# Scripts

Des scripts divers et variés ;)

- triggerMessage BootNotification
- Déploiement BACKEND
- reboot all default
- Nouvelle pageBootNotification aux bornes FAULTED

# triggerMessage BootNotification

```
/**
 * Script POC pour envoyer un triggerMessage BootNotification à toutes les bornes
 *
 * UTILISATION:
 * 1. Ouvrir la console du navigateur (F12) sur votre application Angular
 * 2. Copier-coller ce script dans la console
 * 3. Appeler: triggerBootNotificationToAllStations()
 *    OU avec URL personnalisée: triggerBootNotificationToAllStations("https://portail.we-go.pro/api")
 *
 * POUR ARRÊTER LE SCRIPT:
 * - Appeler: stopBootNotificationScript()
 * - OU rafraîchir la page (F5)
 *
 * IMPORTANT: Assurez-vous d'être connecté et que le token d'authentification est présent
 */

// Variable globale pour contrôler l'arrêt du script
let stopBootNotificationScriptFlag = false;

// Fonction pour arrêter le script
function stopBootNotificationScript() {
  stopBootNotificationScriptFlag = true;
  console.log("□□Arrêt du script demandé. Le script s'arrêtera après la requête en cours...");
}

async function triggerBootNotificationToAllStations(customApiUrl = null) {
  // Réinitialiser le flag d'arrêt
  stopBootNotificationScriptFlag = false;
  console.log("□□Démarrage de l'envoi de BootNotification à toutes les bornes...");
  console.log("□□Pour arrêter le script, appelez: stopBootNotificationScript()");

  try {
    // Récupérer le token d'authentification depuis localStorage
    const token = localStorage.getItem("token");
```

```
    if (!token) {
      throw new Error("Token d'authentification non trouvé dans localStorage (clé: 'token'). Assurez-vous d'être connecté.");
    }

    // Récupérer le groupe sélectionné
    const selectedRoleStr = localStorage.getItem("selectedRole");
    if (!selectedRoleStr) {
      throw new Error("selectedRole non trouvé dans localStorage. Assurez-vous d'avoir sélectionné un groupe.");
    }

    let selectedRole;
    try {
      selectedRole = JSON.parse(selectedRoleStr);
    } catch (e) {
      throw new Error("Erreur lors du parsing de selectedRole: " + e.message);
    }

    if (!selectedRole?.groupId) {
      throw new Error("groupId non trouvé dans selectedRole. Structure: " + JSON.stringify(selectedRole));
    }

    // Détecter l'URL de l'API
    let API_URL;
    const currentOrigin = window.location.origin;

    if (customApiUrl) {
      // URL personnalisée fournie
      API_URL = customApiUrl;
      console.log(`□□URL de l'API (personnalisée): ${API_URL}`);
    } else if (currentOrigin.includes("localhost") || currentOrigin.includes("127.0.0.1")) {
      // Mode développement
      API_URL = currentOrigin.replace(":4200", ":3000").replace(":4201", ":3000") || "http://localhost:3000";
      console.log(`□□URL de l'API (développement): ${API_URL}`);
    } else {
      // Mode production - utiliser le même domaine avec /api
```

```
      API_URL = currentOrigin + "/api";
      console.log(`□□URL de l'API (production): ${API_URL}`);
    }
    console.log(`□□Groupe ID: ${selectedRole.groupId}`);
    console.log(`□□Token présent: ${token ? "Oui (longueur: " + token.length + ")" : "Non"}`);

    const groupId = selectedRole.groupId;
    const pageSize = 100;
    let page = 1;
    let allStations = [];
    let hasMorePages = true;

    // Récupérer toutes les bornes
    console.log("□□Récupération de la liste des bornes...");
    while (hasMorePages && !stopBootNotificationScriptFlag) {
      if (stopBootNotificationScriptFlag) {
        console.log("□□Arrêt demandé pendant la récupération des bornes");
        break;
      }

      const offset = (page - 1) * pageSize;
      const url = `${API_URL}/charging-
stations/group/${groupId}/list?offset=${offset}&limit=${pageSize}`;

      console.log(`□□Requête: ${url}`);

      // Ajouter un timeout pour la récupération des bornes aussi
      const fetchController = new AbortController();
      const fetchTimeoutId = setTimeout(() => fetchController.abort(), 10000); // 10 secondes
pour la liste (peut être plus long)

      let response;
      try {
        response = await fetch(url, {
          headers: {
            Authorization: `Bearer ${token}`,
            "Content-Type": "application/json",
          },
          signal: fetchController.signal,
        });
```

```
        clearTimeout(fetchTimeoutId);
      } catch (fetchError) {
        clearTimeout(fetchTimeoutId);
        if (fetchError.name === "AbortError") {
          throw new Error("Timeout lors de la récupération des bornes (plus de 10 secondes)");
        }
        throw new Error(`Erreur réseau lors de la récupération: ${fetchError.message}`);
      }

      // Vérifier le Content-Type avant de parser
      const contentType = response.headers.get("content-type");
      if (!contentType || !contentType.includes("application/json")) {
        const responseText = await response.text();
        console.error("❌ Réponse non-JSON reçue:");
        console.error(`   Content-Type: ${contentType || "non défini"}`);
        console.error(`   Status: ${response.status} ${response.statusText}`);
        console.error(`   Début de la réponse: ${responseText.substring(0, 200)}...`);

        if (responseText.includes("<!doctype") || responseText.includes("<html")) {
          throw new Error(
            `L'API a retourné du HTML au lieu de JSON. Cela indique probablement:\n` +
            `- L'URL de l'API est incorrecte (actuelle: ${API_URL})\n` +
            `- Une redirection vers une page d'erreur\n` +
            `- Un problème d'authentification\n\n` +
            `Vérifiez que l'URL de l'API est correcte. Si vous êtes en production, l'API
devrait être sur: ${currentOrigin}/api`
          );
        }
        throw new Error(`Réponse non-JSON: ${response.status} ${response.statusText}`);
      }

      if (!response.ok) {
        const errorText = await response.text();
        throw new Error(`Erreur HTTP: ${response.status}
${response.statusText}\n${errorText}`);
      }

      let data;
      try {
        data = await response.json();
```

```
      } catch (parseError) {
        const responseText = await response.text();
        console.error("❌ Erreur lors du parsing JSON:");
        console.error(`   Réponse reçue: ${responseText.substring(0, 500)}...`);
        throw new Error(`Erreur parsing JSON: ${parseError.message}`);
      }

      const totalRecords = parseInt(response.headers.get("X-Total-Count") || "0");

      allStations = allStations.concat(data.records || data);

      const totalPages = Math.ceil(totalRecords / pageSize);
      hasMorePages = page < totalPages;
      page++;

      console.log(`✅ Page ${page - 1}/${totalPages} récupérée (${allStations.length} bornes trouvées)`);
    }

    console.log(`\n📊 Total: ${allStations.length} bornes trouvées\n`);

    // Filtrer les bornes avec un ID valide
    const stationsWithId = allStations.filter((station) => station.id !== undefined && station.id !== null);
    console.log(`✅ ${stationsWithId.length} bornes avec ID valide\n`);

    if (stationsWithId.length === 0) {
      console.warn("⚠️ Aucune borne avec ID valide trouvée!");
      return [];
    }

    // Demander confirmation
    const confirmMessage = `Voulez-vous envoyer BootNotification à ${stationsWithId.length} bornes?\n\nLe script continuera même si certaines bornes échouent.`;
    if (!confirm(confirmMessage)) {
      console.log("❌ Opération annulée par l'utilisateur");
      return [];
    }

    // Envoyer le triggerMessage BootNotification à chaque borne
```

```
    const results = [];
    let successCount = 0;
    let errorCount = 0;
    let timeoutCount = 0;
    let networkErrorCount = 0;

    console.log(`\n□□Démarrage de l'envoi à ${stationsWithId.length} bornes...`);
    console.log(`□□Le script continuera même si certaines bornes échouent ou ne répondent pas.\n`);

    for (let i = 0; i < stationsWithId.length; i++) {
      // Vérifier si l'arrêt a été demandé
      if (stopBootNotificationScriptFlag) {
        console.log(`\n□□Arrêt du script demandé. Arrêt après ${i}/${stationsWithId.length} bornes traitées.`);
        break;
      }

      const station = stationsWithId[i];
      const stationId = station.id;
      const stationName = station.name || station.chargeboxIdentity || `Borne ${stationId}`;

      try {
        console.log(`[${i + 1}/${stationsWithId.length}] Envoi BootNotification à ${stationName} (ID: ${stationId})...`);

        const triggerUrl = `${API_URL}/ocpp/trigger-message/${stationId}`;

        // Créer un AbortController pour gérer les timeouts
        const controller = new AbortController();
        const timeoutId = setTimeout(() => controller.abort(), 1000); // Timeout de 5 secondes

        let triggerResponse;
        try {
          triggerResponse = await fetch(triggerUrl, {
            method: "POST",
            headers: {
              Authorization: `Bearer ${token}`,
              "Content-Type": "application/json",
            },
```

```
            body: JSON.stringify({
              command: "BootNotification",
              connector: 0,
            }),
            signal: controller.signal, // Ajouter le signal pour le timeout
          });
          clearTimeout(timeoutId); // Annuler le timeout si la requête réussit
        } catch (fetchError) {
          clearTimeout(timeoutId);
          if (fetchError.name === "AbortError") {
            timeoutCount++;
            throw new Error("Timeout: La requête a pris plus de 1 secondes");
          }
          networkErrorCount++;
          throw new Error(`Erreur réseau: ${fetchError.message}`);
        }

        if (!triggerResponse.ok) {
          const errorText = await triggerResponse.text();
          throw new Error(`HTTP ${triggerResponse.status}: ${errorText}`);
        }

        // Vérifier le Content-Type
        const triggerContentType = triggerResponse.headers.get("content-type");
        if (!triggerContentType || !triggerContentType.includes("application/json")) {
          const errorText = await triggerResponse.text();
          throw new Error(`Réponse non-JSON (${triggerContentType}): ${errorText.substring(0,
200)}`);
        }

        let triggerResult;
        try {
          triggerResult = await triggerResponse.json();
        } catch (parseError) {
          const errorText = await triggerResponse.text();
          throw new Error(`Erreur parsing JSON: ${parseError.message}. Réponse:
${errorText.substring(0, 200)}`);
        }

        // Afficher la réponse complète pour les 3 premières requêtes (débogage)
```

```
        if (i < 3) {
          console.log(`  □□Réponse complète (débogage):`, JSON.stringify(triggerResult, null,
2));
        }

        // Extraire le statut selon différentes structures possibles
        let status = null;
        let errorMessage = null;

        if (Array.isArray(triggerResult)) {
          // Si c'est un tableau, prendre le premier élément
          const firstResult = triggerResult[0];
          if (firstResult) {
            status = firstResult.result?.status || firstResult.status;
            errorMessage = firstResult.error || firstResult.result?.error;
          }
        } else if (triggerResult && typeof triggerResult === "object") {
          // Si c'est un objet
          status = triggerResult.result?.status || triggerResult.status;
          errorMessage = triggerResult.error || triggerResult.result?.error;

          // Vérifier aussi si la réponse contient directement un message
          if (!status && !errorMessage) {
            // Peut-être que la réponse est directement le statut
            status = triggerResult.message || triggerResult.data?.status;
          }
        }

        // Si on n'a toujours rien trouvé, afficher la structure complète
        if (!status && !errorMessage) {
          console.warn(`  ⚠□ Structure de réponse inattendue pour la borne ${stationId}:`,
triggerResult);
          errorMessage = `Structure de réponse inattendue:
${JSON.stringify(triggerResult).substring(0, 100)}`;
        }

        // Déterminer le succès
        const isSuccess = status === "Accepted" || status === "accepted";

        if (isSuccess) {
```

```
          successCount++;
          results.push({ id: stationId, name: stationName, success: true, status: status });
          console.log(`  □ Succès: ${status}`);
        } else {
          errorCount++;
          const finalError = errorMessage || status || "Statut inconnu";
          results.push({ id: stationId, name: stationName, success: false, error:
String(finalError), status: status });
          console.log(`  □ Échec: ${finalError}${status ? ` (status: ${status})` : ""}`);
        }
      } catch (error) {
        errorCount++;
        const errorMessage = error instanceof Error ? error.message : String(error);
        results.push({ id: stationId, name: stationName, success: false, error: errorMessage
});
        console.log(`  □ Erreur: ${errorMessage}`);
        // Le script continue avec la borne suivante même en cas d'erreur
        console.log(`  □□ Passage à la borne suivante...`);
      }

      // Petit délai pour éviter de surcharger le serveur
      if (i < stationsWithId.length - 1) {
        await new Promise((resolve) => setTimeout(resolve, 100));
      }
    }

    // Résumé
    console.log("\n" + "=".repeat(60));
    console.log("□□RÉSUMÉ");
    console.log("=".repeat(60));
    if (stopBootNotificationScriptFlag) {
      console.log("⚠□ Script arrêté par l'utilisateur");
    }
    console.log(`□ Succès: ${successCount}`);
    console.log(`□ Échecs totaux: ${errorCount}`);
    if (timeoutCount > 0) {
      console.log(`□□ Timeouts: ${timeoutCount}`);
    }
    if (networkErrorCount > 0) {
      console.log(`□□Erreurs réseau: ${networkErrorCount}`);
```

```
    }
    console.log(`□□Total traité: ${results.length}/${stationsWithId.length}`);
    if (stopBootNotificationScriptFlag && results.length < stationsWithId.length) {
      console.log(`□□ Arrêté avant la fin (${stationsWithId.length - results.length} bornes
non traitées)`);
    } else if (results.length === stationsWithId.length) {
      console.log(`□ Toutes les bornes ont été traitées!`);
    }
    console.log("=".repeat(60));

    // Afficher les échecs
    const failures = results.filter((r) => !r.success);
    if (failures.length > 0) {
      console.log("\n□ Bornes en échec:");
      failures.forEach((f) => {
        console.log(`  - ${f.name} (ID: ${f.id}): ${f.error}`);
      });
    }

    return results;
  } catch (error) {
    console.error("□ Erreur fatale:", error);
    throw error;
  }
}

// Exporter pour utilisation dans la console
if (typeof window !== "undefined") {
  window.triggerBootNotificationToAllStations = triggerBootNotificationToAllStations;
  window.stopBootNotificationScript = stopBootNotificationScript;
  console.log("□ Script chargé!");
  console.log("□□Commandes disponibles:");
  console.log("   - triggerBootNotificationToAllStations() : Démarrer");
  console.log("   - stopBootNotificationScript() : Arrêter le script en cours");
}
```

# Déploiement BACKEND

# Script de Déploiement Backend (Pré-production)

Ce script Bash est conçu pour automatiser le déploiement d'une application backend sur un environnement de pré-production. Il gère la mise à jour du code, la sauvegarde des versions précédentes, la configuration spécifique à l'environnement et le déploiement via Docker Compose.

# ? Fonctionnalités

- **Sauvegarde Automatique** : Archive la version précédente du déploiement avant toute mise à jour.
- **Clonage Git Sécurisé** : Récupère la dernière version du code depuis un dépôt Git privé via SSH.
- **Gestion des Variables d'Environnement** : Copie le fichier `.env` de la version précédente pour maintenir la configuration spécifique à l'environnement.
- **Déploiement Dockerisé** : Construit et démarre le service Docker Compose spécifié.

# ?? Prérequis

Avant d'exécuter ce script, assurez-vous que les éléments suivants sont configurés sur le serveur de déploiement :

- **Git** : Installé et configuré.
- **Docker & Docker Compose** : Installés.
- **Clé SSH** : Une clé SSH (`brian_git` dans l'exemple) doit être configurée pour accéder au dépôt Git (`git@github.com:Eiffage-AFC/WegoBackend.git`). Le chemin de la clé est spécifié par la variable `SSH_KEY`.
- **Permissions `sudo`** : L'utilisateur exécutant le script doit avoir les permissions `sudo` pour les opérations de création de répertoire, déplacement de fichiers, clonage Git et commandes Docker.

# ?? Variables Clés

Modifiez ces variables au début du script pour l'adapter à votre projet :

- `FRONT_REPO` : URL SSH du dépôt Git de votre backend.
- `BRANCH` : Branche Git à déployer (par exemple, `preprod` ou `main`).
- `DEPLOY_DIR` : Chemin absolu où l'application sera déployée sur le serveur.
- `DOCKER_SERVICE` : Nom du service Docker à construire et démarrer (défini dans votre `docker-compose.yml`).
- `PREV_DIR` : Répertoire où les sauvegardes des déploiements précédents seront stockées.
- `SSH_KEY` : Chemin vers la clé SSH privée utilisée pour l'authentification Git.

# ? Étapes du Déploiement

1. **Préparation du dossier de sauvegarde** : Crée un répertoire pour stocker les anciennes versions du déploiement.
2. **Sauvegarde** : Si une version de l'application est déjà présente dans `DEPLOY_DIR`, elle est déplacée vers `PREV_DIR` avec un horodatage pour backup.
3. **Clonage du dépôt** : Le script clone la branche spécifiée (`preprod`) du dépôt Git vers le `DEPLOY_DIR`. L'authentification SSH est utilisée.
4. **Configuration de l'environnement** : Le fichier `.env` de la version précédente sauvegardée est copié dans le nouveau dossier de déploiement. **(Il est essentiel que le `.env` de la première installation soit présent dans le dossier `prev` pour être copié).**
5. **Build et Déploiement Docker** : Le script navigue vers le `DEPLOY_DIR`, construit l'image Docker du service spécifié et démarre (ou redémarre) le conteneur en mode détaché.

# ? Utilisation

Pour exécuter ce script, connectez-vous à votre serveur cible via SSH et lancez-le :

bash nom_du_script.sh

```
#!/bin/bash


# ----------------------------
# Déploiement BAKCEND Angular (SSH)
# ----------------------------

# Variables
FRONT_REPO="git@github.com:Eiffage-AFC/WegoBackend.git"
BRANCH="preprod"
DEPLOY_DIR="/var/wego/WegoBackend"
```

```
DOCKER_SERVICE="backend"
PREV_DIR="/var/wego/prev/backend"
TIMESTAMP=$(date +%d%m%Y)
SSH_KEY="$HOME/.ssh/brian_git"

echo "?? Début du déploiement BACKEND"

# 0?? Préparer le dossier prev pour les backups
sudo mkdir -p "$PREV_DIR"

# 1?? Sauvegarder l'ancienne préprod si elle existe
if [ -d "$DEPLOY_DIR" ]; then
  echo "?? Sauvegarde de l'ancienne préprod dans $PREV_DIR/$TIMESTAMP"
  sudo mv "$DEPLOY_DIR" "$PREV_DIR/$TIMESTAMP/"
fi

# 2?? Cloner la branche preprod depuis Git via SSH
echo "?? Clonage de la branche $BRANCH via SSH"
sudo GIT_SSH_COMMAND="ssh -i $SSH_KEY -o StrictHostKeyChecking=no" git clone -b "$BRANCH"
"$FRONT_REPO" "$DEPLOY_DIR"

# 3?? Configuration Backend pour la production


sudo cp "$PREV_DIR/$TIMESTAMP/.env" "$DEPLOY_DIR/.env"


echo "Fichier .env Sauvegarder"

# 4?? Build Docker et déploiement
echo "?? Build Docker et démarrage du service $DOCKER_SERVICE"
cd "$DEPLOY_DIR"
sudo docker compose build "$DOCKER_SERVICE"
sudo docker compose up -d "$DOCKER_SERVICE"

echo "? Déploiement Backend terminé !"
```

# reboot all default

/**

- Script TURBO – RESET (Reboot) des bornes FAULTED
- Commande : OCPP Reset
- Optimisations : Parallélisme (10x plus rapide)
- Utilisation :
- 1. Refresh (F5) pour nettoyer
- 2. Coller le script
- 3. Lancer : triggerResetToFaultedStations() */

let stopResetScriptFlag = false; const RESET_TYPE = "Soft"; // "Soft" (Logiciel) ou "Hard" (Matériel/Électrique)

function stopResetScript() { stopResetScriptFlag = true; console.log("□□ Arrêt du script demandé..."); }

function getStationId(station) { return ( station.id ?? station.chargingStationId ?? station.stationId ?? station.idChargingStation ?? null ); }

async function triggerResetToFaultedStations(customApiUrl = null) { stopResetScriptFlag = false; console.log(`□□Envoi RESET (${RESET_TYPE}) aux bornes FAULTED`);

try { // □□ Auth & Config const token = localStorage.getItem("token"); if (!token) throw new Error("Token non trouvé");

```
const selectedRole = JSON.parse(localStorage.getItem("selectedRole") || "{}");
if (!selectedRole.groupId) throw new Error("groupId manquant");
const groupId = selectedRole.groupId;

const origin = window.location.origin;
let API_URL = customApiUrl || (origin.includes("localhost") ? origin.replace(":4200", ":3000")
: origin + "/api");

console.log(`□□API: ${API_URL} | Groupe: ${groupId}`);

// =========================
// 1. Récupération (Pagination)
// =========================
const pageSize = 100;
```

```
let page = 1;
let hasMorePages = true;
let allStations = [];

console.log("□□Récupération des bornes FAULTED...");

while (hasMorePages && !stopResetScriptFlag) {
  const url = `${API_URL}/charging-stations/group/${groupId}/list?offset=${(page - 1) *
pageSize}&limit=${pageSize}&status=faulted`;

  const res = await fetch(url, {
    headers: { Authorization: `Bearer ${token}`, "Content-Type": "application/json" }
  });

  if (!res.ok) throw new Error(`Erreur HTTP listing: ${res.status}`);

  const data = await res.json();
  const records = data.records || data;
  allStations = allStations.concat(records);

  const totalRecords = parseInt(res.headers.get("X-Total-Count") || "0");
  const totalPages = Math.ceil(totalRecords / pageSize);

  console.log(`□ Page ${page}/${totalPages || '?'} (+${records.length} bornes)`);

  if (records.length === 0 || page >= totalPages) {
    hasMorePages = false;
  } else {
    page++;
  }
}

// ========================
// 2. Filtrage & Préparation
// ========================
const stationsToprocess = allStations
  .map(s => ({ ...s, _stationId: getStationId(s) }))
  .filter(s => s._stationId != null);

console.log(`□□${stationsToprocess.length} bornes FAULTED prêtes.`);
```

```
if (stationsToprocess.length === 0) return console.warn("⚠ Rien à traiter.");

const confirmMessage =
  `⚠ ATTENTION : RESET MASSIF\n\n` +
  `Vous allez redémarrer (${RESET_TYPE}) ${stationsToprocess.length} bornes.\n` +
  `Cela va couper les charges en cours si elles existent.\n\n` +
  `Continuer ?`;

if (!confirm(confirmMessage)) {
  return console.log("❌ Opération annulée");
}

// =========================
// 3. Traitement Parallèle (Concurrency Queue)
// =========================
const CONCURRENCY_LIMIT = 10;
let finishedCount = 0;
let successCount = 0;
let errorCount = 0;

// Fonction worker
const processQueue = async () => {
    while (stationsToprocess.length > 0 && !stopResetScriptFlag) {
        const station = stationsToprocess.shift();
        const name = station.name || station.chargeboxIdentity || station._stationId;

        try {
            const controller = new AbortController();
            const timeoutId = setTimeout(() => controller.abort(), 10000);

            // --- CHANGEMENT VERS ROUTE RESET ---
            const res = await fetch(`${API_URL}/ocpp/reset/${station._stationId}`, {
                method: "POST",
                headers: { Authorization: `Bearer ${token}`, "Content-Type":
"application/json" },
                body: JSON.stringify({ type: RESET_TYPE }), // Payload Reset
                signal: controller.signal
            }).finally(() => clearTimeout(timeoutId));
```

```
            if (!res.ok) throw new Error(`HTTP ${res.status}`);

            const json = await res.json();
            const status = json?.status || json?.result?.status;

            if (String(status).toLowerCase() === "accepted") {
                successCount++;
                console.log(`□ [${++finishedCount}] ${name} : RESET SEND`);
            } else {
                throw new Error(`Refusé (${status})`);
            }
        } catch (err) {
            errorCount++;
            console.warn(`□ [${++finishedCount}] ${name} : ${err.message}`);
        }
    }
};

console.log(`□□Démarrage de ${CONCURRENCY_LIMIT} workers...`);

const workers = [];
for (let i = 0; i < CONCURRENCY_LIMIT; i++) {
    workers.push(processQueue());
}
await Promise.all(workers);

console.log("\n==============================");
console.log("□□TERMINÉ");
console.log(`□ Succès: ${successCount} | □ Erreurs: ${errorCount}`);
console.log("==============================");
```

} catch (err) { console.error("□ Crash:", err); } }

// Exposition globale window.triggerResetToFaultedStations = triggerResetToFaultedStations; window.stopResetScript = stopResetScript;

console.log(`□ Script RESET (${RESET_TYPE}) chargé`); console.log("➡ triggerResetToFaultedStations()");

# Nouvelle pageBootNotification aux bornes FAULTED

/**

- Script TURBO – Envoi BootNotification RAPIDE aux bornes FAULTED
- Optimisations :
- ○ Traitement en parallèle (par lots de 10)
- ○ Suppression des délais artificiels
- Utilisation :
- 1. Faire un "Refresh" (F5) de la page pour nettoyer l'ancien script
- 2. Coller ce nouveau script
- 3. Lancer : triggerBootNotificationToFaultedStations() */

let stopBootNotificationScriptFlag = false;

function stopBootNotificationScript() { stopBootNotificationScriptFlag = true; console.log("□□ Arrêt du script demandé..."); }

function getStationId(station) { return ( station.id ?? station.chargingStationId ?? station.stationId ?? station.idChargingStation ?? null ); }

async function triggerBootNotificationToFaultedStations(customApiUrl = null) { stopBootNotificationScriptFlag = false; console.log("□□ Envoi BootNotification TURBO aux bornes FAULTED");

try { // □□ Auth & Config const token = localStorage.getItem("token"); if (!token) throw new Error("Token non trouvé");

```
const selectedRole = JSON.parse(localStorage.getItem("selectedRole") || "{}");
if (!selectedRole.groupId) throw new Error("groupId manquant");
const groupId = selectedRole.groupId;

const origin = window.location.origin;
let API_URL = customApiUrl || (origin.includes("localhost") ? origin.replace(":4200", ":3000")
: origin + "/api");

console.log(`□□API: ${API_URL} | Groupe: ${groupId}`);

// ========================
```

```
// 1. Récupération (Pagination)
// ========================
const pageSize = 100; // Max souvent autorisé
let page = 1;
let hasMorePages = true;
let allStations = [];

console.log("□□Récupération des bornes FAULTED...");

while (hasMorePages && !stopBootNotificationScriptFlag) {
  const url = `${API_URL}/charging-stations/group/${groupId}/list?offset=${(page - 1) *
pageSize}&limit=${pageSize}&status=faulted`;

  const res = await fetch(url, {
    headers: { Authorization: `Bearer ${token}`, "Content-Type": "application/json" }
  });

  if (!res.ok) throw new Error(`Erreur HTTP listing: ${res.status}`);

  const data = await res.json();
  const records = data.records || data; // Gérer format {records: [], count: n} ou [records]
  allStations = allStations.concat(records);

  const totalRecords = parseInt(res.headers.get("X-Total-Count") || "0");
  const totalPages = Math.ceil(totalRecords / pageSize);

  console.log(`□ Page ${page}/${totalPages || '?'} (+${records.length} bornes)`);

  if (records.length === 0 || page >= totalPages) {
    hasMorePages = false;
  } else {
    page++;
  }
}

// ========================
// 2. Filtrage & Préparation
// ========================
// On garde uniquement celles avec un ID valide (le filtre status est fait par l'API)
const stationsToprocess = allStations
```

```
  .map(s => ({ ...s, _stationId: getStationId(s) }))
  .filter(s => s._stationId != null);

console.log(`□□${stationsToprocess.length} bornes FAULTED prêtes à être traitées.`);

if (stationsToprocess.length === 0) return console.warn("⚠□ Rien à traiter.");

if (!confirm(`□□GO pour envoyer BootNotification à ${stationsToprocess.length} bornes en MODE ACCÉLÉRÉ ?`)) {
  return console.log("□ Opération annulée");
}

// =========================
// 3. Traitement Parallèle (Concurrency Queue)
// =========================
const CONCURRENCY_LIMIT = 10; // Nombre de requêtes simultanées
let finishedCount = 0;
let successCount = 0;
let errorCount = 0;

// Fonction worker : consomme la liste tant qu'il y en a
const processQueue = async (workerId) => {
    while (stationsToprocess.length > 0 && !stopBootNotificationScriptFlag) {
        const station = stationsToprocess.shift(); // Prend le prochain élément
        const name = station.name || station.chargeboxIdentity || station._stationId;

        try {
            const controller = new AbortController();
            const timeoutId = setTimeout(() => controller.abort(), 10000); // 10s timeout par requête

            const res = await fetch(`${API_URL}/ocpp/trigger-message/${station._stationId}`, {
                method: "POST",
                headers: { Authorization: `Bearer ${token}`, "Content-Type": "application/json" },
                body: JSON.stringify({ command: "BootNotification", connector: 0 }),
                signal: controller.signal
            }).finally(() => clearTimeout(timeoutId));

            if (!res.ok) throw new Error(`HTTP ${res.status}`);
```

```
            const json = await res.json();
            const status = json?.result?.status || json?.status || json?.[0]?.result?.status;

            if (String(status).toLowerCase() === "accepted") {
                successCount++;
                console.log(`□ [${++finishedCount}] ${name}`);
            } else {
                throw new Error(`Refusé (${status})`);
            }
        } catch (err) {
            errorCount++;
            console.warn(`□ [${++finishedCount}] ${name} : ${err.message}`);
        }
    }
};

console.log(`□□Démarrage de ${CONCURRENCY_LIMIT} workers parallèles...`);

// Lancer les workers
const workers = [];
for (let i = 0; i < CONCURRENCY_LIMIT; i++) {
    workers.push(processQueue(i));
}

await Promise.all(workers);

console.log("\n==============================");
console.log("□□TERMINÉ");
console.log("==============================");
console.log(`□ Succès: ${successCount}`);
console.log(`□ Échecs: ${errorCount}`);
console.log("==============================");
```

} catch (err) { console.error("□ Erreur fatale:", err); } }

// Exposition globale window.triggerBootNotificationToFaultedStations = triggerBootNotificationToFaultedStations; window.stopBootNotificationScript = stopBootNotificationScript;

console.log("□ Script FAULTED TURBO chargé"); console.log("➡ triggerBootNotificationToFaultedStations()");