EV_CHARGING

1. Problema a resolver

Esta práctica está orientada al diseño, desarrollo e implementación de un sistema de aplicaciones distribuido (en concreto, un sistema de carga de coches eléctricos), en el que la principal tarea es desarrollar una correcta comunicación entre estas aplicaciones mediante Kafka y sockets en tiempo real. Una de las tareas más difíciles es poder coordinar los productores y consumidores de eventos sin acoplarlos ni sobrecargar nada, garantizando que los mensajes no se pierden ni contienen ningún error. Además también se debe crear un canal de comunicación de baja latencia para poder actualizar en tiempo real los datos para los usuarios.

Otro de los inconvenientes que puede surgir al desarrollar esta práctica es evitar fallos al añadir conexiones nuevas, como nuevos puntos de carga o nuevos usuarios. Todo esto se pondrá a prueba en el momento del despliegue, en el que tendremos que lanzar la central del sistema y algunos puntos de carga y usuarios, para poner a prueba nuestro sistema.

2. Tecnologías utilizadas

Tras un análisis de las tecnologías que podríamos utilizar para el desarrollo de este sistema hemos decidido utilizar las siguientes tecnologías:

Apache Kafka 4.1.0
Hemos elegido Apache Kafka para implementar la comunicación de eventos y mensajería ya que es la opción más fácil de implementar y la recomendada en clase. Kafka nos permite gestionar eficientemente las comunicaciones asíncronas entre componentes mediante topics y particiones por zonas geográficas.

Sockets (TCP/IP)
Hemos elegido sockets para la comunicación entre el Monitor y el Engine del propio Punto de Carga, y para el estado y autenticación entre el CP y la Central, ya que es la tecnología que debemos escoger según la práctica. Los sockets proporcionan comunicación bidireccional en tiempo real y baja latencia.

Docker y Docker Compose
Hemos elegido Docker para facilitar el lanzamiento, encapsulando cada aplicación en un contenedor, y así poder lanzar nuestro sistema en cualquier máquina, sin importar el sistema operativo de este. Docker Compose nos permite orquestar múltiples contenedores y gestionar redes y volúmenes de forma declarativa.

PostgreSQL
Hemos migrado de JSON a PostgreSQL como sistema de gestión de base de datos relacional. PostgreSQL nos proporciona persistencia robusta, transacciones ACID, integridad referencial mediante claves foráneas, y capacidades de consulta avanzadas mediante SQL. Es esencial para gestionar usuarios, puntos de carga, recargas, credenciales y claves de cifrado de forma segura y eficiente.

Flask y Flask-SocketIO
Hemos utilizado Flask como framework web ligero para implementar APIs REST (Central Web API, EV Registry) que exponen endpoints HTTP/HTTPS para registro, autenticación y consultas. Flask-SocketIO nos permite comunicación bidireccional en tiempo real con los clientes web (Driver) mediante WebSockets.

API REST (HTTPS)
Implementamos APIs REST con autenticación para la comunicación entre componentes:

- EV Registry API: Registro y baja de CPs con HTTPS/SSL
- Central Web API: Autenticación de CPs y entrega de claves de cifrado
- OpenWeather API: Integración con servicio externo para monitoreo meteorológico

OpenWeather API
Integración con la API externa de OpenWeather para obtener datos meteorológicos en tiempo real de las ubicaciones de los puntos de carga. Permite al sistema Weather Monitor detectar condiciones adversas (temperaturas extremas) y enviar alertas automáticas a la Central.

HTTPS/SSL (Certificados Autofirmados)
Implementamos seguridad en la capa de transporte mediante HTTPS con certificados SSL autofirmados para proteger las comunicaciones entre componentes críticos (CP ↔ Registry, aplicaciones externas ↔ Registry). Los certificados se generan automáticamente al iniciar el sistema.

Criptografía (AES-256-CBC)
Utilizamos la librería cryptography de Python para implementar cifrado simétrico de extremo a extremo en las comunicaciones Engine ↔ Central vía Kafka. Cada CP obtiene una clave AES-256 única tras autenticarse con Central, garantizando confidencialidad de datos sensibles durante las recargas.

Python 3.12
Hemos optado por escoger el lenguaje Python por su facilidad de uso, pero principalmente por su amplio ecosistema de librerías para trabajar con Kafka (kafka-python), sockets, contenedores, criptografía (cryptography), bases de datos (psycopg2), APIs REST (flask, requests) y manejo de datos (json, datetime, hashlib).

HTML y Jinja2
Hemos utilizado HTML con templates Jinja2 para realizar las interfaces web de las aplicaciones (Driver, Engine, Registry), ya que es la opción más utilizada globalmente, muy fácil de utilizar, robusta y se integra perfectamente con Flask.

Scripts Bash
Hemos creado scripts ejecutables (.sh) para facilitar el lanzamiento, detención y gestión de las aplicaciones, permitiendo automatizar el despliegue completo del sistema con un solo comando (start.sh). Los scripts gestionan la construcción de imágenes Docker, creación de redes, inicialización de volúmenes y apertura de terminales.

Sistema de Logs y Auditoría
Implementamos un módulo de auditoría centralizado (audit_logger.py) que registra eventos críticos en cada componente (autenticación, recargas, errores) con marcas de tiempo, permitiendo trazabilidad completa del sistema y facilitando el debugging y cumplimiento normativo.

Arquitectura por Capas y Principios SOLID
Organizamos el código siguiendo principios SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) con separación clara entre lógica de negocio (central_logic.py), acceso a datos (db_utils.py), estado (status_utils.py) y autenticación (auth_utils.py), mejorando la mantenibilidad y escalabilidad del sistema.

3. Desarrollo

3.1 Primeros problemas

Cuando empezamos a pensar e intentar desarrollar el sistema, tuvimos bastantes problemas. Principalmente no sabíamos como afrontar la planificación de desarrollo, ya que nunca habíamos hecho nada similar.

Tras un par de intentos fallidos de intentar abarcar mucho de golpe, nos dimos cuenta que lo mejor era empezar por lo más básico y esencial, que en este caso eran los puntos de carga (monitor y engine) y la central.

Una vez ya tuvimos claro por dónde empezar, nos costó bastante menos concretar y empezar el desarrollo, aquí fue donde tuvimos problemas con kafka y sockets, debido al poco conocimiento que teníamos sobre estos. Pero tras mucho pelear con ello, informarnos y aprender a utilizarlo, pudimos completar la comunicación entre las aplicaciones de forma correcta.

3.2 Comunicación por sockets (Engine-Monitor)

Empezamos el desarrollo por la comunicación mediante sockets entre el engine y el monitor. Implementamos una consola interactiva en la que veíamos si nos llegaba un mensaje o si se mandaba, en ambas terminales.

Al principio planteamos erróneamente el funcionamiento de esta comunicación, ya que pensamos que cuando el engine tenía algún problema, era este el que le mandaba un mensaje al monitor, cuándo realmente si hay un problema en Engine, el engine no le manda ningún mensaje de confirmación de vida al monitor y entonces es cuando el monitor le manda a la central esta incidencia.

Tras darnos cuenta de este error, replanteamos la forma de hacerlo y lo llevamos a cabo de la siguiente manera:
Monitor está constantemente preguntando al engine si está vivo, este responde Ok mientras esté vivo, en el momento que no responda, el monitor le mandará KO a la central, lo que se traduce como el estado “AVERIADO”

3.3 Base de datos

Tras pensarlo bastante y considerar todo tipo de posibilidades para manejar los datos. Consideramos que dada la poca importancia de la base de datos en esta práctica, ya que el núcleo de la práctica es la comunicación entre las aplicaciones, la mejor opción y la más fácil sería utilizar un fichero json para almacenar los datos y utilizarlo como base de datos, ahorrando así problemas y tiempo en conectarse a una base de datos real.

3.4 comunicación por sockets (monitor - central)

Una vez implementada la comunicación entre engine y monitor, empezamos con la implementación de la aplicación de la central, que será la encargada de interactuar con la base de datos y con el resto de aplicaciones.

Primero estuvimos haciendo pruebas con la comunicación. Por ejemplo, si escribíamos ‘incident’ en la terminal del engine, se le mandaba un mensaje al monitor y este a su vez le mandaba otro mensaje a la central, que debía actualizar el estado de ese cp a “AVERIADO”. Toda esta comunicación era por sockets entre el monitor y la central.

Después de estar probando y comprobar que funcionaba correctamente y la central modificaba la base de datos como esperábamos ya implementamos la comunicación correctamente, de manera que el monitor le manda continuamente un mensaje ‘OK’ ,mientras esté conectado, a la central. Y si el engine se desconecta, el monitor responde con ‘KO’ lo que hace que el estado de ese CP se ponga a “Averiado”

3.5 comunicación kafka (Driver - Central)

Una vez funcionando la comunicación entre Central y el punto de carga mediante sockets, empezamos con la implementación de kafka.
La comunicación de kafka entre Driver y central se basa en lo siguiente:

Primero el driver manda un mensaje con las credenciales para verificar que los datos introducidos en el login son válidos (en un primer momento esto no lo implementamos para facilitar las pruebas)

Después el driver manda un mensaje solicitando un suministro en un punto de carga X, y la central responderá dependiendo de si ese punto está disponible para recargar, en caso de que lo esté, empezará el suministro y entonces el driver podrá pararlo, mandándole otro mensaje en el que solicita detener el suministro.

3.6 Comunicación kafka (engine - Central)

En un principio no entendimos muy bien el funcionamiento del engine, no sabíamos que podía solicitar suministro y ponerse como fuera de servicio el propio punto.

En el momento que nos dimos cuenta, y lo implementamos, tuvimos que añadir la comunicación mediante kafka entre el engine y la central. Justo para los casos previamente mencionados, para solicitar un suministro y, en el caso que se este suministrando, solicitar pararlo y para ponerlo como fuera de servicio, o volver a activarlo si ya estaba en ese estado.

3.7 Separación de programas en 3 compose

decidimos que la manera más segura de poder desplegar el dia de la corrección seria hacerlo con 3 compose:

- compose servidor:
    - contenedor con broker de kafka
    - contenedor de python con:
        - EV_central.py:
          programa que gestiona las peticiones tanto de los puntos de carga como de app drivers
        - web.py:
          programa que genera un servidor web del monitor de central asi como algunos de sus controles
    - red interna entre los dos contenedores

- Compose punto de carga:
    - contenedor python ejecutando EV_monitor.py
    - contendor python ejecutando EV_engine.py
- Compose Drive:
    - contenedor python ejecutando EV_driver.py

junto con estos creamos archivos de script para ejecución rápida que funcionan con argumentos y hacen uso de variables de entorno para pasar los datos a los contenedores

hicimos uno por contenedor y uno para pruebas en una sola máquina que los ejecuta los 3 asimismo los script de driver y cp están preparados para ejecuciones de varios a la vez además abren las ventanas de terminal y las páginas correspondientes

Estos scripts compose y dockerfile han ido cambiando constantemente a lo largo de la práctica no lo vamos poniendo todo el rato para no alargar pero han dado un trabajo constante que ha consumido mucho tiempo.

3.8 Web de la Central

Una vez todo funcionaba, comenzamos con el desarrollo de una interfaz para la central, cogiendo como punto de partida la imagen que nos muestra la práctica, en la que podemos ver todos los puntos de carga (identificador, estado (si está suministrando se mostrarán datos en tiempo real del suministro), ubicación y precio del punto), las recargas en curso y los mensajes de la central (tanto errores como mensajes informativos).

Uno de los mayores problemas fue poder mostrar en la web la información del suministro en tiempo real, para que se pudiese ver cuando había consumido el usuario el precio que suponía ese consumo.

Esto lo solucionamos con web sockets (Flask-SocketIO), ya que en el servidor de la web (app.py) tenemos un hilo de fondo que genera actualizaciones periódicas, y el cliente (index.html) Está a la escucha para recibir y realizar esas actualizaciones

3.9 Web del Driver

Cuando ya conseguimos que la web de la central funcionase correctamente, y reflejase la base de datos en tiempo real. Comenzamos el desarrollo de la web del driver, en un primer momento hicimos una interfaz por consola, en la que primero metíamos el id del usuario y si existía ese usuario, metíamos la contraseña, y luego salía un menú con varias opciones, en las que podíamos: “solicitar suministro”, “parar suministro” y “consultar info. suministro”. Para hacer la web, en un primer momento, hicimos que siempre fuese el usuario 1, y no añadimos login, para facilitar las pruebas. Una vez ya comprobamos que en la web se podía solicitar la recarga en los puntos disponibles y ver los datos del suministro, ya añadimos el login y el register, para entrar/crear en la cuenta.

El funcionamiento de la web, se basa en:

Envío de mensaje por kafka con los datos del usuario a la central -> la central comprueba si son correctos, y si lo son -> manda un mensaje de confirmación de login.

Una vez dentro, la central manda, mediante kafka, todos los cp que hay en la base de datos y también las recargas anteriores de ese usuario.
Y en el caso de que ese driver esté realizando una recarga, aparecerán los datos en tiempo real de la recarga (precio y consumo).

3.10 cambio de comunicaciones de kafka

En este punto el profesor nos dio instrucciones de elegir el tópico dinámicamente. Como teníamos dos clases (consumidor y productor), elegir el topic a través simplemente de objeto no era posible, así que los aunamos en un objeto para driver y engine (`kafkaCommExt`) y otro para Central (`kafkaCommCentral`). Este último contiene tanto el consumidor como el productor y los métodos de selección de tópico, apoyándose en el objeto `TopicSelector` para generar los tópicos y asignarlos en Central.

La comunicación ahora funciona de la siguiente manera: el cliente pide un tópico por un "tópico de selección de zona" y la Central le responde por otro "tópico de zona", indicando el tópico específico y la partición por la que se comunicarán. Esta comunicación tiene un límite de 10 minutos, tras lo cual se vuelve a negociar el tópico.

Estructura de Tópicos por Zona Al implementar el sistema de zonas, hemos organizado los tópicos de la siguiente manera para cada zona geográfica:

- **Tópico de selección:** Para la solicitud inicial de canal.
- **Tópico de respuestas de selección:** Por donde la Central asigna el canal.
- **Tópico de status:** Para el estado general.
- **10 Tópicos numerales:** Con sus respectivas preguntas y respuestas, utilizados para distribuir la carga de las operaciones.

Handshake y Seguridad en la Comunicación Adicionalmente, para asegurar la robustez de la comunicación vía Kafka, hemos implementado un mecanismo de **handshake** tanto en el `Engine` como en el `Driver`. Esto nos permite verificar que la conexión bidireccional está establecida y operativa antes de comenzar el intercambio de mensajes críticos, evitando pérdidas de información al inicio de las sesiones.

Esta sección requirió muchas pruebas, pues al principio se intentó solo usar consumer y producer por separado, pero hubo que unificarlos. También hubo que rehacer todo el sistema añadiendo tipos a todos los mensajes de Kafka para asegurar que, diera igual por qué canal vinieran, pudieran ser reconocidos correctamente. Finalmente, toda la comunicación de un "usuario/driver" o "engine" se hace solamente por un topic designado una vez se negocia por `TopicSel_<zona>`

3.11 Separación de backend y Web en contenedores

Una vez que todo el sistema funcionaba correctamente, nos planteamos cómo demostrar la **tolerancia a fallos** ante caídas del sistema de cara a la evaluación.

El problema era que cada aplicación tenía tanto la lógica de negocio como la interfaz web en un mismo contenedor. Si queríamos demostrar que la aplicación se recupera ante fallos, al parar el contenedor se perdía también la web, dificultando la visualización del proceso de recuperación.

Decisión: Separar Central y Engine en **dos contenedores independientes** cada uno:
- **Contenedor Backend**: Solo lógica de negocio (Kafka, sockets, base de datos)
- **Contenedor Web**: Solo interfaz Flask (muestra datos, no gestiona comunicaciones)

Driver se mantuvo en un solo contenedor porque:
- Solo tiene ~500 líneas de código
- Imagen Docker muy ligera (~150MB)
- Web y lógica comparten memoria (sesiones, estado de recarga)
- Separarlo no aporta valor para demostrar tolerancia a fallos

Arquitectura resultante:

Central: 3 contenedores
- 'broker' (Kafka) - Puerto 9092
- 'central_backend' (Central.py) - Puerto 8020 (sockets + Kafka + BD JSON)
- 'central_web' (Flask) - Puerto 5000 (consulta backend vía API REST)

Charging Point: 3 contenedores por CP
- 'monitor' (Monitor.py) - Puerto 8010
- 'engine_backend' (Engine.py) - Kafka + sockets
- 'engine_web' (Flask) - Puerto 5050+ (lee estado desde '/tmp/engine_state_*.json' con 'fcntl' locking)

Driver: 1 contenedor monolítico (no separado)

Comunicación:
- Central: Web → Backend mediante API REST
- Engine: Backend ↔ Web mediante archivos JSON compartidos con file locking
- Ambos usan 'network_mode: service:monitor' para compartir namespace de red

Problema de sincronización de arranque:

Al separar Engine, apareció un error: 'ConnectionRefusedError' al intentar conectarse a Monitor.

Causa: El orquestador original tenía un delay de 12 segundos. Con Docker Compose, los contenedores arrancan en paralelo y Engine intentaba conectarse antes de que Monitor abriera el socket (necesita ~12s para registrarse con Registry y autenticarse con Central).

Solución: Creamos un wrapper script 'start_engine_with_delay.sh' que espera 15 segundos antes de arrancar Engine.py. Configurable con variable 'ENGINE_MONITOR_STARTUP_DELAY'.

Beneficios conseguidos:
- Poder apagar backend sin perder visibilidad de la web
- Poder apagar web sin afectar operaciones de Kafka/sockets
- Demostrar recuperación automática ante fallos
- Builds más rápidos (solo reconstruir el contenedor modificado)
- Logs separados y debugging más sencillo

3.12 API REST para comunicación Web-Backend

Una vez separados los contenedores de backend y web, necesitábamos una forma de comunicación entre ellos. Decidimos implementar una **API REST** en el backend de Central para que la web pudiera consultar datos sin necesidad de acceso directo a Kafka ni a la base de datos.

Endpoints implementados en Central Backend:

GET /api/charging_points - Obtener todos los CPs
GET /api/charging_points/<cp_id> - Obtener un CP específico
GET /api/supplies - Obtener suministros en curso
POST /api/charging_points - Crear nuevo CP
DELETE /api/charging_points/<cp_id> - Eliminar CP
PATCH /api/charging_points/<cp_id>/estado - Cambiar estado CP

Ventajas de usar REST:
- **Desacoplamiento total**: La web no necesita conocer Kafka ni sockets
- **Escalabilidad**: Múltiples webs pueden consultar el mismo backend
- **Estándar HTTP**: Fácil de debuguear con curl, Postman, navegador
- **Stateless**: Cada petición es independiente, más fácil de cachear

Implementación en la web:

```
# En web/app.py
CENTRAL_API_URL = os.getenv('CENTRAL_API_URL', 'http://localhost:8020')

@app.route('/')
```

```python
def index():
    # Consultar CPs desde backend vía REST
    response = requests.get(f'{CENTRAL_API_URL}/api/charging_points')
    cps = response.json()

    # Consultar suministros en curso
    response = requests.get(f'{CENTRAL_API_URL}/api/supplies')
    supplies = response.json()

    return render_template('index.html', cps=cps, supplies=supplies)
```

Autenticación y seguridad:
Por simplicidad y dado que es un entorno de desarrollo/educativo, no implementamos autenticación. En un entorno de producción añadiríamos:
- JWT tokens para autenticación
- HTTPS para cifrado de comunicaciones
- Rate limiting para prevenir abuso
- CORS configurado correctamente

3.13 Sistema de Control Climático (EV_Weather)

Hemos implementado un servicio autónomo de monitoreo climático, **EV_Weather**, que consulta la API de OpenWeather y pone automáticamente los puntos de carga (CPs) fuera de servicio si se detectan temperaturas extremas.

Arquitectura Implementada

El sistema funciona mediante un ciclo de monitoreo continuo:

1. **Sincronización (cada 60s):** EV_Weather consulta a la Central vía API REST para conocer la ubicación actualizada de todos los CPs.
2. **Monitoreo (cada 4s):** Consulta la temperatura de cada ubicación a la API externa de OpenWeather.
3. **Actualización:** Si se detectan condiciones adversas, notifica a la Central, la cual actualiza el estado en la base de datos PostgreSQL (`weather` y `charging`).

Implementación Técnica

A) Carga de Configuración y API Key La carga de la clave de API es flexible, permitiendo el uso de un archivo `api_key.txt`, variables de entorno (`OPENWEATHER_API_KEY`) o un archivo `.env`. El sistema prioriza las variables de entorno para facilitar el despliegue en contenedores.

B) Sincronización Automática con Central Cada 60 segundos, el método `_sync_locations_from_central` realiza una petición GET al endpoint `/api/cp_locations` de la Central. Esto permite al módulo detectar nuevos CPs añadidos, CPs eliminados o cambios de ciudad en tiempo real sin necesidad de reinicios.

C) Monitoreo de Temperatura Cada 4 segundos, se consulta la API de OpenWeather (`api.openweathermap.org/data/2.5/weather`) para obtener la temperatura en grados Celsius, sensación térmica, humedad y descripción del clima.

D) Sistema de Alertas Automáticas Hemos implementado una lógica de umbrales configurables (por defecto 0°C - 40°C).

- **Alerta Activa:** Si la temperatura sale del rango, se envía una notificación `ALERT` a la Central.
- **Normalización:** Si la temperatura vuelve al rango seguro y existía una alerta previa, se envía una notificación `CLEAR`.

E) Procesamiento en Central La Central recibe estas notificaciones en el endpoint `/api/weather_alert`:

- Si recibe **ALERT**: Pone el CP en estado "FUERA DE SERVICIO" automáticamente. Si el CP está suministrando ("SUMINISTRANDO"), **no interrumpe la recarga**, sino que marca el CP para ser desactivado al finalizar.
- Si recibe **CLEAR**: Reactiva automáticamente el CP a estado "ACTIVADO", permitiendo nuevamente su uso por los conductores.

3.14 Sistema de Registry para descubrimiento de servicios

El **EV_Registry** es el servicio centralizado encargado del alta, baja y autenticación de los puntos de carga (CPs). Actúa como el "portero" del sistema, controlando qué CPs tienen permiso para operar en la red y garantizando la seguridad mediante credenciales cifradas.

Arquitectura y Seguridad

El Registry expone una API HTTPS en el puerto 5002 y gestiona dos tablas clave en PostgreSQL: `cp_credentials` (usuarios y contraseñas) y `charging_points` (metadatos).

Generación de Credenciales Seguras Hemos implementado un sistema robusto de generación de credenciales en `EV_Registry/app.py`:

- **Contraseñas aleatorias:** Se generan 32 bytes seguros usando `secrets.token_urlsafe`.
- **Hashing:** Se utiliza SHA-256 para la validación rápida de contraseñas.
- **Cifrado Reversible:** Se utiliza cifrado Fernet (AES-128) para almacenar la contraseña de forma recuperable (necesario para regeneración), protegida por una clave maestra derivada de las variables de entorno.

La contraseña en texto plano solo se devuelve una única vez durante el registro; posteriormente, el CP debe almacenarla localmente.

Funcionalidad Implementada

A) Registro de Puntos de Carga Mediante el endpoint `POST /api/register`, el sistema:

1. Verifica si el CP ya existe.
2. Genera credenciales únicas.
3. Inserta el CP en la base de datos con estado "DESCONECTADO".
4. Registra la ubicación para el monitoreo climático.
5. Devuelve las credenciales al administrador.

B) Baja de Puntos de Carga El endpoint `DELETE /api/unregister` permite dar de baja un CP, validando sus credenciales, revocando sus claves de cifrado y marcándolo como inactivo.

C) Autenticación de CPs Cuando un CP (Monitor o Engine) inicia, se autentica contra la Central mediante `POST /api/authenticate`. La Central consulta internamente al Registry/Base de datos para verificar las credenciales. Si es exitoso, devuelve una **clave de cifrado simétrico** (AES-256) única para ese CP, que se utilizará para cifrar todas las comunicaciones subsiguientes.

3.15 Migración a PostgreSQL para acceso distribuido

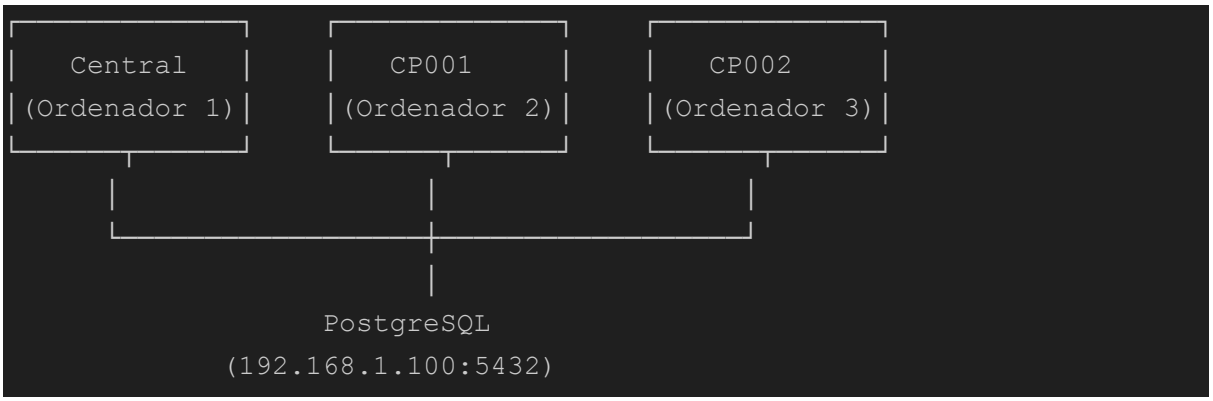
Inicialmente usamos un fichero JSON como base de datos, pero al necesitar que múltiples ordenadores accedan a los mismos datos, **migramos a PostgreSQL**.

Problema con JSON

El fichero JSON (‘bd.json’) solo funciona en un único ordenador:
- No accesible desde otros ordenadores en la red
- File locking solo funciona en el mismo sistema de archivos
- Difícil sincronizar cambios entre máquinas
- No apto para despliegue distribuido

Solución: PostgreSQL en Docker

Implementamos una base de datos PostgreSQL centralizada accesible desde cualquier ordenador de la red.



Nuevas tablas creadas:

1. ‘charging_points’: Puntos de carga

```
CREATE TABLE charging_points (
    cp_id VARCHAR(20) PRIMARY KEY,
    estado VARCHAR(20),
    ubicacion VARCHAR(100),
    precio DECIMAL(5,2),
    potencia INTEGER,
    city VARCHAR(50),
    country VARCHAR(2)
);
```

2. ‘supplies’: Suministros/recargas

```
CREATE TABLE supplies (
    supply_id SERIAL PRIMARY KEY,
```

```
    cp_id VARCHAR(20),
    driver_id VARCHAR(20),
    inicio TIMESTAMP,
    fin TIMESTAMP,
    consumo DECIMAL(10,2),
    precio_total DECIMAL(10,2),
    estado VARCHAR(20)
);
```

3. weather_history: Histórico climático

```
CREATE TABLE weather_history (
    id SERIAL PRIMARY KEY,
    cp_id VARCHAR(20),
    city VARCHAR(50),
    temperature DECIMAL(5,2),
    humidity INTEGER,
    conditions VARCHAR(50),
    wind_speed DECIMAL(5,2),
    timestamp TIMESTAMP
);
```

4. 'weather_alerts': Alertas climáticas

```
CREATE TABLE weather_alerts (
    id SERIAL PRIMARY KEY,
    cp_id VARCHAR(20),
    alert_type VARCHAR(30),
    temperature DECIMAL(5,2),
    description TEXT,
    timestamp TIMESTAMP
);
```

5. 'audit_log': Log de auditoría

```
CREATE TABLE audit_log (
    id SERIAL PRIMARY KEY,
    action VARCHAR(50),
    cp_id VARCHAR(20),
    details TEXT,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

6. 'drivers': Usuarios del sistema

```
CREATE TABLE drivers (
   driver_id VARCHAR(20) PRIMARY KEY,
   nombre VARCHAR(100),
   password_hash VARCHAR(255),
   email VARCHAR(100),
   saldo DECIMAL(10,2)
);
```

Configuración conexión

Docker Compose (Central):

```yaml
services:
 postgres:
   image: postgres:15
   container_name: ev_postgres
   environment:
     POSTGRES_DB: ev_charging
     POSTGRES_USER: ev_user
     POSTGRES_PASSWORD: ev_password
   ports:
     - "5432:5432"
   volumes:
     - postgres_data:/var/lib/postgresql/data
   networks:
     - ev_network

volumes:
 postgres_data:
```

3.15 Sistema de Auditoría Centralizada con Logging

Para cumplir con requisitos de trazabilidad, seguridad y diagnóstico, implementamos un sistema de auditoría completo basado en el módulo logging de Python que registra todas las operaciones críticas del sistema.

Problema que resuelve:

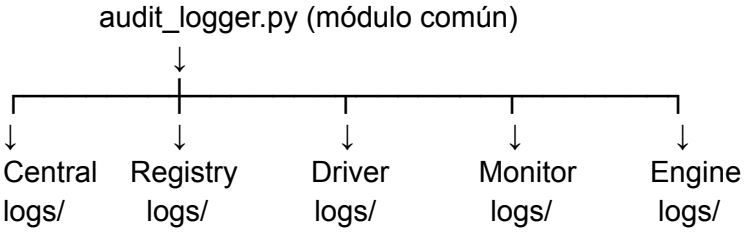
Antes, no había un registro centralizado de eventos:

- Difícil diagnosticar problemas en producción
- Sin trazabilidad de accesos y operaciones
- Imposible auditar actividad de usuarios
- No se detectaban intentos de acceso no autorizados

Después:

- Todos los eventos se registran en formato JSON estructurado
- Logs persistentes en archivos por componente
- Diferenciación por nivel de importancia
- Auditoría completa de autenticación, recargas e incidentes

Arquitectura:



Funcionamiento:

1. Inicialización por componente:

```
import audit_logger as audit
audit.get_audit_logger(log_dir="/app/logs",
component_name="driver")
```

2. Registro de eventos:

```
# Autenticación
audit.audit_auth(user_id="USR001", action="LOGIN",
                success=True, source_ip="192.168.1.50")

# Recargas
audit.audit_charge(cp_id="CP001", user_id="USR001",
                  action="CHARGE_START", success=True)

# Incidentes
audit.audit_incident(incident_type="CENTRAL_DISCONNECTED",
                    description="Sin heartbeat por 15s")

# Errores
audit.audit_error(error_type="CONNECTION_ERROR",
                 error_message="No se pudo conectar con
Central")
```

3. Formato de logs (JSON):

```
{
 "timestamp": "2025-12-13T22:48:16.280515",
 "date": "2025-12-13",
 "time": "22:48:16",
 "category": "AUTH",
 "action": "LOGIN",
 "source_entity": "USR001",
 "source_ip": "192.168.1.50",
 "success": true,
 "parameters": {"user_id": "USR001"},
 "description": "Login exitoso desde Driver"
}
```

Características:

• **Categorización de eventos:**

- AUTH: Autenticación (login, logout, fallos)
- CHARGE: Operaciones de recarga (solicitud, inicio, parada)
- INCIDENT: Incidentes del sistema (desconexiones, timeouts)
- STATUS: Cambios de estado (CP online/offline)
- ERROR: Errores técnicos (conexión, validación)
- SECURITY: Eventos de seguridad (bloqueos, intentos sospechosos)
- SYSTEM: Eventos del sistema (inicio, parada)

• **Diferenciación por nivel:**

- Archivo: DEBUG+ (todo se guarda)
- Consola: WARNING+ (solo eventos importantes)

• **Persistencia:**

- Volúmenes Docker: -v ./logs:/app/logs
- Modo APPEND: logs acumulativos entre reinicios
- Opción --clear-logs en scripts de parada

• **Integración completa:**

```
# Ver logs en tiempo real
tail -f EV_Driver/logs/audit_driver.log

# Buscar eventos específicos
grep "LOGIN" EV_Driver/logs/audit_*.log
```

```
# Analizar errores
cat EV_Central_Server/logs/audit_*.log | grep '"category":
"ERROR"'

# Limpiar logs al detener
./stop.sh --clear-logs
```

Ventajas:

- Trazabilidad completa: Cada operación queda registrada
- Diagnóstico eficiente: Logs estructurados facilitan debugging
- Seguridad mejorada: Detección de accesos no autorizados
- Cumplimiento normativo: Auditoría requerida por estándares
- Análisis forense: Investigación de incidentes post-mortem
- Monitorización: Integrable con herramientas de análisis (ELK, Splunk)

3.16 Integración Completa del Sistema

Para finalizar la arquitectura, hemos logrado una integración completa donde todos los componentes interactúan de forma automatizada y segura.

Flujo de Trabajo:

1. Un script de administración registra el CP en **EV_Registry** (vía HTTPS), generando credenciales seguras que se guardan en PostgreSQL.
2. Al arrancar, el CP se autentica con **Central**, obteniendo su clave de encriptación.
3. Simultáneamente, **EV_Weather** sincroniza las ubicaciones desde la Central y monitorea el clima.
4. Si **EV_Weather** detecta temperaturas extremas, instruye a la Central para desactivar el CP. Si la temperatura se normaliza, ordena su reactivación.

Ventajas del Diseño:

- **Seguridad:** Uso de HTTPS, cifrado de credenciales (Hash + Fernet) y claves de sesión simétricas.
- **Automatización:** El sistema climático actúa sin intervención humana, protegiendo la infraestructura y reactivándola cuando es seguro.
- **Escalabilidad:** El Registry utiliza un pool de conexiones y el Weather realiza llamadas API eficientes, permitiendo gestionar múltiples CPs sin saturar el sistema.
- **Trazabilidad:** Todas las operaciones críticas (registro, autenticación, alertas climáticas) quedan registradas en el sistema de auditoría.

4. Resumen cambios release 2

Trabajo realizado (secciones 3.11-3.16)

Tiempo de desarrollo: Aproximadamente 3 semanas adicionales después del sistema básico

Código añadido:

- 500 líneas: Dockerfiles y docker-compose
- 800 líneas: API REST (endpoints más validación)
- 600 líneas: EV_Weather (monitoreo más alertas)
- 300 líneas: EV_Registry (registro más lookup)
- 400 líneas: Migración y extensiones de Base de Datos PostgreSQL
- 475 líneas: Sistema de auditoría (audit_logger.py × 5 componentes)
- 300 líneas: Scripts de despliegue actualizados
- 200 líneas: Tests automatizados
- TOTAL: Aproximadamente 3575 líneas de código nuevas

Archivos creados o modificados:

- 8 Dockerfiles nuevos (backend y web separados)
- 2 docker-compose.yml actualizados con nueva arquitectura
- 5 módulos audit_logger.py (uno por componente)
- 4 documentos README (SEPARATED, ARCHITECTURE, WEATHER, AUDITORIA_VERIFICACION)
- 6 scripts de test automatizados (incluye test_audit_module.py)
- 1 wrapper script (start_engine_with_delay.sh)
- Scripts SQL de inicialización de PostgreSQL
- Script de migración JSON a PostgreSQL
- Plantilla de error de conexión (error_connection.html)

Tecnologías incorporadas:

- Docker multi-stage builds para optimización de imágenes
- API REST con Flask para comunicación entre servicios
- Integración con OpenWeatherMap API para datos climáticos reales
- Patrón Service Discovery para descubrimiento de servicios
- PostgreSQL como base de datos distribuida
- Transacciones ACID para concurrencia segura
- Sistema de logging con módulo Python logging
- Formato JSON estructurado para logs
- Persistencia de logs con volúmenes Docker
- Health checks y heartbeats para monitorización
- Wrapper scripts para coordinación de arranque
- Variables de entorno para configuración flexible

Funcionalidades implementadas:

1. Separación Backend-Web

    - Central y Engine divididos en contenedores independientes
    - Permite demostrar tolerancia a fallos en tiempo real
    - Posibilidad de apagar backend sin perder visibilidad web
    - Posibilidad de apagar web sin afectar operaciones
2. API REST

    - 10 endpoints para comunicación Central backend-web
    - Desacoplamiento total entre servicios
    - Escalabilidad: múltiples webs pueden consultar mismo backend
    - Fácil debugging con herramientas estándar HTTP
3. Sistema Climático (EV_Weather)

    - Monitoreo automático de temperatura cada 4 segundos
    - Sincronización con Central cada 60 segundos
    - Alertas automáticas por temperaturas extremas
    - Desactivación preventiva de CPs ante condiciones peligrosas
    - Reactivación automática cuando temperatura normaliza
4. Service Registry

    - Directorio centralizado de servicios
    - Descubrimiento automático sin IPs hardcodeadas
    - Heartbeats para detección de caídas
    - Facilita despliegue en múltiples máquinas
5. Base de Datos PostgreSQL

    - Migración de JSON a PostgreSQL
    - Acceso distribuido desde múltiples ordenadores
    - 6 tablas: charging_points, supplies, weather_history, weather_alerts, audit_log, drivers
    - Transacciones ACID para concurrencia real
    - Consultas SQL complejas
    - Backups robustos con pg_dump
6. Sistema de Auditoría Completo

    - Registro de todos los eventos críticos en formato JSON
    - 7 categorías: AUTH, CHARGE, INCIDENT, STATUS, ERROR, SECURITY, SYSTEM
    - Diferenciación por niveles (DEBUG→archivo, WARNING→consola)
    - Logs persistentes por componente (Central, Registry, Driver, Monitor, Engine)
    - Trazabilidad completa: timestamp, usuario, IP, acción, parámetros
    - Opción --clear-logs en scripts de parada
    - Verificación de conexión a Central con bloqueo automático

Beneficios para la demostración:

Tolerancia a fallos: Se puede mostrar en vivo cómo el sistema se recupera ante caídas de componentes específicos (backend, web, weather) sin perder funcionalidad crítica

Sistema realista: Datos climáticos reales de OpenWeatherMap que afectan automáticamente la disponibilidad de los puntos de carga, simulando condiciones reales de operación

Arquitectura moderna: Implementación de patrones de microservicios con API REST, service discovery y base de datos centralizada distribuida

Automatización completa: Scripts de despliegue, health checks automáticos, sincronización automática de servicios, sin intervención manual necesaria

Trazabilidad total: Sistema completo de logs de auditoría que registra todas las acciones importantes (autenticación, recargas, cambios de estado, alertas climáticas, errores) en formato JSON estructurado para debugging, análisis y cumplimiento normativo

Seguridad mejorada: Verificación automática de conexión a Central antes de permitir operaciones, bloqueo preventivo ante desconexiones, registro de todos los intentos de acceso

Despliegue distribuido: Capacidad demostrada de ejecutar componentes en diferentes ordenadores de la red, todos conectándose a la misma base de datos PostgreSQL

Diagnóstico eficiente: Logs estructurados en JSON facilitan análisis automatizado, búsqueda de eventos específicos y troubleshooting en tiempo real

Los cambios principales respecto a la versión anterior son:

- Se añadió la línea de audit_logger (475 líneas × 5)
- Total actualizado: 3100 → 3575 líneas
- Se añadió el punto 6 "Sistema de Auditoría Completo"
- Se actualizaron los beneficios para incluir Trazabilidad, Seguridad y Diagnóstico
- Se añadieron las tecnologías de logging y JSON
- Se añadieron los archivos de auditoría en la lista de archivos creados

5. Problemas comunes durante el desarrollo

5.1 Actualización de datos en la web

Este tema ha sido uno de los mayores quebraderos de cabeza en esta práctica. Hacer que se actualizasen los datos automáticamente en la web, sin tener que recargarla manualmente nos ha dado muchos problemas. También hemos tenido problemas, por

ejemplo, a la hora de mostrar los tickets. Ya que se muestran en el momento que el usuario le da a 'detener suministro', pero si la web se recarga, desaparece el ticket. Y problemas como éste nos han tenido bastante ocupados en el último tramo del desarrollo de la práctica

Conexión desde Python:

```
import psycopg2

conn = psycopg2.connect(
   host="192.168.1.100",  # IP del servidor con PostgreSQL
   port=5432,
   database="ev_charging",
   user="ev_user",
   password="ev_password"
)
```

Funciones actualizadas en db_utils.py

```
# Gestión de CPs
def get_all_charging_points()
def update_cp_state(cp_id, estado)
def add_charging_point(cp_id, ubicacion, precio, city)

# Gestión de suministros
def create_supply(cp_id, driver_id)
def update_supply(supply_id, consumo, precio_total)
def get_active_supplies()

# Gestión de ubicaciones
def set_cp_location(cp_id, city, country)
def get_all_cp_locations()

# Gestión climática
def register_weather_history(cp_id, city, temperature, ...)
def get_latest_weather_for_all_cps()
def register_weather_alert(cp_id, alert_type, ...)

# Auditoría
def audit_log(action, cp_id, details)
def get_audit_log(limit=100)
```

Ventajas de PostgreSQL

- **Acceso distribuido**: Múltiples ordenadores pueden conectarse simultáneamente

- **Concurrencia real**: PostgreSQL maneja transacciones ACID automáticamente
- **Consultas SQL**: Búsquedas y filtros complejos fáciles
- **Escalabilidad**: Preparado para miles de registros
- **Integridad referencial**: Foreign keys y constraints
- **Backups robustos**: ‘pg_dump' para exportar toda la BD
- **Índices**: Búsquedas rápidas en tablas grandes

```
Despliegue distribuido

1. Arrancar PostgreSQL (Ordenador servidor):
cd EV_Central_Server
docker compose up -d postgres

2. Configurar variable de entorno (otros ordenadores):
export
DATABASE_URL="postgresql://ev_user:ev_password@192.168.1.100:5432/ev_ch
arging"

3. Arrancar aplicaciones:
# En ordenador 1
./start.sh  # Central

# En ordenador 2
cd EV_CP
./start.sh  # CP001

# En ordenador 3
cd EV_Driver
./start.sh  # Driver

Todos se conectan a la misma base de datos PostgreSQL centralizada.
```

5.2 Errores en los estados del cp

Otro de los problemas más comunes durante el desarrollo de la práctica ha sido que al cambiar el estado de un punto de carga desde una de las aplicaciones, por ejemplo un driver solicita un suministro. Se cargase ese estado en las demás aplicaciones, y así no se pueda solicitar el suministro desde el engine, o que la central sepa que está suministrando.

5.3 Errores con los topics de Kafka

Muchas veces no llegaban los mensajes de kafka donde queríamos o llegaba pero no se leían como esperábamos. La mayoría de las veces esto se debía a que al cambiar el funcionamiento de mensajería se nos olvidaba cambiar el topic que se usaba para ese caso, o algo tan simple como no escribir el mismo topic en el servidor que en el cliente.

5.4 Sincronización de arranque entre Engine y Monitor

Problema:
Al separar Engine en contenedores backend y web, apareció un error crítico: ConnectionRefusedError: [Errno 111] Connection refused al intentar conectarse a Monitor.

Causa:
Con Docker Compose, los contenedores arrancan en paralelo. Engine intentaba conectarse antes de que Monitor completara su secuencia de inicio:
• Registro con Registry (~5s)
• Autenticación con Central (~10s)
• Apertura del socket en puerto 8010 (~12s)

Antes, el orquestador run_engine_orchestrator.py tenía un delay de 12 segundos que coordinaba este arranque secuencial.

Solución implementada:
Creamos un wrapper script start_engine_with_delay.sh que introduce una espera configurable antes de arrancar Engine:

5.5 Comunicación entre Engine Backend y Engine Web

Problema:
Después de separar Engine en dos contenedores, necesitábamos compartir el estado en tiempo real para:
• Mostrar datos de suministro actualizados cada segundo
• Reflejar cambios de estado inmediatamente
• Mantener sincronización entre backend (que procesa Kafka) y web (que muestra datos)

Los contenedores comparten namespace de red (network_mode: service:monitor) pero NO comparten memoria.

Intentos fallidos:

1. Sockets internos entre contenedores:
   • Problema: Añade complejidad innecesaria
   • Requiere gestión de conexiones y reconexiones
   • Similar problema a la comunicación Engine-Monitor original

2. Variables de entorno:
   • Problema: Solo se leen al arrancar, no actualizan en tiempo real
   • No sirve para datos dinámicos como consumo actual

3. API REST interna:
   • Problema: Overhead de HTTP para actualizaciones cada segundo
   • Requiere servidor Flask en el backend (duplica responsabilidades)

Solución implementada:
Archivos de estado compartidos con file locking (fcntl)

5.6 Migración de JSON a PostgreSQL

Problema:
El fichero JSON (bd.json) funcionaba bien en desarrollo local, pero al intentar desplegar en múltiples ordenadores aparecieron problemas críticos:

1. No accesible desde otros ordenadores:
   • JSON es un archivo local
   • Cada máquina tendría su propia copia
   • Imposible mantener datos sincronizados

2. File locking limitado:
   • fcntl solo funciona en el mismo sistema de archivos
   • No sirve para concurrencia entre máquinas

3. Sincronización manual:
   • Habría que copiar el archivo entre ordenadores
   • Pérdida de datos al haber múltiples "verdades"

Proceso de migración:

1. Diseño del esquema SQL:
   • Convertir estructura JSON anidada a tablas relacionales
   • Definir primary keys y foreign keys
   • Añadir índices para búsquedas frecuentes

2. Creación de tablas
3. Script de migración.
4. Refactorización de db_utils.py.

Problemas durante la migración:

1. Tipos de datos inconsistentes:
   • JSON permitía valores nulos implícitos

- SQL requiere tipos explícitos
- Solución: Valores por defecto y validación

2. Estructura anidada:
- JSON tenía objetos dentro de objetos
- SQL requiere normalización en tablas separadas
- Solución: Tablas relacionadas con foreign keys

3. Timestamps:
- JSON usaba strings para fechas
- PostgreSQL requiere tipo TIMESTAMP
- Solución: Conversión con datetime.fromisoformat()

4. Concurrencia:
- JSON requería locks manuales
- PostgreSQL maneja esto automáticamente con transacciones
- Beneficio: Código más simple

Resultado:
• Sistema completamente funcional en entorno distribuido
• Múltiples CPs y Drivers en diferentes ordenadores
• Todos conectados a la misma BD PostgreSQL centralizada
• Datos consistentes y actualizados en tiempo real

5.7 Coordinación con EV_Weather y temperaturas extremas

Problema:
EV_Weather consulta OpenWeatherMap y envía alertas a Central cuando detecta temperaturas extremas. Sin embargo, surgieron problemas de coordinación:

1. CP en medio de un suministro:
- Weather detecta 45°C y quiere desactivar CP
- Pero hay un usuario cargando su vehículo
- ¿Interrumpir la recarga o esperar?

2. Race condition con estados:
- Weather dice: "Desactivar CP001"
- Simultáneamente Driver solicita: "Iniciar suministro en CP001"
- ¿Qué acción tiene prioridad?

3. Recuperación tras normalización:
- Temperatura vuelve a rango seguro (15°C)
- ¿Reactivar automáticamente o esperar confirmación manual?

Solución implementada:

En web/app.py (endpoint /api/weather_alert):

```
@app.route("/api/weather_alert", methods=["POST"])
def api_weather_alert():
    data = request.json
    cp_id = data['cp_id']
    temperature = data['temperature']
    alert_type = data['alert_type']

    # Obtener estado actual del CP
    cp = dbu.get_charging_point(cp_id)

    if alert_type in ['CRITICAL_TEMP', 'HIGH_TEMP', 'LOW_TEMP']:
        # Temperatura fuera de rango

        if cp['estado'] == 'SUMINISTRANDO':
            # NO interrumpir suministro en curso
            print(f"[Weather] CP {cp_id} suministrando. "
                  f"Se marcará para desactivar al finalizar.")
            # Marcar flag en BD
            dbu.mark_cp_for_deactivation(cp_id, reason='WEATHER')

        elif cp['estado'] == 'ACTIVADO':
            # Desactivar inmediatamente
            dbu.update_cp_state(cp_id, 'FUERA_DE_SERVICIO')
            print(f"[Weather] ⚠ CP {cp_id} DESACTIVADO por temp: {temperature}°C")

        # Registrar alerta en BD
        dbu.register_weather_alert(cp_id, alert_type, temperature, ...)

    elif alert_type == 'TEMP_NORMALIZED':
        # Temperatura vuelve a rango seguro

        if cp['estado'] == 'FUERA_DE_SERVICIO':
            # Verificar que no hay otras razones de desactivación
            if not dbu.has_other_alerts(cp_id):
                # Reactivar automáticamente
                dbu.update_cp_state(cp_id, 'ACTIVADO')
                print(f"[Weather] ✓ CP {cp_id} REACTIVADO - Temp normalizada:
{temperature}°C")

    return jsonify({"status": "ok"})
```

Lógica en Central al finalizar suministro:

```
def finish_supply(cp_id, supply_id):
    # Finalizar suministro
```

```
    dbu.update_supply(supply_id, estado='FINALIZADO')

    # Verificar si hay flag de desactivación pendiente
    if dbu.is_marked_for_deactivation(cp_id):
        reason = dbu.get_deactivation_reason(cp_id)
        dbu.update_cp_state(cp_id, 'FUERA_DE_SERVICIO')
        print(f"[Central] CP {cp_id} desactivado tras finalizar suministro. Razón: {reason}")
        dbu.clear_deactivation_flag(cp_id)
    else:
        # Volver a estado normal
        dbu.update_cp_state(cp_id, 'ACTIVADO')
```

Ventajas de este enfoque:
• No interrumpe suministros en curso (mejor UX)
• Evita race conditions con flags en BD
• Reactivación automática cuando es seguro
• Trazabilidad completa en audit_log

5.8 Gestión de conexiones PostgreSQL

Problema:
Al principio, cada función de db_utils.py abría y cerraba su propia conexión a PostgreSQL:

```
def get_charging_point(cp_id):
    conn = psycopg2.connect(...)  # Nueva conexión
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM charging_points WHERE cp_id = %s", (cp_id,))
    result = cursor.fetchone()
    conn.close()  # Cerrar conexión
    return result
```

Problemas:
• Overhead de crear conexión (~50ms cada vez)
• Límite de conexiones simultáneas (default: 100 en PostgreSQL)
• Errors "too many connections" con múltiples CPs activos
• Mal rendimiento en operaciones frecuentes

Solución: Connection Pool

Implementamos un pool de conexiones con psycopg2.pool:

db_utils.py
```
from psycopg2 import pool
import os

# Crear pool al importar el módulo
```

```
connection_pool = pool.SimpleConnectionPool(
    minconn=1,      # Mínimo 1 conexión siempre abierta
    maxconn=20,     # Máximo 20 conexiones simultáneas
    host=os.getenv('DB_HOST', 'localhost'),
    port=os.getenv('DB_PORT', '5432'),
    database=os.getenv('DB_NAME', 'ev_charging'),
    user=os.getenv('DB_USER', 'ev_user'),
    password=os.getenv('DB_PASSWORD', 'ev_password')
)

def get_charging_point(cp_id):
    # Obtener conexión del pool (rápido)
    conn = connection_pool.getconn()
    try:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM charging_points WHERE cp_id = %s", (cp_id,))
        result = cursor.fetchone()
        cursor.close()
        return result
    finally:
        # Devolver conexión al pool (no cerrarla)
        connection_pool.putconn(conn)
```

Mejoras obtenidas:
• Reducción de latencia: 50ms → 2ms por operación
• Sin errores "too many connections"
• Reutilización eficiente de recursos
• Mejor rendimiento con múltiples CPs y Drivers

5.9 Autodetección de IP en scripts multiplataforma

Problema:
Los scripts de despliegue requieren la IP pública del servidor para configurar Kafka y las comunicaciones. Hardcodear esta IP o pedirla manualmente en cada ejecución resultaba tedioso y propenso a errores, especialmente al cambiar de red (universidad, casa, presentaciones).

Casos problemáticos:

1. Cambio de red frecuente:

   IP diferente en cada ubicación
   Olvidar actualizar la IP en variables de entorno
   Errores de conexión difíciles de diagnosticar

2. Múltiples interfaces de red:

Laptops con WiFi + Ethernet + VPN
Docker con interfaces virtuales (docker0, br-*)
¿Cuál es la IP correcta para comunicación externa?

3. Diferencias entre sistemas operativos:

Linux: hostname -I, ip addr show
Windows: ipconfig, comandos PowerShell
macOS: ifconfig en0

Solución en Linux (start.sh):

```
# Detectar IP automáticamente del adaptador con gateway predeterminado
PUBLIC_IP=$(ip route get 1.1.1.1 | awk '{print $7; exit}')

if [ -z "$PUBLIC_IP" ]; then
    echo "ERROR: No se pudo detectar la IP automáticamente"
    echo "Por favor, proporciona la IP manualmente: $0 <PUBLIC_IP> [COMPONENTES]"
    exit 1
fi

echo "IP detectada: $PUBLIC_IP"
```

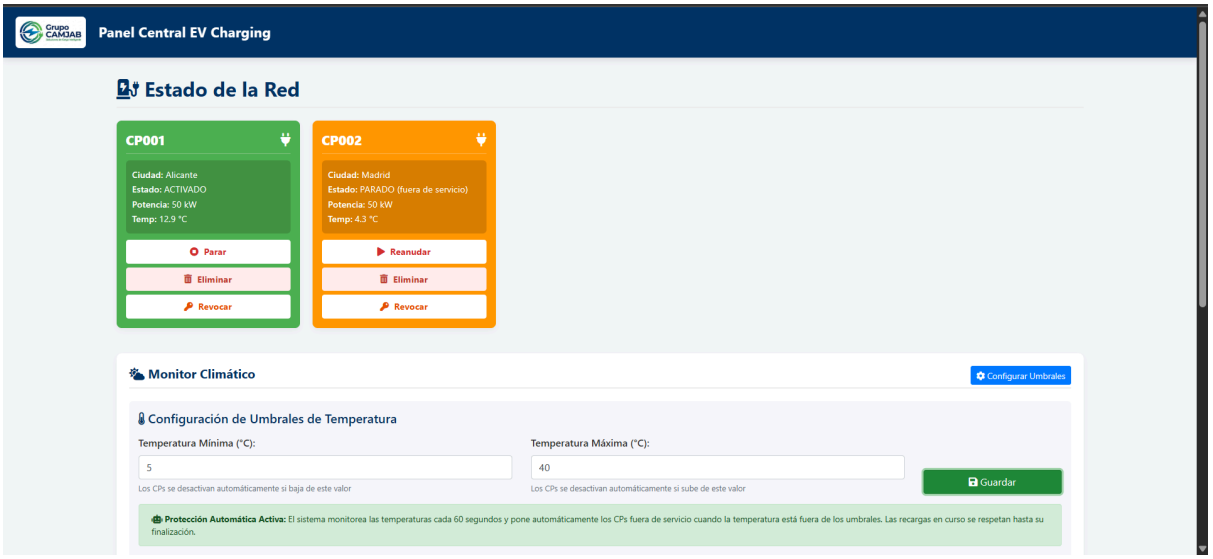
Get-NetIPConfiguration: Obtiene configuraciones de red
Where-Object {$_.IPv4DefaultGateway -ne $null}: Filtra interfaces con gateway (conexión a Internet)
Select-Object -First 1: Toma la primera interfaz válida
ExpandProperty IPv4Address: Extrae solo la dirección IP

Ventajas conseguidas:

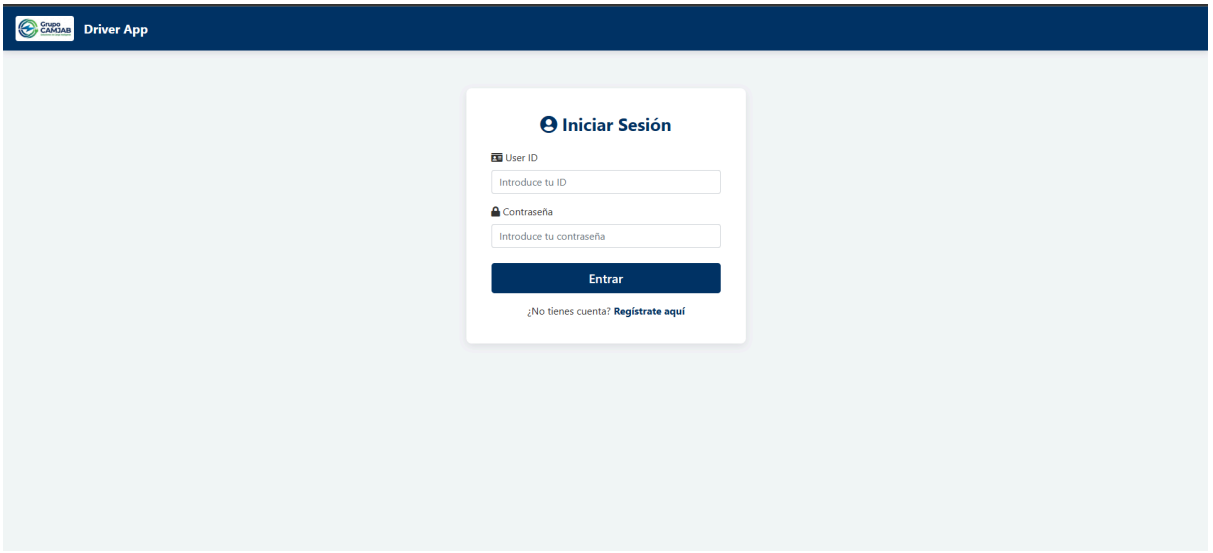
Zero-config: start.sh arranca todo sin preguntar nada
Portabilidad: Funciona en cualquier red sin modificar código
Flexibilidad: Permite override manual si es necesario
Robustez: Fallback a entrada manual si autodetección falla
Debugging sencillo: Se muestra la IP detectada en consola

6. Ejemplos de funcionamiento

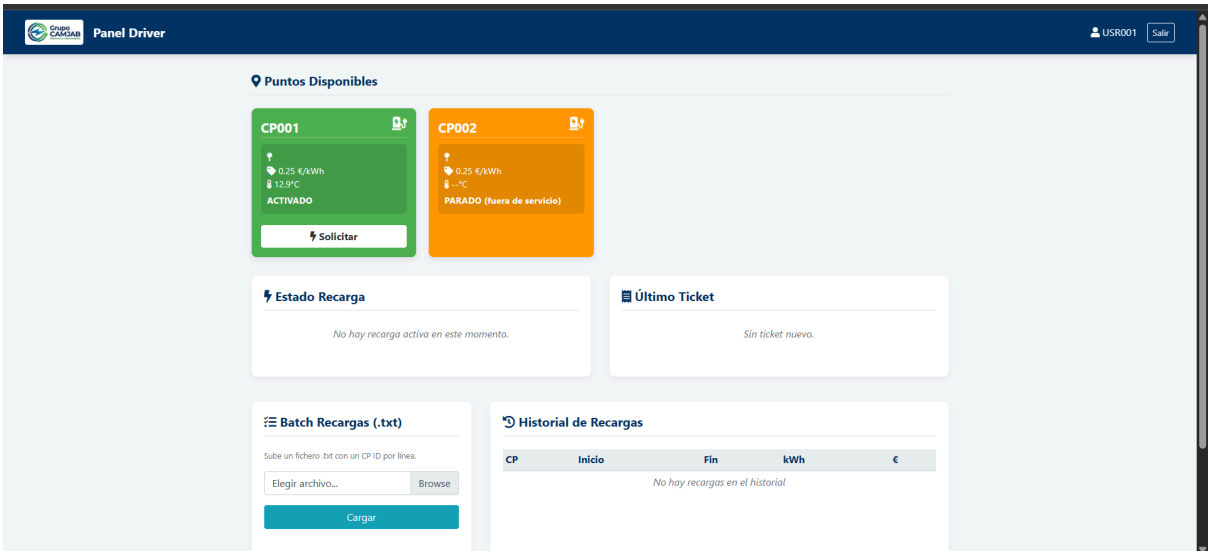
Esta es la interfaz de la central, en la que podemos ver todos los cp que existen y el estado en el que están, una lista de las recargas en curso actuales y mensajes de la aplicación (errores, añadir/quitar cp…etc). Desde esta aplicación, podemos tanto añadir como eliminar puntos de recarga, como poner un punto de recarga como ‘Fuera de servicio’



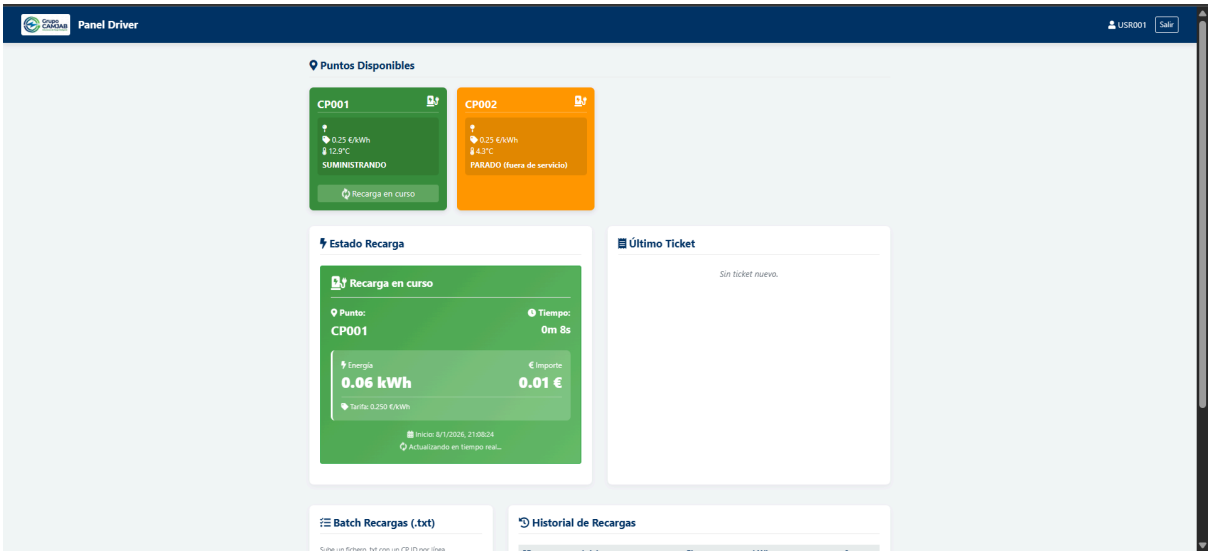
Esta es la interfaz de la aplicación del driver (la parte del login), donde se introducen las credenciales del usuario, y tras la comprobación de la central. Se loguea y entras a la aplicación con tu usuario.



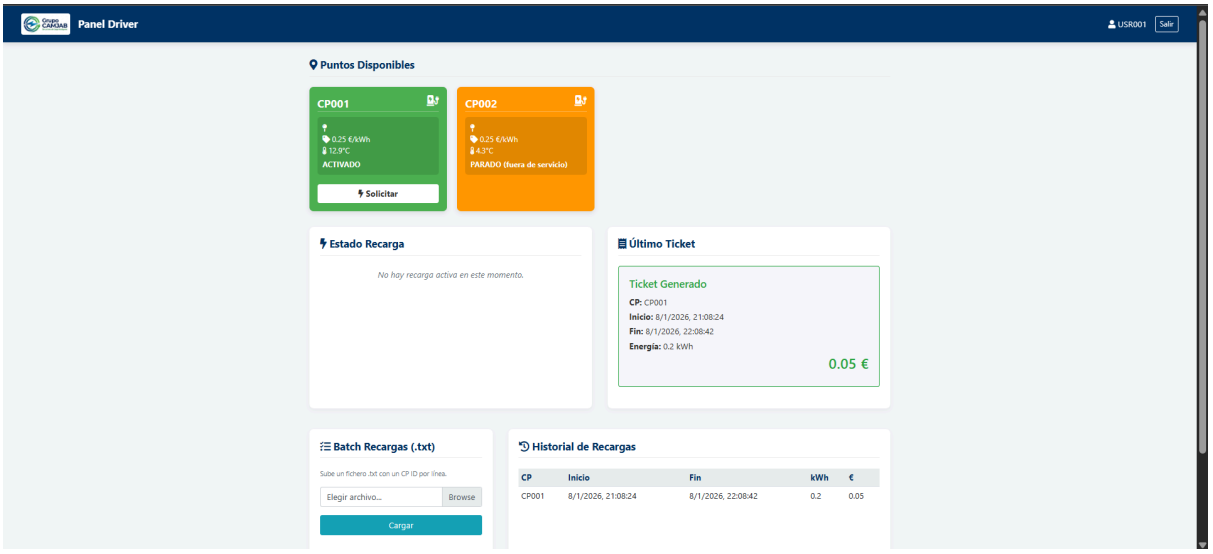
Esta es la aplicación del driver, una vez has hecho el login. Aquí podemos ver todos los cp, y en los que estén ‘Activados’ aparece un botón con el que podemos comenzar un suministro. También podemos ver todas las recargas anteriores del usuario. Y un apartado de mensajes, para mostrar errores o información



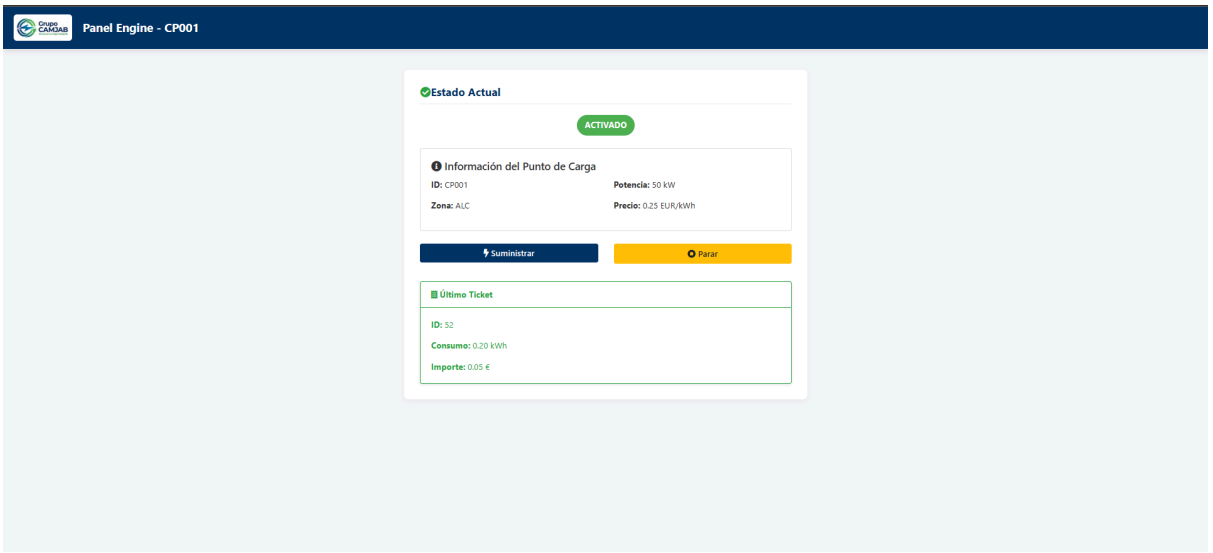
Aqui podemos ver como sería cuando comenzamos un suministro. Nos aparece los datos del suministro en tiempo real.



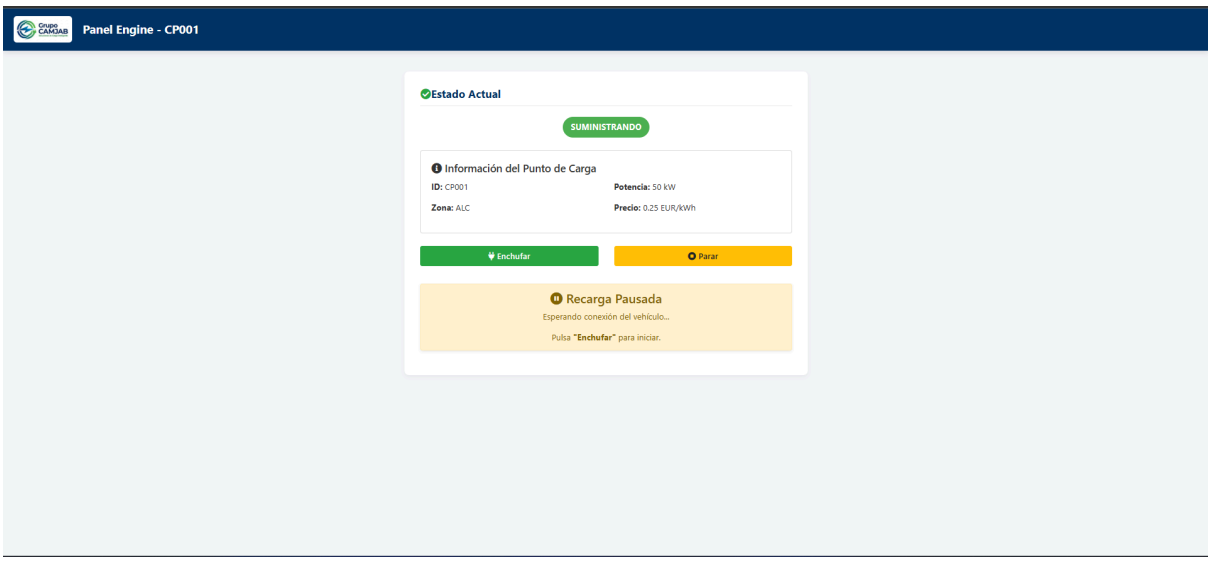
Una vez finalizamos la recarga, podemos ver que nos aparece el ticket con los datos de la recarga



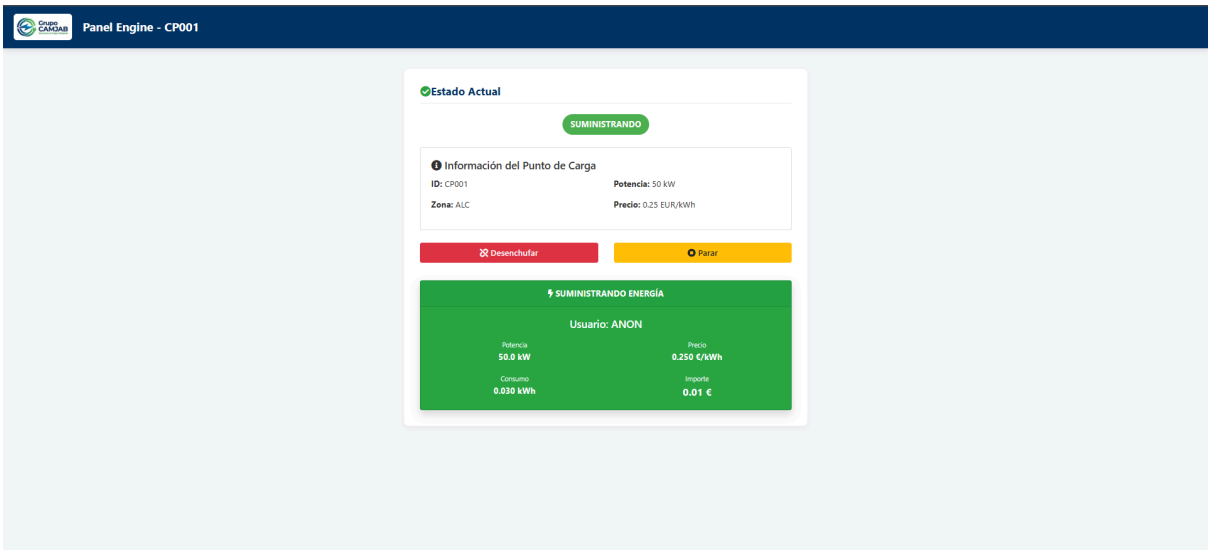
Aquí podemos ver la interfaz de la aplicación del engine, en la que podemos ver el estado en el que está, y podemos utilizar los botones que vemos en la imagen para solicitar una recarga ‘Anónima’ o poner el punto como ‘Fuera de servicio’.



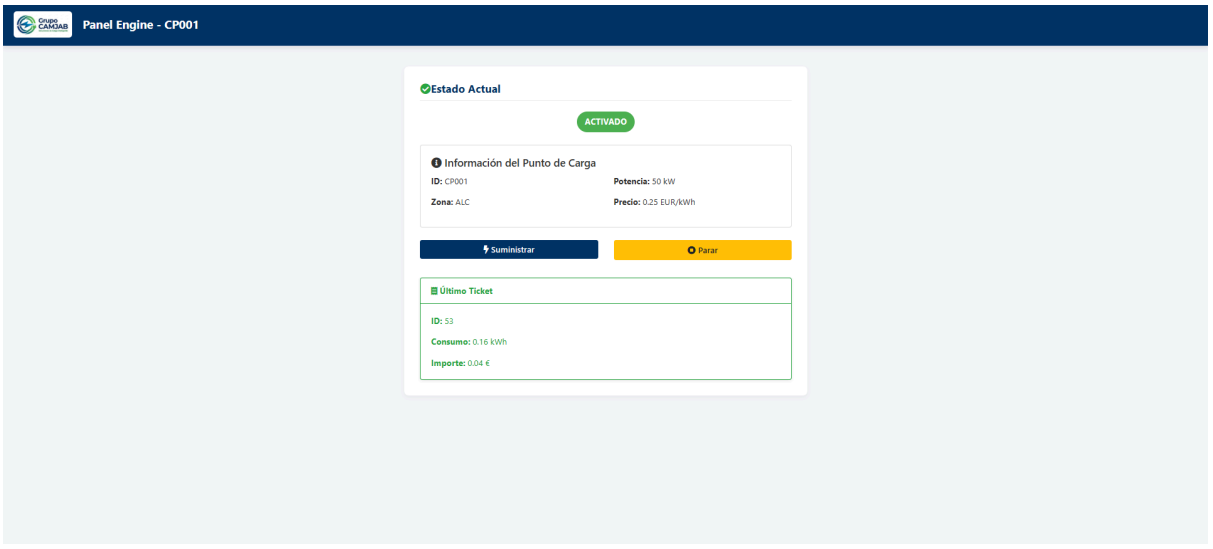
Aquí podemos ver, que cuando solicitamos un suministro, podemos ver que está esperando a ser enchufado



Una vez lo enchufamos podemos ver los datos en tiempo real:



Y aquí podemos ver, que una vez desenchufado, nos aparecerá el ticket de la recarga.



7. Guía de despliegue

Para lanzar nuestra aplicación hay 2 procedimientos para llevarlo a cabo:

7.1 Despliegue local con ejecutable

Para ejecutarlo en local y abrir todas las aplicaciones en un solo ordenador hemos preparado el ejecutable ‘run_all’ tanto .sh para sistemas linux, como .bat para sistemas Windows. El procedimiento será el siguietne:

Requisitos previos:

- Tener instalado docker
- Tener instalado python
- Preparar entorno Python:

```
cd /SD-jabg5-mca119
python3 -m venv .venv
source .venv/bin/activate
pip install --upgrade pip
```

- Instalar dependencias de Python:
pip install -r requeriments.txt

los requerimientos son :
Flask==2.3.3
Flask-SocketIO==5.3.6
python-socketio==5.11.2
kafka-python==2.0.2

- Darle los permisos necesarios al ejecutable

Una vez hechos todos estos pasos, ya podríamos desplegar las aplicaciones ejecutando en el directorio ‘SD-JABG5-MCA119’ el comando: **./run_all.sh**

7.2 Despliegue con docker

como requerimiento hay que tener instalador el docker o docker desktop

para iniciar la central, un servidor y un driver en la maquina local (en 3 compose)
basta con escribir
./start.sh <PUBLIC_IP>
en el directorio raíz
para iniciar solo el compose de central:
/start.sh <BROKER_PUBLIC_IP> [CENTRAL_SOCKET_PORT]
para iniciar solo monitores
./EV_CP/start.sh <CENTRAL_HOST> [CENTRAL_PORT] [CP_ID ...] [IP_HOST] [--no-cache]
para iniciar solo los drivers
./EV_Driver/start.sh <IP_BROKER> <PUERTO_BROKER> <IP_HOST> [DRVNNN ...]

para pararlo todo
./stop_all.sh
para parar individualmente un la central
./EV_Central/stop.sh
para parar individualmente un cp
./EV_CP/stop.sh **[CP_ID]**
se pararan todos si no se pone ningún id
para parar individualmente un Driver
./EV_Driver/stop.sh **[DRV_ID]**
se pararan todos si no se pone ningún id

7. Bibliografía

Copilot/Gemini/ChatGPT: Hemos utilizado estas inteligencias artificiales para consultar acerca de cómo organizarnos, cómo estructurar el proyecto, por donde empezar…etc. También nos han servido como refuerzo para estudiar y aprender sobre las nuevas tecnologías que no conocíamos (principalmente kafka y docker).

Iberdrola: Hemos utilizado esta web, para buscar las características que tiene un punto de carga real, y así utilizar datos realistas y que podrían utilizarse en la actualidad

Pagina de DockerHub sobre kafka: de este pagina sacamos casi toda la información necesaria sobre cómo manejar la imagen oficial de kafka y cogimos la idea de usar variables de entorno para controlar los contenedores