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()
```

Revision #1
Created 2026-05-22 07:07:48 UTC by Clément

Updated 2026-05-22 07:09:28 UTC by Clément