Docker - BROKER MQTT

Documentation des conteneurs docker déployé pourle projet SIGACS pour le fonctionnement du BROKER MQTT

Docker compose

docker-compose.yml

```
version: "3.9"

# ============================================================================
#  Parc – MQTT Bridge stack
# ============================================================================
#
#  Services:
#    mosquitto  – MQTT broker
#    bridge     – Python MQTT?MySQL bridge (this repo)
#
#  The MySQL/MariaDB server is expected to be external (already running).
#  All database credentials and the broker address are passed via environment
#  variables so no secret is baked into any image.
#
# ============================================================================

services:

  # ---------------------------------------------------------------------------
  # Mosquitto MQTT broker
  # ---------------------------------------------------------------------------
  mosquitto:
    image: eclipse-mosquitto:2
    container_name: mosquitto
    restart: unless-stopped
    volumes:
      - ./mosquitto/config/mosquitto.conf:/mosquitto/config/mosquitto.conf:ro
      - ./mosquitto/server-certs:/mosquitto/certs:ro
      - mosquitto-log:/mosquitto/log
    ports:
      - "${MQTT_EXPOSE_PORT:-8883}:8883"
    networks:
      - parc-net
    healthcheck:
      test: ["CMD-SHELL", "mosquitto_sub -t '$$SYS/broker/uptime' -C 1 -W 2 || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 5

  # ---------------------------------------------------------------------------
  # MQTT ? MySQL bridge
  # ---------------------------------------------------------------------------
  bridge:
    build:
      context: .
      dockerfile: Dockerfile
    image: parc-mqtt-bridge:latest
    container_name: parc-bridge
```

```
    restart: unless-stopped
    volumes:
    - ./mosquitto/server-certs/ca.crt:/certs/ca.crt:ro
    - ./mosquitto/client-certs/client.crt:/certs/client.crt:ro
    - ./mosquitto/client-certs/client.key:/certs/client.key:ro
    depends_on:
      mosquitto:
        condition: service_healthy
    environment:
      # --- MQTT ---
      MQTT_HOST:         mosquitto          # service name inside Docker network
      MQTT_PORT:         "8883"
      MQTT_KEEPALIVE:    "60"
      MQTT_USER:         ${MQTT_USER:-}
      MQTT_PASSWORD:     ${MQTT_PASSWORD:-}
      MQTT_RETRY_DELAY:  "5"

      # --- Database ---
      DB_HOST:           ${DB_HOST}
      DB_PORT:           ${DB_PORT:-3306}
      DB_NAME:           ${DB_NAME:-parc}
      DB_USER:           ${DB_USER}
      DB_PASSWORD:       ${DB_PASSWORD}
      DB_RETRY_DELAY:    "5"

      # --- Logging ---
      LOG_LEVEL:         ${LOG_LEVEL:-INFO}
    networks:
      - parc-net
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "5"

# ------------------------------------------------------------------------------
# Shared network
# ------------------------------------------------------------------------------
networks:
  parc-net:
    name: parc-net

# ------------------------------------------------------------------------------
# Volumes
# ------------------------------------------------------------------------------
volumes:
  mosquitto-log:
```

Mosquitto

mosquitto.conf

```
# Mosquitto configuration

# Plain listener - Partie 1
# utilisé uniquement par le healthcheck docker
listener 1883
allow_anonymous true

# TLS listener - Partie 2
listener 8883
protocol mqtt
allow_anonymous true
cafile   /mosquitto/certs/ca.crt
certfile /mosquitto/certs/server.crt
keyfile  /mosquitto/certs/server.key
require_certificate true

# Logging - Partie 3
log_dest file /mosquitto/log/mosquitto.log
log_dest stdout
log_type all
log_timestamp true
log_timestamp_format %Y-%m-%dT%H:%M:%S
connection_messages true
```

Le fichier de configuration est séparé en 3 parties

Partie	Explication
Partie 1	Paramètres des connexions non-sécurisées
Partie 2	Paramètres des connexions sécurisées
Partie 3	Gestions des paramètres des logs MQTT

Partie 1

Paramètre	configuration	Explication
`listener`	1883	défini le port d'écoute sur lequel la configuration suivante va s'appliquer
`allow_anonymous`	false	*Explication ci-dessous*

Partie 2

Paramètre	configuration	Explication
`listener`	8883	défini le port d'écoute sur lequel la configuration suivante va s'appliquer
`require_certificate`	false	autorise **uniquement** les connexions sécurisé sur le port du `listener` concerné (ici 8883)
`protocol`	mqtt	Défini le(s) protocole(s) en attente sur ce port
`allow_anonymous`	true	*Explication ci-dessous*
`cafile`	/mosquitto/certs/ca.crt	Chemin d'accès du certificat d'autorité du serveur, utilisé pour vérifier les clients
`certfile`	/mosquitto/certs/server.crt	Chemin d'accès du certificat public du serveur. Ce que Mosquitto présente aux clients lors de la connexion TLS
`keyfile`	/mosquitto/certs/server.key	Chemin d'accès de la clé privée du serveur

paramètre important Partie 1 & 2 :

`allow_anonymous`:
- `true` = mTLS — le client et le serveur prouvent leur identité mutuelement
- `false` = TLS simple — le serveur prouve son identité au client

`tls_version`:
- `tlsv1.2` = compatible ESP32 et programme Python

Partie 3

Paramètre	configuration	Explication
`log_dest`		Chemin de destination des logs

Application d'écriture en BDD

Parc – MQTT ? MySQL Bridge

Service Python qui fait le pont entre un broker MQTT Mosquitto et une base MySQL pour le système de monitoring de serres **SIGACS / Parc**.

Architecture

```
[ESP32 / M5Stack / autres contrôleurs]
        ?  mTLS port 8883
        ?
 ????????????????????         ??????????????????????????????
 ?   Mosquitto 2.x ? ?????????   bridge.py (Python 3.12) ? ???  MySQL (externe)
 ?   (broker)      ?         ?   subscribe + persist     ?
 ????????????????????         ??????????????????????????????
   Conteneur Docker                 Conteneur Docker
        réseau partagé parc-net
```

Les deux conteneurs partagent le réseau Docker `parc-net`. Le serveur MySQL est externe (conteneurisé sur le web_server).

Stack

Composant	Technologie
Broker MQTT	Eclipse Mosquitto 2.x (Docker)
Bridge	Python 3.12 (Docker)
Base de données	MySQL
Bibliothèque MQTT	paho-mqtt 2.1.0
Bibliothèque DB	PyMySQL 1.1.1
TLS	mTLS — CA auto-signé, certificats par client
Conteneurisation	Docker Compose

Topics MQTT

Mesures capteurs

Pattern	Description
`serre/<numero>/bac/<numero>`	Mesures du bac `<numero>` dans la serre `<numero>`

`<numero>` correspond à la colonne `numero` dans les tables `serre` et `bac` (pas la clé primaire).

Payload JSON :

```
{"humiditeAmbiante": 65.3, "temperatureAmbiante": 22.1, "humiditeSol": 45.0}
```

Plusieurs capteurs peuvent coexister dans un même payload.

Capteurs supportés :

Clé JSON	`id_capteur`	Unité	Min	Max
`humiditeAmbiante`	1	%	0	100
`humiditeSol`	2	%	0	100
`temperatureAmbiante`	3	°C	-40	80

Contrôleurs

Pattern	Direction	Déclencheur
`controleur/<MAC>`	Contrôleur → Broker	À chaque connexion du contrôleur
`controleur/<MAC>/disconnect`	Broker → Bridge	Déconnexion du contrôleur (LWT Mosquitto)

`<MAC>` est l'adresse MAC Wi-Fi du contrôleur au format `XX:XX:XX:XX:XX:XX`. Elle sert d'identifiant unique (`id_controleur`) en base de données.

Payload de connexion (`controleur/<MAC>`) :

```
{"ip": "192.168.42.85", "status": true}
```

Payload de déconnexion (`controleur/<MAC>/disconnect`) :

```
(payload vide)
```

Le bridge passe automatiquement `status = false` en base à la réception de ce message.

Gestion des erreurs

Toutes les erreurs sont persistées dans la table `error` :

`type_erreur`	**Déclencheur**
`BAC_NOT_FOUND`	Numéro de serre ou bac introuvable en base
`UNKNOWN_SENSOR`	Clé JSON absente de la liste des capteurs connus
`INVALID_PAYLOAD`	Erreur de décodage JSON, payload non-objet, ou champ manquant
`INVALID_VALUE`	Valeur non numérique pour un capteur, ou type de champ invalide
`VALUE_OUT_OF_RANGE`	Valeur clampée au min/max du capteur
`INVALID_ENCODING`	Payload non UTF-8
`MQTT_DISCONNECT`	Déconnexion inattendue du broker
`DB_ERROR`	Erreur transitoire de requête base de données
`DB_INSERT_ERROR`	Échec d'insertion en base
`UNEXPECTED_ERROR`	Exception non gérée (attrape-tout)

Quand une valeur est hors limites : la valeur clampée (min ou max) est écrite dans `mesure` **et** une erreur est insérée.

TLS / mTLS

Architecture des certificats

```
ESP32 / M5Stack / autres clients
      ?  mTLS
      ?
 Mosquitto :8883  ??? server.crt (CN=mosquitto, SAN=DNS:mosquitto, IP:x.x.x.x)
      ?
      ?  mTLS
      ?
 bridge.py ????? présente client.crt (CN=mqtt-bridge)
```

Fichiers de certificats

Fichier	Utilisé par	Monté à
`server-certs/ca.crt`	Mosquitto + bridge + clients ESP32	`/mosquitto/certs/ca.crt` et `/certs/ca.crt`
`server-certs/server.crt`	Mosquitto	`/mosquitto/certs/server.crt`
`server-certs/server.key`	Mosquitto	`/mosquitto/certs/server.key`
`client-certs/client.crt`	Bridge	`/certs/client.crt`
`client-certs/client.key`	Bridge	`/certs/client.key`

Exigences critiques

- Le certificat serveur **doit** avoir `CN=mosquitto` **et** `subjectAltName=DNS:mosquitto,IP:<IP_LAN>` (Python SSL ignore le CN et ne vérifie que le SAN — les ESP32 connectent via IP LAN)
- Chaque nouveau client (ESP32, M5Stack, service…) obtient son propre certificat signé par le même CA
- Aucun changement de configuration Mosquitto nécessaire pour de nouveaux clients
- Taille des clés : 2048 bits (compatible ESP32 / M5Stack)
- Format des clés : PKCS#1 (`-----BEGIN RSA PRIVATE KEY-----`) → Sur OpenSSL 3.x, utiliser `openssl genrsa -traditional` pour forcer ce format

`v3.ext` (requis pour la signature du certificat serveur)

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName = @alt_names

[alt_names]
DNS.1 = mosquitto
IP.1  = 192.168.x.x
```

Permissions

```
sudo chown 1883:1883 mosquitto/server-certs/server.key && chmod 600 mosquitto/server-certs/server.key
sudo chown 1001:1001 mosquitto/client-certs/client.key  && chmod 600 mosquitto/client-certs/client.key
```

Démarrage rapide

1. Prérequis

- Docker ≥ 24 avec le plugin Compose
- Serveur MySQL / MariaDB accessible avec le schéma `parc` appliqué
- Certificats générés (voir section TLS)

2. Configuration

```
cp .env.example .env
# Renseigner DB_HOST, DB_USER, DB_PASSWORD dans .env
```

3. Démarrage

```
docker compose up -d --build
```

4. Vérification

```
# Suivre les logs du bridge
docker compose logs -f bridge

# Tester une mesure capteur
docker run --rm --network parc-net eclipse-mosquitto:2 \
  mosquitto_pub -h mosquitto -p 8883 \
  -t "serre/1/bac/1" \
  -m '{"humiditeAmbiante": 65, "temperatureAmbiante": 22}'

# Tester une connexion contrôleur
docker run --rm --network parc-net eclipse-mosquitto:2 \
  mosquitto_pub -h mosquitto -p 8883 \
  -t "controleur/24:D7:EB:38:DC:38" \
  -m '{"ip": "192.168.42.85", "status": true}'
```

Variables d'environnement

Variable	Défaut	Description
`DB_HOST`	*(requis)*	Hôte MySQL
`DB_PORT`	`3306`	Port MySQL
`DB_NAME`	`parc`	Nom de la base
`DB_USER`	*(requis)*	Utilisateur MySQL
`DB_PASSWORD`	*(requis)*	Mot de passe MySQL
`MQTT_HOST`	`mosquitto`	Nom d'hôte du broker

Variable	Défaut	Description
`MQTT_PORT`	`8883`	Port du broker (TLS)
`MQTT_KEEPALIVE`	`60`	Keepalive MQTT (s)
`MQTT_USER`	*(vide)*	Utilisateur broker (optionnel)
`MQTT_PASSWORD`	*(vide)*	Mot de passe broker (optionnel)
`DB_RETRY_DELAY`	`5`	Secondes entre tentatives DB
`MQTT_RETRY_DELAY`	`5`	Secondes entre tentatives MQTT
`LOG_LEVEL`	`INFO`	`DEBUG` / `INFO` / `WARNING` / `ERROR`

Décisions de conception — bridge.py

- **Reconnexion DB** : `ensure_db()` appelé avant chaque requête, reconnecte transparemment en cas de coupure
- **Reconnexion MQTT** : `reconnect_delay_set(min=5, max=60)` + `loop_forever(retry_first_connection=True)`
- **Cache mémoire** : `_bac_cache` et `_capteur_cache` évitent les allers-retours DB sur chaque message ; invalidés à chaque reconnexion DB
- **Clampge de valeurs** : valeurs hors limites clampées au min/max, mesure insérée avec valeur clampée, erreur également insérée
- **Attrape-tout** : chaque callback `on_message` est enveloppé dans try/except pour qu'un seul message mal formé ne fasse jamais planter le bridge
- **Routage de topics** : `on_message()` route vers `process_controleur_message()`, `set_controleur_offline()` ou `process_message()` selon le topic reçu
- **LWT contrôleur** : le bridge écoute `controleur/+/disconnect` — Mosquitto publie ce message automatiquement si un client se déconnecte brutalement

---

Génération d'un nouveau certificat client

Pour chaque nouveau client (ESP32, M5Stack, service...) :

```
cd mosquitto

# Sur OpenSSL 3.x, utiliser -traditional pour forcer le format PKCS#1
openssl genrsa -traditional -out newclient.key 2048

openssl req -new -out newclient.csr -key newclient.key
# Common Name : nom descriptif du client (ex: m5-serre-1)
```

```
openssl x509 -req -in newclient.csr \
  -CA server-certs/ca.crt \
  -CAkey server-certs/ca.key \
  -CAcreateserial -out newclient.crt -days 720

mkdir client-certs-newclient
mv newclient.* client-certs-newclient/
```

Vérifier que le cert et la clé correspondent :

```
openssl x509 -noout -modulus -in newclient.crt | openssl md5
openssl rsa  -noout -modulus -in newclient.key | openssl md5
# Les deux hashes doivent être identiques
```

Commandes de debug fréquentes

```
# Suivre tous les logs
docker compose logs -f

# Logs d'un seul service
docker compose logs -f bridge
docker compose logs mosquitto

# Vérifier le SAN du certificat serveur
openssl x509 -in mosquitto/server-certs/server.crt -noout -ext subjectAltName

# Vérifier la chaîne de confiance du certificat serveur
openssl verify -CAfile mosquitto/server-certs/ca.crt mosquitto/server-certs/server.crt

# Vérifier la connexion TLS depuis le serveur
openssl s_client -connect 127.0.0.1:8883 -CAfile mosquitto/server-certs/ca.crt

# Vérifier correspondance cert / clé client
openssl x509 -noout -modulus -in mosquitto/client-certs/client.crt | openssl md5
openssl rsa  -noout -modulus -in mosquitto/client-certs/client.key | openssl md5

# Vérifier l'exposition des ports Docker
docker ps | grep mosquitto
```

Structure des fichiers

```
mqtt-bridge/
??? bridge.py                        # Programme Python principal
??? Dockerfile                       # Image du bridge
??? docker-compose.yml               # Définition du stack complet
??? requirements.txt                 # Dépendances Python
??? .env.example                     # Modèle de variables d'environnement
??? mosquitto/
```

```
??? config/
?   ??? mosquitto.conf            # Configuration du broker
??? server-certs/                 # CA + certificats serveur
?   ??? ca.crt
?   ??? ca.key
?   ??? server.crt
?   ??? server.key
??? client-certs/                 # Certificat client du bridge
?   ??? client.crt
?   ??? client.key
??? server-certs.sh               # Script de génération CA + serveur
??? clients-certs.sh              # Script de génération d'un cert client
??? v3.ext                        # Extension SAN pour la signature serveur
```

Intégration ESP32 / M5Stack

- Se connecter à l'**IP LAN du serveur hôte** (pas au nom de service Docker `mosquitto`)
- Le port 8883 doit être exposé sur l'hôte dans `docker-compose.yml`
- Chaque appareil obtient son propre certificat client (même CA, nouveau key+CSR+crt, CN différent)
- Utiliser des clés 2048 bits (les clés 4096 bits peuvent dépasser la RAM de l'ESP32)
- Format clé privée : PKCS#1 (`-----BEGIN RSA PRIVATE KEY-----`) — générer avec `-traditional` sur OpenSSL 3.x
- Embarquer les certificats avec la syntaxe `R"EOF(...)EOF"` et `PROGMEM` pour stocker en flash
- Configurer le **Last Will Testament** sur `controleur/<MAC>/disconnect` avant `connect()`
- Publier les infos contrôleur sur `controleur/<MAC>` immédiatement après chaque connexion réussie

Programme "pont" complet

```
"""
MQTT ? MySQL Bridge
====================
Subscribes to  serre/+/bac/+  and  controleur/+  topics on a Mosquitto broker
and persists every measurement and controller info into the `parc` database.

Environment variables (injected via docker-compose):
  MQTT_HOST          – hostname of the Mosquitto container  (default: mosquitto)
  MQTT_PORT          – broker port                          (default: 8883)
  MQTT_KEEPALIVE     – keepalive in seconds                 (default: 60)
  MQTT_USER          – optional broker username
  MQTT_PASSWORD      – optional broker password

  DB_HOST            – MySQL host                           (default: db)
  DB_PORT            – MySQL port                           (default: 3306)
  DB_NAME            – database name                        (default: parc)
  DB_USER            – database user                        (default: parc)
  DB_PASSWORD        – database password

  DB_RETRY_DELAY     – seconds between DB reconnection attempts (default: 5)
  MQTT_RETRY_DELAY   – seconds between MQTT reconnection attempts (default: 5)
  LOG_LEVEL          – DEBUG / INFO / WARNING / ERROR        (default: INFO)
"""

import json
import logging
import os
import re
import sys
import time
from datetime import datetime

import paho.mqtt.client as mqtt
import pymysql
import pymysql.cursors

# ---------------------------------------------------------------------------
# Logging setup
# ---------------------------------------------------------------------------
LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO").upper()
logging.basicConfig(
    stream=sys.stdout,
    level=getattr(logging, LOG_LEVEL, logging.INFO),
    format="%(asctime)s  %(levelname)-8s  %(name)s  %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger("mqtt-bridge")

# ---------------------------------------------------------------------------
# Configuration from environment
# ---------------------------------------------------------------------------
MQTT_HOST        = os.getenv("MQTT_HOST",      "mosquitto")
MQTT_PORT        = int(os.getenv("MQTT_PORT",  "8883"))
MQTT_KEEPALIVE   = int(os.getenv("MQTT_KEEPALIVE", "60"))
```

```
MQTT_USER        = os.getenv("MQTT_USER",       None)
MQTT_PASSWORD    = os.getenv("MQTT_PASSWORD",   None)

DB_HOST          = os.getenv("DB_HOST",     "db")
DB_PORT          = int(os.getenv("DB_PORT", "3306"))
DB_NAME          = os.getenv("DB_NAME",     "parc")
DB_USER          = os.getenv("DB_USER",     "parc")
DB_PASSWORD      = os.getenv("DB_PASSWORD", "")

DB_RETRY_DELAY   = int(os.getenv("DB_RETRY_DELAY",   "5"))
MQTT_RETRY_DELAY = int(os.getenv("MQTT_RETRY_DELAY", "5"))

# Fixed capteur IDs as specified in the project brief
CAPTEUR_IDS: dict[str, int] = {
    "humiditeAmbiante":    1,
    "humiditeSol":         2,
    "temperatureAmbiante": 3,
}

# Topic patterns
TOPIC_RE        = re.compile(r"^serre/(\d+)/bac/(\d+)$")
CONTROLEUR_RE   = re.compile(r"^controleur/([^/]+)$")
DISCONNECT_RE   = re.compile(r"^controleur/([^/]+)/disconnect$")

# ---------------------------------------------------------------------------
# Global DB connection
# ---------------------------------------------------------------------------
_db_conn: pymysql.connections.Connection | None = None

# ---------------------------------------------------------------------------
# DB helpers
# ---------------------------------------------------------------------------

def db_connect() -> pymysql.connections.Connection:
    """Connect to MySQL, retrying indefinitely on failure."""
    while True:
        try:
            conn = pymysql.connect(
                host=DB_HOST,
                port=DB_PORT,
                user=DB_USER,
                password=DB_PASSWORD,
                database=DB_NAME,
                charset="utf8mb4",
                autocommit=True,
                connect_timeout=10,
                cursorclass=pymysql.cursors.DictCursor,
            )
            logger.info("Connected to database %s@%s:%s/%s", DB_USER, DB_HOST, DB_PORT,
DB_NAME)
            return conn
        except pymysql.MySQLError as exc:
            logger.error("DB connection failed: %s – retrying in %ss", exc, DB_RETRY_DELAY)
            time.sleep(DB_RETRY_DELAY)


def ensure_db() -> pymysql.connections.Connection:
    """Return a live DB connection, reconnecting if necessary."""
    global _db_conn
    try:
```

```
        if _db_conn is None:
            raise pymysql.MySQLError("no connection yet")
        _db_conn.ping(reconnect=False)
    except pymysql.MySQLError:
        logger.warning("DB connection lost – reconnecting…")
        try:
            if _db_conn is not None:
                _db_conn.close()
        except Exception:
            pass
        _db_conn = db_connect()
        invalidate_caches()
    return _db_conn


def insert_error(type_erreur: str, message: str, valeur: str | None = None) -> None:
    """Write a row to the error table. Never raises – errors are only logged."""
    try:
        conn = ensure_db()
        with conn.cursor() as cur:
            cur.execute(
                "INSERT INTO error (type_erreur, message, valeur, erreur_a) "
                "VALUES (%s, %s, %s, %s)",
                (type_erreur[:64], message, valeur, datetime.now()),
            )
        logger.debug("Error logged ? [%s] %s", type_erreur, message)
    except Exception as exc:
        logger.critical("Could not write to error table: %s", exc)


def insert_mesure(bac_id: int, capteur_id: int, value: float) -> None:
    """Insert one measurement row."""
    conn = ensure_db()
    with conn.cursor() as cur:
        cur.execute(
            "INSERT INTO mesure (bac, value, mesure_a, capteur) VALUES (%s, %s, %s, %s)",
            (bac_id, value, datetime.now(), capteur_id),
        )


def upsert_controleur(mac: str, ip: str, status: bool) -> None:
    """
    Insert or update a controller row.
    The MAC address extracted from the topic is used as id_controleur (primary key).
    """
    conn = ensure_db()
    with conn.cursor() as cur:
        cur.execute(
            """
            INSERT INTO controleur (id_controleur, ip, status)
            VALUES (%s, %s, %s)
            ON DUPLICATE KEY UPDATE
                ip     = VALUES(ip),
                status = VALUES(status)
            """,
            (mac, ip, int(status)),
        )
    logger.info(
        "Contrôleur enregistré  mac=%s  ip=%s  status=%s",
        mac, ip, status,
```

```
    )


def set_controleur_offline(mac: str) -> None:
    """Set a controller status to false (offline)."""
    try:
        conn = ensure_db()
        with conn.cursor() as cur:
            cur.execute(
                "UPDATE controleur SET status = 0 WHERE id_controleur = %s",
                (mac,),
            )
        logger.info("Contrôleur %s passé hors ligne", mac)
    except Exception as exc:
        msg = f"Erreur DB lors de la mise hors ligne du contrôleur {mac}: {exc}"
        logger.error(msg)
        insert_error("DB_ERROR", msg, None)


def resolve_bac(serre_numero: int, bac_numero: int) -> int | None:
    """
    Return id_bac for (serre.numero=serre_numero, bac.numero=bac_numero).
    Returns None when no matching row is found.
    """
    conn = ensure_db()
    with conn.cursor() as cur:
        cur.execute(
            """
            SELECT b.id_bac
            FROM   bac  b
            JOIN   serre s ON s.id_serre = b.serre
            WHERE  s.numero = %s
              AND  b.numero = %s
            LIMIT  1
            """,
            (serre_numero, bac_numero),
        )
        row = cur.fetchone()
    return row["id_bac"] if row else None


def get_capteur_limits(capteur_id: int) -> tuple[float | None, float | None]:
    """
    Return (valeurMinCapteur, valeurMaxCapteur) for a given capteur ID.
    Returns (None, None) when the capteur is not found.
    """
    conn = ensure_db()
    with conn.cursor() as cur:
        cur.execute(
            "SELECT valeurMinCapteur, valeurMaxCapteur FROM capteur WHERE id_capteur = %s",
            (capteur_id,),
        )
        row = cur.fetchone()
    if row:
        return row["valeurMinCapteur"], row["valeurMaxCapteur"]
    return None, None

# ---------------------------------------------------------------------------
# Cache
# ---------------------------------------------------------------------------
```

```
_bac_cache:     dict[tuple[int, int], int | None]           = {}
_capteur_cache: dict[int, tuple[float | None, float | None]] = {}


def cached_resolve_bac(serre_numero: int, bac_numero: int) -> int | None:
    key = (serre_numero, bac_numero)
    if key not in _bac_cache:
        _bac_cache[key] = resolve_bac(serre_numero, bac_numero)
    return _bac_cache[key]


def cached_capteur_limits(capteur_id: int) -> tuple[float | None, float | None]:
    if capteur_id not in _capteur_cache:
        _capteur_cache[capteur_id] = get_capteur_limits(capteur_id)
    return _capteur_cache[capteur_id]


def invalidate_caches() -> None:
    """Called after a DB reconnection so stale entries don't linger."""
    _bac_cache.clear()
    _capteur_cache.clear()
    logger.debug("Caches invalidated after DB reconnect")

# ---------------------------------------------------------------------------
# Message processing — mesures
# ---------------------------------------------------------------------------

def process_message(topic: str, raw_payload: str) -> None:
    """
    Full processing pipeline for one sensor MQTT message.

    1. Validate topic format
    2. Resolve serre / bac from DB
    3. Parse JSON payload
    4. For each measurement in the payload:
        a. Validate sensor name
        b. Validate value type
        c. Clamp value if out of bounds (and raise an error)
        d. Persist to mesure table
    """
    logger.debug("RECV  topic=%s  payload=%s", topic, raw_payload)

    # 1 – topic validation
    match = TOPIC_RE.match(topic)
    if not match:
        logger.warning("Ignored non-matching topic: %s", topic)
        return

    serre_numero = int(match.group(1))
    bac_numero   = int(match.group(2))

    # 2 – bac resolution
    try:
        bac_id = cached_resolve_bac(serre_numero, bac_numero)
    except Exception as exc:
        logger.error("DB error resolving bac for topic %s: %s", topic, exc)
        insert_error(
            "DB_ERROR",
            f"Impossible de résoudre bac pour le topic {topic}: {exc}",
            raw_payload,
```

```
        )
        return

    if bac_id is None:
        msg = (
            f"Topic {topic} correspond à la serre n°{serre_numero} / bac n°{bac_numero} "
            f"qui n'existe pas dans la base de données."
        )
        logger.warning(msg)
        insert_error("BAC_NOT_FOUND", msg, raw_payload)
        return

    # 3 – payload parsing
    try:
        payload = json.loads(raw_payload)
        if not isinstance(payload, dict):
            raise ValueError("Le payload JSON n'est pas un objet")
    except (json.JSONDecodeError, ValueError) as exc:
        msg = f"Payload JSON invalide sur le topic {topic}: {exc}"
        logger.warning(msg)
        insert_error("INVALID_PAYLOAD", msg, raw_payload)
        return

    # 4 – iterate measurements
    for sensor_name, raw_value in payload.items():
        _process_single_measure(topic, bac_id, sensor_name, raw_value, raw_payload)


def _process_single_measure(
    topic: str,
    bac_id: int,
    sensor_name: str,
    raw_value,
    raw_payload: str,
) -> None:
    """Process one sensor key/value pair from a payload."""

    # 4a – validate sensor name
    capteur_id = CAPTEUR_IDS.get(sensor_name)
    if capteur_id is None:
        msg = (
            f"Capteur inconnu « {sensor_name} » reçu sur le topic {topic}. "
            f"Capteurs attendus : {list(CAPTEUR_IDS.keys())}."
        )
        logger.warning(msg)
        insert_error("UNKNOWN_SENSOR", msg, raw_payload)
        return

    # 4b – validate value type
    try:
        value = float(raw_value)
    except (TypeError, ValueError):
        msg = f"Valeur non numérique pour {sensor_name} sur {topic}: {raw_value!r}"
        logger.warning(msg)
        insert_error("INVALID_VALUE", msg, str(raw_value))
        return

    # 4c – clamp to capteur limits if needed
    try:
        v_min, v_max = cached_capteur_limits(capteur_id)
```

```
    except Exception as exc:
        logger.error("DB error fetching capteur limits for %s: %s", sensor_name, exc)
        insert_error(
            "DB_ERROR",
            f"Impossible de récupérer les limites du capteur {sensor_name}: {exc}",
            str(raw_value),
        )
        return

    clamped_value = value

    if v_min is not None and value < v_min:
        msg = (
            f"Valeur {value} pour {sensor_name} (id={capteur_id}) sur le topic {topic} "
            f"est inférieure au minimum autorisé ({v_min}). Valeur enregistrée : {v_min}."
        )
        logger.warning(msg)
        insert_error("VALUE_OUT_OF_RANGE", msg, str(value))
        clamped_value = v_min

    elif v_max is not None and value > v_max:
        msg = (
            f"Valeur {value} pour {sensor_name} (id={capteur_id}) sur le topic {topic} "
            f"est supérieure au maximum autorisé ({v_max}). Valeur enregistrée : {v_max}."
        )
        logger.warning(msg)
        insert_error("VALUE_OUT_OF_RANGE", msg, str(value))
        clamped_value = v_max

    # 4d – persist
    try:
        insert_mesure(bac_id, capteur_id, clamped_value)
        logger.info(
            "Mesure enregistrée  serre=%s  bac=%s (id=%s)  capteur=%s (id=%s)  valeur=%s",
            topic.split("/")[1],
            topic.split("/")[3],
            bac_id,
            sensor_name,
            capteur_id,
            clamped_value,
        )
    except Exception as exc:
        msg = f"Erreur DB lors de l'insertion de la mesure ({sensor_name}={value}) sur
{topic}: {exc}"
        logger.error(msg)
        insert_error("DB_INSERT_ERROR", msg, str(value))

# ---------------------------------------------------------------------------
# Message processing — contrôleurs
# ---------------------------------------------------------------------------

def process_controleur_message(topic: str, raw_payload: str) -> None:
    """
    Process a message received on controleur/<mac>.

    The MAC address from the topic IS the id_controleur (primary key).
    Expected payload: {"ip": "192.168.x.x", "status": true}
    """
    match = CONTROLEUR_RE.match(topic)
    if not match:
```

```
        return
    mac = match.group(1)

    # Parse JSON
    try:
        payload = json.loads(raw_payload)
        if not isinstance(payload, dict):
            raise ValueError("Le payload JSON n'est pas un objet")
    except (json.JSONDecodeError, ValueError) as exc:
        msg = f"Payload JSON invalide sur le topic {topic}: {exc}"
        logger.warning(msg)
        insert_error("INVALID_PAYLOAD", msg, raw_payload)
        return

    # Validate required fields
    missing = [k for k in ("ip", "status") if k not in payload]
    if missing:
        msg = f"Champs manquants dans le payload contrôleur {topic}: {missing}"
        logger.warning(msg)
        insert_error("INVALID_PAYLOAD", msg, raw_payload)
        return

    # Validate types
    try:
        ip     = str(payload["ip"])
        status = bool(payload["status"])
    except (TypeError, ValueError) as exc:
        msg = f"Type de champ invalide dans le payload contrôleur {topic}: {exc}"
        logger.warning(msg)
        insert_error("INVALID_VALUE", msg, raw_payload)
        return

    # Persist
    try:
        upsert_controleur(mac, ip, status)
    except Exception as exc:
        msg = f"Erreur DB lors de l'enregistrement du contrôleur {mac}: {exc}"
        logger.error(msg)
        insert_error("DB_INSERT_ERROR", msg, raw_payload)

# ---------------------------------------------------------------------------
# MQTT callbacks
# ---------------------------------------------------------------------------

def on_connect(client: mqtt.Client, userdata, flags, reason_code, properties=None):
    if reason_code == 0:
        logger.info("MQTT connecté à %s:%s", MQTT_HOST, MQTT_PORT)
        client.subscribe("serre/+/bac/+",           qos=1)
        client.subscribe("controleur/+",             qos=1)
        client.subscribe("controleur/+/disconnect",  qos=1)
        logger.info(
            "Abonné aux topics  serre/+/bac/+  |  controleur/+  |  controleur/+/disconnect"
        )
    else:
        logger.error("Échec de la connexion MQTT, code retour : %s", reason_code)


def on_disconnect(client: mqtt.Client, userdata, flags, reason_code, properties=None):
    if reason_code == 0:
        logger.info("Déconnexion MQTT propre")
```

```
    else:
        logger.warning(
            "Déconnexion MQTT inattendue (code=%s) – le client va retenter automatiquement",
            reason_code,
        )
        insert_error(
            "MQTT_DISCONNECT",
            f"Déconnexion MQTT inattendue (code={reason_code}). Reconnexion en cours…",
            None,
        )


def on_message(client: mqtt.Client, userdata, msg: mqtt.MQTTMessage):
    # Decode payload
    try:
        payload_str = msg.payload.decode("utf-8")
    except UnicodeDecodeError as exc:
        logger.warning("Payload non-UTF8 sur %s : %s", msg.topic, exc)
        insert_error(
            "INVALID_ENCODING",
            f"Payload non-UTF8 sur {msg.topic}: {exc}",
            repr(msg.payload),
        )
        return

    try:
        # Contrôleur déconnecté : controleur/<mac>/disconnect
        if DISCONNECT_RE.match(msg.topic):
            mac = msg.topic.split("/")[1]
            set_controleur_offline(mac)

        # Contrôleur connecté : controleur/<mac>
        elif CONTROLEUR_RE.match(msg.topic):
            process_controleur_message(msg.topic, payload_str)

        # Mesure capteur : serre/X/bac/Y
        else:
            process_message(msg.topic, payload_str)

    except Exception as exc:
        logger.exception(
            "Erreur inattendue lors du traitement du message %s: %s", msg.topic, exc
        )
        try:
            insert_error("UNEXPECTED_ERROR", str(exc), payload_str)
        except Exception:
            pass


def on_log(client, userdata, level, buf):
    if level == mqtt.MQTT_LOG_ERR:
        logger.debug("MQTT internal: %s", buf)

# ---------------------------------------------------------------------------
# Entry point
# ---------------------------------------------------------------------------

def main() -> None:
    logger.info("=== MQTT Bridge démarrage ===")
    logger.info(
```

```
        "Config MQTT : %s:%s  |  Config DB : %s@%s:%s/%s",
        MQTT_HOST, MQTT_PORT,
        DB_USER, DB_HOST, DB_PORT, DB_NAME,
    )

    # Initial DB connection – blocks until successful
    global _db_conn
    _db_conn = db_connect()

    # Pre-warm capteur cache
    for name, cid in CAPTEUR_IDS.items():
        limits = cached_capteur_limits(cid)
        logger.info("Capteur %-22s (id=%s) : min=%-6s max=%s", name, cid, limits[0],
limits[1])

    # MQTT client setup
    client = mqtt.Client(
        mqtt.CallbackAPIVersion.VERSION2,
        client_id="mqtt-bridge",
        clean_session=True,
    )
    client.on_connect    = on_connect
    client.on_disconnect = on_disconnect
    client.on_message    = on_message
    client.on_log        = on_log

    if MQTT_USER:
        client.username_pw_set(MQTT_USER, MQTT_PASSWORD)

    # TLS setup
    client.tls_set(
        ca_certs="/certs/ca.crt",
        certfile="/certs/client.crt",
        keyfile="/certs/client.key",
    )

    client.reconnect_delay_set(min_delay=MQTT_RETRY_DELAY, max_delay=60)

    # Connect loop – keep trying until the broker is up
    while True:
        try:
            logger.info("Connexion au broker MQTT %s:%s…", MQTT_HOST, MQTT_PORT)
            client.connect(MQTT_HOST, MQTT_PORT, MQTT_KEEPALIVE)
            break
        except (OSError, ConnectionRefusedError) as exc:
            logger.error(
                "Broker MQTT injoignable : %s – retry dans %ss", exc, MQTT_RETRY_DELAY
            )
            time.sleep(MQTT_RETRY_DELAY)

    logger.info("Entrée dans la boucle MQTT principale")
    client.loop_forever(retry_first_connection=True)


if __name__ == "__main__":
    main()
```