Institut National de Radioélectricité et de Cinématographie

Enseignement technique secondaire de qualification

Accès aux études supérieures

Avenue Jupiter, 188

1190 Forest

Casier Personnel Connecté

SmartLocker

Projet personnel de Lamrine Ismail

Pour l'obtention du certificat de qualification

Technicien(ne) en informatique

Année scolaire : 2025-2026

Remerciements

Je tiens à exprimer mes plus sincères remerciements à l'ensemble des personnes qui ont contribué, de près ou de loin, à la réalisation de ce travail de fin d'études.

Je remercie en premier lieu l'équipe enseignante de l'Institut National de Radioélectricité et de Cinématographie (INRACI), et tout particulièrement mes professeurs d'informatique, pour la qualité de l'enseignement dispensé tout au long de ma formation en 6TQ Informatique. Leurs conseils techniques et pédagogiques m'ont permis d'acquérir les compétences indispensables à la conception et au développement de ce projet.

Je remercie également mon promoteur, qui a encadré ce travail et qui m'a guidé dans la structuration de mes idées, dans le choix des technologies et dans la rédaction de ce travail.

Mes remerciements s'adressent aussi à Madame Jeanjean, professeure de français, pour ses conseils relatifs à la rédaction et à la mise en forme du dossier écrit.

Je remercie enfin ma famille et mes proches pour leur soutien constant, leur patience et leurs encouragements tout au long de cette année scolaire.

Table des matières

1. Introduction

1.1 Contexte

Dans de nombreux établissements scolaires, salles de sport et espaces publics, la gestion des casiers repose encore aujourd'hui sur des systèmes traditionnels : cadenas mécaniques, clés physiques ou codes PIN figés. Ces solutions présentent des limites importantes : perte fréquente des clés, absence de traçabilité des accès, gestion administrative entièrement manuelle et impossibilité de s'intégrer aux moyens de paiement numériques.

C'est dans ce contexte que j'ai choisi de développer SmartLocker dans le cadre de mon travail de fin d'études. Mon objectif était de concevoir une application web moderne permettant de gérer des casiers connectés depuis un smartphone, en conjuguant sécurité renforcée, interface intuitive et fonctionnalités avancées telles que le paiement en ligne et la surveillance physique en temps réel.

Ce projet s'inscrit dans le prolongement de ma formation en informatique (6TQ) et m'a permis de mettre en pratique l'ensemble des compétences acquises au cours de ces six années. L'application a été développée puis mise en production sur une infrastructure réelle : trois Raspberry Pi (deux Pi Zero W équipant chacun un casier physique — un casier École et un casier Salle de sport — et un Pi 4B faisant office de passerelle), un serveur web Flask hébergé sur PythonAnywhere et une base de données Firebase synchronisée en temps réel.

1.2 Problématique

Comment concevoir et développer une application web connectée de gestion de casiers, intégrant un matériel physique réel (cartes Raspberry Pi, modules relais, capteurs), une communication en temps réel entre l'application et le matériel via les protocoles MQTT et Firebase, ainsi que des fonctionnalités avancées telles que le paiement par carte bancaire, l'authentification renforcée et les alertes d'intrusion, tout en offrant une expérience utilisateur fluide sur mobile ?

Cette problématique soulève plusieurs défis techniques complémentaires :

- Assurer une communication fiable et en temps réel entre l'interface web (hébergée sur Internet) et le matériel physique (situé sur un réseau local) ;
- Sécuriser l'accès aux casiers tout en garantissant une utilisation simple depuis un smartphone ;
- Intégrer un système de paiement et de réservation par créneaux horaires pour le casier scolaire ;
- Détecter une tentative d'intrusion physique et y réagir automatiquement ;
- Différencier plusieurs profils d'utilisateurs (standard, premium et administrateur).

1.3 Objectifs du TFE

Les objectifs poursuivis dans le cadre de ce travail de fin d'études sont les suivants :

- Développer une application web Flask complète, de l'authentification jusqu'au déploiement en production sur PythonAnywhere ;
- Construire l'infrastructure matérielle des deux casiers : câblage de deux modules relais, de deux capteurs magnétiques de porte, de deux capteurs ultrasoniques HC-SR04 et de deux lecteurs RFID RC522 sur deux Raspberry Pi Zero W ;
- Mettre en place une architecture de communication MQTT entre les deux casiers physiques (Pi Zero W École et Pi Zero W Sport) et la passerelle (Pi 4B), chargée de relayer les commandes vers Firebase ;
- Intégrer Stripe Checkout pour le paiement à la réservation du casier scolaire, avec gestion des créneaux horaires et protection contre la double réservation ;
- Implémenter un système de sécurité physique : détection d'intrusion, mode lockdown automatique de cinq minutes et notifications push via Firebase Cloud Messaging ;
- Déployer l'application sous la forme d'une Progressive Web App (PWA) installable sur smartphone et dotée d'un mode hors ligne ;

1.4 Présentation rapide de la solution

SmartLocker est une application web progressive (PWA) reposant sur une architecture à plusieurs niveaux. Le premier niveau correspond à l'interface web, développée en Flask/Python avec le moteur de gabarits Jinja2, hébergée sur PythonAnywhere et accessible à l'adresse smartlocker.pythonanywhere.com. Elle prend en charge l'authentification (e-mail/mot de passe et Google OAuth), les réservations, les paiements Stripe ainsi que l'affichage en temps réel de l'état des casiers.

Le deuxième niveau est assuré par Firebase, qui joue à la fois le rôle de base de données en temps réel et de bus de messages entre l'interface web et le matériel. L'état de chaque casier (status, active_session, door_physical, lockdown_until, etc.) y est stocké au format JSON et synchronisé automatiquement sur tous les clients connectés.

Le troisième niveau correspond à l'infrastructure physique : deux Raspberry Pi Zero W équipent respectivement le casier École (programme Casier1.py, LOCKER_ID = ECOLE_01) et le casier Salle de sport (programme Casier2.py, LOCKER_ID = SPORT_01). Chacun pilote directement son propre relais, interroge ses capteurs et publie ses événements via MQTT. Un Raspberry Pi 4B exécute bridge_firebase.py, la passerelle commune qui assure le lien entre le réseau local MQTT et Firebase (Internet).

L'application propose deux types de casiers : le casier École (réservation payante par créneaux horaires, au tarif de 5,00 €/heure) et le casier Salle de sport (accès continu, réservé aux membres Premium).

2. Analyse des besoins

2.1 Description du problème

Les systèmes de casiers classiques présentent plusieurs limites majeures. La sécurité repose sur un cadenas ou une clé que l'utilisateur peut aisément perdre ou oublier. Aucune traçabilité n'est assurée : il est impossible de savoir qui a utilisé un casier, à quel moment et pendant combien de temps. La gestion des paiements (location ponctuelle) demeure manuelle et sujette aux erreurs. Enfin, en cas d'effraction ou d'anomalie, aucune alerte n'est transmise au propriétaire du casier.

SmartLocker répond à ces problèmes en connectant les casiers physiques à Internet et en centralisant leur gestion dans une application web accessible depuis n'importe quel smartphone. Chaque accès est authentifié, tracé et, le cas échéant, réglé en ligne. Une alerte d'intrusion déclenche automatiquement un mode lockdown ainsi qu'une notification push sur le téléphone de l'utilisateur concerné.

2.2 Public cible

L'application s'adresse à plusieurs profils d'utilisateurs :

- Les élèves et les membres d'une salle de sport souhaitant accéder à leur casier simplement depuis leur smartphone, sans clé physique ;
- Les utilisateurs occasionnels souhaitant réserver un casier scolaire pour quelques heures, sans abonnement ;
- Les membres Premium bénéficiant d'un accès libre au casier de salle de sport ;
- Les administrateurs (équipe de direction) chargés de superviser l'ensemble des casiers, de gérer les utilisateurs, d'activer ou de désactiver les abonnements et de consulter les revenus générés.

2.3 Cahier des charges

2.3.1 Exigences fonctionnelles

- Inscription et connexion par e-mail/mot de passe ou via un compte Google (OAuth).
- Système d'appairage des casiers : chaque utilisateur scanne un QR code ou saisit un code inscrit sur le casier (par exemple SL-ECOLE-01) afin d'associer celui-ci à son compte, dans la limite d'un seul casier par utilisateur.
- Casier École : réservation par créneaux horaires (de 8 h à 22 h), paiement en ligne via Stripe Checkout (5,00 €/heure) et prolongation d'une heure possible lorsque la session est terminée depuis moins de trente minutes.
- Casier Salle de sport : ouverture et fermeture manuelles réservées aux membres Premium, avec une session illimitée jusqu'à la fermeture manuelle.
- Minuterie de session affichée en temps réel et verrouillage automatique à l'expiration.
- Historique des sessions, assorti d'un export au format CSV.
- Détection d'intrusion physique, mode lockdown automatique de cinq minutes et notification push via Firebase Cloud Messaging (FCM).
- Panneau d'administration : liste des utilisateurs, suivi des revenus, gestion des abonnements et dissociation forcée des casiers.
- Progressive Web App installable sur smartphone, dotée d'un mode hors ligne affichant l'écran « SIGNAL PERDU ».
- Photo de profil avec recadrage, ainsi que modification du pseudo, de l'adresse e-mail et du mot de passe.
- Interface mobile-first et responsive, avec un temps de chargement inférieur à trois secondes.
- Communication chiffrée en HTTPS obligatoire (PythonAnywhere fournit automatiquement un certificat SSL).
- Disponibilité 24 h/24 et 7 j/7.
- Protection contre la double réservation, grâce à un traitement idempotent des paiements Stripe côté serveur.
- Compatibilité avec Chrome, Firefox, Safari et Samsung Internet, sous Android comme sous iOS.

2.4 Contraintes

2.4.1 Contraintes techniques

Le projet devait fonctionner avec du matériel abordable (Raspberry Pi) et des services gratuits ou à faible coût. PythonAnywhere offre un hébergement Flask gratuit ; Firebase propose un palier gratuit suffisant pour la démonstration ; Stripe fonctionne en mode test (carte 4242 4242 4242 4242), sans frais réels.

La communication entre un Pi Zero (casier) et la passerelle (Pi 4B) s'effectue via MQTT sur le réseau local, ce qui impose que les deux machines se trouvent sur le même réseau. Le Pi 4B doit par ailleurs disposer d'une connexion Internet stable afin de synchroniser Firebase.

2.4.2 Contraintes de temps

Le développement s'est étalé sur l'ensemble de l'année scolaire, en parallèle des cours. L'architecture a évolué de façon répétitive : un démarrage avec une interface simple, suivi de l'ajout progressif des fonctionnalités (paiement, matériel physique, sécurité, puis panneau d'administration).

2.4.3 Contraintes matérielles

Le matériel disponible comprend deux Raspberry Pi Zero W, un Raspberry Pi 4B, deux modules relais 5 V, deux capteurs magnétiques de porte (à contact), deux capteurs ultrasoniques HC-SR04 et deux lecteurs RFID RC522. L'alimentation des cartes est assurée par des chargeurs micro-USB standard.

Le budget total du matériel acquis pour le projet est synthétisé dans le tableau ci-dessous :

Composant	Quantité	Prix	Provenance
Câble GPIO	1	9,98 €	Achat personnel
Module relais	2	3,38 €	Achat personnel
Casier physique	2	4,62 €	Achat personnel
Raspberry Pi 4B	1	79,99 €	Achat personnel
Raspberry Pi Zero W	2	—	Prêt de l'école
Capteur ultrasonique HC-SR04	2	—	Prêt de l'école
Lecteur RFID RC522	2	—	Fourni par l'école

3. Étude préalable

3.1 Solutions existantes

Avant de me lancer dans le développement, j'ai cherché à recenser les solutions déjà disponibles sur le marché, afin d'en cerner les forces, les faiblesses et la valeur ajoutée que pouvait apporter SmartLocker.

3.1.1 Casiers mécaniques classiques

Les casiers à cadenas ou à clé constituent la solution la plus répandue dans les établissements scolaires belges. Peu coûteux à l'achat, ils ne requièrent aucune infrastructure numérique. Leurs inconvénients sont toutefois nombreux : pertes de clés fréquentes, absence de traçabilité, gestion manuelle des attributions et impossibilité d'émettre une alerte en cas d'effraction.

3.1.2 Solutions connectées professionnelles

Des entreprises telles que Quadient (ex-Neopost) ou Ojmar proposent des casiers connectés dotés d'interfaces RFID, d'une gestion centralisée et de rapports d'utilisation. Ces solutions sont robustes et bien documentées, mais leur coût est prohibitif (plusieurs centaines d'euros par casier) et elles reposent sur un matériel propriétaire fermé.

3.1.3 Applications mobiles de salles de sport

Des enseignes comme Basic-Fit mettent des casiers à la disposition de leurs membres, fermés par des cadenas physiques (à clé ou à code) que l'utilisateur apporte ou achète sur place. L'application mobile de l'enseigne sert à la gestion de l'abonnement et au suivi d'entraînement, mais ne gère pas l'ouverture ni l'attribution des casiers. L'accès aux installations est conditionné à l'abonnement et le système reste fermé, ce qui le rend difficilement transposable à d'autres contextes (école, gare, espace de coworking).

3.1.4 Solutions DIY

Plusieurs projets open source disponibles sur GitHub proposent des gestionnaires de casiers ou de contrôle d'accès fondés sur des cartes Raspberry Pi. On peut citer *rmcreyes/smart-lock* (système de casier par compte utilisateur via NFC), *chdsbd/rpi-lock* (ouverture par carte RFID avec interface web de gestion des logs) ou *rick1924/RFIDAccessSystem* (contrôle d'accès RFID couplé à un système de réservation en ligne). Ces projets sont intéressants sur le plan technique et librement réutilisables, mais ils demeurent le plus souvent des preuves de concept : ils

n'intègrent ni interface utilisateur aboutie, ni système de paiement, ni authentification sécurisée de niveau production.

3.2 Comparaison des solutions

Le tableau suivant synthétise les principaux critères de comparaison :

Critère	Classique	Solution pro	App mobile	SmartLocker
Coût	Très faible	Très élevé	Moyen	Faible
Traçabilité	Non	Oui	Partielle	Oui
Accessible web	Non	Partielle	Non	Oui
Paiement intégré	Non	Oui	Oui	Oui
Alerte intrusion	Non	Oui	Non	Oui
Open source	N/A	Non	Non	Oui
Matériel abordable	Oui	Non	Non	Oui

3.3 Choix technologiques et justifications

3.3.1 Backend : Flask (Python)

Figure 1 : Logo Flask (micro-framework Python)

Flask est un micro-framework web Python, léger et flexible, parfaitement adapté à un projet de taille moyenne tel que SmartLocker. Il n'impose pas de structure rigide et autorise l'ajout progressif des fonctionnalités. Sa documentation est de grande qualité et Python est le langage que je maîtrise le mieux. Les alternatives Django (trop lourd pour ce besoin) et Node.js/Express ont été écartées.

3.3.2 Base de données temps réel : Firebase Realtime Database

Figure 2 : Logo Firebase (Google)

Firebase Realtime Database stocke les données au format JSON et synchronise automatiquement tous les clients connectés en quelques millisecondes. Cette réactivité est essentielle pour que l'état physique de la porte — mis à jour par le Pi Zero — soit visible quasi instantanément sur l'interface web. Firebase Authentication prend par ailleurs en charge l'authentification (e-mail/mot de passe et Google OAuth) de manière sécurisée, sans que j'aie à gérer moi-même le stockage des mots de passe.

3.3.3 Communication matériel : MQTT

MQTT (Message Queuing Telemetry Transport) est un protocole de messagerie léger, conçu pour les objets connectés aux ressources limitées. Il repose sur le modèle de publication/abonnement (publish/subscribe) : le Pi Zero publie ses événements (état de la porte, alertes) sur des topics, auxquels la passerelle s'abonne pour les transmettre à Firebase. Le choix de MQTT, de préférence à des appels HTTP directs émis depuis le Pi Zero, permet de découpler le casier de la connexion Internet et de garantir la livraison des messages même en cas de latence réseau (qualité de service QoS 1). La bibliothèque Paho-MQTT est employée sur les deux Pi Zero, tandis que le broker MQTT (Mosquitto) s'exécute sur le Pi 4B.

3.3.4 Paiement : Stripe Checkout

Stripe est une référence en matière de paiement en ligne pour les développeurs. Son API est très bien documentée, son mode test permet de simuler des paiements à l'aide de cartes fictives (4242 4242 4242 4242) et sa page de paiement préconstruite (Stripe Checkout) réduit considérablement le temps de développement. Les métadonnées de la session Stripe (date, heure, durée, identifiant utilisateur) permettent d'appliquer la réservation côté serveur de manière idempotente.

3.3.5 Notifications push: Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging (FCM) permet d'envoyer des notifications push sur le smartphone des utilisateurs, y compris lorsque l'application n'est pas ouverte. Lorsqu'une alerte d'intrusion est détectée par le Pi Zero, la passerelle transmet une notification push, via FCM, à l'utilisateur dont le casier est concerné. Le Service Worker dédié de la PWA réceptionne alors la notification et l'affiche.

3.3.6 Hébergement : PythonAnywhere

Figure 3 : Logo PythonAnywhere

PythonAnywhere propose un hébergement Python/Flask assorti d'un domaine en pythonanywhere.com, d'un certificat HTTPS automatique et d'une interface d'administration simple. Son offre gratuite est suffisante pour la démonstration du TFE et la mise en ligne d'une mise à jour s'effectue en quelques clics depuis l'interface web.

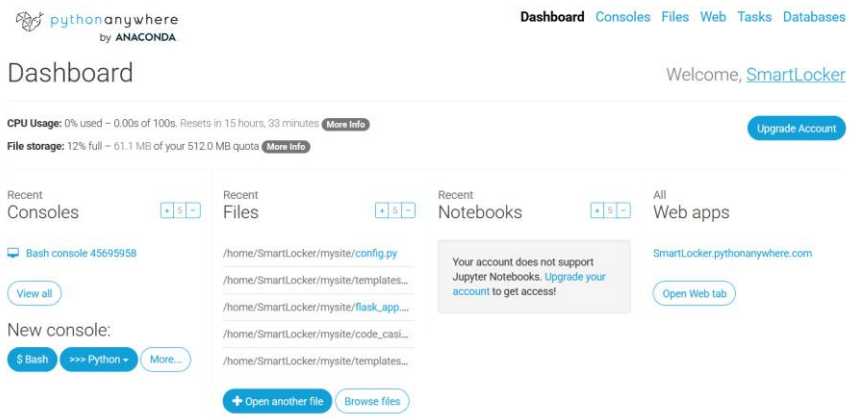

Figure 4 : Tableau de bord PythonAnywhere du projet SmartLocker

3.3.7 Environnement de développement : Visual Studio Code

L'intégralité du code de SmartLocker a été rédigée sous Visual Studio Code (VS Code), l'éditeur développé par Microsoft. Ce choix s'explique par sa légèreté, par ses extensions Python qui offrent la complétion automatique et la détection d'erreurs en temps réel, par sa coloration syntaxique des langages Python, HTML, CSS, JavaScript et JSON, ainsi que par son terminal intégré, qui permet d'exécuter les scripts sans quitter l'éditeur.

Figure 5 : Logo Visual Studio Code

VS Code a servi à développer la totalité des fichiers du projet : flask_app.py, config.py, Casier1.py, Casier2.py, bridge_firebase.py, les gabarits HTML/Jinja2, la feuille de style app.css et les fichiers de configuration de la PWA (manifest.json, service-worker.js). Son principal concurrent, PyCharm, a été écarté car trop gourmand pour le matériel disponible.

3.3.8 Accès distant aux Raspberry Pi : MobaXterm

MobaXterm est un terminal Windows avancé qui intègre un client SSH, un client SFTP et un serveur X11. Il s'est révélé indispensable pour se connecter à distance aux trois Raspberry Pi : le Pi 4B (passerelle MQTT/Firebase et broker Mosquitto) ainsi que les deux Pi Zero W (SPORT_01, exécutant Casier2.py, et ECOLE_01, exécutant Casier1.py).

MobaXterm a notamment permis de :

- Transférer les fichiers Python vers les Pi par SFTP, en glisser-déposer ;
- Lancer et surveiller les programmes dans des terminaux SSH dédiés ;
- Consulter les journaux en temps réel (commande tail -f casier_SPORT_01.log) et déboguer les connexions MQTT.

Figure 6 : Logo MobaXterm

La gestion simultanée de trois connexions SSH dans des onglets distincts a représenté un gain de temps appréciable par rapport à PuTTY. L'environnement de développement complet se compose ainsi d'un PC Windows équipé de VS Code (écriture du code) et de MobaXterm (déploiement et débogage), relié par SSH à un Raspberry Pi 4B et à deux Raspberry Pi Zero W, tous situés sur le même réseau local.

3.3.9 Assistant à la rédaction et au développement : Claude

Figure 7 : Logo Claude (Anthropic)

Claude, l'assistant d'IA développé par Anthropic, a servi d'outil d'aide à la rédaction et au développement. Il a généré l'essentiel du code front-end (gabarits HTML, feuilles CSS au thème cyberpunk et JavaScript côté client : temps réel Firebase, Service Worker, interactions de l'interface) ainsi que certaines portions du code Python (Flask et programmes des Raspberry Pi), qu'il m'a ensuite expliquées en détail. Il m'a aussi aidé à relire et corriger mon code, à améliorer la formulation de ce dossier et à approfondir des concepts techniques (MQTT, Service Worker). La conception d'ensemble, les choix d'architecture, l'intégration matérielle, les tests et la validation finale restent de mon entière responsabilité : chaque élément a été compris, adapté, assemblé et éprouvé sur le prototype réel.

4. Conception du projet

4.1 Architecture générale

L'architecture de SmartLocker repose sur quatre couches distinctes qui communiquent entre elles de façon indépendante : le smartphone (ou le navigateur de l'utilisateur), l'application Flask hébergée sur PythonAnywhere, la base de données Firebase Realtime et l'infrastructure matérielle — un Raspberry Pi 4B en passerelle et deux Raspberry Pi Zero W équipant respectivement le casier École et le casier Salle de sport, reliés par MQTT.

Le flux principal d'ouverture d'un casier se déroule de la manière suivante : l'utilisateur clique sur « Ouvrir » dans l'interface web ; Flask met à jour le champ status ou cmd_open_door dans Firebase ; la passerelle (Pi 4B) détecte ce changement par interrogation périodique (polling) toutes les secondes ; elle publie la commande « ouvrir » sur le topic MQTT locker/SPORT_01/commande ; le Pi Zero reçoit la commande, active le relais pendant quatre secondes et publie l'état réel de la porte sur le topic locker/SPORT_01/etat ; la passerelle réceptionne cet état et le réécrit dans Firebase ; enfin, l'interface web affiche la mise à jour en temps réel.

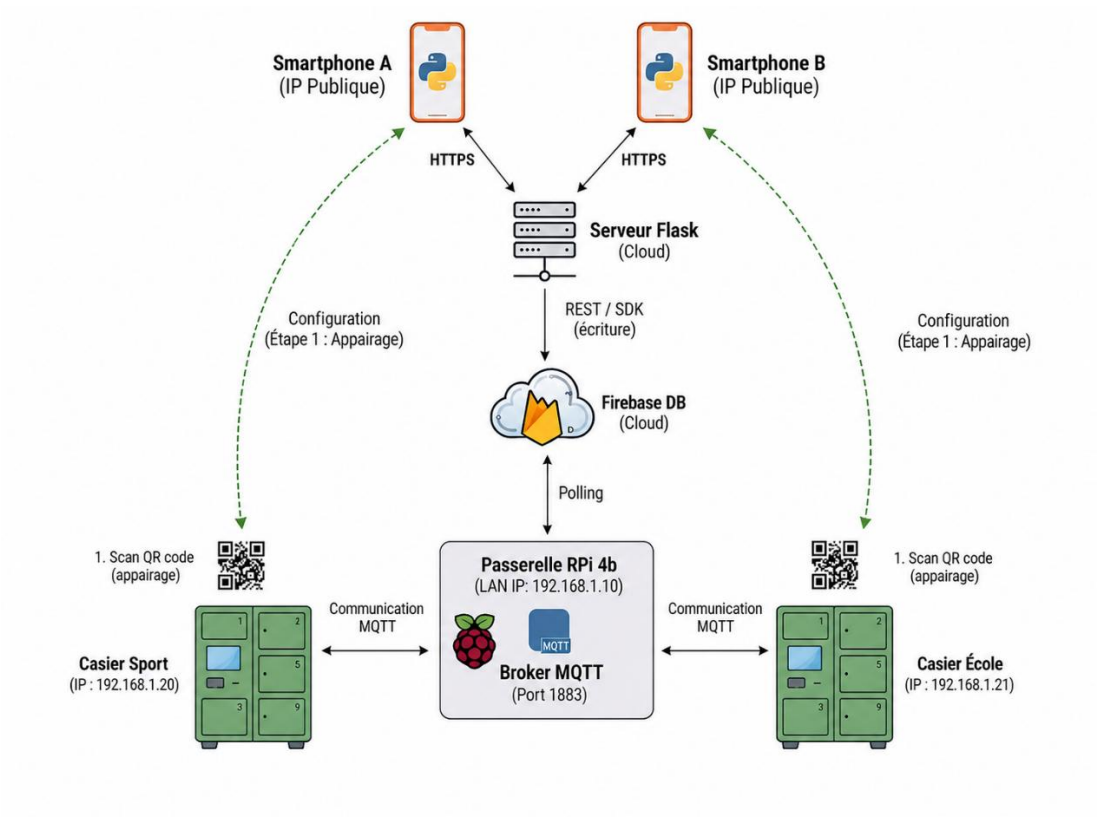

Figure 8 : Topologie réseau du système SmartLocker (adresses IP illustratives)

4.2 Structure du projet

Le projet est organisé selon les conventions de Flask :

```
smartlocker/
├── flask_app.py        # Toutes les routes
├── config.py           # Clés API, configuration Firebase et Stripe
├── templates/          # Pages HTML (Jinja2)
│   ├── index.html      # Tableau de bord principal
│   ├── locker_school.html
│   ├── locker_sport.html
│   ├── history.html
│   ├── settings.html
│   ├── admin.html
│   ├── scan.html
│   ├── payment_success.html
│   └── sport_thanks.html
├── static/
│   ├── service-worker.js
│   ├── manifest.json
│   └── firebase-messaging-sw.js
└── app.css
```

4.3 Structure de la base de données Firebase

La Realtime Database s'articule autour de plusieurs arbres principaux : lockers (état des casiers, mis à jour par la passerelle et par Flask), users (profils, historique et tokens FCM), lockers_index (index inverse casier → propriétaire) et gateways (état de la passerelle). Le schéma a été harmonisé en anglais et sépare clairement les états des commandes.

```
smartlocker-94e11-rtdb/
├── lockers/
│   ├── SPORT_01/
│   │   ├── status              : 'locked' | 'unlocked' | 'reserved'
│   │   ├── active_session      : { user_id, started_at, expires_at, amount_eur }
│   │   ├── door_physical       : 'Ouverte' | 'Fermee' | 'Inconnu'
│   │   ├── cmd_open_door       : timestamp (impulsion d'ouverture)
│   │   ├── online_status       : 'online' | 'offline'
│   │   ├── lockdown_until      : timestamp
│   │   ├── alert_active        : string
│   │   ├── distance_cm         : number
│   │   └── content_state       : 'occupied' | 'empty'
│   └── ECOLE_01/  (mêmes champs + stripe_session_id pour l'idempotence)
├── users/
│   └── {uid}/
│       ├── pseudo, email, photoURL
│       ├── premium       : bool
│       ├── fcm_token     : string
│       ├── paired        : { 'sport': true } | { 'school': true }
│       └── history/{pushId} : { locker_key, started_at, ended_at, ... }
├── lockers_index/   { 'sport': uid, 'school': uid }   (index O(1))
└── gateways/pi4b/   { status, last_seen }
```

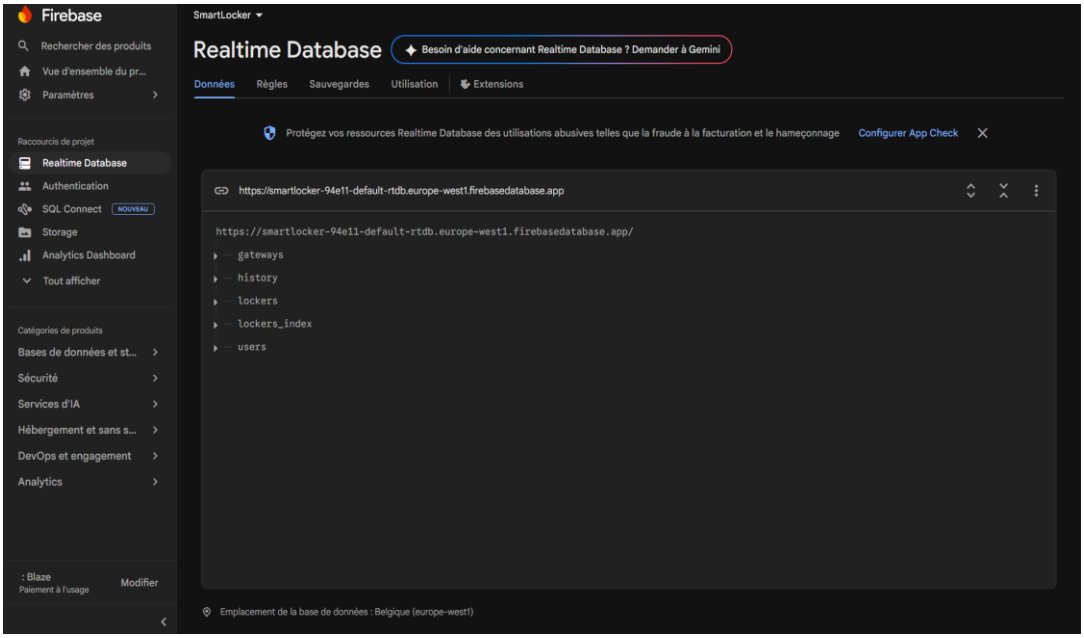

Figure 9 : Console Firebase Realtime Database du projet SmartLocker

4.4 Identité visuelle

L'interface de SmartLocker adopte une esthétique cyberpunk et néon, cohérente sur l'ensemble des pages et définie dans le fichier app.css :

- Un fond sombre, presque noir ;
- Une couleur d'accent principale vert néon, appliquée aux boutons, aux titres et aux bordures actives ;
- Deux polices : Orbitron (titres et identité de marque) et Rajdhani (corps de texte) ;
- Des formes angulaires : les boutons et les cartes recourent à la propriété clip-path: polygon() pour un rendu futuriste ;
- Des effets de lueur (box-shadow de teinte néon) sur les éléments interactifs.

Cette identité visuelle est rigoureusement maintenue sur tous les écrans de l'application, y compris sur l'écran hors ligne de la PWA.

5. Réalisation

5.1 Infrastructure matérielle

5.1.1 Raspberry Pi 4B — Passerelle

Le Raspberry Pi est un nano-ordinateur de la taille d'une carte de crédit, conçu à l'origine pour l'apprentissage de l'informatique ; il est aujourd'hui largement employé pour la domotique, les petits serveurs, le multimédia ou la robotique. Dans ce projet, le modèle 4B — le plus puissant de la gamme — joue le rôle de passerelle entre le réseau local (MQTT) et Internet (Firebase). Il exécute bridge_firebase.py, qui assure deux fonctions principales :

- Du polling de Firebase vers MQTT : chaque seconde, le programme lit l'état de chaque casier dans Firebase. Si le champ status passe à « unlocked » ou si une nouvelle valeur cmd_open_door est détectée, il publie la commande « ouvrir » sur le topic MQTT correspondant. Il respecte le mode lockdown : tant que lockdown_until est postérieur à l'heure courante, toute commande est bloquée ;
- Du relais de MQTT vers Firebase : lorsqu'un Pi Zero publie un événement (état de la porte, alerte, présence), la passerelle réceptionne le message et met à jour Firebase. En cas d'alerte d'intrusion, elle active le lockdown dans Firebase et envoie une notification push FCM au propriétaire du casier.

Figure 10 : Raspberry Pi 4B utilisé comme passerelle MQTT ↔ Firebase

Un signal de présence (heartbeat) émis toutes les trente secondes met à jour le nœud gateways/pi4b dans Firebase, ce qui permet à l'interface web de vérifier que la passerelle est en ligne via la route /api/gateway/status.

5.1.2 Raspberry Pi Zero W — Casier physique

Le Raspberry Pi Zero W est la déclinaison la plus compacte et la plus économique de la gamme : un nano-ordinateur miniature doté du Wi-Fi, parfait pour des objets connectés embarqués. Dans SmartLocker, il constitue le cerveau du casier physique. Chacun des deux casiers dispose de son propre programme : Casier1.py pour le casier École (LOCKER_ID = ECOLE_01) et Casier2.py pour le casier Salle de sport (LOCKER_ID = SPORT_01). Les deux programmes

partagent la même structure et les mêmes constantes (PIN_RELAIS = 17, PIN_PORTE = 27, PIN_TRIG = 23, PIN_ECHO = 24) et se connectent au broker MQTT Mosquitto hébergé sur le Pi 4B (MQTT_BROKER = 192.168.129.34, MQTT_PORT = 1883). Chaque programme pilote en permanence plusieurs composants :

- Le module relais 5 V (GPIO 17) commande le mécanisme qui verrouille physiquement la porte. Il est activé pendant quatre secondes (constante DUREE_OUVERTURE) lors d'une commande d'ouverture ;
- Le capteur magnétique de porte (GPIO 27, classe Button de gpiozero) détecte si la porte est physiquement ouverte ou fermée. Son état est publié sur le topic MQTT locker/SPORT_01/etat ;
- Le capteur ultrasonique HC-SR04 (GPIO 23 pour TRIG, GPIO 24 pour ECHO) mesure la distance à l'intérieur du casier afin de détecter la présence d'un objet (un objet est considéré présent en deçà du seuil SEUIL_OBJET_CM = 30 cm). Lorsque la porte est fermée, les relevés sont publiés toutes les dix secondes sur le topic locker/SPORT_01/presence ;
- Le lecteur RFID RC522 (bus SPI) permet à un badge administrateur de déverrouiller manuellement le casier et de réinitialiser un lockdown, sans passer par l'application web.

Figure 11 : Raspberry Pi Zero W intégré dans chaque casier physique

Le programme s'appuie sur plusieurs threads parallèles : un thread principal pour la boucle MQTT, un thread de surveillance des capteurs (cadencé toutes les 500 ms), un thread dédié au lecteur RFID et un thread de heartbeat (toutes les trente secondes).

5.1.3 Composants électroniques

Chacun des deux casiers physiques (École et Salle de sport) a été câblé selon le même schéma sur une plaque de prototypage. Les composants utilisés pour chaque casier sont les suivants :

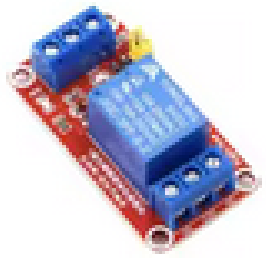

Figure 12 : Module relais 5 V

Le module relais 5 V est un interrupteur électromécanique commandé par un signal logique. Piloté par une sortie GPIO du Raspberry Pi (3,3 V), il referme un circuit de puissance pouvant atteindre 250 V en courant alternatif ou 30 V en courant continu, tout en isolant galvaniquement la partie commande de la partie puissance. Dans le casier, il alimente l'électroaimant (gâche électrique) qui assure le verrouillage de la porte : il est activé pendant quatre secondes (constante DUREE_OUVERTURE) à chaque commande d'ouverture.

Figure 13 : Serrure électrique (électroaimant) du casier

La serrure utilisée est un électroaimant (gâche électrique) installé sur la porte du casier. Au repos, il maintient le pêne verrouillé ; lorsqu'il est alimenté par le relais commandé par le Raspberry Pi, il libère le pêne et la porte peut s'ouvrir. C'est donc ce composant qui assure le verrouillage et le déverrouillage physiques du casier.

Figure 14 : Capteur magnétique de porte

Le capteur magnétique de porte repose sur un interrupteur à lames souples (reed switch). Il se compose de deux lames ferromagnétiques enfermées dans une ampoule de verre, qui se rapprochent et ferment le circuit sous l'effet d'un champ magnétique. Un aimant est fixé sur la porte et le capteur sur le cadre : lorsque la porte est fermée, l'aimant se trouve face au capteur, les lames se ferment et l'entrée GPIO est mise à la masse ; le Pi en déduit alors que la porte est fermée. À l'ouverture, l'aimant s'éloigne, le contact s'ouvre et la porte est signalée ouverte.

Figure 15 : Capteur ultrasonique HC-SR04

Le capteur HC-SR04 émet une onde ultrasonore (40 kHz) via la broche TRIG. Lorsque l'onde rencontre un obstacle, elle revient vers le capteur qui mesure le temps de vol sur la broche ECHO. La distance est calculée selon la formule : distance (cm) = (durée en µs) × 0,0343 / 2. Dans chacun des deux casiers, ce capteur permet de savoir si un objet est présent à l'intérieur.

Figure 16 : Lecteur RFID RC522 (badges Mifare 13,56 MHz)

Le lecteur RFID RC522 communique avec le Pi via le bus SPI (broches GPIO 8, 10, 11 et 25). Il lit les badges Mifare 13,56 MHz. Un badge administrateur permet d'ouvrir le casier sans passer par l'application web (utile en cas de panne réseau) et de réinitialiser un lockdown.

Figure 17 : Câble de raccordement GPIO

Enfin, le raccordement entre le Raspberry Pi et les différents composants est réalisé à l'aide de câbles de connexion (jumpers) reliés aux broches GPIO.

5.1.4 Maquette des casiers

Deux casiers réels ont été acquis et instrumentés pour servir de maquette finale du projet : un casier École et un casier Salle de sport. Chacun a été équipé de l'électronique décrite ci-dessus (Raspberry Pi Zero W, relais, capteur de porte, capteur ultrasonique et lecteur RFID) et raccordé au réseau du prototype.

Figure 18 : Casier École équipé pour la maquette

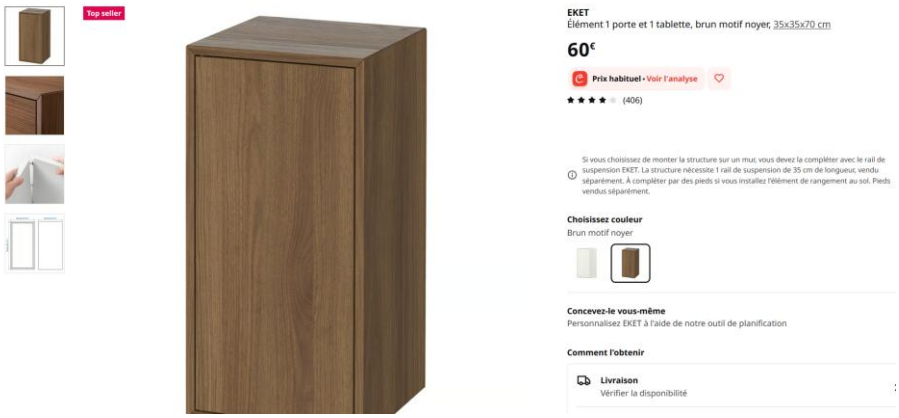

Figure 19 : Casier Salle de sport équipé pour la maquette

5.2 Authentification et gestion des utilisateurs

5.2.1 Inscription et connexion

L'authentification est prise en charge par Firebase Authentication (SDK JavaScript côté client et bibliothèque Pyrebase côté serveur). L'inscription crée un compte Firebase et initialise le nœud utilisateur dans la Realtime Database avec les champs pseudo, email, photoURL, premium (à false) et paired (objet vide).

La connexion via Google (OAuth) est assurée par Firebase Google Sign-In. Lorsqu'un utilisateur se connecte pour la première fois avec Google, son profil est créé automatiquement dans la base de données.

Extrait du fichier flask_app.py — route de connexion :

```
@app.route('/login', methods=['POST'])
def login():
    email    = (request.form.get('email') or '').strip().lower()
    password = request.form.get('password') or ''
    if not email or not password:
        flash("Email et mot de passe requis.")
        return redirect(url_for('home'))
    try:
        user = auth.sign_in_with_email_and_password(email, password)
        token = user['idToken']
        session.permanent = True
        session['user']               = token
        session['refresh_token']      = user.get('refreshToken', '')
        session['token_refreshed_at'] = time.time()
        session['email']              = email
        user_db = db.child('users').child(user['localId']).get(token).val() or {}
        session['pseudo'] = user_db.get('pseudo',
                                user_db.get('username', email.split('@')[0]))
        session['photo']  = user_db.get('photoURL', '')
        return redirect(url_for('home'))
    except Exception as e:
        print("[login] {}".format(e))
        flash("Identifiants incorrects.")
        return redirect(url_for('home'))
```

Les principaux écrans de la partie authentification de l'application sont présentés ci-après :

Figure 20 : Page d'accueil SmartLocker (PWA mobile)

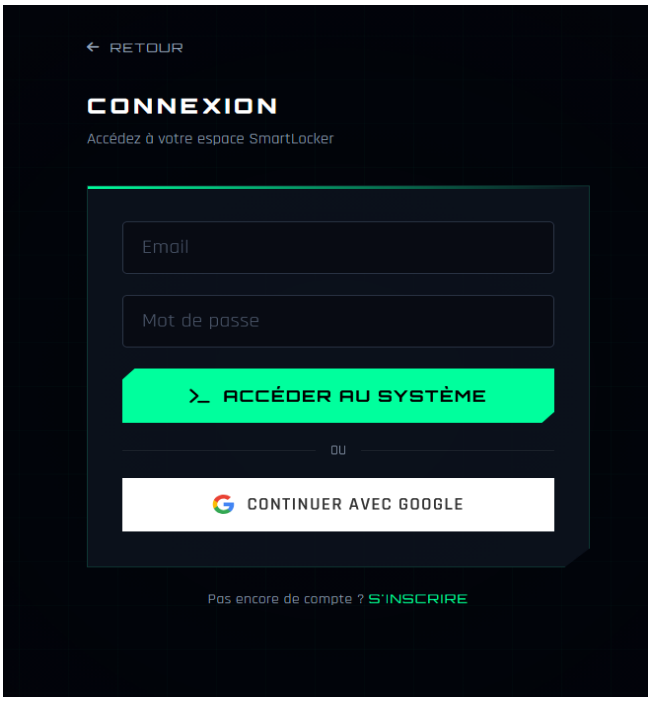

Figure 21 : Page de connexion avec authentification e-mail ou Google

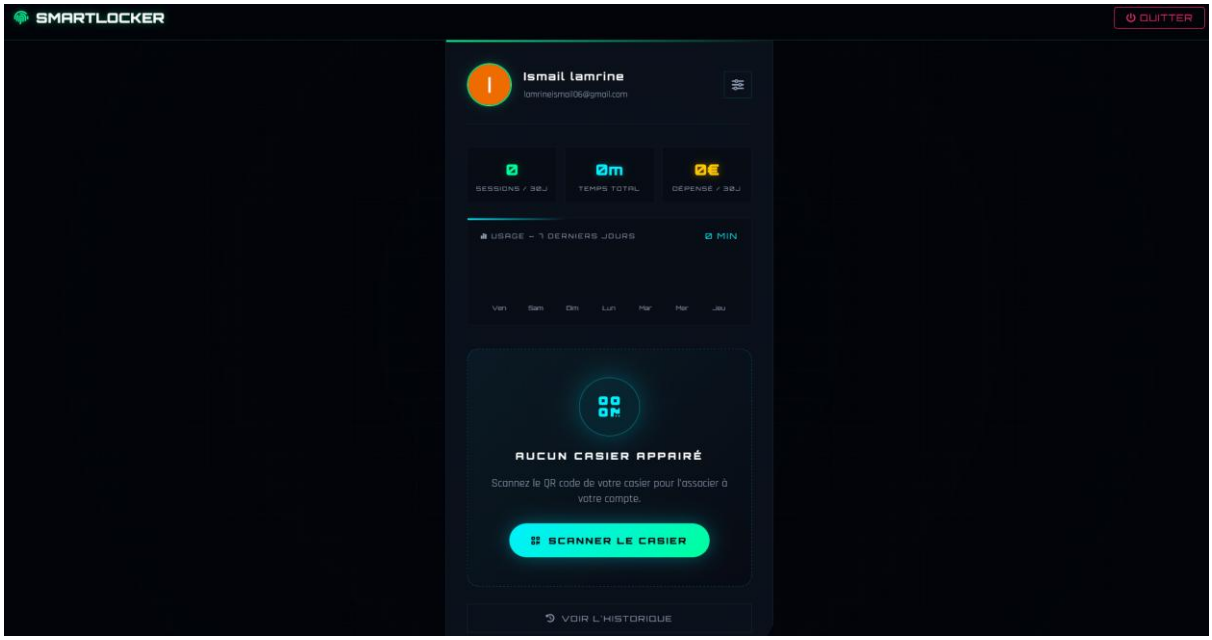

Figure 22 : Tableau de bord utilisateur — statistiques et casiers appairés

5.2.2 Système d'appairage des casiers

Un utilisateur ne peut accéder à un casier qu'après l'avoir appairé à son compte. L'appairage s'effectue en scannant le QR code affiché dans l'application (page /scan, qui s'appuie sur l'API getUserMedia du navigateur).

Lors de l'appairage, le système vérifie successivement que le code est valide (dictionnaire LOCKER_CODES de config.py), que l'utilisateur ne possède pas déjà un casier appairé (limite d'un seul casier par compte), et que le casier n'est pas déjà associé à un autre utilisateur — vérification effectuée grâce à l'index lockers_index — enfin, que le casier est bien en ligne. Les codes définis dans config.py sont SL-SPORT-01 pour le casier Salle de sport et SL-ECOLE-01 pour le casier École.

5.3 Casier Salle de sport — Gestion de session Premium

5.3.1 Contrôle d'accès Premium

L'accès au casier de salle de sport est conditionné au statut premium de l'utilisateur, stocké sous forme de booléen dans Firebase. Le contrôle est réalisé côté serveur, dans la route toggle_sport(), avant toute opération. Pour les besoins de la démonstration au jury, un bouton de la page des paramètres permet d'activer ou de désactiver instantanément le statut Premium (route /toggle_premium).

5.3.2 Cycle de vie d'une session sport

Lorsqu'un membre Premium clique sur « Ouvrir » (action « start ») :

1. Flask crée un objet active_session contenant user_id, user_email, started_at et expires_at fixé à 0 (aucune limite de durée) ;
2. Flask met à jour Firebase : status passe à « unlocked » et cmd_open_door reçoit le timestamp courant ;
3. La passerelle détecte le changement et publie « ouvrir » sur MQTT ;
4. Le Pi Zero active le relais pendant quatre secondes et publie l'état « Ouverte » sur MQTT;
5. La passerelle réceptionne l'état « Ouverte » et met à jour le champ door_physical dans Firebase ;
6. L'interface web affiche « Porte ouverte » en temps réel, grâce à l'interrogation de l'API /api/locker/sport/state toutes les cinq secondes.

Au cours d'une session active, le bouton « Ouvrir » peut être réutilisé (action « open ») pour générer une nouvelle impulsion d'ouverture (cmd_open_door) sans recréer de session. La session se termine lorsque l'utilisateur clique sur « Fermer » (action « end ») : par sécurité, Flask refuse la clôture tant que la porte est physiquement ouverte, puis appelle _log_session() pour consigner l'entrée dans l'historique, repasse status à « locked » et dissocie automatiquement le casier afin de le libérer pour l'utilisateur suivant.

5.4 Casier École — Réservation et paiement Stripe

5.4.1 Flux de réservation

La page du casier École présente un calendrier de réservation. L'utilisateur sélectionne une date (jusqu'à quatorze jours à l'avance), une heure de début (de 8 h à 21 h) et une durée (d'une à six heures, sans dépasser 22 h). Plusieurs validations sont effectuées : date non échue, créneau non déjà occupé et fin de session n'excédant pas 22 h. Le prix est calculé au tarif de 5,00 €/heure (constante PRICE_PER_HOUR = 500 centimes dans config.py) et le montant total est affiché avant confirmation.

5.4.2 Intégration Stripe Checkout

Extrait de la route school_checkout(), qui crée la session de paiement Stripe :

```
checkout = stripe.checkout.Session.create(
    payment_method_types=['card'],
    line_items=[{
        'price_data': {
            'currency': 'eur',
            'unit_amount': amount,   # ex. 1000 pour 2h a 5 EUR/h
            'product_data': {
                'name': 'SmartLocker Ecole - {}h le {}'.format(
                            hours, start_dt.strftime('%d/%m a %Hh')),
                'description': 'Casier Ecole de {}h a {}h'.format(
                                    start_h, start_h + hours),
            },
        },
        'quantity': 1,
    }],
    mode='payment',
    success_url=url_for('school_success', _external=True)
                + '?session_id={CHECKOUT_SESSION_ID}',
    cancel_url=url_for('locker_school', _external=True),
    metadata={
        'user_id':    ctx['local_id'],
        'user_email': session.get('email') or '',
        'hours':      str(hours),
        'amount_eur': str(amount / 100),
        'start_ts':   str(start_ts),
        'expires_ts': str(expires_ts),
        'date_str':   date_str,
        'start_str':  start_str,
        'kind':       'school_booking',
    }
)
return redirect(checkout.url, code=303)
```

5.4.3 Idempotence et protection contre la double réservation

La fonction _apply_school_booking() est appelée aussi bien depuis la route /locker/school/success (retour de Stripe) que depuis le webhook Stripe (/stripe/webhook). Afin d'éviter d'appliquer deux fois la même réservation lorsque les deux appels parviennent au serveur, la fonction vérifie si le champ stripe_session_id de la session active correspond déjà à la session Stripe reçue ; le cas échéant, elle retourne already_applied: True sans modifier Firebase. De plus, si le casier est déjà occupé par une autre réservation au moment de l'application, la fonction retourne conflict: True et l'utilisateur est invité à contacter le support pour obtenir un remboursement.

5.4.4 Prolongation de session

Lorsqu'il reste moins de trente minutes à la session active, le bouton « Prolonger +1 h » devient disponible. Il déclenche un nouveau paiement Stripe de 5,00 € puis, après confirmation, met à jour le champ expires_at dans Firebase en y ajoutant 3 600 secondes.

5.5 Système de sécurité physique

5.5.1 Détection d'intrusion

Dans Casier2.py, le thread surveillance_capteurs() surveille en permanence le capteur de porte. Si la porte s'ouvre alors qu'aucune commande d'ouverture n'est en cours (ouverture_en_cours.is_set() vaut False) et que la période de grâce consécutive à une ouverture (quinze secondes, fin_grace_ouverture) est écoulée, la fonction declencher_alerte('intrusion') est appelée. Celle-ci incrémente le compteur d'alertes (nb_alertes_intrusion) et publie l'alerte sur le topic locker/SPORT_01/alerte. Dès que ce compteur atteint le seuil SEUIL_LOCKDOWN (trois alertes), le lockdown est activé localement et une minuterie de cinq minutes est lancée en vue de sa réinitialisation automatique.

5.5.2 Mode lockdown

Lorsque la passerelle reçoit une alerte d'intrusion via MQTT, elle réalise les opérations suivantes :

- Elle écrit dans Firebase : lockdown_until (timestamp courant + 300 s), lockdown_reason, status à « locked » et active_session à None ;
- Elle identifie, dans Firebase, l'utilisateur propriétaire du casier concerné (champ paired);

- Elle envoie une notification push FCM urgente vers son token, avec un message du type « Intrusion détectée sur votre casier SPORT_01 ! ».

Pendant un lockdown, toute commande d'ouverture — qu'elle provienne de l'application web ou de MQTT — est bloquée. Le déverrouillage s'effectue soit par réinitialisation automatique au bout de cinq minutes, soit par présentation d'un badge RFID administrateur sur le casier en question, soit par action de l'administrateur depuis le panneau d'administration web.

5.6 Panneau d'administration

La page /admin est protégée par le décorateur require_admin(), qui vérifie que l'adresse e-mail de la session figure dans la liste ADMIN_EMAILS définie dans config.py (Admin@inraci.be pour ce projet). Lorsqu'un administrateur se connecte, il est automatiquement redirigé du tableau de bord vers le panneau d'administration.

Le panneau affiche :

- La liste de l'ensemble des utilisateurs (pseudo, e-mail, statut premium, nombre de sessions et revenus) ;
- L'état courant des casiers ;
- Le total des revenus générés.

Les actions disponibles sont les suivantes :

- Activer ou désactiver le statut Premium d'un utilisateur ;
- Dissocier de force un casier d'un utilisateur ;
- Supprimer les données d'un utilisateur de la base.

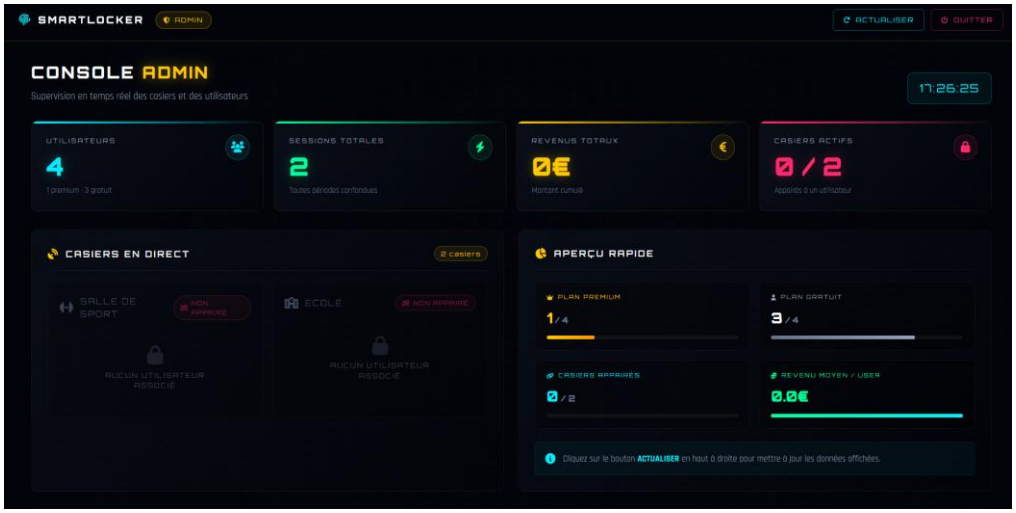

Figure 23 : Console administrateur — vue d'ensemble (utilisateurs, sessions, revenus, casiers)

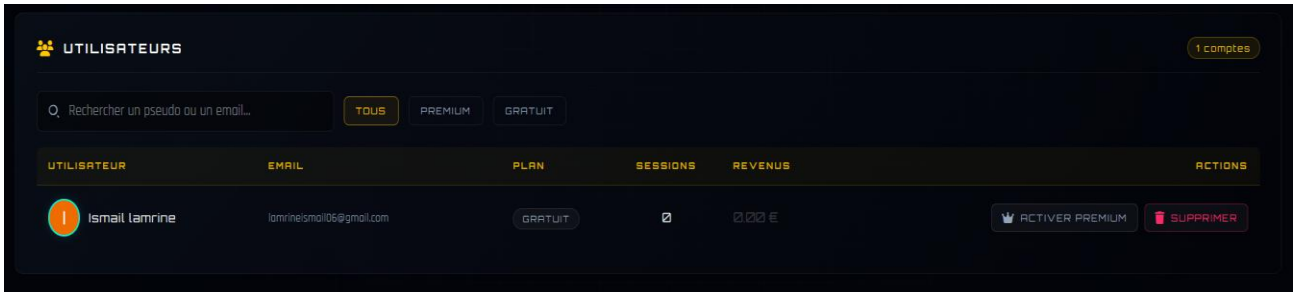

Figure 24 : Console administrateur — gestion des utilisateurs avec actions Activer Premium / Supprimer

5.7 Progressive Web App (PWA)

L'application est configurée en PWA grâce aux fichiers manifest.json et service-worker.js. Le manifeste déclare le nom « SmartLocker », les couleurs de thème et le mode d'affichage standalone (sans barre de navigation du navigateur). La PWA peut ainsi être installée sur l'écran d'accueil d'un appareil Android ou iOS, depuis Chrome ou Safari.

Le Service Worker met en œuvre une stratégie de cache hybride : il laisse systématiquement passer les pages HTML par le réseau (les données des casiers doivent rester fraîches), applique une stratégie « réseau d'abord » (network-first) aux ressources CSS et JavaScript afin de récupérer rapidement les mises à jour, et une stratégie « cache d'abord » (cache-first) aux ressources statiques (images, icônes, polices). La perte de connexion est, quant à elle, gérée directement dans l'interface : un écran de recouvrement affichant le message « SIGNAL PERDU », dans l'esthétique cyberpunk du projet, apparaît grâce aux événements online et offline du navigateur, et disparaît automatiquement au rétablissement du réseau.

Le fichier firebase-messaging-sw.js constitue un second Service Worker, dédié à la réception des notifications push Firebase Cloud Messaging. Il est servi depuis la racine (/firebase-messaging-sw.js) afin de respecter les contraintes de portée (scope) imposées par FCM.

5.8 Page de paramètres utilisateur

La page /settings centralise l'ensemble des actions liées au compte de l'utilisateur. Celui-ci peut y modifier sa photo de profil (avec recadrage côté navigateur), son pseudo et son mot de passe. Lorsque l'utilisateur s'est inscrit via Google OAuth, l'adresse e-mail est verrouillée et un badge « Google — verrouillé » est affiché à côté du champ.

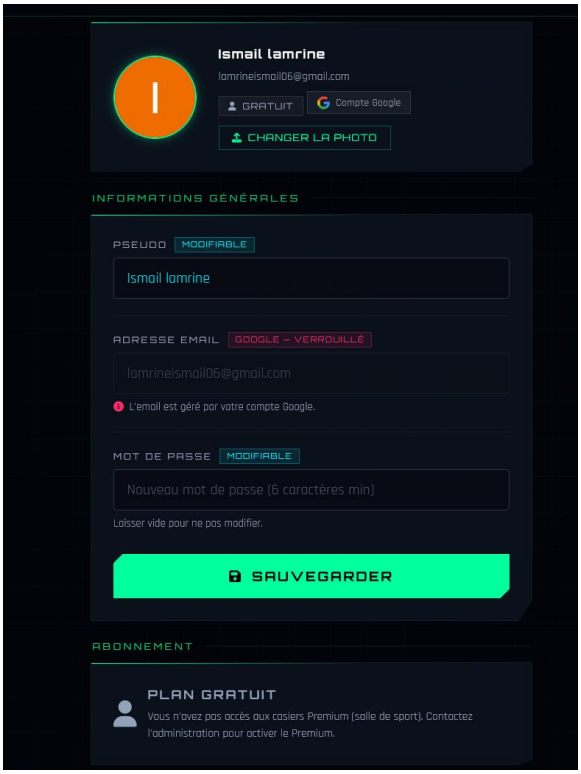

Figure 25 : Page de paramètres — informations générales et abonnement

Dans la partie inférieure de la page, l'utilisateur retrouve la liste des casiers appairés à son compte ainsi qu'une « zone dangereuse » regroupant l'action de déconnexion. Le casier appairé peut également y être dissocié.

Figure 26 : Page de paramètres — casiers appairés et déconnexion

6. Tests et validation

6.1 Tests fonctionnels

Une série de scénarios représentatifs a été éprouvée afin de valider chacune des fonctionnalités de l'application. Le tableau ci-dessous synthétise les principaux tests réalisés et leurs résultats.

Fonctionnalité testée	Scénario / Résultat attendu	Résultat
Inscription	Nouveau compte e-mail valide	Succès
Inscription	E-mail déjà utilisé → message d'erreur	Succès
Connexion Google	Compte Google existant	Succès
Appairage QR code	Scan depuis la caméra native	Succès
Appairage code manuel	SL-ECOLE-01 saisi manuellement	Succès
Appairage double	Casier déjà pris par un autre compte	Refus correct
Paiement Stripe (test)	Carte 4242 4242 4242 4242	Succès
Paiement Stripe (échec)	Carte 4000 0000 0000 0002	Erreur gérée
Réservation créneau	Créneau disponible	Firebase mis à jour
Réservation double	Créneau déjà occupé	Refus correct
Ouverture casier sport	Utilisateur Premium	Relais activé
Ouverture casier sport	Utilisateur non-Premium	Accès refusé
Expiration session	Timer atteint 0 → verrouillage	Succès
Prolongation	Moins de 30 min restantes + paiement	Succès
Alerte intrusion	Porte ouverte sans commande	FCM reçu
Lockdown	Trois intrusions → lockdown	Blocage actif
Reset RFID	Badge admin → déverrouillage	Succès
Admin panel	Connexion Admin@inraci.be	Redirection correcte
Export CSV	Historique exporté	Fichier correct
PWA offline	Connexion coupée	Page « SIGNAL PERDU »

6.2 Tests de compatibilité

L'application a été testée sur plusieurs navigateurs et appareils :

Navigateur / Appareil	Résultat
Chrome Android	Fonctionnel
Safari iOS	Fonctionnel
Chrome Desktop	Fonctionnel
Firefox Desktop	Fonctionnel
Samsung Internet	Fonctionnel

6.3 Bugs rencontrés et solutions

6.3.1 Double déclenchement du webhook Stripe

Problème : la route /locker/school/success (retour du navigateur) et le webhook /stripe/webhook parvenaient parfois au serveur à moins d'une seconde d'intervalle, provoquant une double mise à jour de Firebase et, occasionnellement, des données de session incohérentes.

Solution : implémentation de la fonction _apply_school_booking() avec une vérification d'idempotence fondée sur le champ stripe_session_id. Lorsque la session Stripe a déjà été traitée, la fonction retourne immédiatement, sans modifier Firebase.

6.3.2 Lockdown Firebase non synchronisé avec le Pi Zero

Problème : lorsque le lockdown était réinitialisé depuis l'interface web (par l'administrateur), la passerelle continuait à bloquer les commandes, car elle relisait Firebase avec un léger décalage. Une commande d'ouverture émise juste après la réinitialisation était alors parfois bloquée.

Solution : ajout d'une route d'urgence /admin/force_free_sport permettant à l'administrateur de forcer le champ status à « locked » et de vider active_session, ce qui garantit un état cohérent entre Firebase et le Pi Zero.

6.3.3 Capteur ultrasonique HC-SR04 : valeurs aberrantes

Problème : le capteur renvoyait occasionnellement des valeurs très faibles (inférieures à 1 cm) ou très élevées (supérieures à 200 cm), en raison des interférences électromagnétiques provoquées par le relais lors de son activation.

Solution : filtrage dans la fonction lire_distance_cm() au moyen des constantes ULTRASON_MIN_CM = 1 et ULTRASON_MAX_CM = 200. Les valeurs hors plage sont ignorées et un avertissement est consigné dans les journaux.

6.3.4 Porte ouverte trop longtemps

Problème : lorsqu'une porte demeurait ouverte au-delà de TIMEOUT_OUVERTURE (60 secondes), le programme émettait des rappels successifs et, en l'absence de fermeture, la fonction declencher_alerte() risquait d'être appelée de façon répétée toutes les 500 ms, inondant Firebase d'alertes.

Solution : mise en place d'un mécanisme de rappels progressifs (un premier rappel après 60 secondes, puis un rappel toutes les 30 secondes, dans la limite de RAPPEL_MAX = 3) avant l'escalade en alerte. À chaque escalade, la variable porte_ouverte_depuis est réinitialisée à l'heure courante, ce qui remet le compteur à zéro et évite tout envoi en boucle.

7. Conclusion

7.1 Résumé du travail réalisé

Ce travail de fin d'études a abouti à la conception et au déploiement d'une solution complète de gestion de casiers connectés : SmartLocker. Le projet couvre l'ensemble du cycle de développement — analyse des besoins, étude des technologies, conception de l'architecture, développement logiciel, câblage du matériel et déploiement en production. L'application est pleinement fonctionnelle et accessible publiquement.

L'infrastructure matérielle est opérationnelle : deux Raspberry Pi Zero W, associés à une passerelle Raspberry Pi 4B, communiquent via MQTT. Le système de réservation et de paiement Stripe fonctionne correctement en mode test, et les alertes d'intrusion ainsi que les notifications push FCM ont été validées.

7.2 Objectifs atteints

- Application web Flask complète, déployée sur PythonAnywhere ;
- Infrastructure matérielle opérationnelle : Pi Zero W, capteurs, relais et passerelle Pi 4B (pont MQTT ↔ Firebase) ;
- Intégration de Stripe Checkout, avec gestion des créneaux, idempotence et prolongation ;
- Système de sécurité : détection d'intrusion, lockdown, notifications push FCM et badge RFID administrateur ;
- Panneau d'administration complet ;
- Progressive Web App installable sur smartphone, avec mode hors ligne ;
- Historique des sessions, assorti d'un export au format CSV.

Le seul point non finalisé concerne le passage en production de Stripe (remplacement des clés de test sk_test_ par des clés réelles sk_live_), qui requiert la validation d'un compte Stripe commercial. Cette démarche est reportée à l'issue de la présentation du TFE.

7.3 Compétences acquises

- Développement backend en Python avec Flask : routage, gestion des sessions, intergiciels (middlewares) et gestion des erreurs ;
- Intégration de Firebase Realtime Database et de Firebase Authentication via Pyrebase ;
- Protocole MQTT : broker Mosquitto, modèle publish/subscribe, qualité de service (QoS) et reconnexion automatique ;
- Programmation des GPIO du Raspberry Pi : relais, capteurs numériques et ultrasonique, bus SPI (RFID) ;
- Intégration d'une API de paiement (Stripe Checkout, webhooks, idempotence) ;
- Notifications push web avec Firebase Cloud Messaging ;
- Développement d'une PWA avec Service Worker et stratégies de mise en cache ;
- Déploiement et administration d'une application web en production.

7.4 Améliorations futures

- Passer Stripe en production afin d'accepter de véritables paiements ;
- Intégrer un écran OLED au Pi Zero pour afficher localement l'état du casier ;
- Étendre le système à plusieurs casiers par utilisateur, avec une interface d'administration plus aboutie ;
- Mettre en place une sauvegarde automatique de la base de données Firebase.

8. Bibliographie / Webographie

Les références sont présentées selon la norme APA (7e édition).

8.1 Frameworks et langages

Pallets Projects. (2024). *Flask documentation (3.0.x)*. Consulté le 15 septembre 2025, à l'adresse https://flask.palletsprojects.com/

Horvath, N. (2023). *Pyrebase4 — A simple python wrapper for the Firebase API* [Dépôt logiciel]. GitHub. Consulté le 20 septembre 2025, à l'adresse https://github.com/nhorvath/Pyrebase4

8.2 Firebase et Google

Google. (2024). *Firebase Realtime Database — Guide de démarrage rapide*. Google LLC. Consulté le 28 septembre 2025, à l'adresse https://firebase.google.com/docs/database

Google. (2024). *Firebase Authentication*. Google LLC. Consulté le 28 septembre 2025, à l'adresse https://firebase.google.com/docs/auth

Google. (2024). *Firebase Cloud Messaging — Web/JavaScript*. Google LLC. Consulté le 2 octobre 2025, à l'adresse https://firebase.google.com/docs/cloud-messaging/js/client

8.3 Paiement en ligne

Stripe, Inc. (2024). *Stripe Checkout — Documentation officielle*. Stripe. Consulté le 10 octobre 2025, à l'adresse https://stripe.com/docs/checkout

Stripe, Inc. (2024). *Stripe Webhooks — Réception d'événements*. Stripe. Consulté le 10 octobre 2025, à l'adresse https://stripe.com/docs/webhooks

Stripe, Inc. (2024). *Stripe Python SDK — Référence API*. Stripe. Consulté le 12 octobre 2025, à l'adresse https://stripe.com/docs/api?lang=python

8.4 IoT et protocoles

Eclipse Foundation. (2024). *Paho-MQTT — Python client library*. Eclipse. Consulté le 18 octobre 2025, à l'adresse https://eclipse.dev/paho/clients/python/

OASIS. (2019). *MQTT version 5.0 — OASIS standard*. OASIS Open. Consulté le 18 octobre 2025, à l'adresse https://mqtt.org/mqtt-specification/

Raspberry Pi Foundation. (2024). *GPIO Zero documentation*. Consulté le 25 octobre 2025, à l'adresse https://gpiozero.readthedocs.io/

Raspberry Pi Foundation. (2024). *Getting started with Raspberry Pi*. Consulté le 25 octobre 2025, à l'adresse https://www.raspberrypi.com/documentation/

8.5 PWA et Web

Google Developers. (2024). *Progressive Web Apps*. Google LLC. Consulté le 5 novembre 2025, à l'adresse https://web.dev/progressive-web-apps/

MDN Web Docs. (2024). *Service Worker API*. Mozilla Corporation. Consulté le 8 novembre 2025, à l'adresse https://developer.mozilla.org/fr/docs/Web/API/Service_Worker_API

MDN Web Docs. (2024). *Web app manifests*. Mozilla Corporation. Consulté le 8 novembre 2025, à l'adresse https://developer.mozilla.org/fr/docs/Web/Manifest

8.6 Hébergement et outils de développement

PythonAnywhere. (2024). *Help & documentation — Flask web framework*. Consulté le 15 novembre 2025, à l'adresse https://help.pythonanywhere.com/pages/Flask/

Microsoft. (2024). *Visual Studio Code documentation*. Microsoft Corporation. Consulté le 20 novembre 2025, à l'adresse https://code.visualstudio.com/docs

Mobatek. (2024). *MobaXterm — Enhanced terminal for Windows*. Consulté le 22 novembre 2025, à l'adresse https://mobaxterm.mobatek.net/documentation.html

8.7 Solutions étudiées dans l'analyse comparative

Solutions commerciales

Ojmar. (2026). *Electronic locks for lockers — Gamme RFID OTS*. Consulté le 12 mars 2026, à l'adresse https://ojmar.com/en/solutions/technology/rfid/

Quadient. (2026). *Parcel locker solutions*. Consulté le 12 mars 2026, à l'adresse https://www.quadient.com/

Projets open source (GitHub)

Reyes, R. (2026). *smart-lock — A smart, account based locker system that utilizes NFC communication with a Raspberry Pi* [Dépôt logiciel]. GitHub. Consulté le 5 février 2026, à l'adresse https://github.com/rmcreyes/smart-lock

chdsbd. (2026). *rpi-lock — Raspberry Pi powered door lock* [Dépôt logiciel]. GitHub. Consulté le 5 février 2026, à l'adresse https://github.com/chdsbd/rpi-lock

rick1924. (2026). *RFIDAccessSystem — RFID tag system for access control with online reservations* [Dépôt logiciel]. GitHub. Consulté le 18 octobre 2025, à l'adresse https://github.com/rick1924/RFIDAccessSystem

9. Annexes

Annexe A — Extrait de config.py (structure, clés masquées)

```
# config.py – Variables de configuration SmartLocker

# ── Flask ──────────────────────────────────────────────
SECRET_KEY = '...'

# ── Stripe ─────────────────────────────────────────────
STRIPE_SECRET_KEY      = 'sk_test_...'
STRIPE_PUBLISHABLE_KEY = 'pk_test_...'
STRIPE_WEBHOOK_SECRET  = 'whsec_...'
PRICE_PER_HOUR         = 500   # centimes = 5,00 EUR

# ── Firebase ───────────────────────────────────────────
FIREBASE_CONFIG = {
    'apiKey':        '...',
    'authDomain':    'smartlocker-94e11.firebaseapp.com',
    'databaseURL':   'https://smartlocker-94e11-default-rtdb.europe-
west1.firebasedatabase.app',
    'projectId':     'smartlocker-94e11',
    'storageBucket': 'smartlocker-94e11.firebasestorage.app',
}

# ── Admin ──────────────────────────────────────────────
ADMIN_EMAILS = ['Admin@inraci.be']

# ── Casiers ────────────────────────────────────────────
LOCKERS = {
    'sport':  {'id': 'SPORT_01', 'name': 'Salle de sport',
               'location': 'Zone fitness – Rez-de-chaussée', 'type': 'sport'},
    'school': {'id': 'ECOLE_01', 'name': 'Ecole',
               'location': 'Couloir B – 1er etage', 'type': 'school'},
}

LOCKER_CODES = {
    'SL-SPORT-01': 'sport',
    'SL-ECOLE-01': 'school',
```

```
}

SCHOOL_HOURS = ['08:00', '09:00', '10:00', '11:00', '12:00',
                '13:00', '14:00', '15:00', '16:00', '17:00',
                '18:00', '19:00', '20:00', '21:00']
```

Annexe B — Topics MQTT utilisés

Les topics suivants sont utilisés pour la communication MQTT entre le Pi 4B (passerelle) et chacun des deux Pi Zero W. La structure est identique pour SPORT_01 et ECOLE_01, seul l'identifiant du casier change.

Topic	Direction	Contenu
locker/SPORT_01/commande	Pi 4B → Pi Zero	« ouvrir » | « reset » | « etat »
locker/SPORT_01/etat	Pi Zero → Pi 4B	« Ouverte » | « Fermée » | « commande_recue »
locker/SPORT_01/alerte	Pi Zero → Pi 4B	« intrusion » | « lockdown_actif » | « lockdown_reset »
locker/SPORT_01/presence	Pi Zero → Pi 4B	Distance en cm (float)
locker/SPORT_01/status	Pi Zero → Pi 4B	« en_ligne|timestamp|lockdown=non|alertes=0 »

Annexe C — Extrait : initialisation des GPIO (Casier.py)

Définition des broches selon la norme BCM et initialisation des périphériques (relais, capteurs, lecteur RFID) à l'aide de la bibliothèque gpiozero.

```
# ── Définition des broches GPIO (norme BCM) ──
PIN_RELAIS = 17    # commande du verrou (module relais 5 V)
PIN_PORTE  = 27    # capteur magnétique de porte
PIN_TRIG   = 23    # ultrason HC-SR04 – déclenchement
PIN_ECHO   = 24    # ultrason HC-SR04 – écho


# ── Initialisation des périphériques via gpiozero ──
```

```
try:
    relais           = OutputDevice(PIN_RELAIS, active_high=True,
                                    initial_value=False)
    capteur_porte    = Button(PIN_PORTE, pull_up=True)
    capteur_ultrason = DistanceSensor(echo=PIN_ECHO, trigger=PIN_TRIG,
                                      max_distance=2.0)
    if SimpleMFRC522 is not None:
        rfid_reader = SimpleMFRC522()
    log.info("GPIO initialisés.")
except GPIOZeroError as e:
    log.critical(f"Init GPIO impossible : {e}")
    sys.exit(1)
```

Annexe D — Extrait : ouverture sécurisée du casier (Casier.py)

Fonction d'ouverture du verrou. Elle vérifie l'absence de verrouillage (intrusion) et la protection anti-spam, active le relais pendant 4 secondes, puis confirme l'ouverture réelle via le capteur magnétique.

```
def ouvrir_casier(source="local"):
    # Refuser l'ouverture si le casier est verrouillé (intrusion)
    if lockdown_actif:
        publier(TOPIC_ETAT, "lockdown_actif", qos=1)
        return
    if not anti_spam_autorise():          # limite le nombre d'ouvertures
        publier(TOPIC_ETAT, "anti_spam", qos=1)
        return


    publier(TOPIC_ETAT, "commande_recue", qos=1)
    try:
        relais.on()                       # alimente le verrou
        enregistrer_ouverture()
        sleep(DUREE_OUVERTURE)            # maintient 4 secondes
    finally:
        relais.off()                      # re-verrouille systématiquement


    # Vérifie que la porte s'est réellement ouverte
    if porte_est_fermee() is False:
        publier(TOPIC_ETAT, "Ouverte", qos=1)
    else:
        publier(TOPIC_ETAT, "ouverture_non_confirmee", qos=1)
```

Annexe E — Extrait : réception MQTT et détection d'intrusion

Traitement des commandes reçues par MQTT depuis la passerelle (Pi 4B) et déclenchement d'une alerte lorsqu'une ouverture de porte survient en dehors d'une commande légitime.

```
def sur_message(client, userdata, msg):
    commande = msg.payload.decode('utf-8').strip().lower()
    log.info(f"MQTT {msg.topic} → {commande}")

    if commande == "ouvrir":
        # Ouverture dans un thread pour ne pas bloquer la boucle MQTT
        threading.Thread(target=ouvrir_casier,
                         args=("mqtt",), daemon=True).start()
    elif commande == "reset":
        reset_lockdown()
    elif commande == "etat":
        publier(TOPIC_ETAT, lire_etat_porte(), qos=1)

# Détection d'intrusion : porte ouverte hors commande légitime
if etat_str == "Ouverte":
    if not ouverture_en_cours.is_set() and not en_grace:
        declencher_alerte("intrusion")
```

Annexe F — Extrait : paiement Stripe (flask_app.py)

Route Flask créant la session de paiement Stripe pour la réservation d'un créneau du casier École, avec vérification préalable de la disponibilité du casier.

```
@app.route('/locker/school/checkout', methods=['POST'])
@require_login
def school_checkout():
    token = session['user']
    # Refuser si le casier est hors-ligne
    if not get_locker_state("school", token).get("online"):
        flash("Casier hors-ligne : reservation impossible.")
        return redirect(url_for('locker_school'))

    hours  = int(request.form.get('hours', '1'))
    amount = hours * PRICE_PER_HOUR        # prix en centimes

    # Création de la session de paiement Stripe
    checkout = stripe.checkout.Session.create(
        payment_method_types=['card'],
        line_items=[{'price_data': {
            'currency': 'eur', 'unit_amount': amount,
            'product_data': {'name': f'SmartLocker Ecole - {hours}h'}},
```

```
            'quantity': 1}],
        mode='payment',
        success_url=url_for('school_success', _external=True)
                    + '?session_id={CHECKOUT_SESSION_ID}',
        cancel_url=url_for('locker_school', _external=True),
        metadata={'hours': str(hours), 'kind': 'school_booking'})
    return redirect(checkout.url, code=303)
```

Annexe G — Extrait : passerelle Firebase ↔ MQTT (bridge_firebase.py)

Programme tournant sur le Pi 4B. Il fait le lien entre la base de données Firebase (commandée par l'application web) et les Raspberry Pi Zero des casiers via MQTT. Le premier sens transmet les commandes d'ouverture ; le second remonte l'état physique et les alertes des casiers vers Firebase.

```
# ── Sens 1 : Firebase → MQTT (commande d'ouverture) ──
def polling_firebase():
    while not arret_demande.is_set():
        for locker_id, locker_key in LOCKERS.items():
            data = fdb.reference(f'lockers/{locker_id}').get() or {}
            cmd  = data.get('cmd_open_door')        # timestamp de la demande

            # N'ouvrir que sur une commande NOUVELLE et récente (< 10 s)
            if cmd and cmd != derniers_cmds[locker_id]:
                if time.time() - float(cmd) <= 10 and not is_locked_down:
                    topic = f"locker/{locker_id}/commande"
                    mqtt_client.publish(topic, "ouvrir", qos=1)
                derniers_cmds[locker_id] = cmd
        arret_demande.wait(timeout=POLL_SEC)

# ── Sens 2 : MQTT → Firebase (remontée de l'état du casier) ──
def sur_message(client, userdata, msg):
    parts = msg.topic.split('/')                    # locker/{id}/{type}
    locker_id, event_type = parts[1], parts[2]

    if event_type == 'etat':                        # état physique de la porte
        ecrire_firebase(locker_id, {
            'door_physical':         msg.payload.decode(),
            'door_physical_updated': iso_now()
        })
    elif event_type == 'alerte':                    # intrusion, lockdown...
        ecrire_firebase(locker_id, {'alert_active': msg.payload.decode()})
```