactivas y dinámicas. Es una de las tecnologías esenciales del desarrollo web junto a HTML. JavaScript ha permitido la captura de eventos y manejo de datos de forma que fuera posible enviarlos al backend para que puedan ser procesados por el servidor. Además, ha hecho posible manejar la selección interactiva de coordenadas en el mapa OpenStreetMap, que se explicará en detalle más adelante. [14][15]

3.3.2 Jinja2

Jinja2 es un motor de plantillas para Python, utilizado para generar contenido HTML de forma dinámica. Su principal función es permitir incrustar código Python dentro de archivos HTML con una sintaxis basada en etiquetas, bucles y condicionales, que facilita la generación de interfaces capaces de adaptarse a los datos que se reciben desde el backend. Su uso ha sido indispensable en muchas partes del proyecto. [16]

3.3.3 OpenStreetMap

OpenStreetMaps es un mapa interactivo al estilo de Google Maps, pero estos mapas los edita la comunidad, por tanto, es de código abierto. La gran ventaja de OpenStreetMap es que, al ser de código abierto, es posible integrarlo en el proyecto sin encontrar ninguna restricción de licencias.

En este proyecto, se ha usado OpenStreetMap para introducir un mapa en la interfaz principal que permita introducir las coordenadas del lugar donde se quiere simular. De esta forma, no es necesario conocer las coordenadas exactas del lugar, sino que se puede buscar el sitio en el mapa y obtener las coordenadas con un clic.

Para integrarlo en la aplicación, se ha usado la biblioteca Leaflet, que nos permite incrustarlo en la interfaz con una carga mínima y un rendimiento más que aceptable. [17]

3.3.4 Bootstrap

Bootstrap es un framework de código abierto que nos facilita la creación de una interfaz con un diseño moderno y atractivo, y que ademas sea responsiva. Permite asignar estilos predefinidos a los elementos de HTML asignándoles una clase específica.

Se ha elegido usar Bootstrap para el diseño porque esta aplicación está principalmente centrada en la funcionalidad, y Bootstrap proporcionó un diseño bueno en muchos aspectos sin perder tiempo innecesariamente en crear un diseño desde cero. Algunos podrían considerar este diseño un tanto genérico, pero es funcional y atractivo, y permite centrar los esfuerzos de desarrollo en la lógica de la aplicación. [18]

4. Diseño del Sistema

En este apartado, se va a describir el diseño de toda la aplicación web, indicando todas sus funcionalidades, y se incluirán diagramas de flujo para entender de forma visual y sencilla el proceso de uso de la aplicación.

4.1 Diseño General

Aquí se describe el uso normal que realiza cualquier usuario al usar la plataforma web, se indica todas sus funcionalidades y funcionamiento a nivel de usuario. Esto ayudará a comprender el propósito general y las posibilidades que ofrece el proyecto realizado.

Cuando un usuario accede a la página web, si no está autenticado, se le lleva directamente a la sección de inicio de sesión. No es posible usar el sistema sin estar autenticado. Una vez haya iniciado sesión, será redirigido a la página principal. En esta encontrará un mapa para seleccionar coordenadas y un formulario que necesitará rellenar para poder lanzar una simulación.

En la Figura 1: Diagrama de flujo podemos encontrar un diagrama de flujo para entender mejor el uso de la página web.

4.1.1 Formulario para escenario personalizado

Los datos que hay en el formulario son los siguientes:

- Nombre de la simulación.
- Coordenadas, se rellena automáticamente al pulsar un punto en el mapa, también se pueden introducir a mano.
- Fecha, este campo se refiere a la fecha en la que se quiere simular.
- Hora del día, al igual que el campo anterior, se refiere a la hora que se quiere simular.

- Zona horaria.
- Temperatura (ºC).
- Altitud (m).
- Inclinación de la superficie: Ángulo en grados entre la superficie del panel y el plano horizontal.
- Surface Azimuth: Orientación horizontal del panel, en grados respecto al norte.
  0° indicaría que está orientado hacia el norte, 90° hacia el este, 180° hacia el sur.
- Surface Albedo: Porcentaje de radiación que refleja el suelo.
- Humedad.
- Ozono.
- Aerosol Turbidity 500nm: Cantidad de partículas (polvo, contaminación, etc.) en el aire que dispersan y absorben luz a 500 nm (verde visible).
- Presion (Pa).
- Voc: Voc que estimamos que tendrá el panel que vamos a medir.
- Sample Count: Número de muestras que obtendremos de la simulación.
- Panel: Modelo de panel solar que se está usando.
- SPA Method: Método usado para calcular la posición del sol (altitud y azimut solar) a partir de la fecha, hora y coordenadas.
- Airmass Model: Modelo que calcula la masa de aire atravesada por la radiación solar. Afecta a cuánto se atenúa la radiación.

En la Figura 13: Página principal se puede ver el formulario completo.

4.1.2 Espectros de referencia

Si alternativamente se desea usar un espectro de referencia en vez de uno completamente personalizado, encontrará un botón que le permitirá ir a la página de selección de espectros de referencia. En esta página encontrará un desplegable que le permite elegir entre los espectros de referencia [5]. Las opciones posibles son AM0_1.00, y un rango de espectros AM1.5g, que van desde el AM1.5g_0.10 hasta el AM1.5g_1.20, en incrementos de 0.1. Además, habrá que añadir un nombre para la simulación, seleccionar un panel, establecer un Voc y fijar un Sample Count.

4.1.3 Lanzar simulación

Al final del formulario para escenarios personalizados, así como del formulario para espectros de referencia, encontrará un botón que le permite lanzar la simulación. Este botón solo estará activo si no hay ninguna otra simulación en curso. Una vez finalice la simulación, se le redirigirá automáticamente a la página de resultados, en la que podrá visualizar todos los valores obtenidos de la simulación.

4.1.4 Resultados

En la página de resultados encontraremos toda la información útil relativa a la simulación. Esto incluye todos los puntos característicos (Voc, Isc, Vmpp e Impp), así como una gráfica de la potencia obtenida, construida en base a los valores obtenidos de la simulación. En la gráfica, podremos visualizar también cada uno de los puntos obtenidos de la simulación, así como la curva IV obtenida de aplicar a los resultados los algoritmos TSLLS o TSLLS-refined según decida el usuario.

Después de la gráfica encontraremos la información detallada de la simulación, que, además de los puntos característicos, incluirá las métricas de error (rmse, mae, área_factor) y parámetros SDM (Iph, Isat, a, Rsh, Rs). Adicionalmente podremos revisar los detalles del escenario que ha dado lugar a esta simulación. Aquí se incluirá toda la información aportada cuando se rellenó el formulario de la simulación para escenarios personalizados. Si se hubiera usado un espectro de referencia, esta sección no estaría disponible.

Además, se permitirá la descarga de resultados en formato CSV. Esto se podrá hacer mediante un botón que encontraremos en la sección inicial de la página de resultados.

4.1.5 Historial de simulaciones

Con el objetivo de no perder la información de cada una de las simulaciones, todos los datos de estas se guardan en la base de datos. Para poder acceder a simulaciones anteriores, habrá que pinchar en el menú desplegable superior, y pulsar sobre el botón “Simulaciones”. Esto nos permitirá acceder a un historial de simulaciones. Aquí, el usuario se encontrará con una tabla que tendrá en su interior todas las simulaciones realizadas por él. Cada una de estas simulaciones contiene los siguientes datos:

ID de la simulación, Nombre de la simulación, Tipo (Escenario Personalizado/Espectro de referencia), ID del panel. Además, si pulsamos en “Ver detalle” iremos a la página de resultados con los datos de esta simulación, pudiendo así obtener todos los datos al completo, al igual que cuándo se realizó la simulación.

También encontraremos un botón que nos permite eliminar la simulación. Esto es útil si, por ejemplo, hemos realizado la simulación y los parámetros estaban mal configurados, o hubo algún problema técnico durante la simulación y la prueba no es válida.

4.1.6 Administrador

El usuario administrador de la plataforma podrá acceder a un panel de administrador en el que verá todos los usuarios registrados. Desde este panel podrá dar permisos de administración a los usuarios que él quiera, así como eliminar usuarios o desactivar su cuenta sin eliminar sus datos.

4.1.7 Paneles

Los usuarios pueden añadir paneles a la base de datos. Se les puede asignar un nombre y una descripción. Los paneles que se añadan aparecerán en la lista desplegable de selección de paneles. Al añadir el panel, se podrá elegir si queremos que el panel sea público o privado. Si es privado, solo estará disponible para el usuario que lo haya creado, de lo contrario, este panel será accesible para todos los usuarios de la web.

4.1.7 Diagrama de flujo de uso de la web

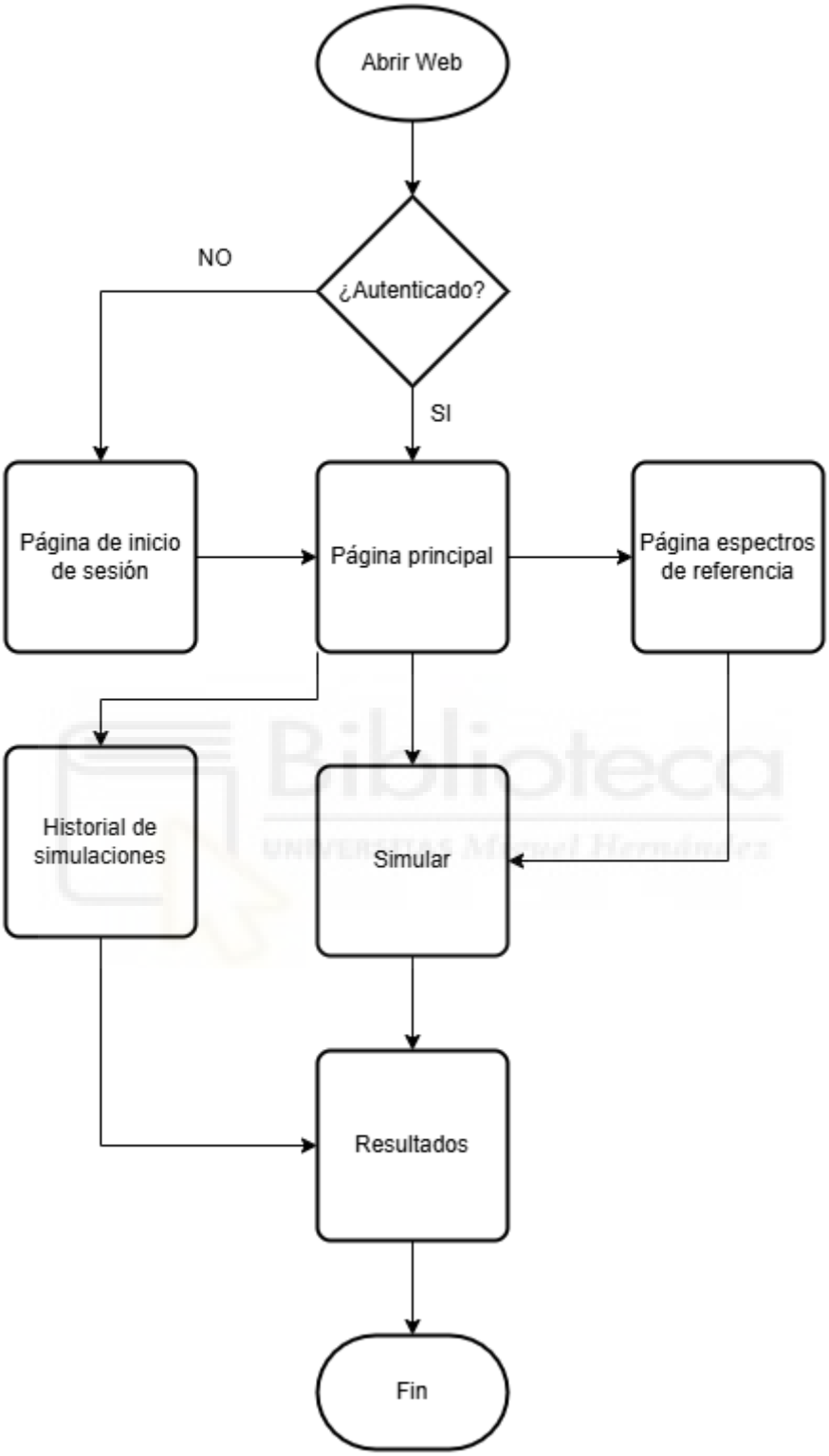


Figura 1: Diagrama de flujo

4.2 Diseño del backend

En este apartado se va a explicar el diseño que ha permitido realizar la implementación de la aplicación de forma detallada. Se explicará el diseño de la base de datos, así como las directrices que se han seguido en todo el desarrollo. Más adelante, en el apartado de Implementación, se explicará con detalle cómo se ha llevado a la realidad este diseño.

4.2.1 Diseño del API RESTful con FastAPI

Como se ha mencionado en otros apartados de este documento, el API RESTful utilizado ha sido FastAPI. El diseño del API se ha estructurado en torno a endpoints, que son rutas específicas que permiten al backend saber qué acción realizar. Por ejemplo, el endpoint /shorts en la página www.youtube.com, le hace saber al servidor que tiene que cargar la página de shorts, con todo lo que ello conlleva (recursos, base de datos, información de cookies, etc).

En nuestra aplicación, la mayoría de los endpoints cargan una página. Para referirnos a los endpoints a partir de ahora, lo haremos con la siguiente estructura: IP del dispositivo/endpoint. La IP del dispositivo dependerá de donde esté instalado, por tanto, no es fija.

Hay que tener en cuenta que esta aplicación web no permite ser usada si no se ha iniciado sesión. Si se intenta acceder a cualquier recurso de la web sin estar autenticado, el usuario será redirigido automáticamente a la página de inicio de sesión.

A continuación, mencionaremos todos los endpoints que se han creado y explicaremos su función sin entrar en detalles sobre la implementación de los mismos:

Endpoint base "/":

- Es el endpoint de entrada a la aplicación web, cargará la página que contiene el mapa con el formulario principal, llamada "index.html".

Endpoint "/dashboard":

- Se accede a él a través del botón "Mi cuenta" del menú desplegable superior (Figura 2).
- Cargará la página "dashboard.html".

Endpoint "/admin":

- Se accede a través del botón "Admin" del menú desplegable superior (Figura 2)
- Solo será visible si el usuario autenticado es administrador. Más adelante se explicará su utilidad.
- Cargará la página "admin.html".

Endpoint "/historial":

- Se accede a través del botón "Simulaciones" del menú desplegable superior (Figura 2).
- Cargará la página "historial.html", que contiene una lista de todas las simulaciones realizadas por el usuario.

Figura 2: Menú Desplegable

Endpoint "/login"

- Se disparará automáticamente si no se está autenticado.
- Cargará la página "login.html" que contiene el formulario para iniciar sesión.

Endpoint "/registro"

- Se accederá a través del enlace "aquí" ubicado en la página "login.html" (Figura 3).
- Cargará la página "registro.html" que contendrá el formulario para registrarse en la web.

Figura 3: Inicio de sesión

Endpoint "/reference_spectra"

- Se accede a través del botón azul "Usar Espectros de Referencia" ubicado debajo del mapa (Figura 4).
- Carga la página "reference_spectra.html", que sustituye el formulario de datos del simulador por una lista desplegable de espectros de referencia.

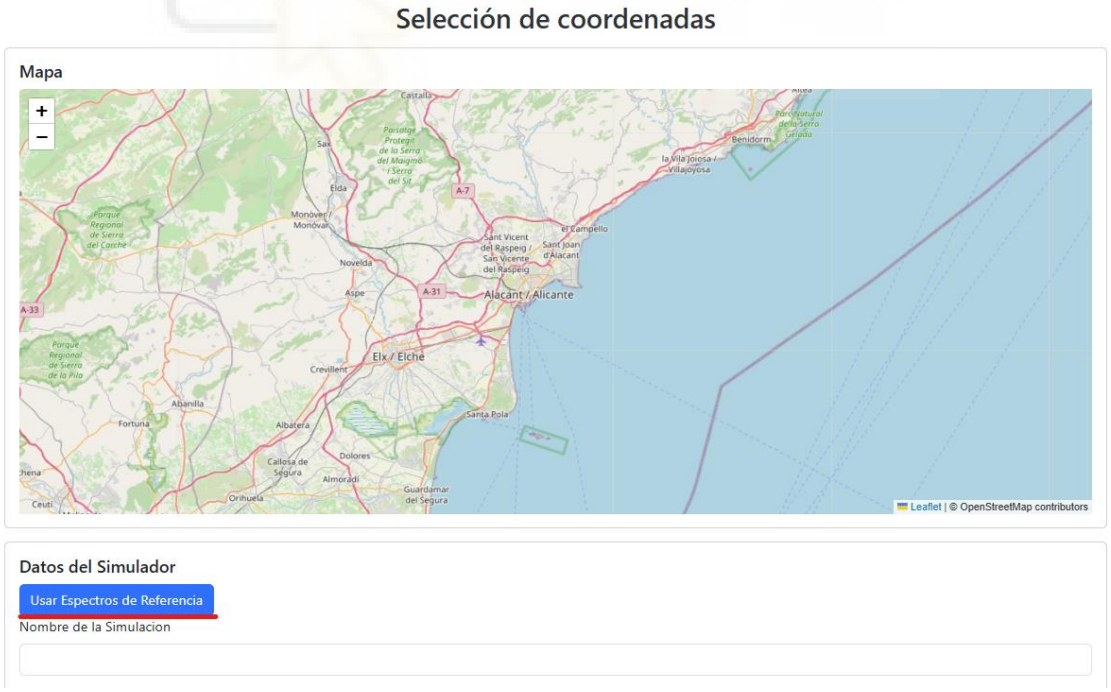


Figura 4: Botón Espectros de Referencia

Endpoint "/panels"

- Se dispara automáticamente al acceder a las páginas "index.html" y "reference_spectra.html".
- Accede a la base de datos y carga la lista de paneles disponibles para que aparezca en el desplegable de los formularios (Figura 5).

Endpoint "/agregarPanel" por Get

- Se accede a través del enlace "agrega un panel" (Figura 5).
- Cargará la página "add_panel.html" que contendrá el formulario para crear un panel nuevo.

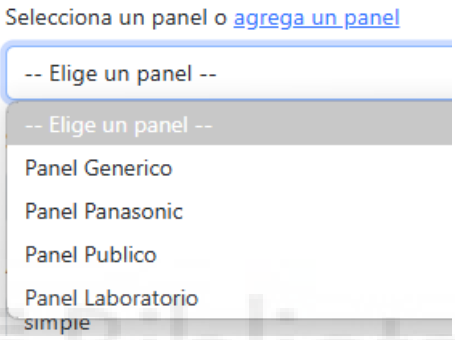


Figura 5: Selección de panel

Endpoint "/agregarPanel" por Post

- Se accede a través del botón guardar de la página "add_panel.html".
- Accede a la base de datos y añade al panel con los datos introducidos por el usuario. Una vez finalizado, redirige al endpoint "/", que nos llevará a la página "index.html"

Endpoint "/cambiarPassword"

- Se accede a través del botón "Cambiar contraseña" de la página "dashboard.html".
- Carga la página "cambiarPassword.html", que permite cambiar la contraseña de la cuenta.

Endpoint "/simulando/status"

- Este endpoint comprueba si algún usuario está lanzando una simulación. Sirve para que dos usuarios no puedan lanzar simulaciones simultáneamente.

Endpoint "/simulate/full"

- Se accede a través del botón "Lanzar simulación" ubicado al final del formulario en la página "index.html"

- Es el endpoint que maneja la lógica para poner en marcha el simulador solar y espera a recibir la matriz resultado del SMU. Comprueba que no haya nadie simulando antes de empezar el proceso. Activa el flag de simulación para que nadie más pueda simular mientras se está realizando esta simulación. Una vez recibe los datos, los procesa y guarda la curva IV en la base de datos. Además, guarda todos los datos del escenario en la base de datos también. Una vez finalizado el proceso, nos redirige al endpoint "/resultado", el cual se explicará más adelante.

Endpoint "/simulate/reference_spectra"

- Se accede a través del botón "Lanzar simulación" ubicado al final del formulario en la página "reference_spectra.html"
- Es un endpoint muy similar al anterior, pero al utilizarse espectros de referencia, las funciones que activan el simulador son distintas; esto se explicará en la parte de implementación. Además, al no haber un escenario personalizado, no se guarda ningún escenario en la base de datos. Al igual que el endpoint anterior, al finalizar este proceso se redirige al endpoint "/resultado".

Endpoint "/resultado"

- Este endpoint se dispara automáticamente por redirección de los endpoints "/simulate". También se puede acceder al pulsar el botón "Ver detalle" de la página historial.html (Figura 6).
- El uso de este endpoint es un poco particular, ya que acepta parámetros en la url. Estos parámetros permiten saber qué simulación, y con qué algoritmo SDM tiene que cargar de la base de datos para mostrar resultados. Se explicará su funcionamiento en detalle en el apartado de implementación.
- En cuanto a diseño, el propósito de este endpoint es cargar los datos necesarios de la base de datos (simulación, algoritmo, etc.) para mostrar la página "resultado.html" de forma correcta.

Endpoint "/simulación/{sim_id}/borrar"

- Para disparar este endpoint debemos pulsar en el botón borrar de la página "historial.html" (Figura 6).
- Le llega en la url el id de la simulación que se quiere borrar, y realiza la operación de borrado en la base datos.

Figura 6: Historial de simulaciones

Endpoint "/simulación/{sim_id}/download_csv"

- Se dispara al pulsar el botón Descargar CSV, en la parte superior de la página "resultado.html". (Figura 7)
- Obtiene los siguientes datos de la base de datos referentes a la simulación especificada en la URL: Algoritmo, Iph, Isat, a, Rsh, Rs, RMSE, MAE, Area Factor, Voc, Isc, Vmpp e Impp. Crea un archivo CSV con los datos y sirve directamente la descarga.
- Como dato adicional, no crea ficheros temporales en el servidor, es una descarga directa. Se extenderá la explicación técnica en la parte de implementación.
- Para entender mejor cómo queda el fichero CSV resultante, se adjunta una imagen en un visualizador de ficheros CSV en la Figura 8 y Figura 9

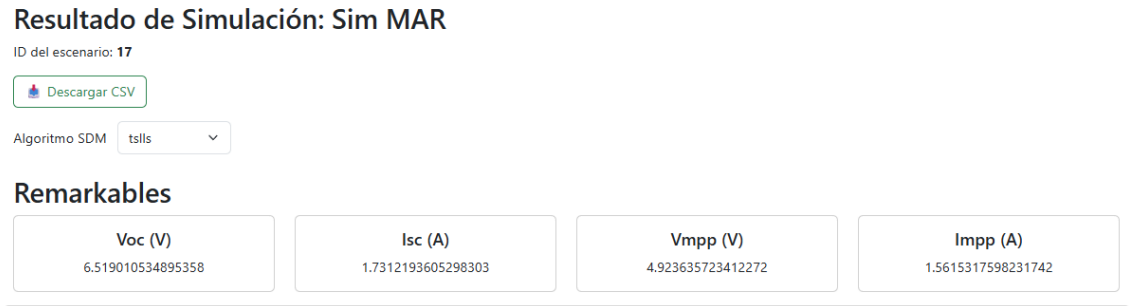


Figura 7: Botón Descargar CSV

	A	B	C	D	E	F
1	algoritmo	Iph	Isat	a	Rsh	Rs
2	tslls	1.033568541	2.27E-06	1.290079615	687.8733659	1.259967401
3	tslls-refined	1.03238234	2.51E-06	1.30015121	744.7139773	1.23928886

Figura 8: CSV descargado Parte 1

G	H	I	J	K	L	M
RMSE	MAE	Area Factor	Voc	Isc	Vmpp	Impp
0.002172280837	0.001724295665	0.001876344415	16.77676335	1.031674888	12.65103652	0.9129710575
0.0020465347	0.001692321069	0.002030794958	16.77699347	1.030663001	12.6552438	0.9126536549

Figura 9: CSV descargado Parte 2

4.2.2 Diagrama de relación y uso de endpoints

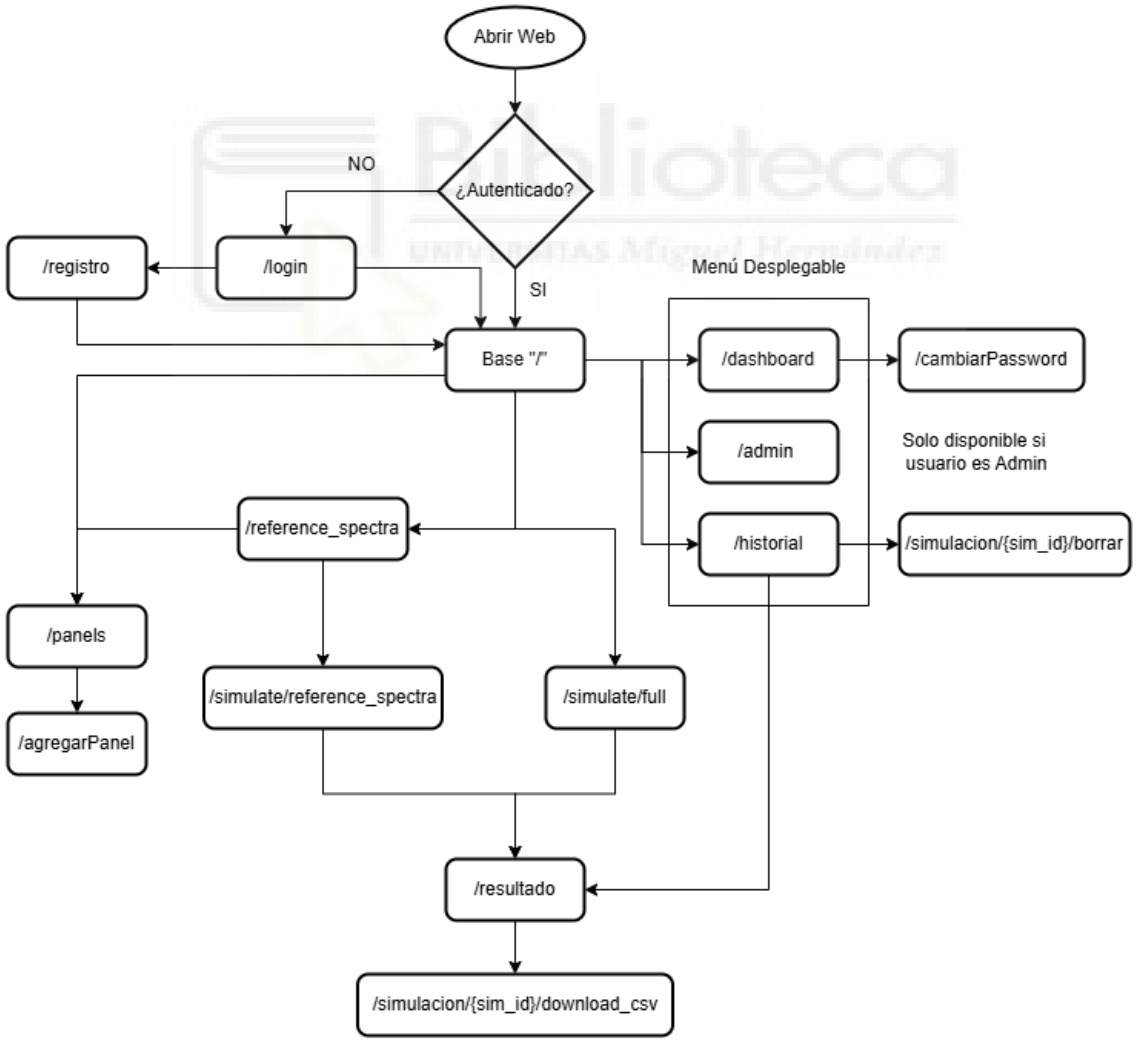


Figura 10: Diagrama de endpoints

4.2.3 Diseño de la Base de Datos

En este apartado, se va a explicar a nivel de diseño la base de datos utilizada para el proyecto. En el apartado de implementación se explicará técnicamente cómo se ha llevado a la realidad este diseño.

Para comprender la base datos, primero se van a especificar todas las tablas con sus respectivos atributos y tipo de dato de los mismos. A continuación, se explicarán las relaciones entre tablas. Debemos tener en cuenta que se trata de una base de datos relacional.

Por defecto, todos los campos son obligatorios; los que no lo sean, se indicarán.

Tabla User

Es heredada directamente de FastAPI Users, contiene campos típicos de usuarios. En este caso, se ha usado el modelo base y solo se han utilizado los campos email y password.

Tabla Escenarios

Esta tabla contiene toda la información respectiva a los escenarios que se van creando conforme el usuario va realizando simulaciones. A nivel lógico, se considera un escenario todos los parámetros geográficos y atmosféricos que el usuario introduce en el formulario de la página “index.html” para realizar simulaciones con todos los parámetros personalizados.

Los campos de esta tabla son los siguientes:

- id (integer, Primary Key)
- datetime (datetime)
- time_zone (string)
- latitude (float)
- longitude (float)
- surface_tilt (float)
- surface_azimuth (float)
- ground_albedo (float)
- water_vapor_content (float)
- ozone (float)
- aerosol_turbidity_500nm (float)
- pressure (float)
- altitude (float, no obligatorio)
- temperatura (float)
- spa_method (string)
- airmass_model (string)
- usuario (string). Hace referencia al id del usuario de la tabla User

Tabla Paneles

Esta tabla guarda información relativa a los paneles que los usuarios agregan. En esta tabla se registra el nombre y descripción del panel, el usuario que ha creado el panel, y se utiliza un atributo llamado public de tipo booleano para saber si el panel es público o privado.

Los campos de esta tabla son:

- id (Integer, Primary Key)
- descripcion (String)
- public (Boolean)
- usuario (String). Hace referencia al id del usuario de la tabla User
- nombre (String)

Tabla Simulacion

Esta tabla contiene los datos relativos a los resultados de una simulación. En ella se almacena el array de Is y Vs medidos por el SMU. Por motivos técnicos, se almacenarán en la base de datos como cadenas de texto. La tabla Simulación está relacionada con la tabla Escenarios, de manera que los escenarios estén asociados con las simulaciones. Además, se almacena también el panel usado y el usuario que ha realizado la simulación.

- id (Integer, Primary Key)
- I (String)
- V (String)
- escenario (Integer, no obligatorio). Hace referencia al id del escenario asociado a esta simulación.
- spectra (String, no obligatorio)
- panel (Integer). Hace referencia al id del panel usado en la simulación.
- nombre (String)
- usuario (String) Hace referencia al id del usuario de la tabla User

Los campos escenario y spectra no son obligatorios, ya que una simulación puede ser lanzada utilizando escenarios personalizados o espectros de referencia. Por tanto, si lanzamos una simulación con un escenario personalizado, el atributo spectra en esa tupla quedará vacío, y viceversa si utilizamos un espectro de referencia.

Tabla SDM

Esta tabla guarda los resultados de aplicar a los datos obtenidos de la simulación los algoritmos SDM. Conceptualmente es necesario entender que, por cada simulación, habrá dos tuplas en esta tabla. Esto es debido a que hay dos posibles algoritmos SDM implementados en la aplicación, entonces se guardan los datos obtenidos de aplicar un algoritmo en una tupla, y luego se crea otra tupla con los datos obtenidos de aplicar el otro algoritmo SDM. ¿Por qué es útil guardarlo en la base de datos y no calcularlo cada vez que el usuario decida cambiar entre

algoritmos? Esto se debe a que guardar esta información en la base de datos consume muy pocos recursos y solo se hace una vez por simulación. Además, mejora la experiencia de usuario al cambiar entre algoritmos, ya que el tiempo de respuesta del sistema es mucho más rápido, puesto que es más rápido computacionalmente hacer una consulta a la base de datos que calcular los parámetros SDM de nuevo, y, como veremos en la sección de implementación, las otras opciones complicaban demasiado el código.

Los atributos de esta tabla son:

- id (Integer, Primary Key)
- simulacion (Integer). Hace referencia a la simulación a la que están asociados estos valores.
- Algoritmo (Integer). Hace referencia al algoritmo al que corresponden los valores obtenidos, se obtiene de la tabla algoritmo.
- iph (Float)
- isat (Float)
- a (Float)
- rsh (Float)
- rs (Float)
- rmse (Float)
- mae (Float)
- areaFactor (Float, no obligatorio)
- voc (Float, no obligatorio)
- isc (Float, no obligatorio)
- vmpp (Float, no obligatorio)
- impp (Float, no obligatorio)
- extra_info (String, no obligatorio)

Tabla Algoritmo

Esta tabla guarda los diferentes algoritmos SDM disponibles. Se ha decidido incluirlos en la base de datos porque esto permite una mayor escalabilidad en el futuro, ya que podemos añadir aquí un algoritmo nuevo y empezar a utilizarlo, por lo menos a nivel de base de datos. Es una tabla muy sencilla que solo guarda el nombre del algoritmo.

Sus campos son:

- id (Integer, Primary Key)
- nombre (String)

Tabla Simulando

Esta tabla solo contiene un campo booleano que por defecto será falso, y servirá como flag para saber si hay algún usuario realizando una simulación.

Atributos de esta tabla:

- id (Integer, Primary Key)
- sim (Boolean)

Si el atributo sim es True, significa que hay alguien simulando y, por lo tanto, nadie más puede lanzar una simulación.

Relaciones entre tablas

Para explicar las relaciones entre tablas, primero vamos a definir las relaciones foráneas de la siguiente forma: Tabla.Atributo FK → Tabla.Atributo.

Por ejemplo:

Escenarios.usuario FK → User.id

Esto quiere decir que el atributo usuario de la tabla Escenarios es clave foránea correspondiente con el id de un usuario en la table User.

Lista de relaciones:

- Escenarios.usuario FK → User.id

- Paneles.usuario FK → User.id

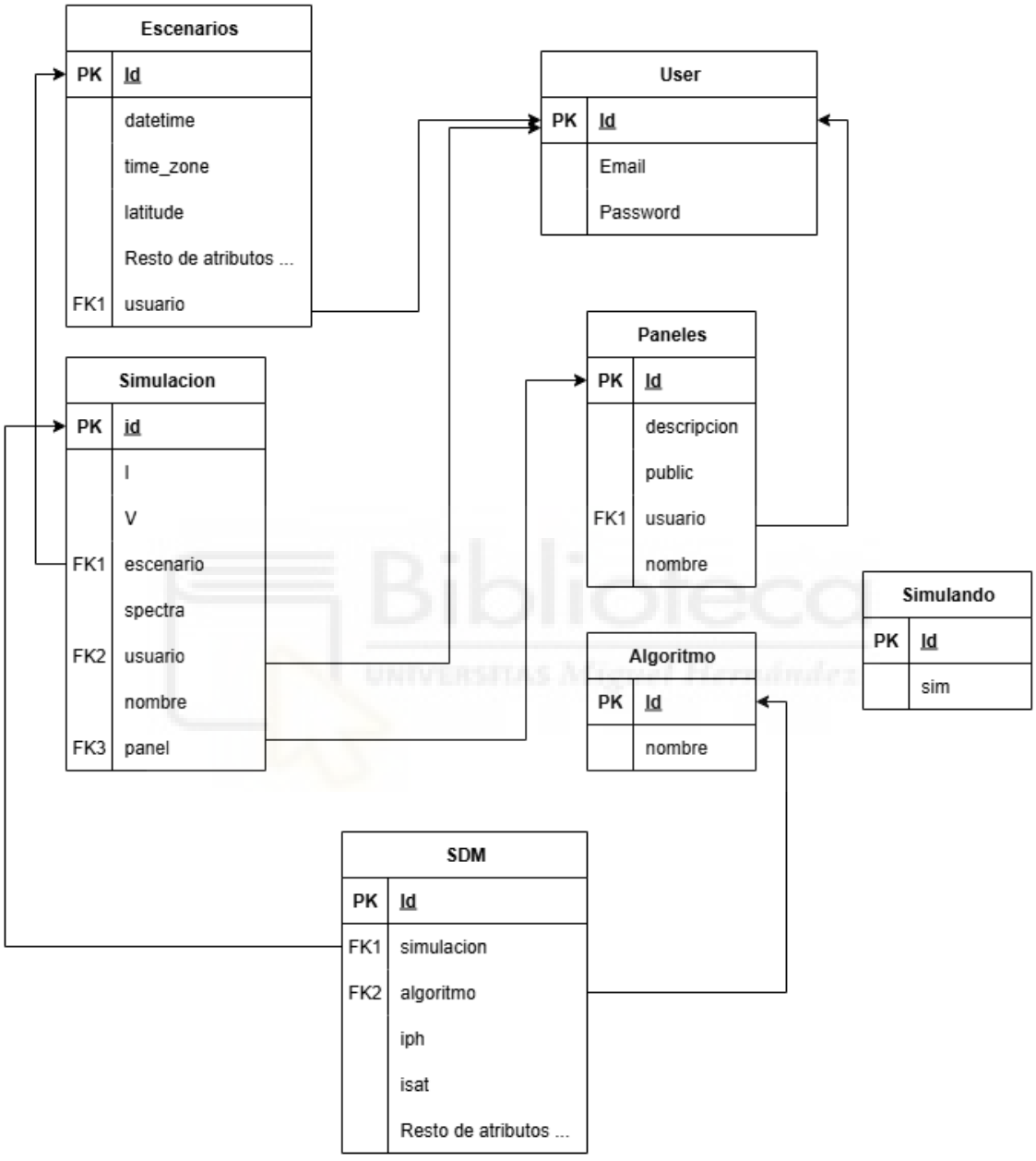
- Simulacion.escenario FK → Escenarios.id
- Simulacion.panel FK → Paneles.id
- Simulacion.usuario FK → User.id

- SDM.simulacion FK → Simulacion.id
- SDM.algoritmo FK → Algoritmo.id

Para entender mejor las relaciones entre tablas, se va a adjuntar un diagrama de la base de datos siguiendo el modelo relacional.

4.2.4 Diagrama Relacional de la Base de Datos

En el diagrama, se han omitido algunos atributos para mantener el tamaño de las tablas reducido y mejorar la legibilidad.



4.2.5 Archivo CRUD

El término CRUD hace referencia a las operaciones básicas que se pueden realizar sobre una base de datos: Crear (Create), Leer (Read), Actualizar (Update) y Eliminar (Delete).

Para simplificar el acceso a la base de datos y reducir el número de consultas SQL en la implementación de los endpoints, se ha creado un fichero CRUD, que simplifica el código y nos permite hacer cambios en las consultas sin tener que modificar el código principal.

Las funciones que se han incluido en este fichero:

- Función para obtener la lista con todos los usuarios (get_all_users)
- Función para obtener la lista de paneles del usuario y de paneles públicos (get_user_panels)
- Función para añadir un nuevo panel a la base de datos (create_panel)
- Función para añadir un nuevo escenario a la base de datos (create_scenario)
- Función para guardar una simulación creada con escenario personalizado (create_IV)
- Función para guardar una simulación creada con espectro de referencia (create_IV_spectra)
- Función para guardar los parámetros SDM de una simulación en la base de datos (create_sdm)
- Función que devuelve verdadero o falso para saber si hay alguien simulando (get_simulando_state)
- Función para cambiar el estado del flag de simulación (set_simulando_state)

Los detalles de implementación, así como las consultas SQL realizadas para poder realizar estas funciones serán explicadas en el apartado de implementación.

4.3 Diseño del frontend

En este apartado, se mostrarán con imágenes todas las páginas de las que dispone la web y se explicarán los motivos y decisiones que se han tomado para haber elegido este diseño. El objetivo principal de este apartado es que el lector vea cómo es la página web a ojos de un usuario cualquiera.

4.3.1 Inicio de sesión

Primero veremos el inicio de sesión, ya que, si no se está autenticado, será lo primero que un usuario encontrará:

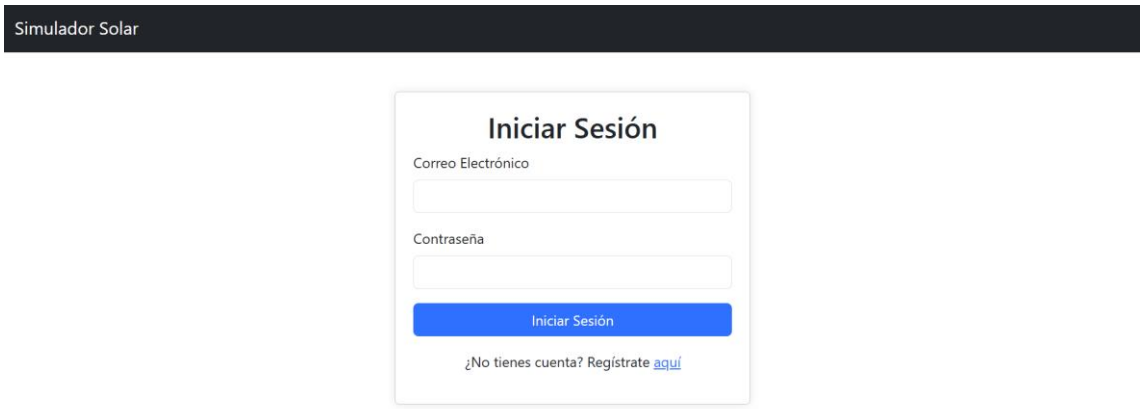


Figura 11: Barra superior

Como se puede apreciar en la Figura 11, vemos una barra superior que se mantendrá en todas las páginas de la web. Además, el aspecto es sobrio y minimalista, sin demasiada complejidad y siguiendo el formulario estandarizado en la mayoría de páginas webs para el inicio de sesión.

4.3.2 Registro

El formulario de registro es muy similar al anterior, como ya se ha mencionado en apartados anteriores, se accede a él pulsando en el enlace “aquí” de la Figura 11.

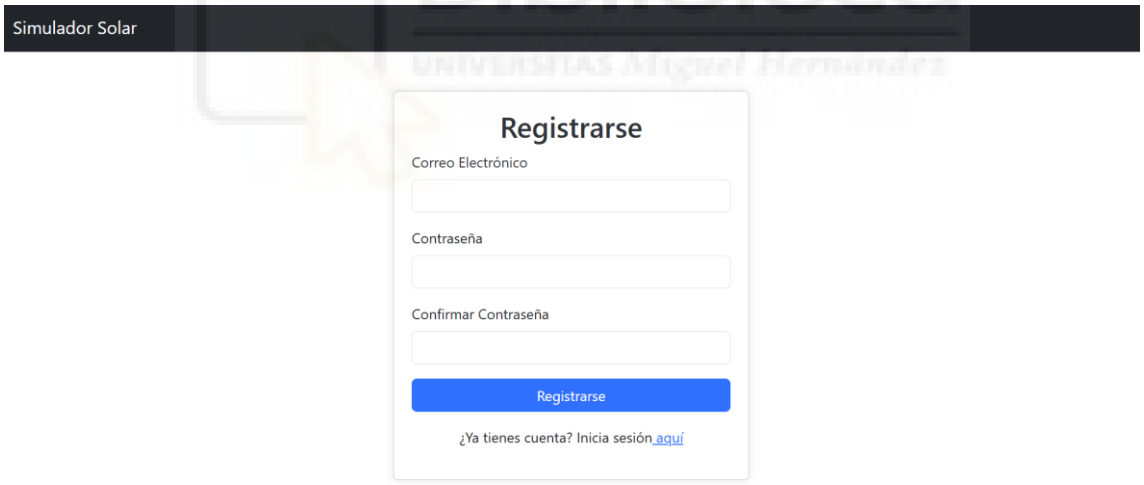


Figura 12: Registro

Se observa en la Figura 12 que se ha incluido un campo para confirmar la contraseña y prevenir de esta forma errores escribiendo la contraseña deseada que pueden llevar a problemas de inicio de sesión por parte del usuario. Se puede apreciar en la parte inferior, que podemos volver al formulario de inicio de sesión si lo deseamos pulsando en el enlace inferior resaltado en color azul.

4.3.4 Página principal

En la página principal encontraremos un mapa de selección de coordenadas y el formulario completo para poder lanzar una simulación en un escenario completamente personalizado. Mantiene un diseño muy limpio, con la barra superior ahora con un menú desplegable y siguiendo la línea de colores que se ha utilizado en las páginas de inicio de sesión y registro.

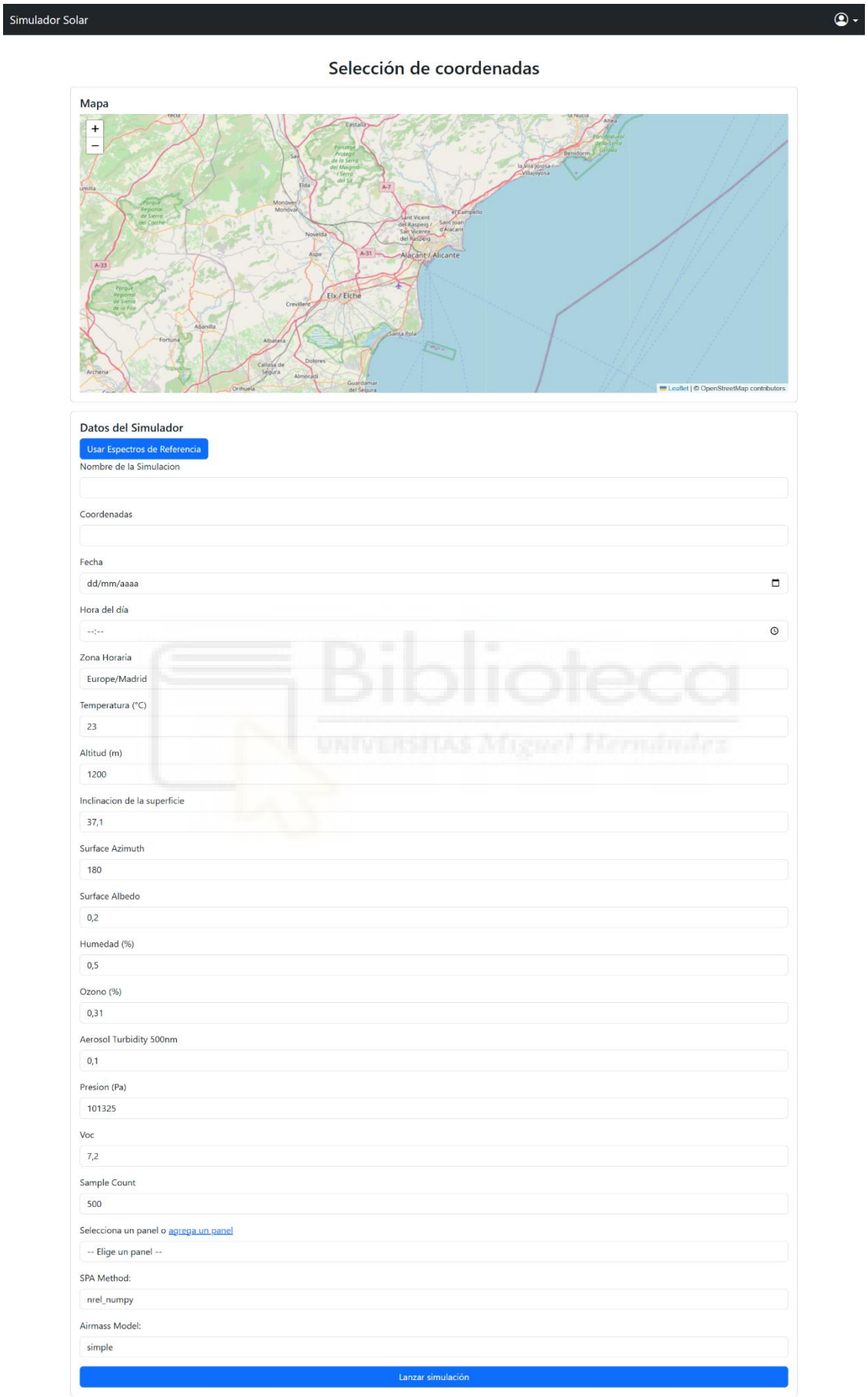


Figura 13: Página principal

En la imagen podemos apreciar que, en el formulario, hay unos valores por defecto para facilitar al usuario entender el rango de valores que son aceptables para una simulación, aunque la mayoría de usuarios que usen este sistema tendrán conocimientos sobre simulación solar. Además, vemos el botón de “Usar Espectros de Referencia”, que nos dirigirá a la página para lanzar simulaciones con espectros de referencia.

Si pulsamos sobre un punto del mapa, veremos que sale un marcador, y que el campo coordenadas se rellena automáticamente (Figura 14). Esto es muy útil si se conoce la ubicación en el mapa, pero no sus coordenadas, ya que evitamos que el usuario vaya a otra web, como Google Maps, para obtener las coordenadas.

Figura 14: Selección de coordenadas

Menú desplegable

El menú desplegable está disponible en todas las páginas pulsando sobre el icono mostrado en la Figura 15. Las opciones de este menú cambian dependiendo de si el usuario autenticado es administrador o no, como podemos ver en la Figura 15 (usuario normal) y en la Figura 16 (administrador)

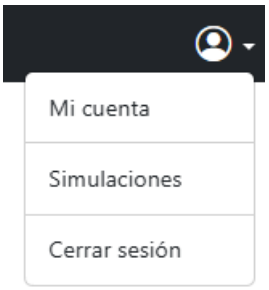


Figura 15: Menú desplegable usuario normal

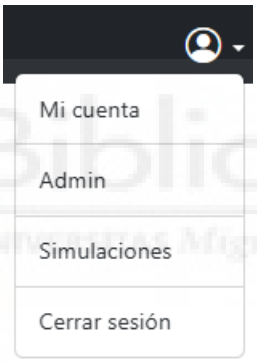


Figura 16: Menú desplegable Administrador

4.3.5 Espectros de referencia

Continuaremos con la página para lanzar la simulación usando espectros de referencia. Esta es mucho más sencilla que la principal, ya que no es necesario introducir información atmosférica ni geográfica. Como podemos ver en la Figura 17, prescindimos del mapa y de la mayoría de datos del formulario, mostrando un desplegable (Figura 18) con los espectros de referencia disponibles y permitiendo solo elegir un nombre, seleccionar un panel y establecer un Voc y Sample Count.

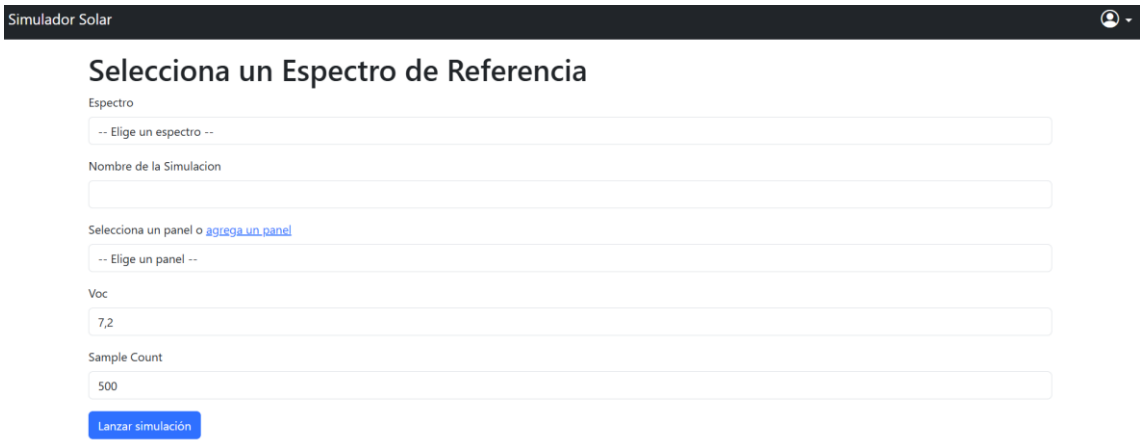


Figura 17: Página espectros de referencia

Figura 18: Desplegable de espectros de referencia

4.3.6 Agregar un panel

Esta es la página a la que se accederá cuando se quiera añadir un nuevo panel a la colección de paneles. Es muy sencilla, se introduce un nombre y una descripción del panel, y si marcamos la casilla “Público”, todos los usuarios podrán ver y usar ese panel.

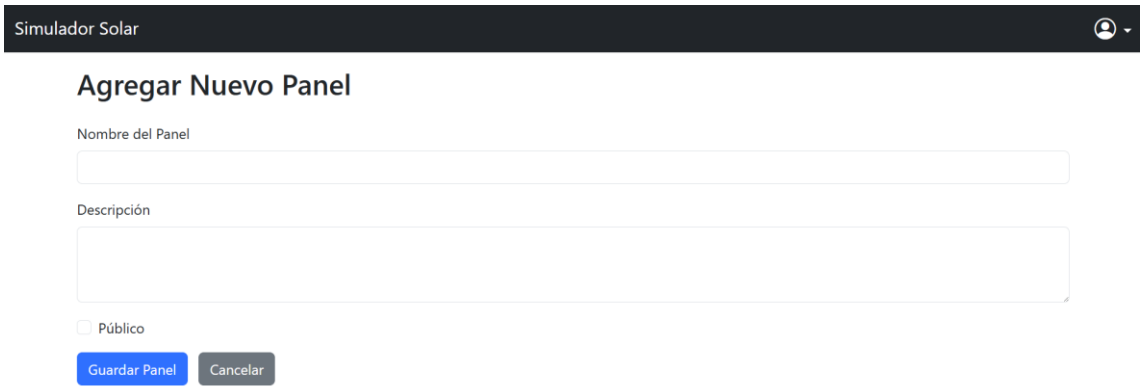


Figura 19: Formulario agregar panel

4.3.7 Mi cuenta

Esta página es uno de los puntos de mejora que se explicarán con detalle en el apartado de propuestas de mejora. Actualmente cuenta con una interfaz muy sencilla que muestra el correo del usuario autenticado y le permite pulsar un botón para cambiar la contraseña.

Figura 20: Página Dashboard

4.3.8 Cambiar contraseña

Esta página permite al usuario autenticado cambiar la contraseña para su cuenta. El usuario encontrará un campo de un formulario para introducir su nueva contraseña, así como un botón azul, siguiendo la línea de colores de la web, para confirmar el cambio.

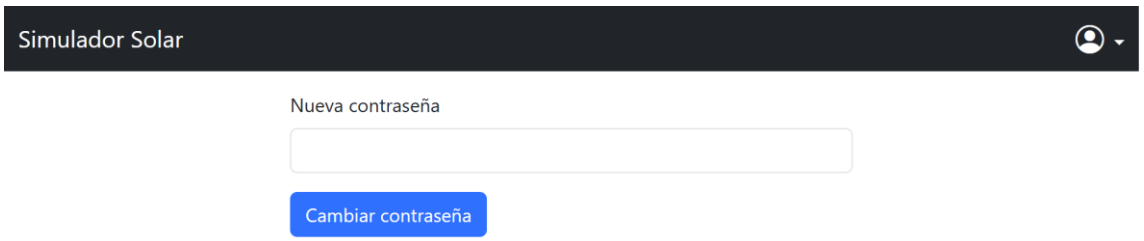


Figura 21: Cambio de contraseña

4.3.9 Página de administrador

En esta página, que solo es accesible por usuarios administradores, se visualizará una lista de usuarios registrados. El administrador podrá activar/desactivar sus perfiles, hacerlos administradores o borrar por completo sus perfiles, incluyendo los datos.

En la Figura 22 se puede ver el aspecto de la página:

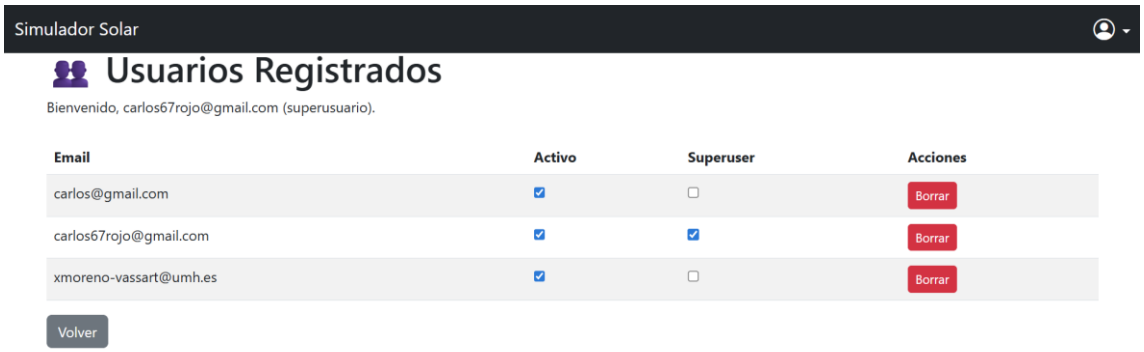


Figura 22: Menú de Administrador

4.3.10 Historial de simulaciones

En esta página el usuario podrá ver una tabla con el historial de simulaciones que ha ido realizando con el paso del tiempo. Aquí la información sobre las simulaciones es muy reducida, pero cuando se pulse en el botón “Ver detalle” el usuario visualizará la simulación completa. También se ha incluido en esta página un botón en color rojo para eliminar simulaciones.

Esta es la página de historial de simulaciones:

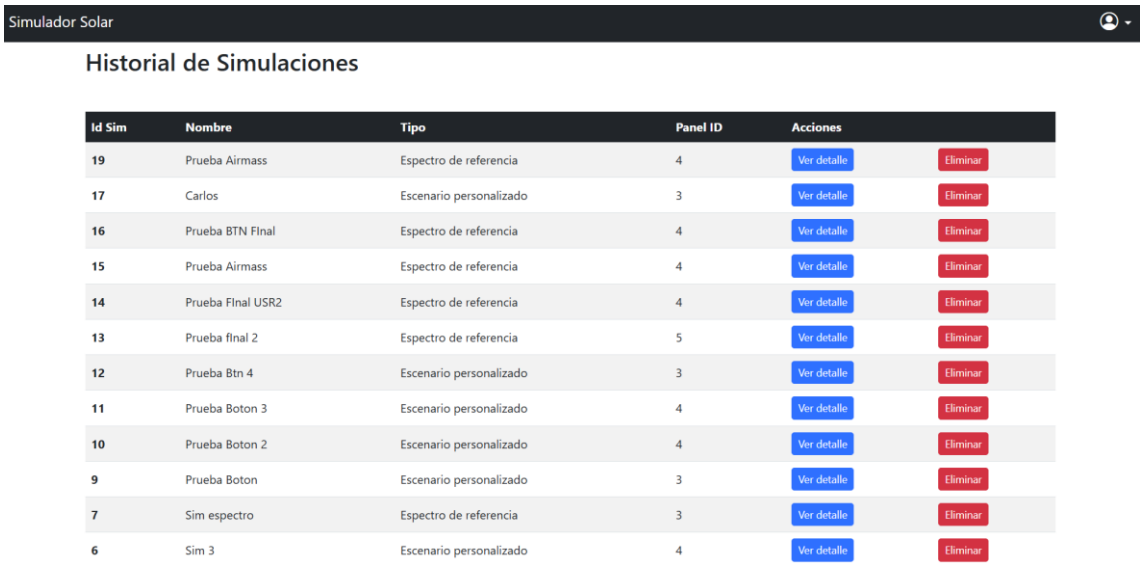


Figura 23: Historial de simulaciones

4.3.11 Resultados

En esta página, el usuario visualiza los resultados de una simulación. Para las imágenes de muestra, utilizaremos el resultado de una simulación real que fue realizada en el laboratorio. En la Figura 24 podemos ver lo primero que encuentra un usuario al entrar a la página de resultados. Primero está el nombre, seguido del botón para descargar los resultados en CSV, y el desplegable de selección de algoritmo SDM. A continuación, se muestran los puntos remarkables, acompañados de una gráfica que representa los resultados.

En esta gráfica, encontramos en el eje X el voltaje, y en el Y la intensidad. Los puntos marcados en negro son cada una de las muestras medidas por el SMU. Habrá tantos puntos como el valor establecido en el Sample Count. La línea roja es la curva IV resultante de aplicar el algoritmo SDM seleccionado. La línea amarilla representa la curva de potencia, es el resultado de multiplicar voltaje e intensidad. Por último, los puntos grandes marcados en rojo, verde y azul representan los remarkables. Se puede observar que el punto verde (punto de máxima potencia) coincide con el punto más alto de la curva de potencia.

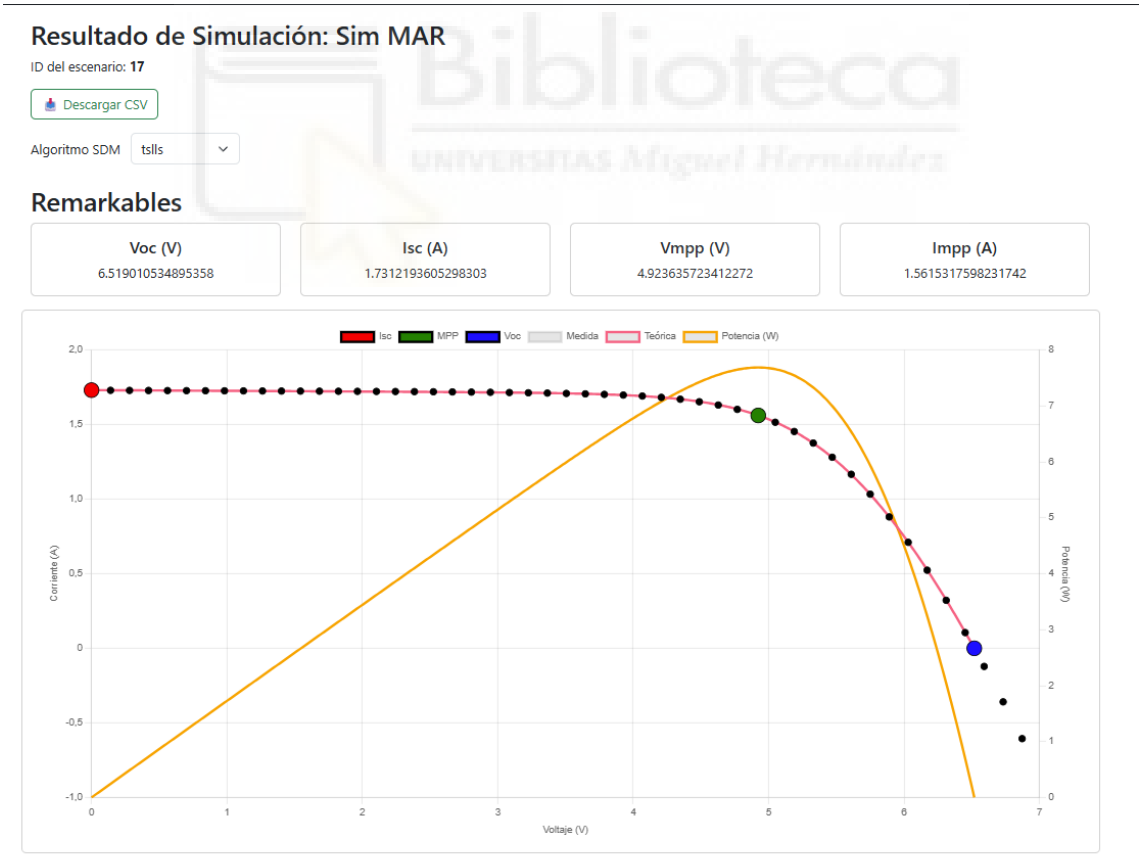


Figura 24: Gráfico de resultados

Si nos seguimos desplazando hacia abajo en la página, encontraremos toda la información de la simulación, que estará recogida en una tabla, como se puede apreciar en la Figura 25

Resultados de la simulación

Voc (V)	6.519010534895358
Isc (A)	1.7312193605298303
Vmpp (V)	4.923635723412272
Impp (A)	1.5615317598231742
RMSE	0.0011805149009921463
MAE	0.0008956090058215055
Area Factor	0.0005912095618750341
Iph (A)	1.7342152406289388
Isat (A)	2.509885964498474e-07
a (n)	0.4144118196103436
Rsh (Ω)	214.75260902496456
Rs (Ω)	0.37151415969393686
Método	TSLLS

Figura 25: Valores resultantes

Por último, encontraremos la sección que muestra los detalles del escenario utilizado para la simulación:

Detalles del Escenario

ID	17
Fecha y Hora	2025-06-02 12:00:00
Zona Horaria	Europe/Madrid
Latitud	34.09634
Longitud	36.061879
Inclinación	37.1
Azimuth	180.0
Albedo	0.2
Agua (cm)	0.5
Ozono (atm-cm)	0.31
Turbidez	0.1
Presión (Pa)	101325.0
Altitud (m)	1200.0
Temperatura (°C)	23.0
SPA Method	nrel_numpy
Airmass Model	simple
Panel	Panel Laboratorio
Descripcion Panel	panel del lab

Figura 26: Detalles de escenario

Esta última sección (Figura 26) no estará disponible si se ha usado un espectro de referencia. En ese caso, encontraremos una sección como la de la Figura 25, pero habiendo añadido la información del panel en esa misma tabla:

Resultados de la simulación

Voc (V)	5.412397944233476
Isc (A)	0.10291521812998262
Vmpp (V)	4.3676628563141096
Impp (A)	0.09023203974189406
RMSE	0.000342204494662818
MAE	0.00026974617440635466
Area Factor	0.0029890282408826974
Iph (A)	0.10296264103191917
Isat (A)	1.3832847509971684e-07
a (n)	0.4020672050875762
Rsh (Ω)	920.2112179380981
Rs (Ω)	0.42388772863860397
Método	TSLLS
Panel	Panel Publico
Descripcion Panel	Panel para todos los publicos

Figura 27: Valores resultantes al usar espectros de referencia

5. Implementación

En este apartado, se va a explicar en detalle cómo se ha llevado a la realidad todo lo descrito en el apartado de diseño. Va a ser la sección más técnica de este documento y es necesario tener conocimientos previos de programación para comprender al completo el proceso de implementación del sistema.

5.1 Implementación del backend

Se comenzará explicando la implementación de la base de datos, ya que, para entender la implementación de los endpoints, es importante saber cómo está construida la base de datos. Después de la base de datos, se explicarán los endpoints y, por último, la implementación del sistema de gestión de usuarios con FastAPI Users.

5.1.1 Implementación Base de Datos

Para la realización de la base de datos, se ha utilizado SQLite en modo asíncrono usando el driver aiosqlite.

Aiosqlite es una interfaz asíncrona para bases de datos sqlite. Replica el módulo estándar de sqlite, pero con versiones asíncronas de todos métodos de conexión y cursores. Se ha optado por usar una base de datos asíncrona para poder aprovechar al máximo el potencial asíncrono nativo de FastAPI. Al disponer de un acceso a la base de datos asíncrono, el servidor no quedará bloqueado esperando a una consulta en la base de datos.

Para definir que estamos usando aiosqlite, tenemos que especificarlo en la DATABASE_URL. En nuestro código:

```
DATABASE_URL = "sqlite+aiosqlite:///./test.db"
```

A continuación, creamos el motor asíncrono de la base de datos con “create_async_engine(DATABASE_URL)” y luego creamos un async_sessionmaker, que nos permitirá gestionar las sesiones de forma eficiente. Seguidamente definimos un generador asíncrono “get_async_session()” que se usará junto con Depends() en los endpoints para poder obtener sesiones de bases de datos.

Como queremos que se creen todas las tablas automáticamente al iniciar la aplicación, tenemos que definir una función de creación de tablas de tal forma que cuando arranque el motor, se creen las tablas. La función resultante es esta:

```
async def create_db_and_tables():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
```

En cuanto a la gestión de usuarios, utilizamos la tabla User heredada de la librería “SQLAlchemyBaseUserTableUUID”, que es el modelo base proporcionado por FastAPI Users. Como se mencionó en el apartado de diseño, nuestros usuarios solo cuentan con los campos email y contraseña, por tanto, no necesitamos modificar la clase base y podemos dejarlo de esta forma:

```
class User(SQLAlchemyBaseUserTableUUID, Base):
    pass
```

donde Base es una clase heredada de DeclarativeBase proporcionado por sqlalchemy.orm:

```
class Base(DeclarativeBase):
    pass
```

Continuamos con la definición del resto de tablas, que han sido definidas como clases ORM, cada una con sus atributos, claves principales y claves foráneas.

Para ver un ejemplo de cómo se han implementado las tablas, se va a mostrar el código de implementación de una de las tablas, que contenga claves foráneas. Se ha elegido la tabla Simulacion:

```
class Simulacion(Base):
    __tablename__ = "simulacion"

    id = Column(Integer, primary_key=True, index=True)
    I = Column(String, nullable=False)
    V = Column(String, nullable=False)
        escenario = Column(Integer, ForeignKey("escenarios.id"),
nullable=True)
    spectra = Column(String, nullable=True)
```

```
    panel = Column(Integer, ForeignKey("paneles.id"), nullable=False)
    nombre = Column(String, nullable=False)
    usuario = Column(String,  ForeignKey("user.id"), nullable=False)
```

Como se puede apreciar, la forma de implementar una tabla es establecer la clase, definir su nombre (será para referirnos a esta tabla en código), los atributos y las relaciones. Para establecer una relación, simplemente debemos escribir “ForeignKey()” e indicar a qué atributo de la tabla con la que se relaciona se refiere.

Con esto, hemos finalizado la implementación de la base de datos. Cabe destacar que esta implementación es compatible con otros motores de base de datos, simplemente habría que modificar la DATABASE_URL y elegir otro motor. Además, al estar preparada para entornos asíncronos, permite escalabilidad para un proyecto más grande.

5.1.2 Fichero CRUD

Para aumentar la legibilidad y sencillez de código en la creación de los endpoints, se ha creado un fichero CRUD con el objetivo de reducir las consultas directas a la base de datos en los endpoints.

Si bien nuestro fichero CRUD no tiene implementadas todas las operaciones que se podrían hacer sobre la base de datos, se han implementado las que requieren de consultas y código complejo. Para explicar la implementación, se explicará la función de cada una de las funciones y las consultas necesarias para realizar la función; en algunas funciones, se explicarán detalles adicionales.

Get_all_users(session):

Devuelve una lista con todos los usuarios registrados en la base de datos.

La consulta realizada en esta función es:

```
result = await session.execute(select(User))
```

Como podemos ver, un simple select de User, y lo almacenamos en result.

Get_user_panels(db, user_id)

Obtiene todos los paneles asociados a un usuario, incluyendo aquellos marcados como públicos. Como parámetros necesita el id del usuario y una sesión de base de datos.

Para realizar la consulta, ya que el user_id es de tipo UUID, debemos convertirlo a string:

```
user_id_str = str(user_id)
```

Ahora ya podemos realizar la consulta:

```
stmt = select(Paneles).where(
        or_(Paneles.usuario == user_id_str, Paneles.public == True)
    )
```

La ejecutamos con “db.execute”, como en el ejemplo anterior, y devolvemos el resultado.

Create_panel(db, user_id, descripcion, public)

Permite crear un nuevo panel vinculado a un usuario. La función requiere que se pasen como parámetros los valores que se quieren guardar con el panel.

Al igual que antes, convertimos el user_id a string. A continuación, creamos un objeto Paneles con los valores de los parámetros y lo añadimos a la base de datos de la siguiente forma:

```
db.add(nuevoPanel)
await db.commit()
await db.refresh(nuevoPanel)
```

Y, por último, devolvemos el objeto Panel.

Create_scenario(...)

Inserta en la base de datos un nuevo escenario. Necesita que se le pasen como parámetros todos los datos que se necesitan guardar en la base de datos. La construcción de la función es muy similar a la anterior. Creamos un objeto Escenarios (nótese que el objeto se corresponde con la clase de la tabla escenarios en la Base de Datos) con los valores obtenidos en los parámetros de la función. Por último, lo añadimos a la base de datos, al igual que en la función anterior, y devolvemos el objeto Escenarios.

Create_IV(...)

Guarda una simulación asociada a un escenario y a un panel, serializando las curvas I-V en formato JSON.

El funcionamiento de esta función es muy similar al de las anteriores: se reciben todos los datos necesarios por parámetros, se crea un objeto Simulacion con los datos, se añade a la base de datos y se devuelve el objeto Simulacion.

Pero esta función tiene la particularidad de que los valores I y V se reciben como una matriz. Esto se debe a cómo trabaja la librería que hace de interfaz entre nuestro código y el simulador solar. Esta librería nos devuelve los resultados medidos en una matriz. Por tanto, ahora debemos separar esa matriz en dos arrays de Is y Vs.

Como es un poco complejo de entender sin el contexto del código de la función al completo, se va a incluir la función completa:

```python
async def create_IV(
    db: AsyncSession,
    user_id: UUID,
    IV_mat,
    panel_id: int,
    escenario_id: int,
    nombre: str,
) -> Simulacion:
    Vs = IV_mat[:, 0].tolist()#separamos en arrays
    Is = IV_mat[:, 1].tolist()
    usuario_str = str(user_id)

    V_json = json.dumps(Vs)#convertimos a JSON
    I_json = json.dumps(Is)
    nuevo = Simulacion(
        nombre=nombre,
        V=V_json,
        I=I_json,
        escenario=escenario_id,
        panel=panel_id,
        usuario=usuario_str,
    )
    db.add(nuevo)
    await db.commit()
    await db.refresh(nuevo)
    return nuevo
```


Create_IV_spectra(...)

Versión alternativa para guardar simulaciones a partir de espectros de referencia.

La única diferencia entre esta función y la anterior es que en vez de guardar el escenario_id, se guarda el nombre espectro utilizado (String).

Create_sdm(…)

Inserta una nueva fila en la tabla SDM con los parámetros calculados.

Requiere que se le pasen por parámetros todos los atributos que hay que rellenar en la base de datos, que son todos los valores eléctricos, métricas de error y puntos característicos, además del id de la simulación asociada y el id del algoritmo usado para calcular esos datos.

El proceso es muy similar a los anteriores: se crea el objeto SDM con los datos, se añade a base de datos y se devuelve el objeto SDM.

Set_simulando_state(db, state)

Permite activar o desactivar un flag en la base de datos (Tabla simulando) para controlar si se está realizando una simulación. Recibe como parámetros la instancia de la base de datos, y un parámetro booleano (state).

Primero hacemos una consulta a la base de datos para obtener la tupla que tiene el valor booleano (solo debe haber una tupla en esta tabla):

```
result = await db.execute(select(Simulando).where(Simulando.id == 1))
sim_obj = result.scalar_one_or_none()
```

Hay que tener en cuenta el caso de que la base de datos esté vacía y no haya ninguna tupla en la tabla “simulando”. Esto se ha tenido en cuenta, si sim_obj es None, significa que no hemos obtenido nada de la consulta, indicando que no había ninguna tupla, por tanto, tendríamos que crearlo:

```
if sim_obj is None:
        sim_obj = Simulando(id=1, sim=state)
        db.add(sim_obj)
else:
        sim_obj.sim = state
```

Ahora hacemos un commit a la base de datos y devolvemos el objeto Simulando:

```
await db.commit()
await db.refresh(sim_obj)
return sim_obj
```

Get_simulando_state(db)

Consulta el estado actual de este flag. Si no existe aún, se asume que el valor es False.

Simplemente se realiza la consulta a la base de datos:

```
result = await db.execute(select(Simulando).where(Simulando.id == 1))
```

Y se devuelve el valor obtenido o False en caso de haber obtenido None.

Es importante recordar que todas estas funciones se han definido como asíncronas, ya que actúan sobre la base de datos. Esto se consigue especificando “async” en la declaración de la función, por ejemplo:

```
async def set_simulando_state
```

Todas estas funciones serán invocadas desde los distintos endpoints, permitiendo separar la lógica de uso de la de acceso a la base de datos, y resultando en un código más limpio, legible y mejor estructurado. Aun así, en los endpoints siguen habiendo algunas consultas a la base de datos.

5.1.3 Librería pvsim

La librería “pvsim” usada en este proyecto ha sido proporcionada por el tutor de este proyecto, Xavier Moreno-Vassart Martinez. Contiene todas las funciones

necesarias para comunicarse con el simulador solar y el SMU, así como las funciones que permiten calcular los parámetros SDM [6]. En este apartado vamos a explicar brevemente las funciones que se han usado en el código de la aplicación web para así poder entender mejor el código de los endpoints que veremos después.

Model_irradiance_SPECTRL2(…)

Esta función obtiene la irradiancia en una ubicación, tiempo y condiciones atmosféricas específicas. Necesitamos proporcionarle los siguientes parámetros:

- Datetime(datetime)
- time_zone(String)
- latitude(float)
- longitude(float)
- surface_tilt(float)
- surface_azimuth(float)
- ground_albedo(float)
- water_vapor_content(float)
- ozone(float)
- aerosol_turbidity_500nm(float)
- pressure(float)
- altitude(float)
- temperature(float)
- spa_method(float)
- airmass_model(float)

Si nos fijamos en los parámetros, veremos que todos los obtenemos en el formulario para escenarios personalizados.

Esta función devuelve un array con los valores de irradiancia. Esta irradiancia la usaremos como parámetro para otras funciones de esta librería.

Adapt_irradiance(…)

Usaremos esta función para obtener los ficheros JSON de configuración, que replican la irradiancia que le pasaremos por parámetros. Estos ficheros servirán como parámetro para la función que lanza la simulación.

Simulate(…)

Esta función pondrá en marcha el simulador solar, necesita que pasemos como parámetros los ficheros de configuración obtenidos de la función "adapt_irradiance(…)", el Sample Count y el Voc. Nos devolverá una matriz con los valores medidos por el SMU en la simulación.

Post_process(…)

Esta función procesa la curva IV para obtener los parámetros SDM y puntos remarkables. Necesitamos pasarle como parámetro la matriz de valores obtenida por la función "simulate(…)". Nos devolverá un objeto IV_Curve. Este es un objeto un tanto complejo y necesario de entender para poder comprender cómo obtenemos los parámetros SDM.

Objeto IV_Curve

La implementación de este objeto es la siguiente:

```
@dataclass(config=_Config)
class IV_Curve:
    time:     datetime
    vi_mat:   npt.NDArray[np.floating]
    sdm_info: list[SDM_Data]
```

La parte que nos interesa es sdm_info, como vemos es una lista de objetos SDM_Data.

Cada objeto SDM_Data contiene: el método utilizado, el resto de parámetros, errores y remarkables. La implementación es la siguiente:

```
class SDM_Data:
    method:      str
    params:      SDM_Params
    errors:      Errors
    remarkables: Remarkables
    extra_info:  str|None = None
```

Nótese que params, errors y remarkables son objetos a su vez. Estos objetos ya contienen los valores en tipos simples que podemos usar.

Se adjunta la implementación de estos por si queda alguna duda:

```
@dataclass
class SDM_Params:
    Iph:  float
    Isat: float
    a:    float
    Rsh:  float
    Rs:   float

@dataclass
class Errors:
    rmse:        float
    mae:         float
    area_factor: float
```

```
@dataclass
class Remarkables:
    Voc:  float
    Isc:  float
    Vmpp: float
    Impp: float
```

Select_reference_spectrum(…)

Devuelve los ficheros JSON de configuración del simulador solar utilizando la irradiancia de un espectro de referencia. Solo necesita que le pasemos por parámetros el nombre del espectro de referencia.

5.1.4 User Manager para gestión de usuarios

La gestión de usuarios en este proyecto se ha implementado mediante la librería FastAPI Users, que proporciona un sistema para autenticación, registro, recuperación de contraseñas, verificación por email, y control de permisos.

Para adaptarla al proyecto, se ha creado un archivo user_manager.py donde se configura todo el sistema de autenticación basado en tokens JWT y cookies.

La clase UserManager hereda de BaseUserManager e implementa la lógica de gestión de usuarios. Además, se combina con UUIDIDMixin ya que los id de usuario son de tipo UUID.

En la clase se han definido tres funciones opcionales que permiten ejecutar acciones específicas cuando ocurren eventos importantes relacionados con la gestión de usuarios:

```
async def on_after_register(self, user: User, request: Optional[Request] =
None):
        print(f"User {user.id} has registered.")

    async def on_after_forgot_password(
        self, user: User, token: str, request: Optional[Request] = None
    ):
         print(f"User {user.id} has forgot their password. Reset token:
{token}")

    async def on_after_request_verify(
        self, user: User, token: str, request: Optional[Request] = None
    ):
         print(f"Verification requested for user {user.id}. Verification
token: {token}")
```

Como podemos ver, solo imprimen mensajes por consola, pero se podrían modificar estas funciones para enviar, por ejemplo, un correo electrónico después del registro.

Se define una función asíncrona que instancia UserManager, y que obtiene previamente el acceso a la base de datos de usuarios mediante Depends(get_user_db):

```
async def get_user_manager(user_db=Depends(get_user_db)):
    yield UserManager(user_db)
```

Esto permite integrar el gestor de usuarios con los endpoints de autenticación y con FastAPI de forma automática.

Ahora veremos la estrategia de autenticación utilizada y su implementación. Para autenticar usuarios se ha usado una estrategia basada en tokens JWT (JSON Web Tokens), que se almacenan en cookies:

```
def get_jwt_strategy() -> JWTStrategy:
    return JWTStrategy(secret=SECRET, lifetime_seconds=3600)

cookie_transport=CookieTransport(cookie_max_age=3600,
cookie_secure=False)

auth_backend = AuthenticationBackend(
    name="jwt",
    transport=cookie_transport,
    get_strategy=get_jwt_strategy,
)
```

Se puede ver que la duración de la cookie es de 3600 segundos, lo que equivale a una hora, es decir, después de una hora de uso es necesario volver a autenticarse. Este tiempo se puede incrementar si se desea aumentar el tiempo entre autenticaciones. Cookie_secure se ha establecido en False para facilitar el desarrollo sin HTTPS.

Con todo lo anterior, se crea una instancia de FastAPIUsers que permite obtener rutas predefinidas para el manejo de los usuarios y protección de los endpoints. La instancia es la siguiente:

```
fastapi_users=FastAPIUsers[User,uuid.UUID](get_user_manager,
[auth_backend])
```

Esta instancia nos permitirá definir dependencias que luego usaremos en los endpoints para obtener al usuario autenticado. Se han definido las siguientes:

```
current_active_superuser=fastapi_users.current_user(active=True,
superuser=True)
current_active_user = fastapi_users.current_user(active=True)
```

“current_active_user” obtiene al usuario autenticado y activo, mientras que “current_active_superuser” solo lo obtendrá si tiene permisos de superusuario.

Estas dependencias se usan en los endpoints para proteger rutas que solo deben ser accesibles por usuarios registrados o administradores.

5.1.5 Implementación de endpoints

En este apartado se detallará la implementación de todos los endpoints que permiten a los usuarios interactuar con el sistema.

La aplicación se inicializa con un lifespan que se encargará de crear las tablas necesarias en la base de datos al arrancar:

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    await create_db_and_tables()
    yield

app = FastAPI(lifespan=lifespan)
```

Podemos ver que inicializamos la variable “app” con el lifespan definido justo arriba que espera a la función create_db_and_tables() implementada en el fichero de implementación de la base de datos.

Ahora, antes de comenzar con la implementación de los endpoints, se ha creado un manejador de excepciones personalizado que modifica el comportamiento de la excepción 401 Unauthorized, redirigiendo automáticamente al endpoint de login:

```
@app.exception_handler(HTTPException)
async def custom_http_exception_handler(request: Request, exc:
HTTPException):
    if exc.status_code == status.HTTP_401_UNAUTHORIZED:
        return RedirectResponse(url="/login")
    return await http_exception_handler(request, exc)
```

Esto es muy útil, porque como todos los endpoints dependen de un usuario, si no hay ningún usuario autenticado, no se rompe el uso del programa y se redirige al usuario al login sin complicaciones.

Ahora debemos inicializar una variable para los templates de JinJa2 que permitirá al frontend utilizar variables del backend para ser construido, como veremos en el apartado de implementación del frontend. Esto se hace de la siguiente forma:

```
templates = Jinja2Templates(directory="templates")
```

Como vemos, guardamos en la variable templates el objeto Jinja2Templates con el directorio “templates”. En este directorio es donde deben estar todos los ficheros HTML de la aplicación.

Endpoint “/”

Comenzamos con el endpoint principal. Su implementación es la siguiente:

```
@app.get("/", response_class=HTMLResponse)
async def index(
    request: Request,
    user=Depends(current_active_user),
    error: str | None = None
):
    spa_methods    = _spa_methods_list
    airmass_models = _airmass_models_list
    return templates.TemplateResponse(
        "index.html",
        {
            "request": request,
            "user": user,
            "spa_methods": spa_methods,
            "airmass_models": airmass_models,
            "error": error
        }
    )
```

Utilizamos endpoints asíncronos: “async def index(…)” y hacemos que sea necesario estar autenticado con “user=Depends(current_active_user)”. Esto nos lanzará la excepción 401 Unauthorized mencionada anteriormente si el usuario no está autenticado. A continuación, cargamos las listas de spa_methods y airmass_models para mostrarlas en el formulario y poder seleccionar uno. Las listas las obtenemos de la librería pvsim con _spa_methods_list y _airmass_models_list, que son variables que importamos directamente de pvsim con:

```
from pvsim.simulation import _spa_methods_list, _airmass_models_list
```

Por último, definimos que la página de respuesta tiene que ser index.html y enviamos las variables request, user, spa_methods, airmass_models y error, que podrán ser usadas en el frontend.

Endpoint “/dashboard”

Este endpoint es bastante sencillo ya que simplemente carga la plantilla dashboard.html enviando el objeto User:

```python
@app.get("/dashboard", response_class=HTMLResponse)
async def dashboard(
    request: Request,
    user: User = Depends(current_active_user)

):
    return templates.TemplateResponse("dashboard.html", {
        "request": request,
        "user": user
    })
```

Endpoint "/admin"

En este endpoint hay algunos matices que es importante destacar, ya que hay que obtener toda la lista de usuarios de la base de datos y enviarla al frontend. Es el primer endpoint de los que hemos visto hasta ahora que realiza una conexión con la base de datos. Su implementación es la siguiente:

```python
@app.get("/admin", response_class=HTMLResponse)
async def admin_dashboard(
    request: Request,
    superuser: User = Depends(current_active_superuser),
    session: AsyncSession = Depends(get_async_session),
):
    users = await get_all_users(session)

    return templates.TemplateResponse("admin.html", {
        "request": request,
        "user": superuser,
        "users": users,
    }
```

Los cambios respecto al resto de endpoints comienzan al principio, precisamente en la línea "superuser: User = Depends(current_active_superuser)". Para poder acceder a este endpoint debemos ser superusuarios y para ello utilizamos la dependencia current_active_superuser definida en "user_manager.py".

Luego establecemos una sesión asíncrona con la base de datos con "session: AsyncSession = Depends(get_async_session)" que nos permitirá usar la función get_all_users(…) para obtener todos los usuarios de la base de datos.

Obtenemos los usuarios de la base de datos con "users = await get_all_users(session)", y, por último, devolvemos la plantilla admin.html con el usuario autenticado y la lista de todos los usuarios.

Endpoints "/login" y "/registro"

Se han agrupado estos endpoints en un mismo apartado ya que son los únicos que no requieren estar autenticados para acceder a ellos y, además, solo

devuelven las plantillas login.html y registro.html respectivamente. El código es el siguiente:

```
@app.get("/login", response_class=HTMLResponse)
def login(request: Request):
    return templates.TemplateResponse("login.html", {
        "request": request})

@app.get("/registro", response_class=HTMLResponse)
def signup(request: Request):
    return templates.TemplateResponse("registro.html", {
        "request": request})
```

Endpoint "/panels"

Este endpoint, a diferencia del resto, devuelve una lista de paneles obtenida de la base de datos haciendo uso de la función CRUD get_user_panels(db, id), en vez de devolver una plantilla HTML.

Esta es la implementación:

```
@app.get("/panels", response_model=List[PanelRead])
async def list_panels(
    user: User = Depends(current_active_user),
    db: AsyncSession = Depends(get_async_session),

):
    panels = await get_user_panels(db, user.id)
    return panels
```

La clase PanelRead se ha definido en el archivo "schemas.py" y esta es su implementación:

```
class PanelRead(BaseModel):
    id: int
    nombre: str
    descripcion: str

    model_config = ConfigDict(from_attributes=True)
```

Teniendo en cuenta esto, vemos que el endpoint "/panels" devuelve una lista de objetos PanelRead cada uno de los cuales contiene los atributos id, nombre y descripción (de la misma forma que se guardan en la base de datos). Este endpoint se llamará desde el frontend para obtener la lista de paneles y poder mostrarla en su respectivo desplegable.

Endpoint "/agregarPanel" por Get

Este endpoint simplemente renderiza la plantilla add_panel.html enviando al frontend el usuario, como hacen la mayoría de endpoints:

```
@app.get("/agregarPanel", response_class=HTMLResponse)
async def form_agregar_panel(request: Request, user: User =
Depends(current_active_user)):
    return templates.TemplateResponse(
        "add_panel.html",
        {"request": request, "user": user}
    )
```

Es un endpoint bastante sencillo que se entiende simplemente con echarle un vistazo.

Endpoint "/agregarPanel" por Post

Este endpoint se llama igual que el anterior, pero se distinguen por la petición HTTP dependiendo de si es con método Get o Post. Su funcionamiento interno no tiene nada que ver, ya que este endpoint maneja la lógica necesaria para recoger los datos del formulario que rellena el usuario cuando va a crear un panel, crear el panel y redirigirnos a la página principal.

Esto se ha conseguido de la siguiente forma:

```
@app.post("/agregarPanel")
async def procesar_agregar_panel(
    nombre: str         = Form(...),
    descripcion: str    = Form(...),
    public: bool        = Form(False),
    user: User = Depends(current_active_user),
    db: AsyncSession    = Depends(get_async_session),
):
    await create_panel(db, user.id, nombre, descripcion, public)
    return RedirectResponse(url="/", status_code=303)
```

Con Form(…) vamos recogiendo los valores del formulario y guardándolo en variables que luego pasaremos como argumentos a create_panel(…), que se definió en el archivo CRUD y creará el objeto panel y lo guardará en la base de datos.

Por último, redirigimos a la página principal con el código 303, que indica redirección.

Endpoint "/cambiarPassword"

Endpoint que devuelve la plantilla cambiarPassword.html con el usuario autenticado:

```
@app.get("/cambiarPassword", response_class=HTMLResponse)
async def cambiarPassword(
```

```
    request: Request,
    user: User = Depends(current_active_user)

):
    return templates.TemplateResponse("cambiarPassword.html", {
        "request": request,
        "user": user
    })
```

Endpoint "/historial"

Este endpoint realizará la consulta a la base de datos para obtener todas las simulaciones y devolverá la plantilla historial.html enviando la lista de simulaciones y el usuario al frontend.

Se adjunta su implementación y luego se explicará por partes:

```
@app.get("/historial", response_class=HTMLResponse)
async def historial(
    request: Request,
    user: User = Depends(current_active_user),
    db: AsyncSession = Depends(get_async_session),
):
    user_id = str(user.id)
    q = (
        select(Simulacion)
        .where(Simulacion.usuario == user_id)
        .order_by(Simulacion.id.desc())
    )
    result = await db.execute(q)
    sims = result.scalars().all()
    return templates.TemplateResponse(
        "historial.html",
        {
            "request": request,
            "simulaciones": sims,
            "user": user,
        }
    )
```

Podemos ver que, en la definición de argumentos de la función, obtenemos el usuario y obtenemos la sesión con la base de datos.

A continuación, convertimos el id de usuario a un string para poder usarlo en la consulta SQL con "user_id = str(user.id)"

Procedemos guardando la consulta en la variable "q". Esta consulta obtiene de la tabla Simulación todas las simulaciones pertenecientes al usuario y lo ordena

por id de forma descendiente, así aparecerán al principio las simulaciones más recientes, pues tienen un id mayor.

Ejecutamos la consulta y guardamos el resultado en la variable result con “result = await db.execute(q)”. El problema es que en result obtenemos una instancia de Result, un objeto intermedio que contiene los resultados en bruto de la consulta. Para obtener una lista de objetos Simulacion necesitamos utilizar scalars(). Pero haciendo esto obtenemos un iterador de objetos Simulacion; para convertirlo en una lista, usamos all(). Esto mismo se repetirá en consultas futuras en otros endpoints, por tanto, se obviará su explicación.

Por último, se devuelve la plantilla historial.html junto con el usuario y la lista de simulaciones guardada en la variable sims.

Endpoint “/simulando/status”

Para obtener el valor del flag de la base de datos que nos indica si hay alguien simulando, utilizaremos la función CRUD correspondiente. Esta es la implementación completa del endpoint:

```
@app.get("/simulando/status")
async def simulando_status(db: AsyncSession = Depends(get_async_session)):
    estado = await get_simulando_state(db)
    return {"sim": estado}
```

Para obtener el estado del flag, simplemente hacemos una llamada a get_simulando_state(db) y ya obtenemos un valor booleano que nos indica si se está produciendo una simulación.

Endpoint “/reference_spectra”

Para cargar la lista de espectros de referencia y mostrarlos en el desplegable es necesario instanciar un objeto Simulador utilizando el archivo de configuración “config.json” y, a continuación, devolver la plantilla “reference_spectra.html”.

Esta es la implementación de este endpoint:

```
@app.get("/reference_spectra", response_class=HTMLResponse)
async def reference_spectra(request: Request, error: str | None = None,
user: User = Depends(current_active_user)):
    sim = Simulator("pvsim/config.json")
    spectra = sim.reference_spectra_list
    return templates.TemplateResponse(
        "reference_spectra.html",
        {
            "request": request,
            "spectra_list": spectra,
            "user": user,
            "error": error
        }
```

```
    )
```

Como se puede ver en la implementación, utilizamos el atributo reference_spectra_list del objeto Simulator, referenciado con la variable “sim”, para obtener la lista de espectros de referencia y poder mostrarla en el desplegable.

Endpoint “/simulate/reference_spectra”

Como este endpoint es bastante extenso, se va a desglosar en diferentes partes que se irán explicando una a una.

Comenzamos con la definición de la función simulate_reference_spectra(…). Aquí recogemos los datos del formulario del frontend y se almacenan los valores en variables:

```
@app.post("/simulate/reference_spectra", response_class=RedirectResponse)
async def simulate_reference_spectra(
    request:       Request,
    nombre:        str = Form(...),
    spectrum_name: str = Form(...),
    panel_id:      int    = Form(...),
    voc:           float  = Form(...),
    sc:            int    = Form(...),
    user:          User   = Depends(current_active_user),
    db:            AsyncSession = Depends(get_async_session),
):
```

Podemos ver que obtenemos los valores del formulario con Form(…), luego obtenemos la dependencia del usuario activo y establecemos una sesión con la base de datos.

Después de este proceso, debemos comprobar si hay alguien simulando. Si lo hay, mostraremos un mensaje de error y haremos un return para detener el proceso de simulación:

```
if await get_simulando_state(db):
        error_msg = quote("Ya hay una simulación en curso. Por favor,
inténtalo de nuevo más tarde.")
        return RedirectResponse(
        url=f"/reference_spectra?error={error_msg}",
        status_code=302
    )
```

Para hacer lo mencionado en el párrafo anterior, utilizamos la función CRUD get_simulando_state(db) en un if() y si se entra en el if (lo que significa que hay

alguien simulando), se lanza el mensaje y se redirige al endpoint que carga la página de espectros de referencia.

A continuación, si no había nadie simulando, se establecerá el flag de simulación en True, se instanciará un objeto simulador con el fichero de configuración "config.json", se obtendrán los ficheros de configuración con "select_reference_spectrum (spectrum_name)", se lanzará la simulación y se procesarán los resultados para obtener un objeto IV_curve. Todo este proceso se puede ver en este fragmento de código

```
await set_simulando_state(db, True)
    try:
        sim = Simulator("pvsim/config.json")
        spec_files = sim.select_reference_spectrum(spectrum_name)
        vi_mat = sim.simulate(spec_files, sample_count=sc, Voc=voc)
        iv_curve = sim.post_process(vi_mat)
```

"set_simulando_state(db, True)" establece el flag de simulación en True, luego se encapsula el proceso de simulación en un "try". Instanciamos el objeto simulador con "sim = Simulator("pvsim/config.json")", obtenemos los ficheros de configuración del simulador con "spec_files = sim.select_reference_spectrum(spectrum_name)". Nótese que spectrum name, sample count y voc se han obtenido del formulario del frontend. Lanzamos la simulación con "vi_mat = sim.simulate(spec_files, sample_count=sc, Voc=voc)" y procesamos la matriz obtenida de la simulación con "iv_curve = sim.post_process(vi_mat)"

Si fallase algo, al estar en un "try", se ejecutaría el siguiente fragmento de código:

```
except Exception as e:
        raise HTTPException(status_code=400, detail=f"Error simulando:
{e}")
```

Que mostrará el mensaje "Error simulando: + la excepción". Por último, falle la simulación o no, se ha incluido un "finally" que se ejecutará siempre, para volver a dejar el flag de simulación en False.

Este es el código:

```
finally:
        await set_simulando_state(db, False)
```

El siguiente paso será guardar la simulación en la base de datos, para ello se hace uso de la función CRUD "create_IV_spectra(…)":

```
IV_bdd = await create_IV_spectra(
        db,
```

```
        user.id,
        IV_mat = vi_mat,
        panel_id=panel_id,
        reference_spectra = spectrum_name,
        nombre = nombre,
    )
```

Por último, queda rellenar dos tuplas en la tabla SDM con los valores obtenidos de aplicar los dos algoritmos SDM a los resultados. Para ello se hará una consulta a la base de datos que obtendrá los algoritmos disponibles en la tabla Algoritmo:

```
result = await db.execute(select(Algoritmo))
algos  = result.scalars().all()
```

En la variable “algos” tenemos una lista de objetos Algoritmo. Ahora creamos un diccionario con “columnas = { alg.nombre: alg.id for alg in algos }” que servirá para acceder rápidamente al id de un algoritmo a partir de su nombre, que obtendremos desde iv_curve.sdm_info:

```
columnas = { alg.nombre: alg.id for alg in algos }
```

Debemos recorrer los resultados SDM a partir de la iv_curve que hemos calculado. Obtenemos el id del algoritmo y guardamos los parámetros SDM con la función CRUD create_sdm(…). Todo esto se hace de esta forma:

```
for sdm_data in iv_curve.sdm_info:
        if sdm_data.method not in columnas:
            continue
        algoritmo_id = columnas[sdm_data.method]
        await create_sdm(
            db,
            simulacion_id   = IV_bdd.id,
            algoritmo_id    = algoritmo_id,
            iph             = sdm_data.params.Iph,
            isat            = sdm_data.params.Isat,
            a               = sdm_data.params.a,
            rsh             = sdm_data.params.Rsh,
            rs              = sdm_data.params.Rs,
            rmse            = sdm_data.errors.rmse,
            mae             = sdm_data.errors.mae,
            area_factor     = sdm_data.errors.area_factor,
            voc             = sdm_data.remarkables.Voc,
            isc             = sdm_data.remarkables.Isc,
            vmpp            = sdm_data.remarkables.Vmpp,
            impp            = sdm_data.remarkables.Impp,
            extra_info      = getattr(sdm_data, "extra_info", None),
```

```
    )
```

Y para finalizar este proceso, hacemos un redirect a “/resultado” añadiendo en la URL el id de la simulación y el nombre del método que queremos que se muestre por defecto en los resultados:

```
return RedirectResponse(
        url=f"/resultado?sim_id={IV_bdd.id}&method_index={'tslls-
refined'}",
        status_code=302
    )
```

Endpoint “/simulate/full”

Como este endpoint es muy similar al anterior (/simulate/reference_spectra), se omitirá la explicación de los fragmentos que ya han sido detallados anteriormente, y se resaltarán solo las partes que cambian.

Comenzamos con la definición de la función “simulate_full(…)”, que recibe un objeto IrradianceRequest, definido en schemas.py:

```
class IrradianceRequest(BaseModel):
    panel_id: int
    datetime: datetime
    time_zone: str
    latitude: float
    longitude: float
    surface_tilt: float
    surface_azimuth: float
    ground_albedo: float
    water_vapor_content: float
    ozone: float
    aerosol_turbidity_500nm: float
    pressure: float
    altitude: float | None = None
    temperature: float
    voc: float
    sc: int
    nombre: str
    spa_method: str
    airmass_model: str
```

El objeto tiene todos los atributos necesarios con los datos necesarios para lanzar una simulación personalizada.

Además del objeto IrradianceRequest con todos los datos del formulario, recibe también al usuario autenticado y la sesión con la base de datos:

```
async def simulate_full(request: IrradianceRequest, user: User =
Depends(current_active_user), db: AsyncSession =
Depends(get_async_session)):
```

A continuación, igual que en el caso anterior, se comprueba si ya hay una simulación en curso. Si es así, se redirige con un mensaje de error y se interrumpe el proceso:

```
if await get_simulando_state(db):
        error_msg = quote("Ya hay una simulación en curso. Por favor,
inténtalo de nuevo más tarde.")
        return RedirectResponse(
        url=f"/?error={error_msg}",
        status_code=302)
```

Si no hay ninguna simulación activa, se establece el flag de simulación en True y se encapsula todo el proceso en un "try". Aquí es donde empieza la parte diferente respecto al endpoint anterior.

En lugar de seleccionar un espectro de referencia, se calcula la irradiancia directamente a partir de los datos introducidos en el formulario utilizando la función model_irradiance_SPECTRL2(...). Esta función se explicó en el apartado de la librería pvsim.

Esto lo hacemos con este fragmento de código:

```
await set_simulando_state(db, True)
    try:
        sim = Simulator("pvsim/config.json", )
        irradiance = model_irradiance_SPECTRL2(
            datetime=request.datetime,
            time_zone=request.time_zone,
            latitude=request.latitude,
            longitude=request.longitude,
            surface_tilt=request.surface_tilt,
            surface_azimuth=request.surface_azimuth,
            ground_albedo=request.ground_albedo,
            water_vapor_content=request.water_vapor_content,
            ozone=request.ozone,
            aerosol_turbidity_500nm=request.aerosol_turbidity_500nm,
            pressure=request.pressure,
            altitude=request.altitude,
            temperature=request.temperature,
            spa_method=request.spa_method,
            airmass_model=request.airmass_model,
        )
```

Una vez calculada la irradiancia, esta se adapta con sim.adapt_irradiance(...) para generar los archivos de configuración que se enviarán al simulador. Luego se lanza la simulación y se procesa la curva I-V resultante:

```
sunbrick_files, sim_irradiances = sim.adapt_irradiance(irradiance)
vi_mat = sim.simulate(sunbrick_files, sample_count=request.sc,
Voc=request.voc)
iv_curve = sim.post_process(vi_mat)
```

En caso de error durante este proceso, se lanza una excepción con el mensaje correspondiente y, como ya se explicó anteriormente, en el bloque “finally” se vuelve a dejar el flag de simulación en False:

```
except Exception as e:
        raise HTTPException(status_code=400, detail=f"Error calculando
irradiancia: {e}")
    finally:
        await set_simulando_state(db, False)
```

Después de simular, hay que guardar en la base de datos toda la información. Primero, se guarda el escenario personalizado utilizando la función CRUD create_scenario(...):

```
escenario = await create_scenario(
        db,
        user.id,
        datetime_val        = request.datetime,
        time_zone           = request.time_zone,
        latitude            = request.latitude,
        longitude           = request.longitude,
        surface_tilt        = request.surface_tilt,
        surface_azimuth     = request.surface_azimuth,
        ground_albedo       = request.ground_albedo,
        water_vapor_content = request.water_vapor_content,
        ozone               = request.ozone,
        aerosol_turbidity_500nm = request.aerosol_turbidity_500nm,
        pressure            = request.pressure,
        altitude            = request.altitude,
        temperature         = request.temperature,
        spa_method          = request.spa_method,
        airmass_model       = request.airmass_model,
    )
```

A continuación, se guarda la simulación utilizando create_IV(...), asociándola al escenario que acabamos de crear:

```
IV_bdd = await create_IV(
        db,
        user.id,
        IV_mat = vi_mat,
        panel_id=request.panel_id,
        escenario_id = escenario.id,
        nombre = request.nombre,
    )
```

Los siguientes pasos son idénticos al endpoint anterior: se consulta la tabla Algoritmo, se crea un diccionario para acceder rápidamente al id de cada método, se recorren los datos SDM generados, y se almacenan en la tabla correspondiente utilizando la función create_sdm(...). No se volverá a revisar detalladamente esta parte, ya que está explicada en el apartado anterior.

Por último, una vez que se ha completado todo el proceso de simulación y se han guardado los resultados, se redirige al endpoint /resultado indicando en la URL el id de la simulación y el método SDM que se mostrará por defecto:

```
return RedirectResponse(
        url=f"/resultado?sim_id={IV_bdd.id}&method_index={columnas['tslls
-refined']}",
        status_code=302
    )
```

Endpoint "/resultado"

Este endpoint cargará los datos de una simulación almacenada en la base de datos y devolverá la plantilla "resultado.html".

Se va a detallar por partes el proceso seguido para conseguir esto.

La definición de la función resultado(...) es la siguiente:

```
@app.get("/resultado", response_class=HTMLResponse)
async def resultado(
    request:      Request,
    sim_id:       int            = Query(..., description="ID de la
simulación"),
    metodo_id:    int            = Query(1, description="ID de algoritmo
SDM"),
    user:         User           = Depends(current_active_user),
    db:           AsyncSession   = Depends(get_async_session),
):
```

Como podemos ver, sim_id y método_id, se reciben directamente de la URL al usar Query(…).

A continuación, se realiza una consulta a la base de datos para obtener el objeto Simulacion asociado al ID recibido. Si no se encuentra, se lanza una excepción con código 404:

```
sim_obj = (await db.execute(
        select(Simulacion).where(Simulacion.id == sim_id)
    )).scalars().first()
    if not sim_obj:
        raise HTTPException(404, "Simulación no encontrada")
```

Después, si la simulación tiene asociado un escenario (es decir, no fue realizada con espectro de referencia), se hace otra consulta para obtener el objeto Escenarios:

```
esc_obj = None
if sim_obj.escenario is not None:
     esc_obj = (await db.execute(
        select(Escenarios).where(Escenarios.id == sim_obj.escenario)
     )).scalars().first()
```

Además, se recuperará el panel utilizado en la simulación con:

```
pan_obj = (await db.execute(
        select(Paneles).where(Paneles.id == sim_obj.panel)
    )).scalars().first()
```

Después, se realiza una consulta para obtener los parámetros SDM asociados a la simulación y al método SDM solicitado. En caso de no obtener nada en esta consulta, se lanzará una excepción 404:

```
if not sim_obj:
        raise HTTPException(404, "Simulación no encontrada")
    sdm_obj = (await db.execute(
        select(SDM).where(
            SDM.simulacion == sim_id,
            SDM.algoritmo  == metodo_id
        )
    )).scalars().first()
    if not sdm_obj:
        raise HTTPException(404, "Datos SDM no encontrados para este
método")
```

Ahora tenemos que generar una curva I-V teórica a partir de los parámetros obtenidos utilizando la función “from_diode” de la librería pvtools.singlediode.

Para ello, primero se genera un vector V2 de 250 puntos lineales desde 0 hasta Voc, y luego se calcula el vector de intensidades I2:

```
V2 = np.linspace(0, sdm_obj.voc, 250)
I2 = sd_intensity.from_diode(
       Iph=sdm_obj.iph,
       Isat=sdm_obj.isat,
       a=sdm_obj.a,
       Rsh=sdm_obj.rsh,
       Rs=sdm_obj.rs,
       V=V2
    )
```

Esta curva se utilizará para representar gráficamente el comportamiento ideal del panel.

Por último, se devuelve la plantilla “resultado.html”, pasando todos los datos necesarios:

```
return templates.TemplateResponse(
        "resultado.html",
        {
            "request": request,
            "user":    user,
            "simulacion":     sim_obj,
            "sdm":            sdm_obj,
            "esc":            esc_obj,
            "pan":            pan_obj,
            "metodo_id": metodo_id,
            "V2" : V2.tolist(),
            "I2" : I2.tolist(),
        }
    )
```

Endpoint “/simulacion/{sim_id}/borrar”

Este endpoint recibe en la URL el id de la simulación que se desea borrar de la base de datos.

Esta es su implementación:

```
@app.post("/simulacion/{sim_id}/borrar")
async def borrar_simulacion(
    sim_id: int,
    db:     AsyncSession = Depends(get_async_session),
    user:   User = Depends(current_active_user),
```

```
):
    user_id = str(user.id)
    await db.execute(
        delete(Simulacion)
        .where(Simulacion.id == sim_id,
               Simulacion.usuario == user_id )
    )
    await db.commit()
    return RedirectResponse(url="/historial",
status_code=status.HTTP_303_SEE_OTHER)
```

La función borrar_simulacion() recibe como parámetro el sim_id, establece sesión con la base de datos y requiere que haya un usuario autenticado. Luego convertimos el id del usuario en string para usarlo en la consulta.

Vemos que la consulta es muy sencilla, un delete de la tupla que coincida con el id de la URL y que pertenezca al usuario para mayor seguridad. Esto se hace por si alguien cambiase el id de la URL, ya que, si no estuviera la comprobación del usuario, se podría modificar la URL con otro id y borrar una simulación no perteneciente a ese usuario.

Por último, redirigimos a "/historial" con código 303 para mostrar la página de historial de simulaciones.

Endpoint "/simulacion/{sim_id}/download_csv/"

Al igual que el endpoint anterior, recibe el id de la simulación que se va a querer descargar en CSV en la URL. Como es un endpoint extenso en código, lo analizaremos por partes.

Comenzamos con la definición de función "download_simulation_csv(…)"

```
@app.get("/simulacion/{sim_id}/download_csv/")
async def download_simulation_csv(
    sim_id:       int,
    user:         User         = Depends(current_active_user),
    db:           AsyncSession = Depends(get_async_session),
):
```

Esta parte es idéntica al endpoint anterior, recibimos el parámetro sim_id, el usuario autenticado y establecemos la sesión con la base de datos.

Ahora debemos hacer una consulta para encontrar la simulación que queremos descargar en CSV. Al igual que en el endpoint anterior, se ha comprobado que la simulación que se pretende descargar pertenezca al usuario que la intenta descargar:

```
sim = (await db.execute(
        select(Simulacion).where(
```

```
      Simulacion.id == sim_id,
            Simulacion.usuario == str(user.id)
        )
    )).scalars().first()
```

Si no se encuentra la simulación, se lanzará una excepción 404 con el mensaje “Simulación no encontrada”:

```
if not sim:
        raise HTTPException(404, "Simulación no encontrada")
```

Si existe la simulación, se hace una consulta para recuperar todos los resultados almacenados en la tabla SDM asociados a esa simulación, junto con el nombre del algoritmo utilizado en cada caso:

```
rows = await db.execute(
        select(SDM, Algoritmo.nombre)
        .join(Algoritmo, SDM.algoritmo == Algoritmo.id)
        .where(SDM.simulacion == sim_id)
    )
```

Con los datos obtenidos, se recorre cada fila y se construye una lista de diccionarios con los parámetros SDM que se desean incluir en el CSV:

```
for sdm_obj, algo_name in rows.all():
        records.append({
            "algoritmo":       algo_name,
            "Iph":          sdm_obj.iph,
            "Isat":         sdm_obj.isat,
            "a":            sdm_obj.a,
            "Rsh":          sdm_obj.rsh,
            "Rs":           sdm_obj.rs,
            "RMSE":             sdm_obj.rmse,
            "MAE":              sdm_obj.mae,
            "Area Factor":      sdm_obj.areaFactor,
            "Voc":          sdm_obj.voc,
            "Isc":          sdm_obj.isc,
            "Vmpp":         sdm_obj.vmpp,
            "Impp":         sdm_obj.impp,
        })
```

Lo que queda guardado en “records” es lo que irá en el CSV.

Una vez tenemos todos los datos, debemos generar el archivo CSV. Para ellos hemos utilizado la librería Polars, que nos permitirá convertir la lista de registros en un DataFrame y exportarlo a texto CSV utilizando un buffer de memoria:

```
df = pl.DataFrame(records)
buf = io.StringIO()
df.write_csv(buf)
buf.seek(0)
```

Con “df = pl.DataFrame(records)” convertimos la lista de diccionarios “records” en una estructura con forma de tabla (DataFrame). Después, con “buf = io.StringIO()” creamos el buffer de memoria que está en la memoria RAM y guarda texto. Con “df.write_csv(buf)” convertimos el DataFrame a texto en formato CSV, y en vez de guardarlo en disco en forma de archivo temporal, lo guardamos en el buffer que hemos creado. Por último, movemos el cursor de lectura al inicio del buffer con “buf.seek(0)”. Esto es necesario para que la descarga del archivo se realice correctamente con StreamingResponse, ya que FastAPI va a intentar leer desde la posición actual, y si no se hiciera “seek(0)”, el cursor estaría al final, así que no se leería nada para poder descargar.

Como último paso, se prepara la cabecera para establecer el nombre del archivo que se va a descargar y se realiza la descarga en el “return”:

```
headers = {
        "Content-Disposition": f"attachment;
filename={sim.nombre}_metrics.csv"
    }
    return StreamingResponse(buf, media_type="text/csv", headers=headers)
```

5.1.6 Routers FastAPI

FastAPI permite unos routers predefinidos que nos permiten implementar rápidamente un sistema de autenticación y gestión de usuario.

En el proyecto se han incluido unos cuantos, que se añaden al objeto FastAPI con app.include_router(…).

Se va a analizar y explicar uno por uno estos routers.

Autenticación con JWT (inicio de sesión)

Este router proporciona las rutas necesarias para realizar el login mediante JSON Web Tokens (JWT), utilizando cookies como método de transporte, que fue definido en auth_backend. El prefijo /auth/jwt se añade a todas estas rutas. Estas rutas las usaremos en el JavaScript del frontend al pulsar un botón para, por ejemplo, iniciar sesión. Algunas de las rutas incluidas son:

- POST /auth/jwt/login: iniciar sesión.

- POST /auth/jwt/logout: cerrar sesión.

El código que permite usar este router es el siguiente:

```
app.include_router(
    fastapi_users.get_auth_router(auth_backend), prefix="/auth/jwt",
tags=["auth"]
)
```

Registro de usuarios

Este router permite registrar nuevos usuarios. Incluye la ruta que se usará en el JavaScript del frontend cuando pulsemos el botón de registrar. La ruta será "POST /auth/register"

El código que permite incluir este router es:

```
app.include_router(
    fastapi_users.get_register_router(UserRead, UserCreate),
    prefix="/auth",
    tags=["auth"],
)
```

Las clases UserRead y UserCreate están definidas en schemas.py, y no han sido modificadas, son heredadas directamente de la librería schemas proporcionada por FastAPI Users:

```
class UserRead(schemas.BaseUser[uuid.UUID]):
    pass
class UserCreate(schemas.BaseUserCreate):
    pass
```

Recuperación de contraseña

Este router nos proporciona las rutas necesarias para restablecer la contraseña. En este caso hemos usado la ruta POST /auth/reset-password, que permite cambiar la contraseña. El código del router es:

```
app.include_router(
    fastapi_users.get_reset_password_router(),
    prefix="/auth",
    tags=["auth"],
)
```

Routers para mejoras

Se han incluido routers para verificar usuarios (get_verify_router) y para poder actualizar datos de usuarios (get_users_router), pero sus rutas no están en uso actualmente. En mejoras futuras, será muy sencillo añadir estas funcionalidades

y no será necesario modificar el backend, ya que las rutas estarán disponibles directamente.

5.2 Implementación del frontend

En este apartado veremos toda la implementación de las plantillas HTML utilizadas en el proyecto. Veremos cómo se ha implementado el diseño y la funcionalidad del frontend y nos permitirá conocer al completo cómo interactúa con el backend del programa.

Se ha utilizado HTML, JinJa2, JavaScript y Bootstrap. Es recomendable tener conocimientos previos sobre estas tecnologías para comprender este apartado en su totalidad.

Todos los archivos que vamos a desglosar en este apartado se encuentran dentro del directorio “templates”, ya que tiene que coincidir con lo que establecimos en el backend cuando especificamos la ubicación de las plantillas para JinJa2.

5.2.1 Navbar.html

Esta plantilla contiene la barra superior con el menú desplegable disponible en todas las pantallas de la aplicación. Para no repetir el mismo código en cada plantilla, se ha creado esta plantilla que será muy sencilla de incluir en las demás plantillas. Solo necesitamos utilizar “{% include 'navbar.html' %}” en el resto de plantillas, y se incluirá el código que veremos a continuación.

Primero comprobamos si hay un usuario enviado desde el backend como contexto con “{% if user %}”. Esto está hecho para discernir la forma del menú y no mostrar el desplegable si no hay un usuario autenticado.

Luego, para mostrar la opción adicional “Admin”, utilizaremos “{% if user.is_superuser %}” e incluiremos el código necesario para mostrar esa opción.

El código HTML con las partes de código JinJa2 es el siguiente:

```
<nav class="navbar navbar-dark bg-dark fixed-top">
  <div class="container-fluid">
    <a class="navbar-brand" href="/">Simulador Solar</a>
    {% if user %}
    <div class="dropdown">
      <a class="nav-link dropdown-toggle text-white" href="#"
role="button" id="userMenu" data-bs-toggle="dropdown"
        aria-expanded="false">
        <i class="bi bi-person-circle" style="font-size:1.5rem;"></i>
      </a>
      <ul class="dropdown-menu dropdown-menu-end" aria-
labelledby="userMenu">
        <li><a class="dropdown-item" href="/dashboard">Mi cuenta</a></li>
```

```
        {% if user.is_superuser %}
      <li>
          <hr class="dropdown-divider">
        </li>
        <li><a class="dropdown-item" href="/admin">Admin</a></li>
        {% endif %}
        <li>
          <hr class="dropdown-divider">
        </li>
        <li><a class="dropdown-item"
href="/historial">Simulaciones</a></li>
        <li>
          <hr class="dropdown-divider">
        </li>
        <li>
          <form id="logoutForm" method="POST" action="/auth/jwt/logout">
            <button type="submit" class="dropdown-item">Cerrar
sesión</button>
          </form>
        </li>


      </ul>
    </div>
    {% else %}
    <div class="d-flex ms-auto gap-2">
      <a href="/login" class="btn btn-outline-light">Entrar</a>
      <a href="/registro" class="btn btn-primary">Registrarse</a>
    </div>
    {% endif %}
  </div>
</nav>
```

Podemos ver que, si no hubiera un usuario autenticado, en vez de mostrarse el menú desplegable, se mostrarían dos botones para acceder al registro o al inicio de sesión. Este caso no se va a dar en la aplicación tal y como está configurado el backend ahora mismo, pero si se quisiera modificar para que la web se pudiera usar sin estar autenticado, el menú se comportaría de forma correcta.

Para la explicación de las siguientes plantillas no se va a incluir todo el código HTML, ya que la mayoría de lógica la encontraremos en los scripts de JavaScript.

Por último, nos queda analizar el script que maneja el cierre de sesión. El elemento con id=”logoutForm” hace la acción “auth/jwt/logout”, que es una ruta de FastAPI Users para cerrar sesión. Interceptaremos esta acción con JavaScript para poder recargar la página justo al cerrar sesión. El script que hemos utilizado es este:

```
<script>
  const logoutForm = document.getElementById('logoutForm');
  logoutForm.addEventListener('submit', async e => {
    e.preventDefault();
    await fetch(logoutForm.action, {
      method: 'POST',
      credentials: 'include'
    });
    window.location.reload();
  });
</script>
```

Con el script esperamos a que se cierre sesión y luego recargamos la página.

5.2.2 Index.html

Es la página principal. Solo se van a destacar los aspectos más importantes para no extender el documento innecesariamente con código HTML.

Primero incluimos el menú con “{% include 'navbar.html' %}” como se ha mencionado anteriormente. A continuación, incluimos el div para el mapa de OpenStreetMap en el código HTML, lo manejaremos después con JavaScript:

```
<!-- Mapa OpenStreetMap -->

<div class="card mt-3">
    <div class="card-body">
        <h5 class="card-title">Mapa</h5>
        <div id="map"></div>
    </div>
</div>
```

Luego creamos un formulario en el que introduciremos todos los datos de la simulación. La acción del formulario será llamar a la función validarFormulario(), que permite controlar la entrada de datos de forma precisa:

```
 <form action="/resultado" method="post" onsubmit="return
validarFormulario()">
```

A continuación, se incluirán todos los campos del formulario. Todos son obligatorios, lo indicamos con “required”. Por ejemplo:

```
<div class="mb-3">
    <label class="form-label">Nombre de la Simulacion</label>
    <input type="text" name="nombre" id="nombre" class="form-control"
required>
</div>
```

Hay dos campos cuyo comportamiento es manejado por JavaScript. Son la Zona Horaria y el desplegable de paneles, los veremos a continuación:

```
<div class="mb-3">
    <label class="form-label" for="timezone">Zona Horaria</label>
    <select id="timezone" name="timezone" class="form-control">
        <!-- Opciones generadas con JavaScript -->
    </select>
</div>
```

```
<div class="mb-3">
    <label for="panelSelect" class="form-label">Selecciona un panel o <a
href="/agregarPanel">agrega un panel</a></label>
     <select id="panelSelect" name="panel_id" class="form-control"
required>
         <option value="" disabled selected>Cargando paneles…</option>
      </select>
</div>
```

Antes de pasar al script, los desplegables de spa_methods y airmass_models se han hecho utilizando un bucle for recorriendo las listas obtenidas por contexto desde el backend:

```
<div class="mb-3">
    <label for="spa_method" class="form-label">SPA Method:</label>
    <select name="spa_method" id="spa_method" class="form-control"
required>
        {% for m in spa_methods %}
            <option value="{{ m }}">{{ m }}</option>
        {% endfor %}
     </select>
</div>
```

El de airmass_models es igual, pero sustituyendo “{% for m in spa_methods %}” por “{% for m in airmass_models %}”.

Ahora sí, se va a analizar la parte de script. Comenzaremos con la parte del mapa interactivo, para ello debemos incluir el siguiente script para importar Leaflet:

```
<script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
```

Y a continuación comenzar con su manejo con JavaScript. Crearemos una variable “map” y seleccionaremos las coordenadas que queremos que se muestren por defecto al cargar el mapa:

```
var map = L.map('map').setView([38.3452, -0.4810], 10);
```

En este caso, se ha centrado el mapa en Elche con un nivel 10 de zoom.

Ahora cargaremos las capas visuales desde OpenStreetMap:

```
// Cargar mapa base de OpenStreetMap
L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '&copy; OpenStreetMap contributors'
}).addTo(map);
```

Ahora debemos capturar las coordenadas cuando se clica en el mapa. Para ello capturaremos la latitud y la longitud, y limitaremos los decimales a 6, y luego escribiremos estas coordenadas en el elemento con id coordsForm, que es el campo de coordenadas del formulario:

```
var marker;
map.on('click', function (e) {
    var lat = e.latlng.lat.toFixed(6);
    var lng = e.latlng.lng.toFixed(6);
    document.getElementById('coordsForm').value = lat + ', ' + lng;
```

Por último, debemos añadir un marcador al mapa para que el usuario sepa dónde ha pinchado en el mapa. La lógica seguida es: si ya había un marcador antes, lo borra y luego crea un nuevo marcador en la posición seleccionada. El código necesario es:

```
    if (marker) {

       map.removeLayer(marker);
    }
    marker = L.marker([lat, lng]).addTo(map);
});
```

Continuamos con la función validarFormulario. Actualmente simplemente está definida en JavaScript, pero no contiene código. Es un punto de mejora, solo habría que incluir aquí las comprobaciones pertinentes y el formulario automáticamente realizaría estas comprobaciones antes de enviar sus datos al backend.

Para crear el desplegable con las zonas horarias, se ha usado el estándar Intl y se creado un desplegable <select> de forma dinámica. Por último, se ha establecido Europe/Madrid como zona horaria por defecto:

```
const tzSelect = document.getElementById('timezone');
Intl.supportedValuesOf('timeZone').forEach(tz => {
    const option = document.createElement('option');
    option.textContent = tz;
    tzSelect.appendChild(option);
});
tzSelect.value = 'Europe/Madrid'; //Establecer por defecto Europe/Madrid
```

Ahora debemos cargar la lista de paneles desde el backend y rellenar el elemento <select> desde JavaScript de forma dinámica.

Primero obtenemos el elemento del DOM 'panelSelect'. Luego hacemos una petición fetch a la ruta /panels, la cual nos devolverá un array de paneles:

```
fetch('/panels')
    .then(async res => {
        if (!res.ok) {
            console.error('Error al cargar paneles', await res.text());
            return;
        }
         return res.json();
     })
```

Si la respuesta es válida, se limpia el contenido del desplegable y se inserta un placeholder que indica al usuario que debe seleccionar una opción.

Por último, se recorrerán todos los paneles recibidos y se irá creando un elemento <option> para cada uno de ellos:

```
  .then(panels => {
       // Limpia el placeholder
       panelSelect.innerHTML = '<option value="" disabled selected>--
Elige un panel --</option>';
       // Añadir cada panel como opción
       panels.forEach(p => {
           const opt = document.createElement('option');
           opt.value = p.id;
           opt.textContent = p.nombre;
           panelSelect.append(opt);
           });
       })
       .catch(err => {
           console.error('Fetch panels failed:', err);
       });
});
```

Para evitar que dos usuarios lancen una simulación simultáneamente, se ha implementado una comprobación periódica mediante la función checkSimulando(), que se ejecuta inicialmente al cargar la página y luego cada dos segundos mediante setInterval. La función envía una petición GET a la ruta /simulando/status, la cual devuelve un objeto JSON con una clave sim que indica si el simulador está ocupado (true) o libre (false). Según el valor de sim, se cambia el diseño del botón de lanzamiento de simulación. Si hay una simulación en curso, el botón se desactiva, se cambia su estilo a btn-secondary y se muestra el texto "Simulación en curso". Si no hay nadie simulando, el botón vuelve a estar habilitado, con estilo btn-primary y el texto "Lanzar simulación".

Esta es la implementación de la función:

```
async function checkSimulando() {
    const resp = await fetch('/simulando/status');
    const { sim } = await resp.json();    // { sim: true/false }
    const btn = document.getElementById('launchBtn');
    if (sim) {
        btn.disabled = true;
        btn.classList.remove('btn-primary');
        btn.classList.add('btn-secondary');
        btn.textContent = 'Simulación en curso';
    } else {
        btn.disabled = false;
        btn.classList.remove('btn-secondary');
        btn.classList.add('btn-primary');
        btn.textContent = 'Lanzar simulación';
    }
}
```

Y aquí el intervalo de comprobación:

```
document.addEventListener('DOMContentLoaded', () => {
    checkSimulando();
    setInterval(checkSimulando, 2_000);
});
```

Por último, queda explicar el envío de los datos del formulario al backend.

Cuando el usuario completa el formulario con los parámetros deseados para la simulación, se intercepta su envío con JavaScript para manipular los datos de forma personalizada y enviarlos al backend a través de fetch, sin recargar la página.

Identificamos el formulario con:

```
const simForm = document.querySelector('form[action="/resultado"]');
```

Ahora definimos un eventListener para el evento submit del formulario y así poder detener el comportamiento por defecto del navegador con e.preventDefault().

Primero debemos procesar las coordenadas, ya que las recibimos como una cadena de esta forma: "38.123456, -0.123456", y tenemos que separarlas en latitud y longitud, que lo haremos con Split() y parseFloat():

```
const coordText = document.getElementById('coordsForm').value;
const [latStr, lngStr] = coordText.split(',').map(s => s.trim());
const latitude = parseFloat(latStr);
const longitude = parseFloat(lngStr);
```

Ahora obtenemos la fecha del elemento del DOM con id “fecha” y la hora del elemento de tipo Time que la contiene y lo juntamos en un String siguiendo el formato ISO YYYY-MM-DDTHH:mm:

```
const fecha = document.getElementById('fecha').value;
const hora = document.querySelector('input[type="time"]').value;
const datetime = `${fecha}T${hora}`;
```

Obtenemos el id del panel seleccionado:

```
const panelId = parseInt(document.getElementById('panelSelect').value,
10);
```

Luego construimos el objeto JavaScript que contiene todos los campos requeridos y se extraen todos los valores del formulario:

```
const data = {
    panel_id: panelId,
    datetime: datetime,
    time_zone: document.getElementById('timezone').value,
    latitude: latitude,
    longitude: longitude,
    surface_tilt: parseFloat(document.getElementById('incS').value),
    surface_azimuth: parseFloat(document.getElementById('surAzi').value),
    ground_albedo: parseFloat(document.getElementById('surAlb').value),
    water_vapor_content:
parseFloat(document.getElementById('hum').value),
    ozone: parseFloat(document.getElementById('ozo').value),
    aerosol_turbidity_500nm:
parseFloat(document.getElementById('turb').value),
    pressure: parseFloat(document.getElementById('pres').value),
    altitude: parseFloat(document.getElementById('alt').value),
```

```
    temperature: parseFloat(document.getElementById('temp').value),
    voc: parseFloat(document.getElementById('voc').value),
    sc: parseInt(document.getElementById('sc').value),
  nombre: document.getElementById('nombre').value,
    spa_method: document.getElementById('spa_method').value,
    airmass_model: document.getElementById('airmass_model').value,
};
```

Finalmente, realizamos la petición POST al endpoint simulate/full, enviando los datos como JSON. Si el backend responde con una redirección, se navega automáticamente a la nueva URL, normalmente para mostrar el resultado. En caso de error, se notifica al usuario mediante un alert con el mensaje de error devuelto:

```
try {
    const resp = await fetch('/simulate/full', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(data),
        redirect: 'follow'
    });
    if (resp.redirected) {
        window.location.href = resp.url;
        return;
    }
    if (!resp.ok) {
        const ebody = await resp.json();
        alert('Error en simulación: ' + ebody.detail);
    }
} catch (err) {
    console.error(err);
    alert('Error de red o del servidor.');
}
});
```

5.2.3 Reference_spectra.html

Esta plantilla es la que nos muestra el formulario para lanzar simulaciones usando espectros de referencia. Muchas partes son idénticas al apartado anterior, ya que la forma de cargar la lista de paneles y de comprobar si hay alguien simulando es exactamente igual, así que se van a obviar en este apartado.

Sí que es necesario explicar la forma de cargar el desplegable que muestra los espectros de referencia disponibles. Esto lo hacemos usando un bucle for de JinJa2 que recorre la lista enviada como contexto desde el backend y creando un option por cada uno de los elementos de la lista:

```
<div class="mb-3">
    <label for="spectrum_name" class="form-label">Espectro</label>
    <select id="spectrum_name" name="spectrum_name" class="form-control"
required>
        <option value="" disabled selected>-- Elige un espectro --
</option>
        {% for name in spectra_list %}
        <option value="{{ name }}">{{ name }}</option>
        {% endfor %}
    </select>
</div>
```

Esta vez, el formulario se envía con el comportamiento por defecto del navegador, haciendo una petición POST a "/simulate/reference_spectra":

```
<form action="/simulate/reference_spectra" method="post">
```

5.2.4 Login.html

En esta plantilla encontraremos un formulario con los campos usuario y contraseña.

Interceptaremos el envío con JavaScript para poder incluir un mensaje que indique que se está iniciando sesión y para poder personalizar y controlar el funcionamiento. Se realizará una petición POST a la ruta auth/jwt/login incluyendo los datos del formulario en un FormData. Si el servidor nos devuelve una respuesta con código 204 (No content), se considera que el login ha sido exitoso y se redirige a la página principal. Si hay un error, se muestra el mensaje correspondiente en el elemento msg.

El script es el siguiente:

```
const form = document.getElementById('loginForm');
const msg = document.getElementById('msg');
form.addEventListener('submit', async e => {
    e.preventDefault();
    msg.textContent = 'Iniciando sesión…';
    msg.className = '';
    const formData = new FormData(form);

    try {
        const res = await fetch('/auth/jwt/login', {
            method: 'POST',
            credentials: 'include',
            body: formData
        });
        if (res.status !== 204) {
            const text = await res.text();
```

```
            throw new Error(text || `Error ${res.status}`);
        }
        msg.textContent = '¡Sesión iniciada correctamente!';
        msg.className = 'text-success';
        setTimeout(() => window.location.href = '/', 500);
    } catch (err) {
        msg.textContent = 'Error: ' + err.message;
        msg.className = 'text-danger';
    }
});
```

5.2.5 Registro.html

Esta plantilla es idéntica a la anterior en cuanto a lógica se refiere, simplemente cambiamos la petición del fetch y utilizamos la ruta /auth/register en vez de /auth/jwt/login.

5.2.6 Admin.html

Esta plantilla está compuesta por una tabla en HTML que se rellena dinámicamente con la lista de usuarios que hemos recibido como contexto desde el backend. Cada fila de esta tabla contiene el correo del usuario, un checkbox para activar o desactivar al usuario y un checkbox para poder hacer administrador al usuario que se desee. Por último, habrá un botón para eliminar a un usuario. Su funcionamiento se manejará en JavaScript

Aquí podemos ver el cuerpo de la tabla con el bucle JinJa2 para recorrer la lista de usuarios:

```
<tbody>
    {% for u in users %}
    <tr>
        <td>{{ u.email }}</td>
        <td>
            <input type="checkbox" {% if u.is_active %}checked{% endif %}
                onchange="updateUser('{{ u.id }}', { is_active:
this.checked })" />
        </td>
        <td>
            <input type="checkbox" {% if u.is_superuser %}checked{% endif
%}
                onchange="updateUser('{{ u.id }}', { is_superuser:
this.checked })" />
        </td>
        <td>
            <button class="btn btn-sm btn-danger" onclick="deleteUser('{{
u.id }}', '{{ u.email }}')">
                Borrar
            </button>
```

```
        </td>
    </tr>
    {% endfor %}
</tbody>
```

También se puede apreciar que se han usado condicionales if para poner marcados o desmarcados los checkbox dependiendo del estado actual de los usuarios. Por otra parte, cada vez que marcamos o desmarcamos alguno de los checkbox de is_active o is_superuser , se dispara la función updateUser(…), definida en JavaScript. Si pulsamos en el botón de borrar, se dispara la función deleteUser(…)

La función updateUser(id, data) permite modificar los atributos is_active y is_superuser de un usuario. Se envía una solicitud PATCH a la ruta /users/{id}, siendo {id} el Id del usuario a actualizar. En el cuerpo de la solicitud se envía un objeto JSON con la propiedad a modificar. Esta es su implementación:

```
async function updateUser(id, data) {
      const res = await fetch(`/users/${id}`, {
        method: 'PATCH',
        credentials: 'include',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(data)
      });
      if (!res.ok) {
        alert('Error al actualizar usuario: ' + res.status);
      }
    }
```

La función deleteUser(id, email) elimina a un usuario de la base de datos. Cuando se pulse el botón “borrar”, se mostrará un mensaje de confirmación para evitar errores. Si se acepta la confirmación, se envía una solicitud DELETE al backend con el Id del usuario a eliminar. Si la operación se completa con éxito, se recarga la página para reflejar los cambios en la tabla. La implementación de la función es la siguiente:

```
async function deleteUser(id, email) {
      if (!confirm(`¿Eliminar usuario ${email}?`)) return;
      const res = await fetch(`/users/${id}`, {
        method: 'DELETE',
        credentials: 'include'
      });
      if (res.ok) {
        window.location.reload();
      } else {
```

```
        alert('Error al borrar usuario: ' + res.status);
      }
    }
```

Podemos ver que el confirm dentro del if, se encargará de mostrar el mensaje de confirmación y finalizará la ejecución de la función si no se acepta la confirmación.

5.2.7 Dashboard.html

Esta plantilla es muy sencilla en cuanto a HTML se refiere, ya que solo cuenta con un texto que muestra el email del usuario autenticado, y un botón para ir a la plantilla de cambio de contraseña. Si pulsamos este botón, iremos a la plantilla cambiarPassword.html, que veremos a continuación.

5.2.8 CambiarPassword.html

Esta plantilla también es muy sencilla a nivel de HTML, ya que solo contiene un campo de tipo password para introducir la nueva contraseña y un botón para confirmar la operación:

```
<form id="change-password-form" method="POST">
    <div class="mb-3">
        <label for="newPassword" class="form-label">Nueva
contraseña</label>
        <input type="password" id="newPassword" name="password"
class="form-control" required>
    </div>
    <button type="submit" class="btn btn-primary">Cambiar
contraseña</button>
```

Ahora, para controlar el comportamiento al pulsar en el botón “cambiar contraseña", se ha usado JavaScript. Primero obtenemos el elemento con id “change-password-form”, que es el formulario completo. Luego anulamos el envío normal del formulario con e.preventDefault(), y hacemos una petición PATCH a la ruta /users/me, incluyendo en el cuerpo de la petición la nueva contraseña en formato JSON. Si la contraseña se ha cambiado exitosamente mostramos una alerta indicándolo. En caso contrario, mostraremos el mensaje de error correspondiente. La implementación es la siguiente:

```
document
    .getElementById('change-password-form')
    .addEventListener('submit', async e => {
        e.preventDefault();
        const pw = document.getElementById('newPassword').value;
        const resp = await fetch('/users/me', {
```

```
            method: 'PATCH',
            headers: {
                'Content-Type': 'application/json'
            },
            credentials: 'include',
            body: JSON.stringify({ password: pw })
        });
        if (resp.ok) {
            alert('Contraseña cambiada con éxito 🎉');
        } else {
            const err = await resp.json();
            alert('Error: ' + (err.detail ?? JSON.stringify(err)));
        }
    });
```

5.2.9 Add_panel.html

Esta plantilla es la que permite al usuario añadir un panel. Es muy sencilla, solo cuenta con un formulario básico en HTML con un campo de texto para el nombre y un elemento <textarea> para introducir la descripción. Además, cuenta con un checkbox que, si se marca, indica que el panel será público y lo podrán usar todos los usuarios. El envío del formulario será estándar, no lo modificaremos mediante JavaScript. Al pulsar en Guardar Panel, se realizará una petición POST al endpoint “/agregarPanel”.

5.2.10 Historial.html

En esta plantilla, la estrategia seguida para mostrar un historial de simulaciones ha sido rellenar una tabla en HTML de forma dinámica mediante bucles de JinJa2 que recorran la lista de simulaciones obtenida como contexto desde el backend.

Se ha comenzado con un if que comprueba si hay simulaciones:{% if simulaciones %}. Si no hay simulaciones, mostraremos el mensaje “Aún no tienes simulaciones en tu historial”.

Luego construimos la cabecera de la tabla, con los campos que se ha decidido incluir:

```
<div class="table-responsive">
    <table class="table table-striped table-hover align-middle">
        <thead class="table-dark">
            <tr>
                <th scope="col">Id Sim</th>
                <th scope="col">Nombre</th>
                <th scope="col">Tipo</th>
                <th scope="col">Panel ID</th>
                <th scope="col">Acciones</th>
                <th scope="col"></th>
            </tr>
```

```
        </thead>
```

En el cuerpo de la tabla, crearemos un bucle for que recorrerá la lista simulaciones: {% for sim in simulaciones %}. A continuación, rellenaremos las filas (<td>) con los datos:

```
{% for sim in simulaciones %}
<tr>
    <th scope="row">{{ sim.id }}</th>
    <td scope="row">{{ sim.nombre }}</td>
    <td>
        {% if sim.spectra %}
        Espectro de referencia
        {% else %}
        Escenario personalizado
        {% endif %}
    </td>
    <td>{{ sim.panel }}</td>
```

Por último, queda explicar el botón “ver detalle” y el botón “borrar”.

El botón “ver detalle” es un enlace <a> con el href establecido en el endpoint “/resultado” y añadiendo en la URL el id de la simulación a mostrar y el method_index establecido en el valor 1. Esto se entenderá mejor al visualizar el código:

```
 <td>

    <a href="/resultado?sim_id={{ sim.id }}&method_index=1" class="btn
btn-sm btn-primary">
        Ver detalle
    </a>
</td>
```

El botón de borrar, sin embargo, se ha incrustado dentro de un formulario, para poder mostrar una alerta de confirmación antes de proceder con el borrado de una simulación:

```
<form action="/simulacion/{{ sim.id }}/borrar" method="post"
style="display:inline;">
    <button type="submit" class="btn btn-sm btn-danger"
      onclick="return confirm('¿Seguro que quieres eliminar la simulación
#{{ sim.id }}?');">
      Eliminar
    </button>
</form>
```

5.2.11 Resultado.html

Esta plantilla muestra el resultado de la simulación. Recordemos primero el aspecto visual de la misma. Este lo podemos encontrar en las figuras: Figura 24, Figura 25, Figura 26 y Figura 27. Como es una plantilla bastante extensa y con mucho código, la analizaremos por partes.

Comenzamos con la estructura HTML y con la lógica JinJa2 usada. Primero mostramos el nombre de la simulación con {{ simulacion.nombre }}. Recordemos que el objeto simulación nos llega como contexto desde el backend. Luego haremos una distinción entre si se trata de una simulación realizada con espectros de referencia o con escenarios personalizados para mostrar el nombre del espectro de referencia usado o, en su defecto, mostrar el id del escenario que ha dado lugar a la simulación:

```
<h2>Resultado de Simulación: {{ simulacion.nombre }}</h2>
{% if simulacion.spectra is not none %}
<p>Espectro utilizado: <strong>{{ simulacion.spectra }}</strong></p>
{% else %}
<p>ID del escenario: <strong>{{ simulacion.escenario }}</strong></p>
{% endif %}
<a href="/simulacion/{{ simulacion.id }}/download_csv" class="btn btn-
outline-success mb-3">
    Descargar CSV
</a>
```

Se puede apreciar que hemos hecho uso del atributo spectra del objeto simulación para discernir entre los dos tipos de simulaciones mencionadas anteriormente. Además, se ha incluido el botón de descarga en CSV, que simplemente es un enlace que, al ser pulsado, nos lleva al endpoint “/simulacion/{sim.id}/download_csv y nos permite descargar los datos.

Seguidamente nos encontramos la parte de selección del algoritmoo, que nos permite elegir entre los dos modelos de ajuste (TSLLS o TSLLS-refined) mediante un formulario.

El formulario envía una petición GET a “/resultado” cuando el usuario cambia de opción en el desplegable, enviando los parámetros que requiere este endpoint, que son el id de la simulación y el índice del método para el cual se quieren mostrar los resultados. Al hacer un cambio, se recargará la página con los datos del algoritmo elegido. El código necesario para hacer esto es el siguiente:

```
<form method="get" action="/resultado" class="mb-4">
    <input type="hidden" name="sim_id" value="{{ simulacion.id }}">
    <label for="variable_metodo" class="form-label">Algoritmo SDM</label>
    <select id="variable_metodo" name="metodo_id" class="form-select w-
auto d-inline-block ms-2"
        onchange="this.form.submit()">
```

```
    <option value="1" {% if metodo_id==1 %}selected{% endif
%}>tslls</option>
        <option value="2" {% if metodo_id==2 %}selected{% endif %}>tslls-
refined</option>
    </select>
</form>
```

A continuación, encontraremos los 4 puntos remarkables. Es muy fácil acceder a su valor, ya que hemos recibido el objeto SDM desde el backend. Por ejemplo, para acceder al valor Voc, es tan sencillo como hacer {{ sdm.voc }}. Como el código es idéntico para los 4 puntos, cambiando únicamente el nombre de la variable, solo se va a incluir el código para un punto:

```
<h2>Remarkables</h2>
    <div class="row mb-4">
      <div class="col-md-3 mb-3">
        <div class="card text-center">
          <div class="card-body">
            <h5 class="card-title">Voc (V)</h5>
            <p class="card-text">{{ sdm.voc }}</p>
          </div>
        </div>
      </div>
```

Para los siguientes, cambiaríamos {{ sdm.voc }} por {{ sdm.isc }}, {{ sdm.vmpp }} y {{ sdm.impp }}.

Ahora, debemos crear el elemento HTML <canvas> que modificaremos después en JavaScript para construir el gráfico de resultados. De momento solo incluimos en el HTML este código:

```
<div class="card mb-4">
    <div class="card-body">
      <canvas id="ivCurveSingle"></canvas>
    </div>
</div>
```

Pasamos ahora a la tabla que muestra los resultados en formato atributo/valor. Esta tabla se construye accediendo a cada uno de los valores que nos interesan y poniéndolos en un elemento <td> (fila). Como son todos idénticos cambiando simplemente la variable, al igual que en el ejemplo anterior, solo se va a incluir uno:

```
<h3 class="mt-4">Resultados de la simulación</h3>
    <div class="table-responsive">
      <table class="table table-striped table-bordered table-hover">
        <tbody>
          <tr>
```

```
      <th scope="row">Voc (V)</th>
            <td>{{ sdm.voc }}</td>
          </tr>
```

Se replica lo mismo con casi todo el resto de variables que se muestran en la tabla. Los dos únicos campos de la tabla que cambian son el método y si se muestra o no el Panel y su descripción, dependiendo de si la simulación se ha realizado con espectros de referencia o no.

Para el método, se usa un simple if, que indica que si el method_index es igual a 1, se mostrará el texto TSLLS, y si es igual a 2, TSLLS-refined:

```
 {% if metodo_id == 1 %}

<tr>
    <th scope="row">Método</th>
    <td>TSLLS</td>
</tr>
{% endif %}
{% if metodo_id == 2 %}
<tr>
    <th scope="row">Método</th>
    <td>TSLLS-refined</td>
</tr>
{% endif %}
```

Si la simulación se ha realizado con un espectro de referencia, como se elimina la segunda tabla que muestra todos los detalles del escenario, se ha decidido incluir en esta tabla el panel usado y su descripción:

```
{% if simulacion.spectra is not none %}
<tr>
    <th scope="row">Panel</th>
    <td>{{ pan.nombre }}</td>
</tr>
<tr>
    <th scope="row">Descripcion Panel</th>
    <td>{{ pan.descripcion }}</td>
</tr>
{% endif %}
```

La siguiente tabla solo se mostrará si la simulación se ha realizado con un escenario personalizado, y mostrará los detalles de ese escenario. Como hemos recibido desde el backend un objeto Escenario llamado esc, podemos acceder a los valores muy fácilmente y rellenar la tabla de forma muy sencilla. Solo se mostrará como ejemplo el primer campo, ya que los siguientes son iguales, pero

cambiando la variable que nos proporciona el valor, igual que en los ejemplos anteriores:

```
{% if simulacion.spectra is none %}
<h3 class="mt-4">Detalles del Escenario</h3>
<div class="table-responsive">
    <table class="table table-striped table-bordered">
        <tbody>
            <tr>
                <th scope="row">ID</th>
                <td>{{ esc.id }}</td>
            </tr>
```

Para la parte de JavaScript que nos permitirá rellenar el gráfico, primero necesitamos recuperar los arrays V e I y convertirlos en arrays de JavaScript. Para ello hacemos:

```
const V = JSON.parse(`{{ simulacion.V | replace("'", "\\'") }}`);
const I = JSON.parse(`{{ simulacion.I | replace("'", "\\'") }}`);
```

Se usa replace("'", "\\'") como protección para evitar errores de sintaxis si las cadenas contienen comillas simples.

Ahora guardaremos los arrays de datos simulados teóricamente, también los recibimos desde el backend:

```
const V2j = {{ V2 | tojson }};
const I2j = {{ I2 | tojson }};
```

Seguidamente crearemos un objeto que llamaremos R, que contendrá los remarkables:

```
const R = {
  Voc: {{ sdm.voc }},
  Isc: {{ sdm.isc }},
  Vmpp: {{ sdm.vmpp }},
  Impp: {{ sdm.impp }}
};
```

Procedemos a calcular y guardar la curva de potencia (W) que se representará de color naranja en el gráfico:

```
const P2 = V2j.map((v, i) => ({ x: v, y: v * I2j[i] }));
```

Obtendremos el elemento canvas creado en el body del HTML:

```
const ctx = document.getElementById('ivCurveSingle').getContext('2d');
```

Ahora debemos definir los datasets, que son lo que representará puntos o curvas en el gráfico. Crearemos un dataset por cada remarkable, tres en total, ya que el Impp y Vmpp se multiplican para obtener el Pmpp (Punto de potencia máxima). Como ejemplo se mostrará un solo dataset:

```
new Chart(ctx, {
      type: 'line',
      data: {
        datasets: [
          {
            label: 'Isc',
            data: [{ x: 0, y: R.Isc }],
            showLine: false,
            pointStyle: 'RectRot',
            pointRadius: 9,
            backgroundColor: 'red',
            borderColor: 'black',
            yAxisID: 'y'
          },
```

Ahora pasamos a la representación de los puntos medidos directamente de la simulación. Recordemos que estos puntos también se representan con un dataset:

```
{
    label: 'Medida',
    showLine: false,
    data: V.map((v, i) => ({ x: v, y: I[i] })),
    pointBackgroundColor: 'black',
    pointBorderColor: 'black',
    pointRadius: 4,
    pointHitRadius: 0,
    pointHoverRadius: 4,
},
```

Ahora dibujaremos la curva teórica en color rojo. No se va a adjuntar el código porque es muy similar al anterior, pero cambiando los arrays usados en data, estableciendo un border color y dejando showLine en True (para unir los puntos de los arrays con una línea).

Por último, representaremos la curva de potencia, calculada anteriormente, con el siguiente dataset:

```
{
    label: 'Potencia (W)',
    data: P2,
    borderColor: 'orange',
```

```
    fill: false,
    pointRadius: 0,
    pointHitRadius: 0,
    pointHoverRadius: 0,
    tension: 0.2,
    yAxisID: 'y2'
}
```

Adicionalmente configuraremos los nombres y tipos de los ejes que se quieren representar y se ajustan los plugins desactivando los tooltips emergentes para simplificar la visualización.

Con todo esto, tendríamos el gráfico generado con todos los datos importantes. No se ha incluido todo el código de configuración de ejes y plugins porque no se ha considerado relevante para la comprensión general de esta plantilla y puede dificultar la lectura del documento.

6. Propuestas de mejora

Una vez finalizado tanto el diseño como la implementación de esta aplicación, solo nos queda revisar los puntos que se pueden mejorar de cara a futuras versiones. Si bien todas las propuestas que vamos a revisar se podrían haber implementado desde un principio, el tiempo de desarrollo es limitado y, a veces, hay que priorizar qué partes son imprescindibles y cuáles no. Por eso mismo, vamos a valorar diferentes mejoras que se podrían implementar de cara al futuro.

6.1 Página Dashboard

Esta página no tiene practicamente ningún contenido, solo muestra el correo y permite cambiar el correo. Como mejoras se podría añadir información adicional, por ejemplo, número de simulaciones realizadas, fecha de creación del perfil, número de paneles públicos, acceso rápido a las simulaciones más recientes y un resumen de actividad mostrando el método SDM más utilizado y el panel más utilizado.

6.2 Uso sin Login

Sería muy interesante permitir el uso de la aplicación sin haber iniciado sesión en el sistema, pudiendo solo utilizar los paneles públicos ya disponibles. Cuando se realizara la simulación se redirigiría al usuario a la página de resultados, y aquí se le permitiría descargarlos en CSV, así como descargar un PDF en el que se guardara también el gráfico generado, para tener un documento más visual. No se guardaría información en el sistema sobre estas simulaciones.

6.3 Modo simplificado

Para el formulario de escenarios personalizados, se podría poner un botón que permitiera visualizar un modo simplificado del formulario, mostrando solo el mapa, los campos hora, día, zona horaria, panel, voc, sample count y altitud. El resto de campos avanzados, se establecerían con valores predeterminados. Esto permitiría a usuarios menos expertos en simulación solar experimentar y poder usar el sistema con más facilidad.

6.4 Cambiar contraseña

Sería interesante añadir un enlace que permitiera cambiar la contraseña desde el menú de login, ya que, actualmente, solo se puede cambiar la contraseña habiendo iniciado sesión. Si algún usuario pierde su contraseña, no sería posible recuperar su cuenta. El problema de hacer esto, es que habría que enviar un correo electrónico de recuperación al usuario, lo que nos lleva al siguiente apartado.

6.5 Verificación de Usuarios y Email

Para poder implementar el apartado anterior, habría que poder enviar tanto correos de recuperación, como correos de verificación de cuentas cuando se registran los usuarios. FastAPI Users ya tiene routers que nos permiten realizar estas acciones, pero se necesitaría poner en funcionamiento un servidor SMTP.

6.6 Barra de progreso

Para mejorar la usabilidad de la página es importante incluir una barra de progreso mientras se ejecuta una simulación, ya que, si no, estamos incumpliendo la regla heurística 1 de Jackob Nielsen, que nos indica que siempre debe haber visibilidad del estado del sistema. Si el usuario no ve una barra de progreso mientras se ejecuta una simulación (2-6 minutos), puede pensar que el sistema no está respondiendo y desesperarse o, en el peor de los casos, abandonar la aplicación.

6.7 Modo de simulación con barrido temporal

Esta mejora consiste en añadir un modo de simulación que repita la simulación cambiando el parámetro de la hora de simulación según un intervalo de tiempo preestablecido. Es decir, si seleccionamos como hora inicial las 13:00 del día 16/07/2025 y como hora final las 15:00 del mismo día, a intervalos de 15 minutos, el sistema ejecutaría en total 8 simulaciones consecutivas. La única diferencia entre las simulaciones sería la hora de simulación establecida. En la primera simulación, la hora sería las 13:00, en la segunda las 13:15, en la tercera 13:30, y así sucesivamente hasta llegar a la hora final establecida. Esto permitiría ver con exactitud la variación de resultados conforme avanza la hora y compararlos de manera más sencilla, ahorrando tiempo a los usuarios.

7. Conclusiones

El desarrollo de este proyecto muestra el proceso completo de creación de una aplicación web, desde una definición inicial de requisitos y objetivos, pasando por las distintas fases de diseño (funcional, estructural -base de datos-, visual, backend, frontend…) hasta llegar a tener un producto de software completamente funcional.

Bajo un punto de vista académico, este proyecto de desarrollo web full-stack, permite aplicar muchos de los conocimientos adquiridos a lo largo de la realización del grado de Ingeniería Informática en Tecnologías de la Información.

Durante el desarrollo del proyecto, se han aplicado conceptos clave de diseño de interfaces, estructuración de bases de datos, programación orientada a objetos, ingeniería del software y usabilidad, entre otros. Al tratarse de un desarrollo full-stack, se han podido aplicar y afianzar muchos de los conocimientos que se estudian pero que, a veces, no se aplican de forma práctica.

Por otra parte, se han utilizado tecnologías actuales y relevantes en el mundo laboral. El lenguaje principal utilizado en el backend, Python, es uno de los lenguajes de programación más usado a día de hoy que, junto con FastAPI, un framework moderno y optimizado, ha dado lugar a un proyecto computacionalmente ligero y eficiente.

Como se ha desarrollado código tanto para frontend como para backend, el proyecto, junto con su respectiva memoria, permite entender cómo se relacionan ambos entre sí y cómo comparten información entre ellos.

Al haberse desplegado y desarrollado en una Raspberry, se ha podido adquirir experiencia en el uso de entornos Linux que en el futuro podría ser muy útil, además de haberse adquirido mucha práctica al usar interfaces por línea de comandos, no gráficas.

Finalmente, lo más relevante es que este proyecto se ha llevado a la realidad, no se ha quedado en una propuesta teórica. La aplicación está actualmente en funcionamiento en una Raspberry Pi y está lista para ser utilizada por el personal del laboratorio de simulación solar, facilitando su trabajo diario y permitiendo un control más intuitivo y accesible del simulador.

8. Bibliografía

En este apartado se listarán las fuentes consultadas para la creación de todo el código usado en el proyecto, y para la obtención de información en la redacción de este documento.

[1] «Ministerio de Fomento, Documento Básico HE: Ahorro de energía, Código Técnico de la Edificación, Real Decreto 314/2006, Art. 15.6, España». [En línea]. Disponible en: https://www.codigotecnico.org

[2] S. R. Kurtz et al., «Outdoor rating conditions for photovoltaic modules and systems», Solar Energy Materials and Solar Cells, vol. 62, n.o 4, pp. 379-391, jun. 2000, doi: 10.1016/S0927-0248(99)00160-9.

[3] F. J. Toledo, J. M. Blanes, y V. Galiano, «Two-Step Linear Least-Squares Method For Photovoltaic Single-Diode Model Parameters Extraction», IEEE Trans. Ind. Electron., vol. 65, n.o 8, pp. 6301-6308, ago. 2018, doi: 10.1109/TIE.2018.2793216.

[4] «Implementation Limits For SQLite». Accedido: 16 de julio de 2025. [En línea]. Disponible en: https://sqlite.org/limits.html

[5] R. E. Bird y C. Riordan, «Simple Solar Spectral Model for Direct and Diffuse Irradiance on Horizontal and Tilted Planes at the Earth's Surface for Cloudless Atmospheres», Journal of Climate and Applied Meteorology, vol. 25, n.º 1, pp. 87-97, 1986.

[6] K. S. Anderson, C. W. Hansen, W. F. Holmgren, A. R. Jensen, M. A. Mikofski, y A. Driesse, «pvlib python: 2023 project update», *JOSS*, vol. 8, n.º 92, p. 5994, dic. 2023, doi: 10.21105/joss.05994.

[7] G. Allen y M. Owens, «Introducing SQLite», en The Definitive Guide to SQLite, Berkeley, CA: Apress, 2010, pp. 1-16. doi: 10.1007/978-1-4302-3226-1_1.

[8] «FastAPI». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://fastapi.tiangolo.com/

[9] «SQLite Documentation». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://www.sqlite.org/docs.html

[10] «Python 3.13 documentation», Python documentation. Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://docs.python.org/3/

[11] «SQLAlchemy». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://www.sqlalchemy.org

[12] «FastAPI Users». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://fastapi-users.github.io/fastapi-users/latest/

[13] «Uvicorn». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://www.uvicorn.org/

[14] «JavaScript | MDN». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[15] «HTML: HyperText Markup Language | MDN». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/HTML

[16] «Jinja — Jinja Documentation (3.1.x)». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://jinja.palletsprojects.com/en/stable/

[17] «Documentation - Leaflet - a JavaScript library for interactive maps». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://leafletjs.com/reference.html

[18] M. O. contributors Jacob Thornton, and Bootstrap, «Introduction». Accedido: 17 de julio de 2025. [En línea]. Disponible en: https://getbootstrap.com/docs/4.1/getting-started/introduction/