PRACTICA 3 SD
Práctica no guiada: Seguridad y API Rest
EV Charging Network

YOUSSEF SERROUKH AFELLAD 49436184F
ISRAEL IZQUIERDO SANCHIS 50387916E

CURSO: 2025/26

ÍNDICE

1. Introducción

1. Componentes Software Desarrollados

1.1. EV_Central (Sistema Central)

- Actúa como núcleo de la arquitectura, integrando Kafka, Base de Datos y API Rest.
- **Novedad:** Expone una API REST para comunicación con el Front-End y el módulo de clima (EV_W).
- **Novedad:** Implementa el descifrado simétrico (AES) de los mensajes entrantes de los CPs.
- **Novedad:** Gestiona alertas meteorológicas para detener CPs en condiciones adversas.
- Mantiene el log de auditoría de seguridad en la base de datos.

Funcionalidades del código

El módulo EV_Central inicializa la conexión con la base de datos a través de DBManager y levanta un servidor HTTPS (API_Central) en el puerto 5000 para atender peticiones externas (alertas de clima y estado para la web). Utiliza dos manejadores principales: WeatherHandler para recibir las alertas críticas del módulo EV_W y también tenemos el StatusHandler para que se pueda hacer el front con todos sus elementos donde convertimos el JSON en un HTML como Front-End.

Paralelamente, configura el Productor y Consumidor de Kafka, donde aplicamos la política offset earliest para que así nos aseguramos la recuperación de mensajes pendientes cuando se cae un servicio, es decir, si se cae la central y tenemos suministrando, cuando se vuelva a recuperar está reciba los mensajes del suministro junto con el ticket si finaliza. La lógica principal reside en el HiloServidor (Servidor Concurrente), que procesa los mensajes de los distintos tópicos donde cada petición se hace en un hilo independiente para que así no se bloqueen entre ellos. En esta versión, el servidor discrimina entre mensajes claros (conductores) y cifrados (puntos de carga); para estos últimos, recupera la clave simétrica de la base de datos y utiliza CryptoUtils para descifrar el contenido antes de procesarlo. Además, incorpora un WeatherHandler que recibe alertas HTTPS del módulo EV_W y emite órdenes de "Parada de Emergencia" a Kafka si la temperatura es crítica. Finalmente, permite comandos de administrador por consola para revocar credenciales de seguridad, lo que elimina las claves del CP en la base de datos y fuerza su desconexión.

1.2. EV_Registry (Servicio de Registro)

- **Novedad:** Autoridad de registro independiente para la autenticación segura.
- Valida la identidad de los Puntos de Carga mediante criptografía asimétrica.
- Genera y distribuye credenciales de sesión.
- Expone una API REST segura para el alta y baja de dispositivos.

Funcionalidades del código
El módulo EV_Registry es un servidor HTTPS seguro que escucha en el puerto 4444. Su principal núcleo es la clase de RegistroHandler, que es un APi REST completo que se encarga de gestionar el ciclo de vida de los CPs:

1. Registro(POST): Procesa las peticiones que incluyen el ID, certificado X.509 y firma digital. Esto lo que hace es realizar una validación criptográfica para asegurar que la firma corresponde con el certificado que hemos hecho. Si esta verificación es correcta, genera el token para que luego en el proceso de autenticación la central es la que nos genera las claves de cifrado AES y luego todo esto se devuelve al monitor para indicar que el registro y la autenticación es correcta, es resumen, lo que hace en el registro es proporcionar el token para que luego la central sea la encargada de dar las claves simétricas para hacer el cifrado, pero para esto tiene que tener un token.

2. Baja(DELETE): Gestiona la baja, es decir, la eliminación del CP, y la revocación de sus permisos si la central lo ejecuta ese módulo, para poder dar de baja es necesario que el CP esté registrado de antes, sino no se puede dar de baja.

3. Consulta(GET): Ésta permite verificar el estado de registro de un CP, el servidor responde con los datos del CP si está activo, es decir, cuando lo está devuelve un código 200, o indica que no está registrado ya que devuelve un código 404, esto permite que el monitor pueda saber el estado de registro de su engine.

1.3. EV_CP_M (Monitor)

- Supervisa el Engine y gestiona la seguridad del Punto de Carga.
- **Novedad:** Realiza el "Handshake" de seguridad con el Registry.
- **Novedad:** Transfiere las credenciales de cifrado al Engine de forma segura.
- Detecta caídas del Engine y reporta averías a la Central.

Funcionalidades del código
El módulo EV_CP_M actúa como el gestor de seguridad y salud del punto de carga. Al iniciar, lee del disco el certificado digital y la clave privada del CP. Utiliza estos archivos para firmar una petición de registro y enviarla vía HTTPS al módulo EV_Registry.

El proceso de alta sigue un flujo de seguridad de dos pasos:

1. Obtiene el token de sesión del Registry tras validar la firma digital
2. Se autentica con la central utilizando ese token para así obtener la clave de cifrado AES

Una vez obtenido todo esto, establece una conexión TCP local con el Engine y las transfiere mediante el protocolo interno SET_CREDS. Además mantiene una consistencia del estado local ya que verifica la existencia del CP antes de permitir las operaciones tanto de bajo como la de gestionar las respuestas inconsistentes para así limpiar las credenciales residuales, es decir, que la central las ha revocado. Finalmente, tiene un bloque de supervisión con el engine que ya teníamos en la práctica anterior, el "heartbeat" para detectar una caída o recuperación del servicio y esto lo reporta automáticamente a la central a través de Kafka.

1.4. EV_CP_E (Engine)

- Simula el hardware físico de carga y el suministro de energía.
- **Novedad:** Cifra todas las comunicaciones salientes hacia la Central (AES).
- Recibe credenciales del Monitor y se conecta a Kafka.
- Gestiona el ciclo de vida de la carga.

Funcionalidades del código

El módulo EV_CP_E representa la lógica operativa del cargador donde simula el hardware físico. Al arrancar, levanta un MonitorServer en un puerto local y permanece en espera hasta que el monitor le proporcione las credenciales de seguridad que ya hemos comentado antes con el protocolo de SET_CREDS.

Una vez ya estamos autenticados, se inicia la comunicación con Kafka con el CentralConnector donde se implementa una seguridad bidireccional usando la clase CryptoUtils:

- Salida: lo que hace es interceptar el mensaje y lo cifra mediante las clave AES que se le ha proporcionado, esto lo hace para todos los mensajes, es decir, para el estado de estos, las de consumo, los tickets, etc.

- Entrada: esté escucha todos los tópicos de comandos(comandos-cp) donde descifra las órdenes que recibe de la central (central-to-cp) antes de ejecutarlas.

El Engine gestiona órdenes como la de Revocar_Credenciales que lo que hace es forzar el borrado de las claves y una desconexión de la central automática, y también las parte de las paradas de emergencia que hace por las alertas climaticas. También incorpora un mecanismo de seguridad donde si se pierde la conexión con el monitor lo que va a hacer el engine va a ser detener el suministro y se bloquea hasta que el monitor vuelva a estar disponible. Finalmente, incluye lo de la práctica anterior, es decir, lo de las averías, recuperaciones, etc.

1.5. EV_W (Weather Control Office)

- **Novedad:** Monitoriza condiciones ambientales externas.
- Consulta APIs de terceros en tiempo real.

- Evalúa reglas de negocio (Temp < 0ºC) y notifica a la Central.

Funcionalidades del código
El módulo EV_W ejecuta un temporizador que consulta periódicamente la API pública de OpenWeatherMap para obtener la temperatura de la ubicación configurada. El código procesa la respuesta JSON y evalúa si la temperatura desciende por debajo del umbral de congelación (temperatura < 0ºC). Si esto se detecta, construye una carga útil JSON con la alerta y la notifica mediante una petición HTTPS POST segura al endpoint /api/alertas de EV_Central.

Este módulo dispone de una buena resiliencia ya que encapsula la comunicación en los controles de las excepciones, lo que hace que se pueda sobrevivir a las caídas que pueda haber de la API de la central o cuando la red no funciona correctamente, ya que cuando vuelve lo que hace es reenviar los datos automáticamente. Además permite que el cliente lo reconfigure durante la ejecución, ya que se puede cambiar la ciudad que se monitoriza o la API Key mediante el menú de la consola sin tener que detenerlo como ya hemos dicho antes.

1.6. EV_Driver

- Simula la interacción del usuario final.
- Solicita servicios de carga y recibe tickets de facturación.
- Funciona en modo interactivo o mediante script de pruebas.

Funcionalidades del código
El módulo EV_Driver gestiona la interacción del conductor con el sistema. Utiliza Kafka para enviar solicitudes de servicio y se suscribe a tópicos específicos de su ID para recibir respuestas de autorización y tickets finales. Implementa una máquina de estados simple que le impide solicitar un nuevo servicio si ya tiene una sesión activa. Incluye la capacidad de procesar un archivo de texto con una lista de CPs para automatizar pruebas de carga masivas o secuenciales, facilitando la validación del sistema distribuido. Además el driver puede emitir las solicitudes de servicio aunque la central esté caída ya que la petición se queda en la cola del tópico de “driver-solicitud” y cuando la central se recupere se procesa ya que estaba en la cola sin tener que volver a pedirla.

1.7. Front

- **Novedad:** Interfaz web pública accesible desde navegador.
- Visualiza en tiempo real el estado de toda la infraestructura.
- Muestra el registro de auditoría de seguridad, alertas climáticas, y de las funciones de la práctica anterior.

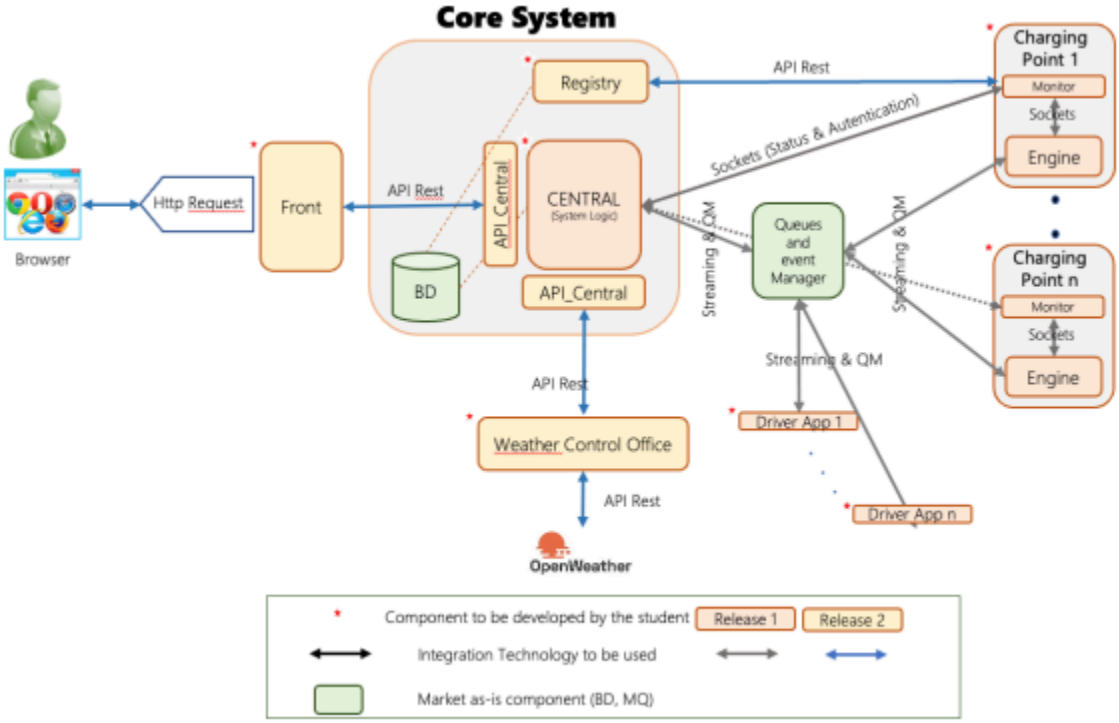
Funcionalidades del código
El Front-End es una aplicación web ligera desarrollada en HTML5, CSS y JavaScript sin dependencias externas ni compilación.

La arquitectura se basa en un cliente que consume la API REST con HTTPS de la central mediante la Fetch API de JavaScript.

Utiliza un patrón de comunicación con un sistema que hace consultas periódicas que son asíncronas para así refrescar el estado del sistema casi en tiempo real. Además la comunicación se hace mediante la implementación de las cabeceras CORS en el servidor Central para así que se pueda que el navegador acepte las respuestas de la API local.

Además este se va actualizando el DOM para así reflejar los cambios que pueda haber como las alertas climáticas o las desconexiones de seguridad y también el registro de auditoría. Además gestiona la resiliencia ya que si la API Central se cae y deja de responder, la interfaz cambia indicando la pérdida de la conexión pero cuando esta se vuelve a levantar el front va a estar actualizada con los mensajes y estados que hayan ocurrido mientras la API estaba caída

2. Arquitectura del sistema distribuido



Como se puede ver en la imagen que se nos proporcionó en la práctica, como novedad en esta práctica tenemos el EV_Registry que se va a conectar a la central mediante un API Rest y el EV_W que se conecta tanto a OpenWeather como a la API de la central mediante API Rest

como se hacía con el registro también, todas estas conexiones las vamos a hacer con HTTPS para así tener mayor seguridad y también tenemos que hacer un front que se va a poder desplegar en cualquier navegador y las veces que se quiera que se comunicara con el API Central para sacar la información a mostrar.

3. Base de datos

Se utiliza MariaDB con Docker
Tablas utilizadas:

- driver: Guarda la información de los drivers
- charging_point: Guarda la información y estado de los puntos de carga
- charging_session: Guarda la información de las sesiones activas y finalizadas
- event_log: Registra los logs que se reportan
- auditoría: Guarda todos los eventos que ocurren en el sistema con la dirección origen como novedad, esta lo que hace va a ser garantizar que todo lo que pase en nuestro sistema quede registrado.

4. Modelo de seguridad y criptografía

El sistema implementa una seguridad híbrida para así poder garantizar la confidencialidad, integridad y la autenticidad en el entorno distribuido:

- Asimetrica - RSA: Utilizamos una infraestructura de clave pública con PKI que simulamos. Durante el proceso de registro, lo que hacemos es que el monitor emplea certificados digitales X.509 que los firma digitalmente también para la petición de su clave privada, haciendo que un dispositivo que no tenga esto no pueda entrar en las comunicaciones.

- Simétrica - AES128: Tras la autenticación lo que se hace es generar una clave de sesión AES, esto lo hacemos para permitir cifrar los mensajes en Kafka para garantizar la seguridad de los mensajes y nadie pueda interceptarlos y leerlos ya que no tienen la clave para desencriptarlos.

En cuanto a la seguridad en las capas, lo que hacemos es que todas las comunicaciones de las API, es decir, el Registry, el Front, el Weather, y todo esto se hace con HTTPS. Además los mensajes, como ya he dicho antes, viajan cifrados por Kafka para así proteger los datos.

5. Resiliencia y tolerancia a fallos

Se han implementado mecanismos para así asegurar la continuidad ante las caídas de la infraestructura, aunque en la práctica 1 ya teníamos una parte de esto, aquí se ha mejorado por lo que contamos con lo siguiente:

- Desacoplamiento: La central configura su consumidor con auto.offset.reset = 'earliest' para que si esta se cae, al volver a levantarse se recupera automáticamente con todas las solicitudes tanto de los drivers como de los CPs que han hecho cuando estaba caída, haciendo que ninguna se pierda.

- Auto-reparación: El weather hace un bucle de reintento, es decir, que si detecta que la API Central no responde, lo que hacen es esperar a que se vuelva indicando por terminal que no puede establecer conexión y sigue intentando automáticamente la conexión de vuelta, haciendo que cuando esta se vuelva a levantar continúe sin que se tenga que intervenir manualmente

- Seguridad física: El monitor lo que hace es monitorizar constantemente la conexión con el engine, por lo que si este monitor se desconecta el engine para cualquier suministro que esté haciendo y se bloquea hasta que no se vuelva a conectar el monitor

6. Guía de Despliegue Paso a Paso

6.1. Preparación del entorno Kafka

cd ~/Kafka
bin/kafka-server-start.sh config/server.properties

6.2. Creación de topics

cd ~/Kafka
bin/kafka-topics.sh --create --topic cp-registro --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic cp-estado --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic cp-autorizacion --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic actualizacion-recarga --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

bin/kafka-topics.sh --create --topic fallo-cp --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic recuperacion-cp --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic central-to-cp --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic sistema-eventos --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic monitor-registro --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic ticket --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic comandos-cp --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --create --topic monitor-averias --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

Para ver que los topics se han creado de manera correcta:
bin/kafka-topics.sh --list --bootstrap-server localhost:9092

6.3. Despliegue de MariaDB en Docker

cd p3/
docker compose build
docker compose up -d
docker compose ps

Entrar en la base de datos (la contraseña de la base de datos es evpass):

docker exec -it p3_mariadb_1 mysql -u evuser -pevpass ev_charging_system

Insertar datos iniciales:

docker exec -i p3_mariadb_1 mysql -u evuser -p evpass ev_charging_system < p3/central/sql/inserts.sql

6.4. Generación de certificados

Antes de ejecutar, es necesario generar las claves y credenciales antes de iniciar los componentes.

Primeramente hay que generar el almacén de claves (JKS) para el servidor, la contraseña es password123, y se hace con el siguiente comando y lo tienes que guardar en p3/registry:

keytool -genkeypair -alias registry -keyalg RSA -keysize 2048 \ -storetype JKS -keystore registry.jks -validity 365 \ -storepass password123 -keypass password123

Después, hay que darle permisos al script de generación de la identidad del CP:
chmod +x generar_identidad.sh

Ahora una vez ya tiene los permisos se ejecuta de la siguiente manera para que se generen los certificados y claves, voy a poner un ejemplo para el CP001:

./generar_identidad.sh CP001

Esto va a generar cp001.key, cp001_java.key y cp001.crt y estos tienen que estar al mismo nivel que la carpeta p3

6.5. Ejecución

EV_Registry:
java -cp ".:p3/libs/*" p3.registry.EV_Registry

Central:
java -cp ".:p3/libs/*:p3" p3.central.EV_Central localhost:9092

Front:
No requiere comando de consola, con abrir el archivo index.html que se encuentra en la carpeta front.

Engine:
java -cp ".:p3/libs/*" p3.evcharging.cp.EV_CP_E CP001 "Calle Principal 123" 0.15 localhost:9090 localhost:9092 8080

Monitor:
java -cp ".:p3/libs/*:p3" p3.evcharging.cp.EV_CP_M localhost 8080 CP001 localhost:9092 8080

EV_M(Control Climatico):
java -cp "../p3/libs/*:.." p3.ev_w.EV_W
a2331b49f4c9d2656d837c291d852c12

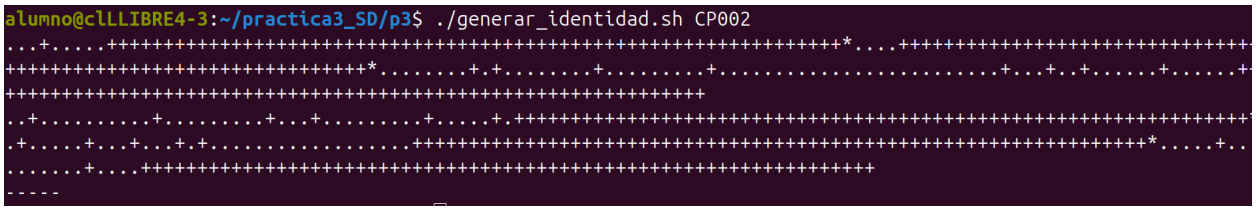
Driver (modo menú):
java -cp ".:p3/libs/*:p3" p3.driver.EV_Driver localhost:9092 DRIVER001

Para iniciar el driver con el archivo de texto lo que hay que ejecutar es lo siguiente: java -cp ".:p3/libs/*:p3" p3.driver.EV_Driver localhost:9092 DRIVER001 ruta_del_archivo/servicios.txt

7. Capturas de funcionamiento

Registro de un CP:

Primero hay que empezar con la generación de los certificados:



```
alumno@clLLIBRE4-3:~/practica3_SD$ java -cp ".:p3/libs/*" p3.evcharging.cp.EV_CP_E CP002 "Avenida Central 456, A
licante" 0.15 localhost:9090 172.20.243.101:9092 8081

--- MENÚ PRINCIPAL ---
1.  Iniciar suministro manual
2.  Finalizar suministro
3.  Activar CP
4.  Parar CP
5.  Simular avería
6.  Reparar avería
7.  Estado completo
8.  Salir
```

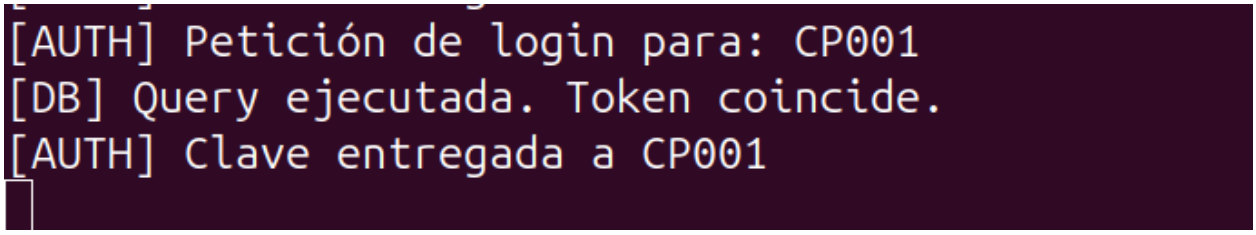
```
alumno@clLLIBRE4-3:~/practica3_SD$ java -cp ".:p3/libs/*" p3.evcharging.cp.EV_CP_M localhost 8081 CP002 172.20.243.101:9092 8081

Monitor iniciado para el CP: CP002
Conectado a Central en: 172.20.243.101:9092
Conectado al Engine en: localhost:8081

--- MENÚ MONITOR CP (CP002) ---
Engine conectado - 10:00:52.405395216
1. Registrar CP en Sistema
2. Eliminar CP en Sistema
3. Consultar estado CP en Sistema
4. Salir
```

```
1
Engine OK - 10:05:11.487382188
-> Registro OK en Registry. Token obtenido: aa23ce54
-> Iniciando autenticación con Central para obtener claves...
-> AUTENTICACIÓN EXITOSA CON CENTRAL. Clave recibida.
Engine OK - 10:05:12.488828703
Engine responde: CREDS_OK
```

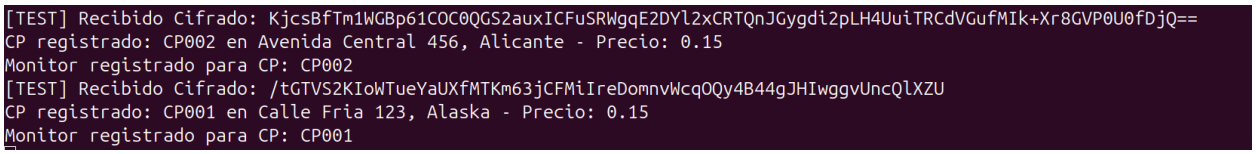
```
alumno@clLLIBRE4-4:~/practica3_SD$ java -cp ".:p3/libs/*" p3.registry.EV_Registry
--- EV_REGISTRY Iniciado ---
   >> Puerto: 4444
   >> Esperando peticiones de CPs...

[REGISTRY] Petición recibida: POST
ALTA OK: CP002 en Avenida Central 456, Alicante, ES

[REGISTRY] Petición recibida: POST
ALTA OK: CP001 en Calle Fria 123, Alaska
```

Esta es la API Central:

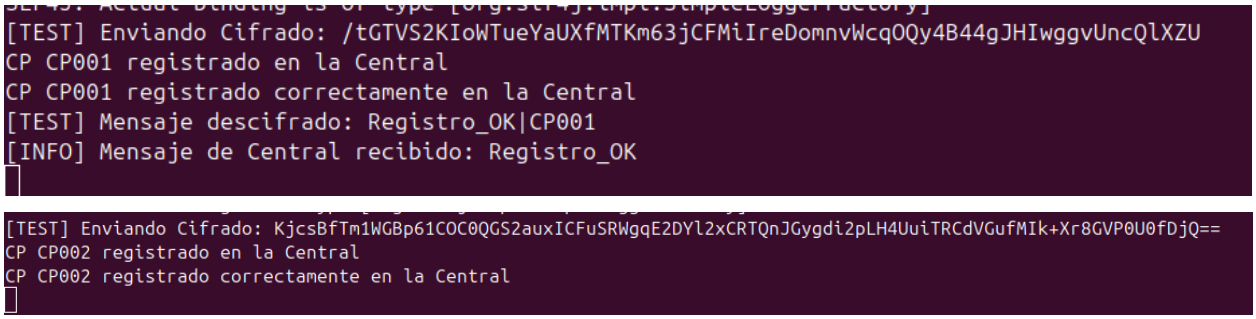
```
[AUTH] Petición de login para: CP002
[DB] Query ejecutada. Token coincide.
[AUTH] Clave entregada a CP002
```



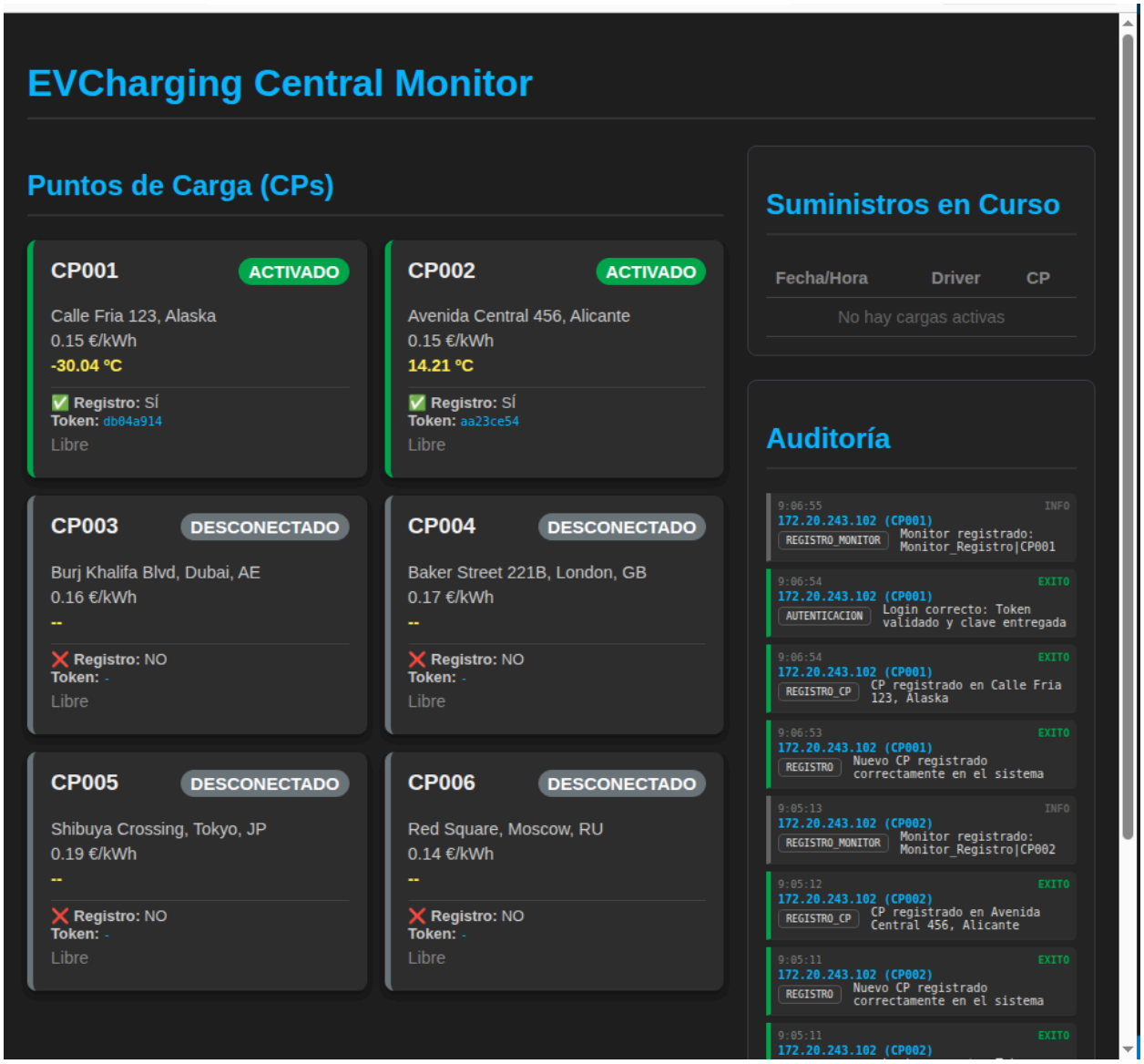
Esta es la central:



Esto es lo que hace el engine de ambos CPs:



Y este el front:



Ahora prueba de consultar el estado del CP, en el monitor con la opción 3 muestra esto:

```
--- ESTADO ACTUAL DEL CP ---
{"status":"OK", "id":"CP002", "estado":"ACTIVADO", "ubicacion":"Avenida Central 456, Alicante", "registrado":true}
```

Y en el registry esto:

```
[REGISTRY] Petición recibida: GET
CONSULTA OK: Estado enviado a CP002
```

Ahora vamos a dar de baja un CP:

En el monitor se muestra esto:

```
Engine confirma reseteo: OK_RESET
Engine OK - 10:12:38.675769616
BAJA COMPLETADA. El CP ha sido eliminado del sistema.
```

En el registry llega esto:

```
[REGISTRY] Petición recibida: DELETE
BAJA OK: CP002 ha sido eliminado del registro.
```

En el engine se muestra esto:

```
[TEST] Enviando Cifrado: 4Y0VIkCx30PjzHNB7pbAVg==
Solicitud de baja enviada a Central
```

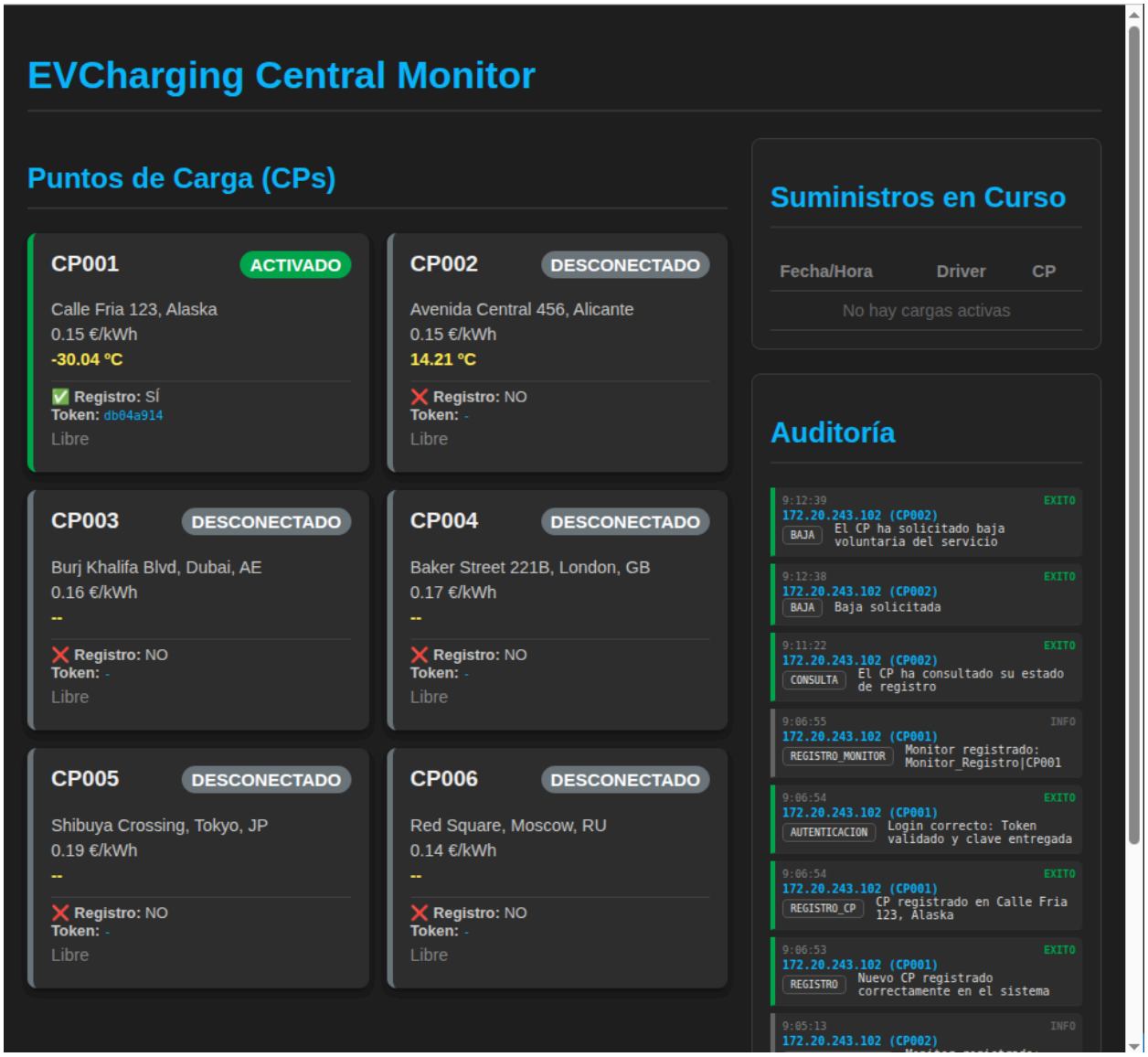
La central muestra esto:

```
[TEST] Recibido Cifrado: 4Y0VIkCx30PjzHNB7pbAVg==
Baja completada y credenciales eliminadas para CP002
```

Para ver que la baja se ha completado vamos a ver el estado desde el monitor:

```
--- ESTADO ACTUAL DEL CP ---
{"status":"OK", "id":"CP002", "estado":"DESCONECTADO", "ubicacion":"Avenida Central 456, Alicante", "registrado":false}
```

Y el front:



Ahora una vez demostrado la parte del registry con el alta y la baja de un cp, vamos a mostrar la parte de openweather con la api:

Por defecto se inicia en Alicante pero tenemos la posibilidad de cambiar tanto la ciudad como el Api Key desde el menú, y además la clave se pasa por argumentos:

```
alumno@clLLIBRE4-3:~/practica3_SD/p3$ java -cp "../p3/libs/*:.." p3.ev_w.EV_W a2331b49f4c9d2656d837c291d852c12

--- MENÚ DE CONTROL CLIMÁTICO ---
Ciudad actual monitorizada: Alicante,ES
1. Cambiar ciudad
1. Cambiar API KEY
3. Salir
Seleccione opción: [AUTO] Alicante,ES -> 14,93ºC. Temperatura Operativa -> Enviando OK.
[AUTO] Alicante,ES -> 14,93ºC. Temperatura Operativa -> Enviando OK.
```

Ahora para probar lo de la temperatura cambiamos la ciudad:

```
>>> CIUDAD CAMBIADA A: Alaska
(El próximo chequeo automático usará la nueva localización)

--- MENÚ DE CONTROL CLIMÁTICO ---
Ciudad actual monitorizada: Alaska
1. Cambiar ciudad
1. Cambiar API KEY
3. Salir
Seleccione opción: [AUTO] Alaska -> -30,04°C. ¡ALERTA BAJA TEMPERATURA! -> Notificando a Central.
[AUTO] Alaska -> -30,04°C. ¡ALERTA BAJA TEMPERATURA! -> Notificando a Central.
[AUTO] Alaska -> -30,04°C. ¡ALERTA BAJA TEMPERATURA! -> Notificando a Central.
[AUTO] Alaska -> -30,04°C. ¡ALERTA BAJA TEMPERATURA! -> Notificando a Central.
```

En el API central se ve lo siguiente:

```
[TEST] CP: CP001 | Comando: Parada_Emergencia | Encriptado: S5NtyiBGCNLv8iNsaRD2dwu9bJwhcFidqRU8/oDP40w=
[API REST] Alerta recibida de EV_W: {"ciudad": "Alaska", "temperatura": -30.04, "alerta": true}
ALERTA FRÍO: Enviando PARADA a CP001
```

Y en la central:

```
[TEST] Recibido Cifrado: b1hSPILj88psNgVlbALDk07ccJD/2co0AKGiSI0K1kjJTIkZF76i/yncKekysbx+
Estado actualizado CP: CP001: PARADO - Funciona: Ok
```

En el engine se recibe la solicitud de la parada:

```
[TEST] Mensaje descifrado: Actualizacion_OK|CP001
[INFO] Mensaje de Central recibido: Actualizacion_OK
[TEST] Mensaje descifrado: Parada_Emergencia
[TEST] Enviando Cifrado: b1hSPILj88psNgVlbALDk07ccJD/2co0AKGiSI0K1kjJTIkZF76i/yncKekysbx+
CP fuera de servicio
```

Y todo esto se puede ver en la auditoría como en el front:

8. Capturas de resiliencia

Se puede observar que cuando el CP no está registrado e intentas darte de baja obviamente no te lo permite.

```
Error: No puedes dar de baja un CP que no está registrado.
```

Para la seguridad del registry y los cps, el api la podemos consumir desde otra aplicación externa, en este caso, ejecutamos este comando desde la terminal:

curl -v -k -X POST -H "Content-Type: application/json" -d '{"id": "CP_HACKER"}' https://172.20.243.105:4444/api/registro

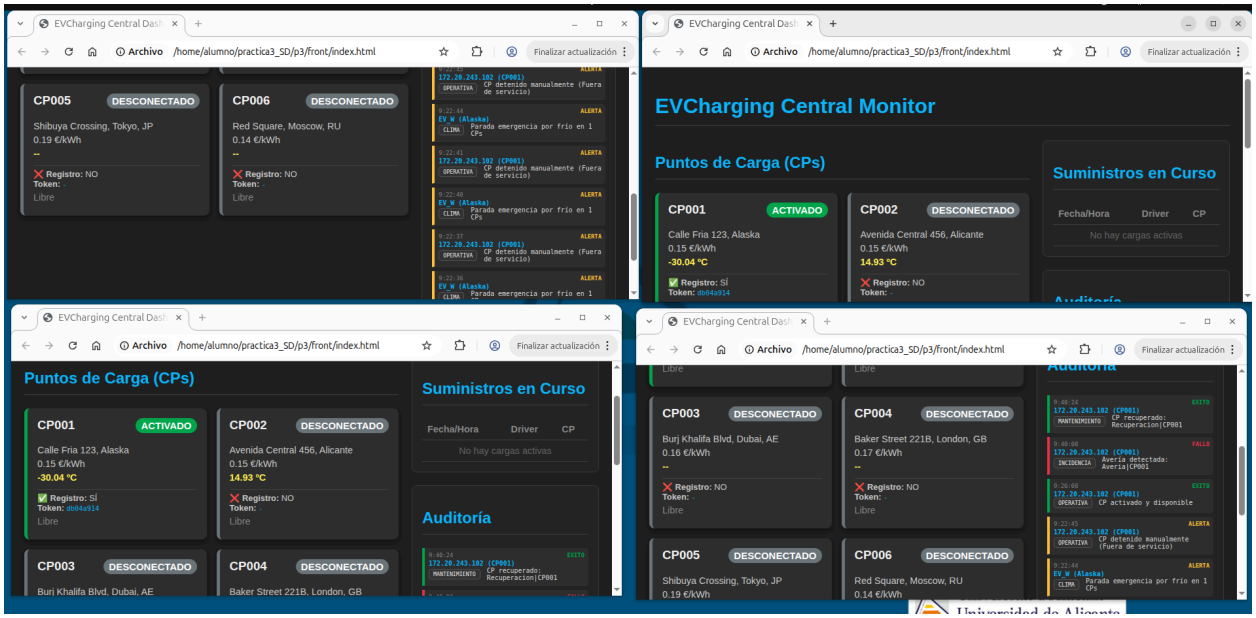
En las capturas se va a ver como si que consume la api, pero al no tener un token el registro no se puede hacer y no se procesa, lo que garantiza la seguridad de nuestro sistema:

```
alumno@clLLIBRE4-3:~/practica3_SD$ curl -v -k -X POST -H "Content-Type: application/json" -d '{"id": "CP_HACKER"}' https://172.20.243.105:4444/api/registro
Note: Unnecessary use of -X or --request, POST is already inferred.
*   Trying 172.20.243.105:4444...
* Connected to 172.20.243.105 (172.20.243.105) port 4444
* ALPN: curl offers h2,http/1.1
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384 / X25519 / RSASSA-PSS
* ALPN: server did not agree on a protocol. Uses default.
* Server certificate:
*  subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*  start date: Dec 13 10:01:22 2025 GMT
*  expire date: Dec 13 10:01:22 2026 GMT
*  issuer: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*  SSL certificate verify result: self-signed certificate (18), continuing anyway.
*   Certificate level 0: Public key type RSA (2048/112 Bits/secBits), signed using sha256WithRSAEncryption
* using HTTP/1.x
> POST /api/registro HTTP/1.1
> Host: 172.20.243.105:4444
> User-Agent: curl/8.5.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 19
>
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
< HTTP/1.1 400 Bad Request
< Date: Thu, 18 Dec 2025 09:33:36 GMT
< Access-control-allow-origin: *
< Content-length: 37
<
* Connection #0 to host 172.20.243.105 left intact
{"error":"Faltan datos de seguridad"}alumno@clLLIBRE4-3:~/practica3_SD$
```

En el registry se ve que llega, pero no se hace el ok como en las demás altas:

```
[REGISTRY] Petición recibida: POST
```

Como se puede observar en la imagen, se pueden abrir varios front simultáneamente, completamente funcionales.



También funciona en diferentes navegadores, en este caso lo he hemos abierto en Firefox: