06/10/2025

Veronique Cloutier & Félix-Alexandre Morneau-Carrier

Cégep Limoilou

Projet – partie 3

Distributeur automatique de nourriture avec reconnaissance

Table des matières

[Introduction 1](#_Toc199407214)

[Schéma 2](#_Toc199407215)

[Interface et base de données 3](#_Toc199407216)

[Problèmes rencontrés 5](#_Toc199407217)

[Annexe 1 : Queries SQL pour la création des tables et l’insertion de données test 6](#_Toc199407218)

[Annexe 2 : Backend (Node.JS) 7](#_Toc199407219)

[Annexe 3 : Frontend (React) 14](#_Toc199407220)

[Annexe 4 : Code arduino 22](#_Toc199407221)

# Introduction

Ce projet vise à concevoir un distributeur de nourriture automatique et intelligent pour chats, basé sur les technologies IoT. L’objectif est de contrôler l’accès à la nourriture à l’aide de la reconnaissance RFID, afin que seuls les chats autorisés puissent manger. Le système détecte la présence de l’animal, vérifie son identité via une puce RFID, et ouvre automatiquement le compartiment à l’aide d’un servomoteur.

Le présent document est le troisième d’une série de quatre, et il concerne particulièrement la partie interface du projet. Seront donc présentés un schéma des connexions entre les différents éléments physiques, puis on présentera notre interface en la documentant, suite à quoi nous présenterons notre base de données. Finalement, les problèmes rencontrés seront discutés.

# Schéma

Une image contenant texte, capture d’écran, Police, diagramme

Le contenu généré par l’IA peut être incorrect.

**Légende :**

Flèches bleues = requêtes

Flèches orange = réponses

# Interface et base de données

Nous voulions que l’utilisateur puisse utiliser une interface facilement accessible en tout temps pour pouvoir changer la fréquence de distribution de la nourriture. Nous avons donc choisi de faire une application web, comme cela permet d’avoir accès via différentes machines (téléphone, pc, etc.) et à différents endroits (à la maison ou hors de la maison).

Également, nous avons souhaité que l’utilisateur puisse consulter les moments où son chat est allé manger à son bol.

Voici donc notre interface web qui présente ces 2 aspects :

Une image contenant texte, capture d’écran, Police, conception

Le contenu généré par l’IA peut être incorrect.

L’utilisateur peut donc incrémenter ou décrémenter de 30 minutes à la fois, en cliquant sur les boutons de chevrons, la fréquence de distribution de la nourriture. On bloque à 30 minutes minimum et 24 heures maximum. Ensuite il faut cliquer sur le bouton pour sauvegarder le changement.

Ensuite l’utilisateur voit un petit tableau de l’historique d’actionnement du servomoteur qu’il peut simplement consulter pour voir quand son chat a mangé.

Les informations consignées dans ce tableau proviennent de notre base de données, dans la table « ouverture\_distributeur », qui contient pour chaque entrée la date et le nombre d’ouvertures. On conserve une entrée par date, et à chaque ouverture le nombre va s’incrémenter dans la table.

On a aussi en BD une table « interval\_distributeur » qui contiendra toujours une seule entrée qui sera écrasée à chaque changement. On a un intervalle pour stocker la fréquence de distribution de la nourriture et une date de mise à jour pour savoir quand on a fixé cet intervalle.

Voir en Annexe 1 notre code SQL pour la création de nos tables en BD et l’insertion de données de test.

# Problèmes rencontrés

1. Problème : Avec l’ajout des requêtes HTTP dans le code arduino, nous manquions d’espace mémoire pour toutes nos variables donc on ne pouvait pas compiler.

Solution : On a déplacé une constante de la RAM à la PROGMEM pour laisser plus de place dans la RAM (stepSequence), car elle contient un tableau de tableaux ce qui semblait prendre pas mal de RAM.

1. Problème : On n’était pas en mesure de faire fonctionner le transfert de données via le ENC28J60 (pour la connexion à internet).

Solution : Après multiples essais, on a laissé tomber ce composant et on a fait notre connexion via le serial. Ça nécessite d’avoir le Arduino Uno branché dans un PC mais cela fonctionne bien.

1. Problème : On acait un problème d’affichage du nombre d’ouvertures dans notre tableau dans botre interface (mettons c’était initialement 1 et on en faisait une de plus, ça mettait 11, 111, etc. au lieu de 2, 3, 4...

Solution : Notre table SQL avait la colonne du nombre d’ouvertures en varchar au lieu de int donc ce n’était pas considéré comme un nombre. On a changé pour un int et ça fonctionne.

# Annexe 1 : Queries SQL pour la création des tables et l’insertion de données test

-- Table de configuration (1 seule entrée)

CREATE TABLE interval\_distributeur (

id\_int INT PRIMARY KEY,

interval\_ms\_int BIGINT NOT NULL,

updated\_at\_int DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

);

-- Table des logs d'ouvertures

CREATE TABLE ouverture\_distributeur (

id\_ouv INT AUTO\_INCREMENT PRIMARY KEY,

nb\_ouv VARCHAR(20),

date\_ouv DATE DEFAULT (CURRENT\_DATE),

UNIQUE (date\_ouv)

);

INSERT INTO interval\_distributeur (id\_int, interval\_ms\_int)

VALUES (1, 3600000);

INSERT INTO `object`.`ouverture\_distributeur`(`nb\_ouv`,`date\_ouv`)

VALUES (4, now());

# Annexe 2 : Backend (Node.JS)

// backend.js

// === IMPORT DES MODULES NECESSAIRES ===

// Framework web minimaliste pour créer des API REST

const express = require('express');

// Module HTTP natif de Node.js, utilisé pour créer un serveur

const http = require('http');

// WebSocket pour la communication bidirectionnelle en temps réel avec les clients

const WebSocket = require('ws');

// Middleware pour parser les données JSON envoyées via les requêtes HTTP

const bodyParser = require('body-parser');

// Client MySQL pour interagir avec la base de données

const mysql = require('mysql2');

// Middleware pour gérer les requêtes cross-origin (CORS)

const cors = require('cors');

// Module pour gérer la communication série avec un périphérique (ex: Arduino)

const { SerialPort, ReadlineParser } = require('serialport');

// === CONSTANTES DE CONFIGURATION ===

// Chemin vers le port série utilisé par l'Arduino

const SERIAL\_PORT\_PATH = 'COM18';

// Vitesse de communication série (doit correspondre à celle de l'Arduino)

const BAUD\_RATE = 9600;

// Informations de connexion à la base de données MySQL

const DB\_CONFIG = {

    host: 'localhost',

    user: 'root',

    password: '',

    database: 'object'

};

// === INITIALISATION DES SERVEURS ===

// Création d'une application Express

const app = express();

// Création d'un serveur HTTP basé sur Express

const server = http.createServer(app);

// Initialisation du serveur WebSocket attaché au serveur HTTP

const wss = new WebSocket.Server({ server });

// === GESTION DES CONNEXIONS WEBSOCKET ===

// Événement déclenché lorsqu’un client WebSocket se connecte

wss.on('connection', (ws) => {

    console.log('Client WebSocket connecté');

    // Événement déclenché à la déconnexion du client

    ws.on('close', () => {

        console.log('Client WebSocket déconnecté');

    });

    // Gestion des erreurs WebSocket

    ws.on('error', (err) => {

        console.error('Erreur WS côté serveur:', err);

    });

});

// Deuxième déclaration du même événement "connection" (fusionnable avec l'autre en réalité)

wss.on('connection', (ws) => {

    // Attribut pour vérifier que le client est actif

    ws.isAlive = true;

    // Quand un pong est reçu, on note que le client est encore vivant

    ws.on('pong', () => {

        ws.isAlive = true;

    });

});

// Intervalle régulier pour envoyer des "ping" aux clients et détecter les connexions inactives

const interval = setInterval(() => {

    wss.clients.forEach(ws => {

        if (!ws.isAlive) return ws.terminate(); // Si pas de réponse, fermeture de la connexion

        ws.isAlive = false; // Réinitialisation de l'état

        ws.ping(); // Envoi du ping

    });

}, 30000); // Toutes les 30 secondes

// Nettoyage du timer si le serveur WebSocket est fermé

wss.on('close', () => clearInterval(interval));

// === MIDDLEWARES EXPRESS ===

// Autoriser les requêtes cross-origin

app.use(cors());

// Parser le corps des requêtes en JSON

app.use(bodyParser.json());

// === CONNEXION A LA BASE DE DONNEES ===

// Connexion à la base de données MySQL

const db = mysql.createConnection(DB\_CONFIG);

db.connect(err => {

    if (err) console.error('Erreur MySQL :', err);

    else console.log('Connecté à MySQL');

});

// === BROADCAST VIA WEBSOCKET ===

// Fonction pour envoyer des données à tous les clients connectés en WebSocket

function broadcast(data) {

    const json = JSON.stringify(data); // Transformation des données en JSON

    wss.clients.forEach(client => {

        // Envoi uniquement aux clients dont la connexion est ouverte

        if (client.readyState === WebSocket.OPEN) {

            client.send(json);

        }

    });

}

// === COMMUNICATION SERIE AVEC L'ARDUINO ===

// Configuration du port série pour communiquer avec l’Arduino

const serialPort = new SerialPort({

    path: SERIAL\_PORT\_PATH,

    baudRate: BAUD\_RATE

});

// Parser pour découper les messages terminés par un saut de ligne

const parser = serialPort.pipe(new ReadlineParser({ delimiter: '\n' }));

// Événement déclenché quand le port série est ouvert

serialPort.on('open', () => {

    console.log('Connexion série avec Arduino ouverte');

});

// Événement en cas d'erreur de communication série

serialPort.on('error', (err) => {

    console.error('Erreur série :', err.message);

});

// Traitement des données reçues de l’Arduino

parser.on('data', (line) => {

    const cleaned = line.trim(); // Nettoyage des retours chariot / espaces

    console.log('Reçu de l’Arduino:', cleaned);

    // Si l’Arduino envoie une mise à jour de fréquence

    if (cleaned.startsWith('FREQ\_UPDATE:')) {

        const freqMs = parseInt(cleaned.substring('FREQ\_UPDATE:'.length).trim(), 10);

        if (!isNaN(freqMs)) {

            console.log('Nouvelle fréquence reçue depuis Arduino:', freqMs);

            updateIntervalInDB(freqMs, true); // Mise à jour en BDD + envoi WS

        }

    }

    // Si l’Arduino signale une ouverture

    else if (cleaned === 'LOG\_OUVERTURE') {

        console.log('Ouverture détectée – log en base...');

        addOuverture(true); // Log en base de données + notification WS

    }

});

// === FONCTIONS DE GESTION DES DONNEES ===

// Met à jour la fréquence dans la base de données (insert ou update)

// Peut également diffuser la mise à jour via WebSocket

function updateIntervalInDB(intervalMs, broadcastChange = false) {

    db.query(

        `INSERT INTO interval\_distributeur (id\_int, interval\_ms\_int)

         VALUES (1, ?) ON DUPLICATE KEY UPDATE interval\_ms\_int = VALUES(interval\_ms\_int)`,

        [intervalMs],

        (err) => {

            if (err) console.error("Erreur DB :", err);

            else {

                console.log("Intervalle mis à jour :", intervalMs);

                if (broadcastChange) {

                    broadcast({ type: 'update\_frequency', intervalMs });

                }

            }

        }

    );

}

// Enregistre ou met à jour le nombre d'ouvertures du jour

function addOuverture(broadcastChange = false) {

    const date = new Date().toISOString().slice(0, 10); // Format AAAA-MM-JJ

    // Vérifie si une entrée existe déjà pour aujourd'hui

    db.query(`SELECT \* FROM ouverture\_distributeur WHERE date\_ouv = ?`, [date], (err, results) => {

        if (err) return console.error("Erreur DB (SELECT):", err);

        if (results.length > 0) {

            // Mise à jour du nombre d'ouvertures

            const newNbOuv = results[0].nb\_ouv + 1;

            db.query(

                `UPDATE ouverture\_distributeur SET nb\_ouv = ? WHERE date\_ouv = ?`,

                [newNbOuv, date],

                (updateErr) => {

                    if (updateErr) return console.error("Erreur DB (UPDATE):", updateErr);

                    console.log(`Ouverture du ${date} mise à jour à ${newNbOuv}`);

                    if (broadcastChange) broadcast({ type: 'update\_ouverture', nb\_ouv: newNbOuv, date\_ouv: date });

                }

            );

        } else {

            // Première ouverture de la journée

            db.query(

                `INSERT INTO ouverture\_distributeur (date\_ouv, nb\_ouv) VALUES (?, ?)`,

                [date, 1],

                (insertErr) => {

                    if (insertErr) return console.error("Erreur DB (INSERT):", insertErr);

                    console.log(`Ouverture enregistrée pour la première fois à la date ${date}`);

                    if (broadcastChange) broadcast({ type: 'update\_ouverture', nb\_ouv: 1, date\_ouv: date });

                }

            );

        }

    });

}

// === ROUTES EXPRESS POUR LE FRONTEND ===

// Route POST pour définir une nouvelle fréquence (en minutes)

app.post('/set-frequency', (req, res) => {

    const { minutes } = req.body;

    // Validation de l'entrée

    if (typeof minutes !== 'number' || minutes <= 0 || minutes > 1440) {

        return res.status(400).json({ error: 'La fréquence doit être entre 1 et 1440 minutes.' });

    }

    const ms = minutes \* 60 \* 1000; // Conversion en millisecondes

    // Envoi de la nouvelle fréquence à l’Arduino via le port série

    serialPort.write(`FREQ:${ms}\n`, (err) => {

        if (err) {

            console.error("Erreur d'envoi série :", err);

            return res.status(500).send("Erreur d'envoi vers Arduino.");

        }

        // Mise à jour en BDD et diffusion WS

        updateIntervalInDB(ms, true);

        res.send("Fréquence mise à jour.");

    });

});

// Route GET pour obtenir la fréquence actuelle depuis la BDD

app.get('/current-frequency', (req, res) => {

    db.query("SELECT interval\_ms\_int FROM interval\_distributeur WHERE id\_int = 1", (err, results) => {

        if (err || results.length === 0) {

            return res.status(500).json({ error: "Non trouvé." });

        }

        res.json({ intervalMs: results[0].interval\_ms\_int });

    });

});

// Route GET pour obtenir les logs d’ouvertures triés par date

app.get('/logs-ouvertures', (req, res) => {

    db.query("SELECT \* FROM ouverture\_distributeur ORDER BY date\_ouv DESC", (err, results) => {

        if (err) return res.status(500).json({ error: "Erreur récupération." });

        // Formatage des dates avant envoi au client

        const formattedResults = results.map(row => ({

            ...row,

            date\_ouv: row.date\_ouv.toISOString().slice(0, 10)

        }));

        res.json(formattedResults);

    });

});

// === LANCEMENT DU SERVEUR ===

// Le serveur écoute sur le port 3001

server.listen(3001, () => {

    console.log(`Backend démarré sur http://localhost:3001`);

});

# Annexe 3 : Frontend (React)

// === IMPORT DES MODULES ===

import React, { useState, useEffect, useRef } from 'react'; // React + hooks

import axios from 'axios';                                  // Client HTTP pour faire des requêtes à l’API

import styles from './FrequenceControl.module.css';         // Import des styles CSS modulaires

// === COMPOSANT PRINCIPAL ===

function FrequenceControl() {

  // État local pour stocker le nombre total de minutes (fréquence d’alimentation)

  const [totalMinutes, setTotalMinutes] = useState(480); // 480 min = 8 heures (valeur par défaut)

  // État pour gérer les erreurs éventuelles

  const [error, setError] = useState(null);

  // État pour indiquer si la donnée est en cours de chargement

  const [loading, setLoading] = useState(true);

  // Référence au WebSocket, stockée dans un useRef (persistant entre rendus)

  const ws = useRef(null);

  // === useEffect #1 : Chargement initial + Connexion WebSocket ===

  useEffect(() => {

    // Appel API pour récupérer la fréquence actuelle depuis le backend

    axios.get('http://localhost:3001/current-frequency')

      .then(response => {

        // Conversion des millisecondes en minutes

        setTotalMinutes(response.data.intervalMs / 60000);

        setLoading(false); // Fin du chargement

      })

      .catch(err => {

        // Gestion d’erreur lors de la récupération des données

        setError('Erreur lors de la récupération de la fréquence');

        setLoading(false);

      });

    // === Initialisation du WebSocket ===

    ws.current = new WebSocket('ws://localhost:3001');

    // Quand la connexion WebSocket est établie

    ws.current.onopen = () => {

        console.log('Connexion WebSocket ouverte');

    };

    // Lors de la réception d’un message WebSocket

    ws.current.onmessage = (event) => {

      const message = JSON.parse(event.data); // On parse les données reçues

      // Si c’est une mise à jour de la fréquence

      if (message.type === 'update\_frequency') {

        setTotalMinutes(message.intervalMs / 60000); // Mise à jour de l’état

      }

    };

    // Gestion de la fermeture de la connexion WebSocket

    ws.current.onclose = () => {

        console.log('Connexion WebSocket fermée');

    };

    // Nettoyage lors du démontage du composant

    return () => {

      if (ws.current) {

        ws.current.close(); // Fermeture propre de la connexion WebSocket

      }

    };

  }, []); // Exécuté une seule fois au montage

  // === useEffect #2 : Tentative de reconnexion WebSocket (non finalisée ici) ===

  useEffect(() => {

    // Timer pour retenter une connexion après un délai

    const timer = setTimeout(() => {

      ws.current = new WebSocket('ws://localhost:3001');

    }, 500); // Délai de 500 ms

    // Nettoyage du timer au démontage

    return () => clearTimeout(timer);

  }, []); // Exécuté aussi au montage (peut créer un double appel)

  // === Fonctions pour augmenter ou diminuer la fréquence par pas de 30 minutes ===

  // Ajoute 30 minutes, sans dépasser 1440 min (24h)

  const incrementTime = () => {

    setTotalMinutes(prev => Math.min(prev + 30, 1440));

  };

  // Retire 30 minutes, sans descendre en dessous de 30 min

  const decrementTime = () => {

    setTotalMinutes(prev => Math.max(prev - 30, 30));

  };

  // Formate les minutes en "hh h mm min" (ex : 02 h 30 min)

  const formatTime = (minutes) => {

    const heures = Math.floor(minutes / 60);

    const mins = minutes % 60;

    return `${heures.toString().padStart(2, '0')} h ${mins.toString().padStart(2, '0')} min`;

  };

  // === Soumission du formulaire ===

  const handleSubmit = (event) => {

    event.preventDefault(); // Empêche le rechargement de la page

    // Envoi de la nouvelle fréquence au backend

    axios.post('http://localhost:3001/set-frequency', { minutes: totalMinutes })

      .then(() => {

        // Message de confirmation

        alert(`L'alimentation sera programmée pour ${formatTime(totalMinutes)} !`);

      })

      .catch(() => {

        // Gestion d’erreur si la requête échoue

        setError('Erreur lors de la mise à jour de la fréquence');

      });

  };

  // === Affichage conditionnel pendant le chargement ===

  if (loading) return <p>Chargement...</p>;

  // === Rendu JSX du composant ===

  return (

    <div className={styles.container}>

      {/\* Titre principal \*/}

      <h2 className={styles.heading}>Configuration de la fréquence d'alimentation</h2>

      {/\* Affichage de l’erreur s’il y en a une \*/}

      {error && <p className={styles.error}>{error}</p>}

      {/\* Formulaire de configuration \*/}

      <form className={styles.form} onSubmit={handleSubmit}>

        <div className={styles.timeSelector}>

          <label className={styles.label}>

            Fréquence :

            <div className={styles.buttonGroup}>

              {/\* Bouton pour diminuer \*/}

              <button type="button" onClick={decrementTime}>-</button>

              {/\* Affichage de l’heure formatée \*/}

              <span className={styles.timeValue}>{formatTime(totalMinutes)}</span>

              {/\* Bouton pour augmenter \*/}

              <button type="button" onClick={incrementTime}>+</button>

            </div>

          </label>

        </div>

        {/\* Bouton de soumission \*/}

        <button className={styles.button} type="submit">Enregistrer la fréquence</button>

      </form>

      {/\* Résumé de la configuration actuelle \*/}

      <p className={styles.paragraph}>

        L'alimentation sera distribuée à chaque {formatTime(totalMinutes)}.

      </p>

    </div>

  );

}

// === EXPORT DU COMPOSANT ===

export default FrequenceControl;

// === IMPORT DES MODULES REQUIS ===

import React, { useState, useEffect, useRef } from 'react';  // React + hooks

import axios from 'axios';                                   // Pour les requêtes HTTP

import styles from './OuverturesTable.module.css';           // Styles CSS modulaires

// === COMPOSANT PRINCIPAL ===

function OuverturesTable() {

  // État contenant les logs d'ouvertures (tableau de dates + nb d'ouvertures)

  const [logs, setLogs] = useState([]);

  // État pour stocker un éventuel message d'erreur

  const [error, setError] = useState('');

  // Référence persistante à l’objet WebSocket

  const ws = useRef(null);

  // === useEffect principal : récupération initiale + WebSocket ===

  useEffect(() => {

    // Fonction asynchrone pour récupérer les logs depuis l’API backend

    const fetchLogs = async () => {

      try {

        // Requête GET vers l'endpoint des logs

        const response = await axios.get('http://localhost:3001/logs-ouvertures');

        // Mise à jour de l'état avec les données reçues

        setLogs(response.data);

      } catch {

        // En cas d’échec, afficher une erreur

        setError('Erreur lors de la récupération des logs d\'ouverture');

      }

    };

    // Exécution de la fonction au montage

    fetchLogs();

    // === Connexion WebSocket à l’API backend ===

    ws.current = new WebSocket('ws://localhost:3001');

    // Événement déclenché quand la connexion WebSocket s'ouvre

    ws.current.onopen = () => {

        console.log('Connexion WebSocket ouverte');

    };

    // Événement déclenché lorsqu’un message est reçu via WebSocket

    ws.current.onmessage = (event) => {

      const message = JSON.parse(event.data); // On parse les données

      // Si le message indique une mise à jour d’ouverture

      if (message.type === 'update\_ouverture') {

        setLogs(prevLogs => {

          // Cherche si une entrée existe déjà pour cette date

          const idx = prevLogs.findIndex(log => log.date\_ouv === message.date\_ouv);

          if (idx !== -1) {

            // Si l’entrée existe, on met à jour le nombre d’ouvertures

            const updatedLogs = [...prevLogs];

            updatedLogs[idx].nb\_ouv = message.nb\_ouv;

            return updatedLogs;

          } else {

            // Sinon, on ajoute une nouvelle entrée en début de tableau

            return [{ date\_ouv: message.date\_ouv, nb\_ouv: message.nb\_ouv }, ...prevLogs];

          }

        });

      }

    };

    // Événement déclenché quand la connexion WebSocket se ferme

    ws.current.onclose = () => {

        console.log('Connexion WebSocket fermée');

    };

    // Nettoyage à la destruction du composant

    return () => {

      if (ws.current) {

        ws.current.close(); // Fermeture propre de la connexion

      }

    };

  }, []); // Ne s’exécute qu’au montage

  // === Deuxième effet : reconnexion WebSocket après délai (incomplet) ===

  useEffect(() => {

    // Timer de 500 ms pour déclencher une reconnexion

    const timer = setTimeout(() => {

      ws.current = new WebSocket('ws://localhost:3001');

    }, 500); // Délai en ms

    // Nettoyage du timer si le composant est démonté entre-temps

    return () => clearTimeout(timer);

  }, []); // Exécuté aussi une seule fois

  // === RENDU DU COMPOSANT ===

  return (

    <div className={styles.container}>

      {/\* Titre principal \*/}

      <h2 className={styles.heading}>Logs d'Ouverture</h2>

      {/\* Affichage d’un message d’erreur s’il y en a un \*/}

      {error && <div className={styles.error}>{error}</div>}

      {/\* Table HTML avec en-têtes et contenu dynamique \*/}

      <table className={styles.table}>

        <thead>

          <tr>

            <th>Date</th>

            <th>Nombre d'ouvertures</th>

          </tr>

        </thead>

        <tbody>

          {/\* Si des logs sont présents, on les affiche \*/}

          {logs.length > 0 ? (

            logs.map((log, i) => (

              <tr key={i}>

                <td>{log.date\_ouv}</td>

                <td>{log.nb\_ouv}</td>

              </tr>

            ))

          ) : (

            // Si aucun log trouvé, on affiche une ligne vide informative

            <tr><td colSpan="2">Aucun log trouvé</td></tr>

          )}

        </tbody>

      </table>

    </div>

  );

}

// === EXPORT DU COMPOSANT POUR UTILISATION EXTERNE ===

export default OuverturesTable;

# Annexe 4 : Code arduino

#include <SPI.h>      // Pour la communication SPI avec le module RFID

#include <MFRC522.h>  // Pour le lecteur RFID RC522

#include <Servo.h>    // Pour contrôler le servo moteur (couvercle)

#include <Wire.h>     // Pour la communication I2C (utilisée avec le module RTC)

#include <RTClib.h>   // Pour l’utilisation de l’horloge temps réel DS1307

// === Définition des broches utilisées ===

#define MOTOR\_STEP\_PIN 3    // Broche IN1 du moteur pas à pas

#define MOTOR\_DIR\_PIN 4     // Broche IN2 du moteur pas à pas

#define MOTOR\_IN3\_PIN 5     // Broche IN3 du moteur pas à pas

#define MOTOR\_IN4\_PIN 6     // Broche IN4 du moteur pas à pas (aussi utilisée pour le servo dans certains cas)

#define SERVO\_PIN 7         // Broche de signal pour contrôler le servo moteur

#define RFID\_RST\_PIN 9      // Broche RST pour le module RFID (Reset)

#define RFID\_SDA\_PIN 10     // Broche SDA (SS/CS) du module RFID (Chip Select)

#define BTN\_INC\_PIN A0      // Bouton connecté à A0 pour augmenter la fréquence

#define BTN\_DEC\_PIN A1      // Bouton connecté à A1 pour diminuer la fréquence

#define TRIG\_PIN A2         // Broche TRIG du capteur ultrason

#define ECHO\_PIN A3         // Broche ECHO du capteur ultrason

// === Constantes de configuration ===

#define STEPS\_PER\_REVOLUTION 2048  // Nombre de pas pour une rotation complète du moteur pas à pas

#define STEPPER\_SPEED 30           // Vitesse du moteur en tours par minute (non utilisée ici directement)

#define STEP\_DELAY 5               // Délai entre chaque pas du moteur (ms)

#define ANGLE\_OUVERT 90           // Angle pour ouvrir le couvercle avec le servo

#define ANGLE\_FERME 0             // Angle pour fermer le couvercle avec le servo

#define DEBOUNCE\_DELAY 50         // Délai anti-rebond pour les boutons (ms)

// === Création des objets ===

MFRC522 rfid(RFID\_SDA\_PIN, RFID\_RST\_PIN); // Instance du module RFID avec les broches définies

Servo servo;                              // Création du servo moteur

RTC\_DS1307 rtc;                           // Création du module RTC

// === Variables globales ===

bool estOuvert = false;                  // État du couvercle : false = fermé, true = ouvert

unsigned long lastMotorTime = 0;         // Temps de la dernière activation du moteur

unsigned long motorInterval = 8UL \* 60 \* 60 \* 1000; // Fréquence de distribution de nourriture (8h en ms)

unsigned long motorDuration = 10000;     // Durée de rotation du moteur (pas utilisée directement)

// Séquence de pas pour le moteur pas à pas (stockée en mémoire flash pour optimiser la RAM)

const byte stepSequence[8][4] PROGMEM = {

  { 1, 0, 0, 0 },

  { 1, 1, 0, 0 },

  { 0, 1, 0, 0 },

  { 0, 1, 1, 0 },

  { 0, 0, 1, 0 },

  { 0, 0, 1, 1 },

  { 0, 0, 0, 1 },

  { 1, 0, 0, 1 }

};

unsigned long lastDebounceTimeInc = 0;   // Dernier appui du bouton +

unsigned long lastDebounceTimeDec = 0;   // Dernier appui du bouton -

String expectedUID = "ac1ff21";          // UID autorisé (badge RFID valide)

// Déclaration des fonctions

void activerMoteur();

void tournerUnPas();

long mesurerDistance();

// === Fonction d'initialisation ===

void setup() {

  Serial.begin(9600);             // Démarre la communication série

  SPI.begin();                    // Initialise le bus SPI

  rfid.PCD\_Init();                // Initialise le module RFID

  if (!rfid.PCD\_PerformSelfTest()) {

    Serial.println("Test du module RFID échoué !");

  } else {

    Serial.println("Module RFID initialisé avec succès.");

  }

  servo.attach(SERVO\_PIN);       // Attache le servo à sa broche

  servo.write(ANGLE\_FERME);      // Met le servo en position fermée au démarrage

  pinMode(BTN\_INC\_PIN, INPUT\_PULLUP);  // Bouton + avec résistance de tirage interne

  pinMode(BTN\_DEC\_PIN, INPUT\_PULLUP);  // Bouton - avec résistance de tirage interne

  pinMode(MOTOR\_STEP\_PIN, OUTPUT);     // Broches du moteur en sortie

  pinMode(MOTOR\_DIR\_PIN, OUTPUT);

  pinMode(MOTOR\_IN3\_PIN, OUTPUT);

  pinMode(MOTOR\_IN4\_PIN, OUTPUT);

  pinMode(TRIG\_PIN, OUTPUT);     // Capteur ultrason : TRIG en sortie

  pinMode(ECHO\_PIN, INPUT);      // Capteur ultrason : ECHO en entrée

  Wire.begin();                  // Initialisation du bus I2C pour le RTC

  if (!rtc.begin()) {

    Serial.println("Impossible de trouver un RTC DS1307");

    while (1); // Arrêt du programme

  }

  if (!rtc.isrunning()) {

    Serial.println("RTC n'est pas en marche, mise à jour de l'heure");

    rtc.adjust(DateTime(F(\_\_DATE\_\_), F(\_\_TIME\_\_))); // Mise à l'heure automatique

  }

}

// === Boucle principale du programme ===

void loop() {

  DateTime now = rtc.now();  // Récupère l’heure actuelle (peut servir pour des logs)

  // --- Communication série pour changement de fréquence ---

  if (Serial.available()) {

    String input = Serial.readStringUntil('\n');  // Lire une ligne de texte depuis le port série

    input.trim();                                 // Retire les espaces inutiles

    if (input.startsWith("FREQ:")) {

      unsigned long newInterval = input.substring(5).toInt(); // Extrait la valeur numérique

      if (newInterval >= 30UL \* 60 \* 1000 && newInterval <= 24UL \* 60 \* 60 \* 1000) {

        motorInterval = newInterval;  // Applique la nouvelle fréquence

        Serial.print("Nouvel intervalle reçu : ");

        Serial.println(motorInterval);

      } else {

        Serial.println("Intervalle invalide reçu");

      }

    }

  }

  // --- Activation automatique du moteur ---

  if ((millis() - lastMotorTime >= motorInterval) && !estOuvert) {

    activerMoteur();  // Si assez de temps s’est écoulé, active le moteur

  }

  // --- Lecture RFID ---

  if (rfid.PICC\_IsNewCardPresent() && rfid.PICC\_ReadCardSerial()) {

    Serial.println("Puce RFID détectée !");

    if (!estOuvert) {

      String uid = "";

      for (byte i = 0; i < rfid.uid.size; i++) {

        uid += String(rfid.uid.uidByte[i], HEX);  // Concatène les octets en HEX

      }

      if (uid == expectedUID) {

        Serial.println("Puce autorisée détectée !");

        servo.write(ANGLE\_OUVERT);     // Ouvre le couvercle

        estOuvert = true;

        Serial.println("Couvercle ouvert.");

        Serial.println("LOG\_OUVERTURE"); // Message envoyé au backend

      } else {

        Serial.println("Puce non autorisée.");

      }

    }

    delay(1000);  // Pause pour éviter plusieurs lectures rapides

  }

  // --- Vérifie la fermeture du couvercle ---

  long distance = mesurerDistance();  // Mesure de la distance

  if (estOuvert && distance > 15) {

    servo.write(ANGLE\_FERME);   // Ferme si la gamelle est vide (plus d'objet proche)

    estOuvert = false;

    Serial.println("Couvercle fermé.");

    delay(1000);  // Laisse le temps au servo

  }

  // --- Bouton pour augmenter l’intervalle ---

  if (digitalRead(BTN\_INC\_PIN) == LOW && (millis() - lastDebounceTimeInc) > DEBOUNCE\_DELAY) {

    lastDebounceTimeInc = millis();

    if (motorInterval < 24.0 \* 60 \* 60 \* 1000) {  // Limite supérieure : 24h

      motorInterval += 30.0 \* 60 \* 1000;          // +30 minutes

      Serial.print("Intervalle augmenté à : ");

      Serial.print(motorInterval / 1000 / 60 / 60);  // Heures

      Serial.print("h");

      unsigned long minutes = motorInterval / 1000 / 60 % 60;

      if (minutes < 10) Serial.print("0");

      Serial.println(minutes);

      Serial.print("FREQ\_UPDATE:");

      Serial.println(motorInterval); // Envoi au backend

      delay(1000);

    }

  }

  // --- Bouton pour diminuer l’intervalle ---

  if (digitalRead(BTN\_DEC\_PIN) == LOW && (millis() - lastDebounceTimeDec) > DEBOUNCE\_DELAY) {

    lastDebounceTimeDec = millis();

    if (motorInterval > 30.0 \* 60 \* 1000) {  // Limite inférieure : 30 minutes

      motorInterval -= 30.0 \* 60 \* 1000;     // -30 minutes

      Serial.print("Intervalle réduit à : ");

      Serial.print(motorInterval / 1000 / 60 / 60);

      Serial.print("h");

      unsigned long minutes = motorInterval / 1000 / 60 % 60;

      if (minutes < 10) Serial.print("0");

      Serial.println(minutes);

      Serial.print("FREQ\_UPDATE:");

      Serial.println(motorInterval); // Envoi au backend

      delay(1000);

    }

  }

}

// === FONCTIONS ===

// Active le moteur pour une rotation complète

void activerMoteur() {

  Serial.println("Activation du moteur pas à pas...");

  lastMotorTime = millis();  // Mise à jour du temps

  for (int i = 0; i < STEPS\_PER\_REVOLUTION; i++) {

    tournerUnPas();          // Effectue un pas

    delay(STEP\_DELAY);       // Pause entre les pas

  }

  Serial.println("Moteur désactivé.");

}

// Effectue un seul pas du moteur

void tournerUnPas() {

  static int stepIndex = 0;  // Indice du pas courant

  digitalWrite(MOTOR\_STEP\_PIN, pgm\_read\_byte(&(stepSequence[stepIndex][0])));

  digitalWrite(MOTOR\_DIR\_PIN, pgm\_read\_byte(&(stepSequence[stepIndex][1])));

  digitalWrite(MOTOR\_IN3\_PIN, pgm\_read\_byte(&(stepSequence[stepIndex][2])));

  digitalWrite(MOTOR\_IN4\_PIN, pgm\_read\_byte(&(stepSequence[stepIndex][3])));

  stepIndex = (stepIndex + 1) % 8;  // Étape suivante

}

// Mesure la distance en cm avec le capteur ultrason

long mesurerDistance() {

  digitalWrite(TRIG\_PIN, LOW);

  delayMicroseconds(2);

  digitalWrite(TRIG\_PIN, HIGH);     // Déclenche un pulse

  delayMicroseconds(10);

  digitalWrite(TRIG\_PIN, LOW);

  long duration = pulseIn(ECHO\_PIN, HIGH);  // Temps aller-retour

  long distance = duration \* 0.034 / 2;     // Convertit en cm

  return distance;

}