Institut National de Radioélectricité et de Cinématographie

Enseignement technique secondaire de qualification

Accès aux études supérieures

Avenue Jupiter, 188

1190 Forest

FootArena

KADI TOUZANI Amjad

Pour l’obtention du certificat de qualification

Technicien en informatique

Année scolaire : 2025-2026

Remerciements :

Je tiens à remercier mes amis **Elias**, **Dorian** et **Martial** pour leur soutien et leur bonne humeur tout au long de cette année. Même dans les moments plus compliqués, ils ont toujours été présents pour m'encourager et me motiver à avancer dans mon travail.

Je souhaite également remercier mon confrère de toujours, **Peter Calva**, pour son soutien, ses conseils et ses encouragements durant la réalisation de ce projet. Sa présence et son aide m'ont permis de rester motivé jusqu'à la fin de cette année scolaire.

Table des matières

1. Introduction

1.1. Contexte

Le football amateur occupe une place importante dans la vie sportive et sociale de la région bruxelloise. Des dizaines de clubs locaux organisent chaque semaine des rencontres. Pourtant, la gestion de la vente de billets reste aujourd'hui très artisanale dans ce milieu : vente en espèces à l'entrée, billets papier non sécurisés et files d'attente.

En tant que passionné de football et témoin de cette réalité, j'ai constaté que les clubs amateurs n'ont ni les ressources financières ni les compétences techniques pour mettre en place une solution de billetterie professionnelle. Les outils existants sur le marché sont soit trop coûteux, soit trop complexes pour des structures avec un budget limité.

1.2. Problématique

Ce constat m'a amené à formuler la problématique suivante :

> Comment fournir aux clubs de football amateur de la région bruxelloise une plateforme de billetterie en ligne accessible, sécurisée et gratuite, leur permettant de vendre des billets à leurs membres sans investissement majeur ?

À cela s'ajoute une dimension physique : comment garantir, à l'entrée du stade, que seul le détenteur d'un billet valide et non utilisé puisse accéder au stade ?

1.3. Objectifs du TFE

Ce travail de fin d'études poursuit plusieurs objectifs :

- Développer une plateforme web gratuite pour les clubs amateurs, leur permettant de gérer et vendre des billets en ligne ;
- Offrir aux supporters une expérience simple : consulter les matchs, réserver et payer en ligne depuis n'importe quel appareil ;
- Garantir la sécurité des billets grâce à un QR code unique par achat, transmis par email et scannable à l'entrée ;
- Concevoir un système automatisé de contrôle d'accès physique à l'aide d'un Raspberry Pi 4, d'un capteur ultrason et d'un servomoteur ;

- Mettre en pratique les compétences acquises tout au long de ma formation en informatique, notamment en développement web, en base de données et en électronique embarquée.

1.4. Présentation rapide de la solution

La solution développée est une application web construite avec le framework Flask (Python), déployée en ligne sur la plateforme Render et connectée à une base de données PostgreSQL hébergée sur Railway.

Le site propose deux interfaces distinctes :

- Une interface administrateur, destinée aux clubs, permettant de créer des rencontres, gérer des tribunes et consulter les réservations ;
- Une interface client, destinée aux supporters, permettant de consulter les matchs à venir, de filtrer par club ou par date, de réserver des billets et d'effectuer un paiement sécurisé via Stripe.

À l'achat, le client reçoit automatiquement un email contenant un QR code unique, valable pour un seul match et invalidé après son premier scan. Ce QR code est lu à l'entrée du stade par un système physique développé sur Raspberry Pi 4, qui commande l'ouverture d'une porte via un servomoteur et détecte le passage du visiteur grâce à un capteur ultrason.

2. Analyse des besoins

2.1. Description du problème

Les clubs de football amateur de la région bruxelloise font face à plusieurs difficultés dans la gestion de leur billetterie. Aujourd'hui, la grande majorité d'entre eux pratique encore la vente de billets de façon manuelle : paiement en espèces à l'entrée, billets papier sans sécurité, absence de contrôle des capacités et aucune traçabilité des ventes.

Cette situation engendre plusieurs problèmes concrets :

- Un risque élevé de fraude : les billets papier peuvent être falsifiés ou revendus illégalement devant les stades ;
- Une mauvaise expérience pour les supporters, qui doivent se déplacer physiquement pour acheter leur billet ou faire la file à l'entrée ;
- Un manque de visibilité sur les matchs à venir pour les membres des clubs ;
- Une gestion comptable peu fiable et peu transparente pour les clubs.

Mettre en place un système de billetterie en ligne professionnel représente un coût important, que les clubs amateurs aux budgets limités ne peuvent généralement pas assumer.

2.2. Public cible

Le projet s'adresse à deux types d'utilisateurs distincts :

Les clubs de football amateur (administrateurs) :

- Clubs évoluant dans la région bruxelloise ;
- Disposant de peu de ressources financières et techniques ;
- Souhaitant digitaliser leur billetterie sans investissement.

Les supporters et membres des clubs (clients) :

- Toute personne souhaitant assister à un match d'un club inscrit sur la plateforme ;
- Utilisateurs ayant accès à internet et capables d'effectuer un paiement en ligne.

2.3. Cahier des charges

Le projet doit répondre aux exigences suivantes :

Pour les clubs (interface administrateur) :

- Pouvoir s'inscrire gratuitement sur la plateforme ;
- Créer et gérer des rencontres (matchs) ;
- Créer des tribunes avec une capacité et un prix définis ;
- Avoir la possibilité de ne louer qu'un nombre prédéfini de sièges par tribune ;
- Consulter les réservations effectuées par les clients ;
- Recevoir automatiquement le montant des ventes, déduction faite de la commission.

Pour les supporters (interface client) :

- Créer un compte et se connecter de façon sécurisée ;
- Consulter la liste des matchs à venir avec des options de filtrage (par date, par club, par ligue) ;
- Réserver un ou plusieurs billets pour un match ;
- Effectuer un paiement en ligne sécurisé ;
- Recevoir une confirmation par email contenant un QR code unique.

Pour le système de contrôle d'accès :

- Scanner le QR code à l'entrée du stade ;
- Ouvrir automatiquement la porte en cas de billet valide ;
- Invalider le QR code après son premier scan (pas de double entrée possible) ;
- Détecter le passage du visiteur et refermer la porte automatiquement.

2.4. Fonctionnalités attendues

Fonctionnalité	Description
Inscription / Connexion	Création de compte sécurisé pour clubs et supporters
Gestion des matchs	Création, modification et suppression de rencontres par l'admin
Gestion des tribunes	Définition des capacités, prix et disponibilités
Réservation en ligne	Sélection d'un match et achat de billets par le client
Paiement sécurisé	Intégration de Stripe pour le traitement des paiements
QR code unique	Génération et envoi par email d'un billet numérique sécurisé
Contrôle d'accès physique	Système Raspberry Pi pour scan, ouverture de porte et détection de passage
Filtrage des matchs	Recherche par date, club ou ligue
Commission automatique	Prélèvement automatique d'une commission sur chaque vente

2.5. Contraintes

Contraintes techniques

- Le projet doit mobiliser les compétences acquises dans l'ensemble des cours d'informatique : développement web, base de données, réseaux et électronique embarquée ;
- L'application doit être déployée en ligne et accessible depuis n'importe quel appareil ;
- La sécurité des données (mots de passe, informations de paiement) doit être assurée.

Contraintes de temps

- Le projet a été développé sur une année scolaire complète, en parallèle des cours ;
- Cette contrainte a limité le développement de certaines fonctionnalités avancées, comme l'envoi de notifications push lors de la création d'une nouvelle rencontre.

Contraintes matérielles

- Aucune contrainte matérielle majeure n'a été rencontrée.

3. Étude préalable

Avant de commencer le développement du projet, il était important d'étudier les solutions déjà présentes sur le marché. Cette analyse permet de comprendre les fonctionnalités existantes et les besoins auxquels mon projet devra répondre.

3.1. Solutions existantes

Avant de commencer le développement, j'ai analysé les solutions de billetterie déjà disponibles sur le marché afin de comprendre ce qui existait et d'identifier les lacunes auxquelles mon projet pourrait répondre.

Ticketmaster est la plateforme de billetterie la plus connue au monde. Elle permet la vente de billets pour des événements sportifs, musicaux et culturels à grande échelle. Cependant, elle s'adresse exclusivement aux grandes organisations et implique des frais fixes et des contrats complexes, totalement inaccessibles pour un club amateur.

Weezevent est une solution française de billetterie en ligne orientée événementiel. Elle est plus accessible que Ticketmaster mais reste payante, avec des abonnements mensuels et des commissions variables. De plus, elle n'est pas spécialisée dans le football amateur et ne propose pas de système de contrôle d'accès physique intégré.

Eventbrite est une plateforme généraliste de gestion d'événements. Elle permet la création d'événements gratuits ou payants, mais prélève une commission importante sur chaque billet vendu et ne propose pas de gestion spécifique des tribunes ou des rencontres sportives.

Aucune des solutions ne répond pleinement aux besoins des clubs amateurs bruxellois : elles sont soit trop coûteuses, soit trop généralistes, soit dépourvues d'un système de contrôle d'accès physique.

3.2. Comparaison des solutions

Critère	Ticketmaster	Weezevent	Eventbrite	FootArena
Gratuit pour les clubs	Non	Non	Non	Oui
Spécialisé en football amateur	Non	Non	Non	Oui
Gestion des tribunes	Oui	Non	Non	Oui
Paiement en ligne	Oui	Oui	Oui	Oui
QR Code sécurisé	Oui	Oui	Oui	Oui
Contrôle d'accès physique	Non	Non	Non	Oui
Accessible aux petits clubs	Non	Non	Non	Oui

3.3. Choix technologiques

Après cette analyse, j'ai sélectionné les outils et technologies les mieux adaptés au projet, en tenant compte de mes compétences, des contraintes de temps et des exigences fonctionnelles.

En ce qui concerne le langage de programmation, **Python** est le langage que je maîtrise le mieux grâce à ma formation. Il est polyvalent, lisible et dispose d'un écosystème très riche pour le développement web et la gestion de bases de données. Il est également compatible avec le Raspberry Pi, ce qui permet d'utiliser un seul langage pour l'ensemble du projet.

Afin de lancer l'application en local, j'ai choisi d'utiliser **Flask**. Ce dernier est un micro-framework Python léger et flexible. Cette flexibilité est idéale pour créer des pages web, pour faciliter la communication avec les bases de données. Enfin, il est facile à prendre en main.

En ce qui concerne la création de la base de données, **PostgreSQL** sur Railway permet d'héberger ma base de données en ligne gratuitement. Elle stocke l'ensemble des données comme clubs, matchs, tribunes, réservations, utilisateurs.

De plus, l'utilisation de **SQLAlchemy** me permet de manipuler la base de données directement en Python, sans écrire de requêtes SQL brutes. Il facilite la création des tables, la gestion des relations entre elles et rend le code plus lisible et maintenable.

J'ai également eu recours à **Render** qui est une plateforme d'hébergement cloud qui propose un plan gratuit. Il permet de déployer une application Flask et de la rendre accessible depuis des appareils connectés à internet.

En ce qui concerne les solutions de paiement en ligne pour mon application, **Stripe** est l'une des solutions les plus utilisées et les mieux documentées au monde. Elle propose une API claire, une gestion sécurisée des transactions et un environnement de test complet.

Pour ce qui est de la session des utilisateurs, j'ai utilisé **Flask-Login** qui est une extension Flask. Elle permet de gérer facilement la connexion, la déconnexion et la protection des pages réservées aux administrateurs ou aux clients connectés.

Enfin, le système de contrôle d'accès a été fait avec un **Raspberry Pi 4** qui est un mini-ordinateur qui pilote des composants électroniques comme un servomoteur (ouverture de porte) et un capteur ultrason (détection de passage).

3.4. Justification globale des choix

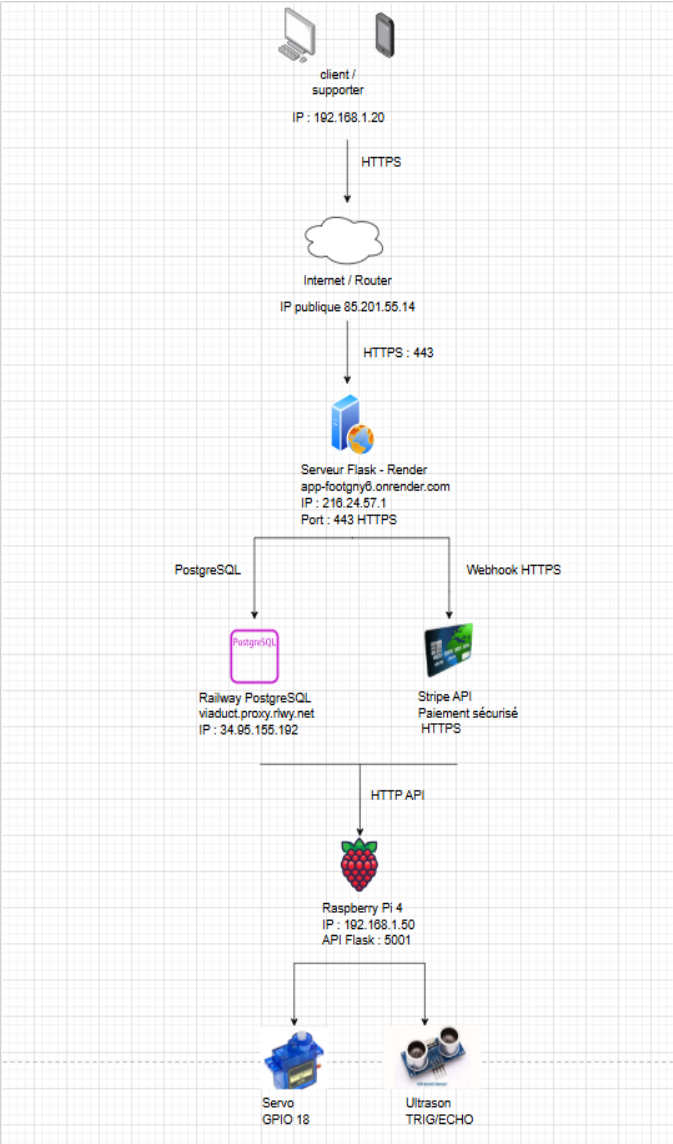
L'ensemble des technologies retenues répond à trois critères principaux :

- **Cohérence** : Python est utilisé aussi bien pour le site web que pour le système embarqué, ce qui simplifie le développement et la maintenance ;
- **Accessibilité** : toutes les solutions choisies sont gratuites ou disposent d'un plan gratuit ;
- **Pédagogie** : chaque outil choisi correspond à une compétence enseignée durant la formation, permettant de mobiliser l'ensemble des apprentissages dans un projet.

Ces choix technologiques ont permis de développer une solution complète, moderne et accessible, tout en respectant les contraintes techniques et financières du projet. Ils ont également permis de combiner développement web, base de données, hébergement cloud, cybersécurité et électronique embarquée dans une seule application cohérente.

4. Conception du projet

4.1. Topologie



L'architecture générale du projet repose sur une application web connectée à une base de données, à une plateforme de paiement en ligne ainsi qu'à un système physique de contrôle d'accès.

Le supporter accède au site web depuis un ordinateur ou un smartphone via Internet en HTTPS. L'application web, développée avec Flask en Python, est hébergée sur la plateforme Render.

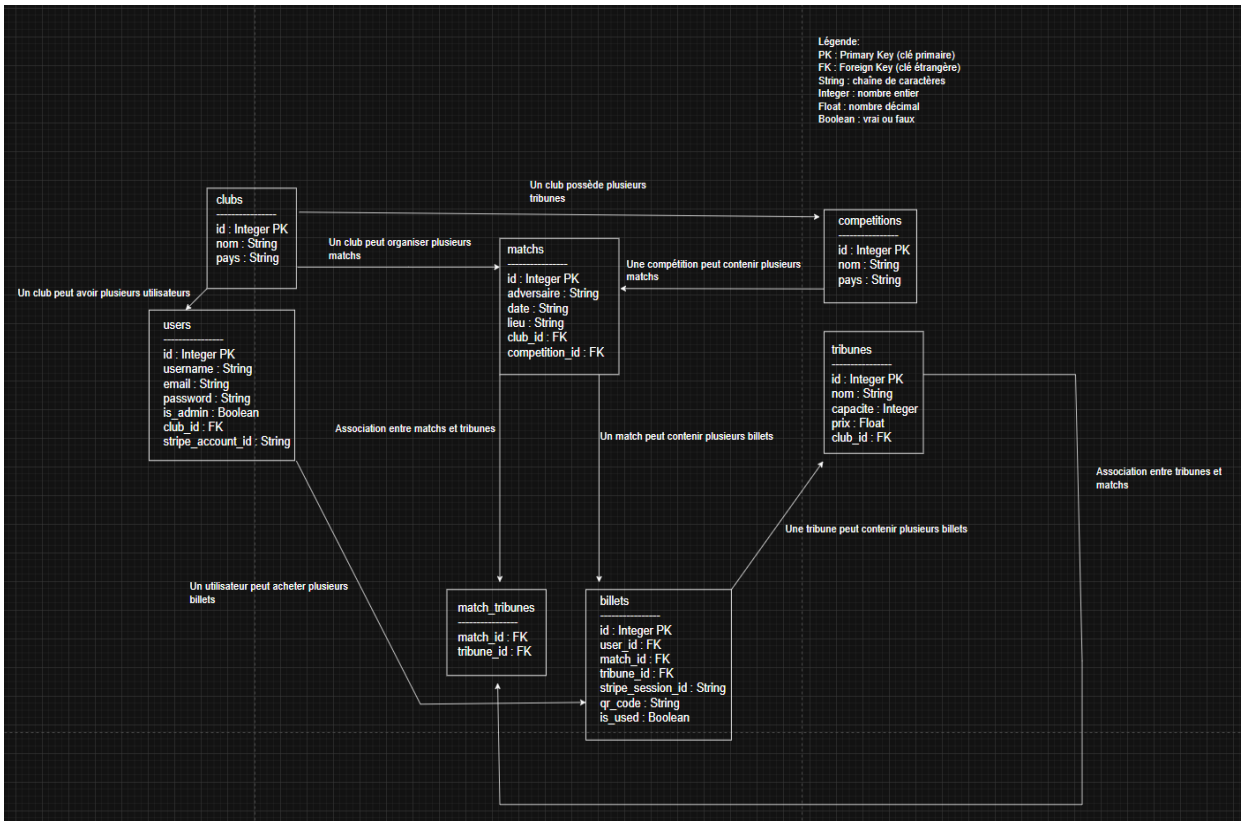
Le serveur Flask communique avec une base de données PostgreSQL hébergée sur Railway. Cette base de données stocke les utilisateurs, les matchs, les tribunes et les billets.

Le paiement en ligne est assuré par Stripe. Après un paiement validé, Stripe envoie une confirmation au serveur grâce à un webhook HTTPS. Cette confirmation permet de générer automatiquement les billets et les QR codes.

Le système comprend également un Raspberry Pi 4 chargé du contrôle d'accès physique. Celui-ci pilote un servomoteur pour ouvrir la porte et utilise un capteur ultrason pour détecter le passage du supporter.

Cette architecture permet de combiner développement web, base de données, paiement sécurisé et électronique embarquée dans une seule solution.

4.2. Base de donnée



La base de données du projet a été conçue avec PostgreSQL afin de garantir une bonne organisation des informations ainsi qu'une gestion fiable des relations entre les différentes données du système.

Le schéma relationnel présenté ci-dessus représente l'ensemble des tables utilisées dans l'application ainsi que les relations entre elles. Chaque table possède une clé primaire (PK) permettant d'identifier chaque élément de manière unique. Les clés étrangères (FK) permettent quant à elles de créer des liens entre les différentes tables.

- La table *users* contient les informations des utilisateurs et des administrateurs du système. Chaque utilisateur possède un nom d'utilisateur, une adresse email, un mot de passe ainsi qu'un rôle administrateur ou non.
- La table *clubs* permet de stocker les clubs de football présents dans l'application. Chaque club peut posséder plusieurs utilisateurs, plusieurs tribunes ainsi que plusieurs matchs.
- La table *competitions* contient les différentes compétitions disponibles, par exemple les championnats nationaux ou internationaux.

- La table *matchs* représente les matchs disponibles à la réservation. Chaque match est associé à un club ainsi qu'à une compétition.
- La table *tribunes* permet de gérer les différentes zones du stade, avec leur capacité maximale ainsi que leur prix.
- La table *billets* est l'une des tables principales du projet. Elle relie un utilisateur à un match et à une tribune précise. Elle contient également les informations liées au paiement Stripe ainsi qu'au QR code utilisé pour le contrôle d'accès.
- Enfin, la table *match_tribunes* sert de table d'association entre les matchs et les tribunes. Elle permet d'indiquer quelles tribunes sont ouvertes pour chaque match.

Cette structure relationnelle permet de garantir une organisation claire des données tout en facilitant les opérations importantes du système comme les réservations, les paiements, la génération des billets et la validation des accès au stade.

4.3. Architecture du projet

L'architecture du projet repose sur une organisation modulaire permettant de séparer les différentes parties de l'application afin de faciliter le développement, la maintenance et l'évolution du système.

Le projet a été développé principalement en Python avec le framework Flask. Cette architecture permet de gérer à la fois l'interface web, la base de données, le système de réservation, les paiements Stripe, la génération des QR codes ainsi que la communication avec le Raspberry Pi.

Le fichier *app.py* contient les routes principales de l'application ainsi que la logique générale du système. Il permet de gérer les différentes pages du site, les connexions utilisateurs, les réservations de billets et les paiements.

Le fichier *config.py* contient les paramètres de configuration du projet comme la connexion à la base de données PostgreSQL, les clés Stripe, la configuration des emails ainsi que les variables de sécurité.

Les modèles de la base de données sont définis grâce à SQLAlchemy. Chaque modèle représente une table de la base de données comme les utilisateurs, les matchs, les tribunes ou les billets.

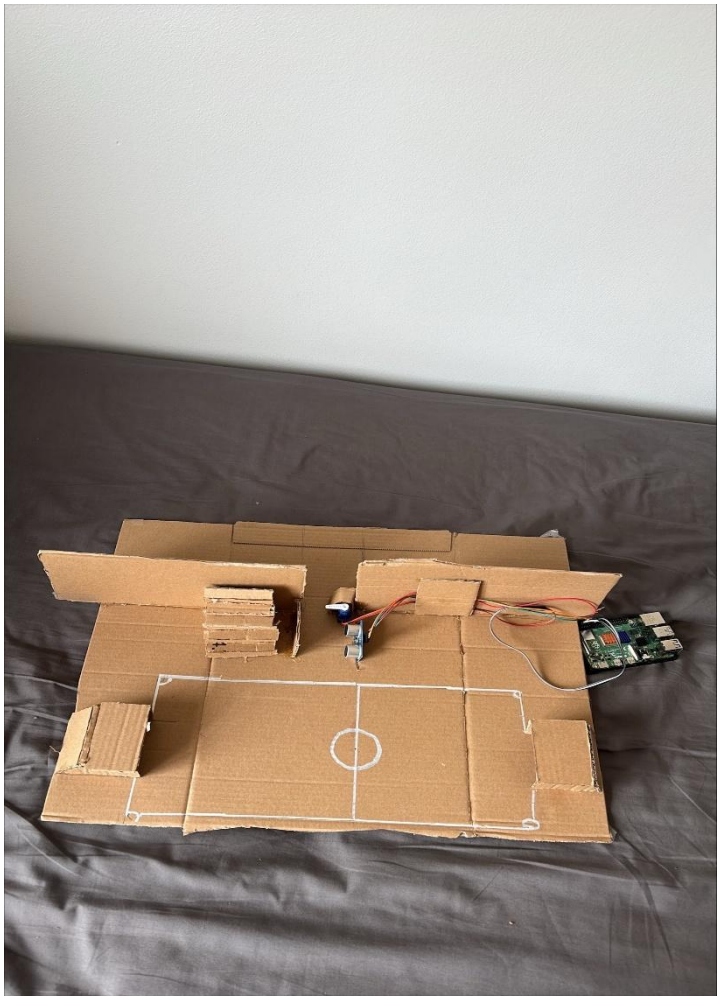
Le dossier templates contient toutes les pages HTML du site web. Ces pages sont affichées dynamiquement grâce au moteur de templates Jinja2 intégré à Flask.

Le dossier static contient les fichiers statiques du projet comme le CSS, les images, les icônes ainsi que les fichiers JavaScript.

L'application communique également avec plusieurs services externes comme Render pour l'hébergement du site, Railway pour la base de données PostgreSQL, Stripe pour les paiements en ligne ainsi que le Raspberry Pi pour le contrôle d'accès physique.

Cette architecture modulaire permet d'obtenir une application claire, organisée et plus facile à maintenir. Elle facilite également l'ajout futur de nouvelles fonctionnalités sans devoir modifier l'ensemble du projet.

4.4. Maquettes/interfaces



Pour montrer que mon projet ne sert pas seulement à vendre des billets en ligne, j'ai réalisé une maquette physique. Le but était de reproduire une entrée de stade avec une porte qui s'ouvre automatiquement lorsqu'un billet valide est scanner.

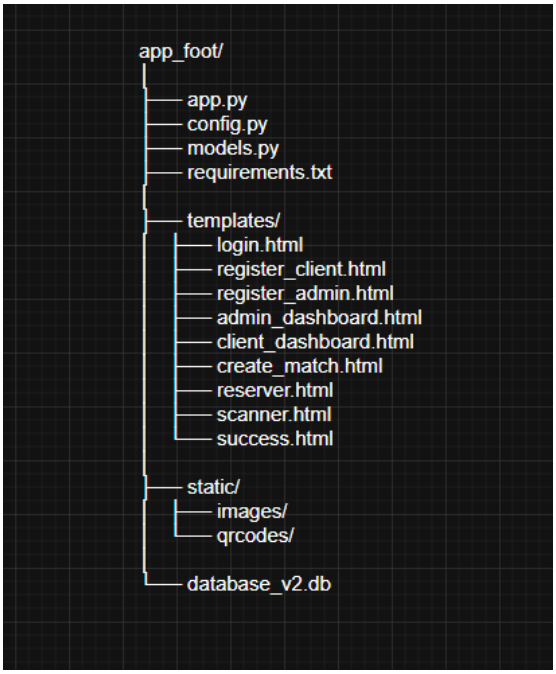
Pour cela, j'ai utilisé un Raspberry Pi un servomoteur et un capteur ultrason. Le Raspberry Pi reçoit les informations du site et décide si la porte doit s'ouvrir ou non. Si le billet est valide, le servomoteur ouvre la porte. Ensuite, le capteur ultrason détecte le passage de la personne et la porte se referme automatiquement après quelques secondes.

Cette maquette m'a permis de relier la partie informatique du projet à quelque chose de concret. Elle montre que le système peut être utilisé non seulement pour vendre des billets, mais aussi pour gérer l'accès à un stade de manière automatique.

4.5. Organisation des fichiers

Afin de faciliter le développement et la maintenance de l'application, les différents fichiers ont été organisés selon une structure claire et hiérarchisée.

L'arborescence générale du projet est présentée ci-dessous.



Cette organisation permet de séparer les différentes parties de l'application et de faciliter l'ajout de nouvelles fonctionnalités. Elle contribue également à rendre le code plus lisible et plus simple à maintenir.

5. Réalisation (développement)

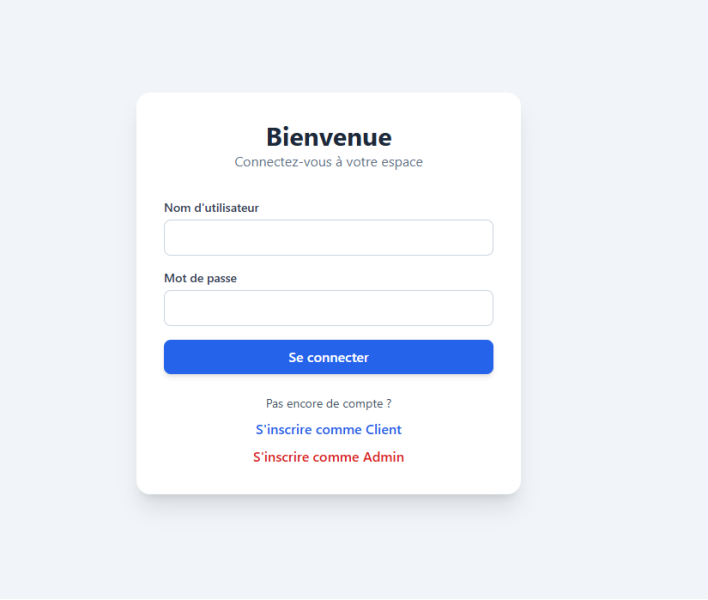
5.1. Explication et Présentation

Cette partie présente le fonctionnement général de l'application développée dans le cadre du projet. L'objectif est de montrer comment les différentes fonctionnalités interagissent entre elles, depuis l'inscription d'un utilisateur jusqu'à l'achat d'un billet et son utilisation à l'entrée du stade.

5.1.1. Connexion et inscription

L'application dispose de deux types d'utilisateurs : les administrateurs représentant les clubs de football et les supporters souhaitant acheter des billets.

Lorsqu'un utilisateur crée un compte, les informations saisies dans le formulaire sont envoyées au serveur Flask. Le système vérifie d'abord qu'aucun autre compte n'utilise déjà le même nom d'utilisateur ou la même adresse email. Si les informations sont valides, un nouvel utilisateur est créé dans la base de données PostgreSQL.



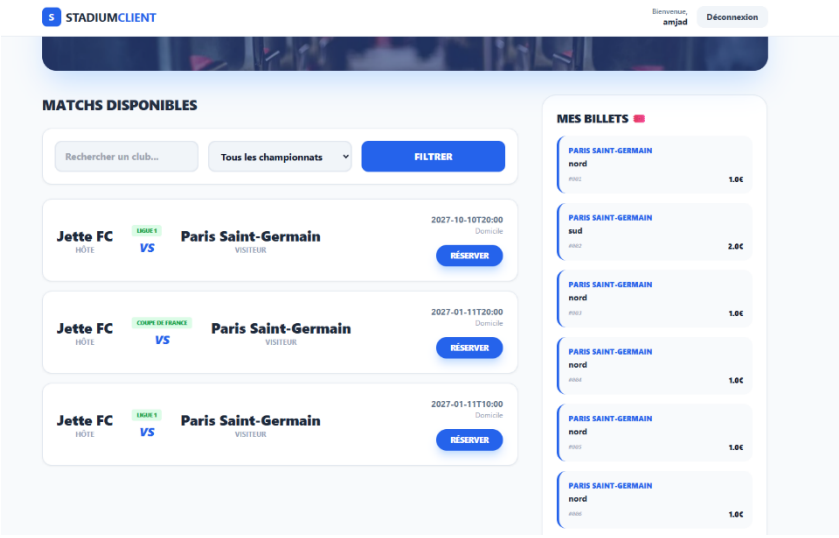
Après son inscription, l'utilisateur peut se connecter grâce à son nom d'utilisateur et à son mot de passe. Le serveur recherche alors les informations correspondantes dans la base de données. Si les identifiants sont corrects, une session utilisateur est créée.

Une session est un mécanisme permettant de conserver l'identité d'un utilisateur pendant sa navigation sur le site. Grâce à cette session, le système sait à tout moment quel utilisateur est connecté sans lui demander de se reconnecter à chaque page.

Selon le type de compte utilisé, le système redirige automatiquement l'utilisateur vers l'interface administrateur ou vers l'interface client.

5.1.2. Affichage des matchs

Une fois connecté, le supporter accède à une page regroupant l'ensemble des matchs disponibles à la réservation.



Les informations affichées ne sont pas enregistrées directement dans les pages HTML. Elles sont récupérées dynamiquement depuis la base de données PostgreSQL grâce à SQLAlchemy. Une requête est exécutée afin de sélectionner uniquement les matchs dont la date est supérieure à la date actuelle.

Cette vérification permet de masquer automatiquement les rencontres déjà jouées et de ne présenter à l'utilisateur que les événements encore disponibles.

Les données récupérées sont ensuite transmises à la page HTML grâce au framework Flask. Le moteur de templates Jinja2 génère automatiquement l'affichage de chaque match à partir des informations présentes dans la base de données.

Chaque carte contient notamment :

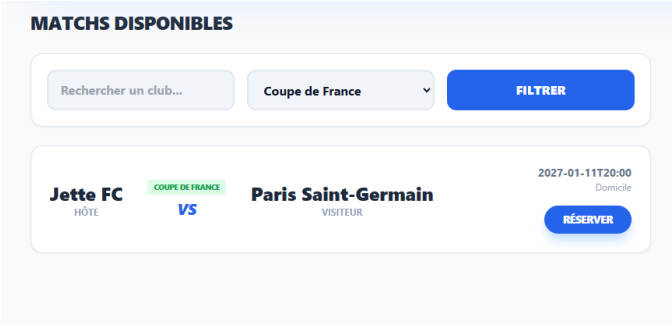
- le club organisateur ;
- l'adversaire ;
- la date ;
- le lieu ;
- la compétition associée.

Cette génération dynamique permet d'ajouter de nouveaux matchs sans devoir modifier manuellement les pages du site.

5.1.3. Filtrage des matchs

Afin de faciliter la recherche d'un événement, plusieurs outils de filtrage ont été intégrés à l'interface client. L'utilisateur peut notamment rechercher un match en fonction du club ou de la compétition.

Lorsqu'un filtre est sélectionné, les informations sont envoyées au serveur Flask sous forme de paramètres. Le serveur récupère ensuite ces paramètres et adapte la requête exécutée sur la base de données PostgreSQL.

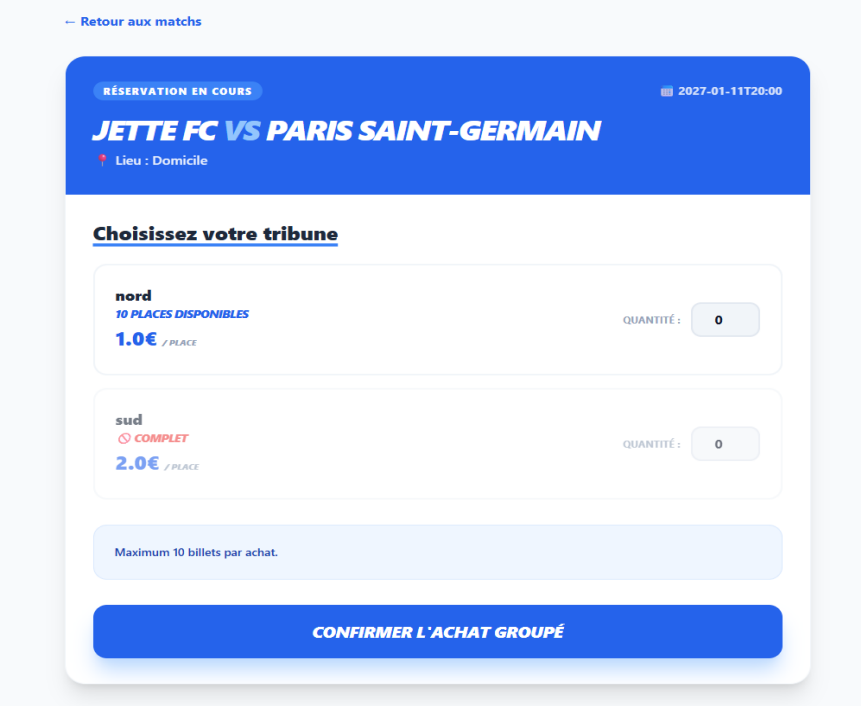


Seuls les matchs correspondant aux critères sélectionnés sont alors renvoyés à l'utilisateur. Cette méthode permet de réduire la quantité d'informations affichées et d'améliorer la navigation sur la plateforme.

Le filtrage est effectué dynamiquement à chaque recherche, ce qui garantit que les résultats affichés correspondent toujours aux données les plus récentes enregistrées dans la base de données.

5.1.4. Réservation des billets

Lorsqu'un utilisateur sélectionne un match, il accède à une page de réservation affichant les tribunes disponibles ainsi que leur prix.



Chaque tribune possède une capacité maximale définie par l'administrateur lors de sa création. Afin d'éviter les surréservations, le système calcule automatiquement le nombre de places restantes en comparant la capacité totale de la tribune avec le nombre de billets déjà vendus.

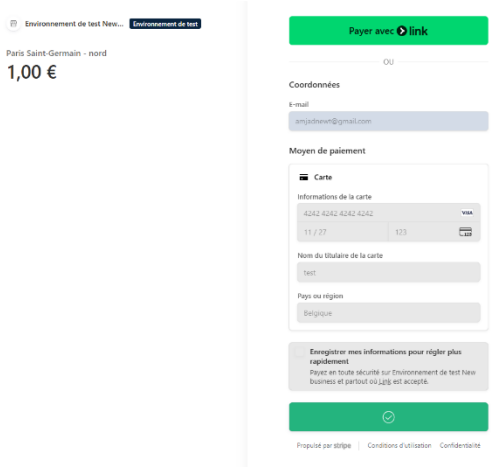
L'utilisateur peut sélectionner le nombre de places souhaitées directement depuis l'interface. Une limite de dix billets par achat a été mise en place afin de limiter les abus et de réduire les risques d'erreurs lors des commandes importantes.

Avant de poursuivre vers le paiement, plusieurs vérifications sont réalisées afin de s'assurer que les quantités demandées sont valides et que suffisamment de places sont encore disponibles.

5.1.5. Paiement en ligne avec Stripe

Une fois les billets sélectionnés, le client est redirigé vers une page de paiement sécurisée générée par Stripe.

Stripe est une plateforme spécialisée dans le traitement des paiements en ligne. Son utilisation permet de sécuriser les transactions sans stocker les informations bancaires des utilisateurs sur le serveur du projet.



Lors de la création du paiement, le serveur Flask génère une session Stripe contenant les informations relatives à la commande : montant total, billets sélectionnés et bénéficiaire du paiement.

Le client complète ensuite le paiement directement sur l'interface Stripe. Une fois la transaction validée, Stripe informe automatiquement l'application grâce à un système appelé webhook.

Cette méthode garantit qu'un billet n'est créé que lorsqu'un paiement a réellement été accepté.

5.1.6. Génération des billets et des QR codes

Après la validation du paiement, le webhook Stripe transmet les informations de la commande au serveur Flask. Le système crée alors les billets correspondants dans la base de données. Chaque billet est associé à un utilisateur, à un match et à une tribune spécifique. Pour chaque billet généré, un identifiant unique est créé automatiquement. Cet identifiant est ensuite utilisé pour générer un QR code grâce à une bibliothèque Python spécialisée. Le QR code constitue la preuve de réservation du supporter. Chaque QR code est unique et ne peut être associé qu'à un seul billet. Cette méthode permet d'éviter la duplication ou la falsification des billets.

5.1.7. Envoi automatique des emails

Une fois les billets générés, l'application envoie automatiquement un email au client.

Cet email contient le QR code correspondant au billet acheté. L'envoi est réalisé grâce à la bibliothèque Flask-Mail, configurée avec un compte Gmail dédié au projet.

Cette automatisation permet au client de recevoir immédiatement son billet sans intervention humaine.

L'utilisation du courrier électronique offre également une solution simple et universelle, compatible avec tous les appareils connectés à Internet.

5.1.8. Contrôle d'accès et validation des QR codes

Lorsqu'un supporter arrive au stade, son QR code est scanné à l'aide du système développé pour le projet.

Le système envoie alors le contenu du QR code au serveur Flask afin de vérifier sa validité.

Plusieurs contrôles sont effectués :

- existence du billet ;
- correspondance avec le club concerné ;
- vérification que le billet n'a pas déjà été utilisé.

Si toutes les vérifications sont validées, l'accès est autorisé. Dans le cas contraire, le système refuse l'entrée.

Après une validation réussie, le billet est automatiquement marqué comme utilisé dans la base de données afin d'empêcher toute réutilisation du même QR code.

5.1.9. Raspberry Pi et automatisation de la porte

Afin d'ajouter une dimension physique au projet, un système de contrôle d'accès a été développé à l'aide d'un Raspberry Pi 4.

Le Raspberry Pi agit comme un mini-ordinateur chargé de communiquer avec l'application web et les composants électroniques.

Lorsqu'un QR code valide est détecté, le Raspberry Pi reçoit l'autorisation d'ouverture. Il commande alors un servomoteur connecté à ses broches GPIO.

Le servomoteur permet de simuler l'ouverture d'un portique ou d'une barrière d'accès.

Après l'ouverture, un capteur ultrason mesure en permanence la distance devant la porte. Lorsque le passage d'une personne est détecté, le Raspberry Pi attend quelques secondes puis referme automatiquement la porte.

Cette partie du projet combine le développement logiciel avec l'électronique embarquée et démontre l'intégration d'un système informatique complet allant de la réservation en ligne jusqu'au contrôle physique de l'accès au stade.

5.2. Extraits de code commentés

5.2.1. Authentification d'un utilisateur

```
user = User.query.filter_by(username=username).first()

if user and user.password == password:
    session["user_id"] = user.id
```

Cet extrait permet de vérifier l'identité d'un utilisateur lors de la connexion. Le programme recherche d'abord l'utilisateur dans la base de données grâce à son nom d'utilisateur. Si celui-ci existe et que le mot de passe correspond, son identifiant est enregistré dans la session Flask. Cette session permet ensuite de garder l'utilisateur connecté pendant sa navigation sur le site.

5.2.2. Création d'une session de paiement Stripe

```
checkout_session = stripe.checkout.Session.create(
    payment_method_types=["card"],
    line_items=line_items,
    mode="payment",
    success_url=url_for("success", _external=True),
    cancel_url=url_for("cancel", _external=True)
)
```

Cet extrait crée une session de paiement Stripe. Les billets sélectionnés par l'utilisateur sont ajoutés dans la liste des articles à payer. Stripe génère ensuite une page de paiement sécurisée permettant à l'utilisateur de régler sa commande par carte bancaire. Une fois le paiement effectué, l'utilisateur est redirigé vers une page de confirmation.

5.2.3. Génération d'un QR Code

```
qr = qrcode.make(qr_data)
qr.save(qr_path)
```

Cet extrait génère automatiquement un QR Code unique pour chaque billet acheté. Les informations du billet sont encodées dans l'image afin de permettre une vérification rapide lors de l'entrée au stade. Le QR Code est ensuite enregistré sur le serveur pour pouvoir être utilisé ultérieurement.

5.2.4. Réception d'un paiement Stripe (Webhook)

```
if event["type"] == "checkout.session.completed":
    session = event["data"]["object"]
```

Ce code fait partie du webhook Stripe. Lorsqu'un paiement est terminé, Stripe envoie automatiquement un événement au serveur. L'application récupère alors les informations de la transaction afin de créer les billets correspondants dans la base de données. Cette méthode garantit que les billets ne sont créés qu'après validation effective du paiement.

5.3. Difficultés rencontrées

Au cours du développement du projet, plusieurs difficultés ont été rencontrées. La première a été le changement de langage et de technologies en au cours de l'année . Au départ, le projet avait été commencé avec l'application power apps, mais il a finalement été nécessaire de réorganiser une grande partie du travail afin d'utiliser Python, Flask et les technologies associées. Cela a demandé un temps d'adaptation important, car il a fallu comprendre le fonctionnement du framework Flask et restructurer certaines parties déjà développées.

Une autre difficulté importante a été la mise en place du système de paiement en ligne avec Stripe. Cette fonctionnalité était nouvelle et nécessitait la compréhension des sessions de paiement, des clés de sécurité, des webhooks ainsi que de la communication entre Stripe et l'application.

La gestion de la base de données a également représenté un défi. Il a fallu concevoir une structure cohérente permettant de gérer les utilisateurs, les clubs, les matchs, les tribunes et les billets tout en assurant les relations entre ces différentes tables.

La génération automatique des billets et des QR Codes a aussi demandé plusieurs essais. Chaque billet devait être unique, associé à un utilisateur précis et pouvoir être vérifié rapidement lors de l'entrée au stade. Il a donc fallu mettre en place un système fiable de génération et de stockage des QR Codes.

Le déploiement de l'application sur Internet a aussi été dure. Contrairement aux tests réalisés en local, l'application devait fonctionner correctement sur un serveur distant. Des problèmes de configuration, de connexion à la base de données PostgreSQL et d'intégration avec les services externes ont dû être résolus avant d'obtenir une version fonctionnelle accessible en ligne.

Enfin, la mise en place du système d'envoi d'e-mails automatiques a posé plusieurs problèmes techniques. Bien que le système fonctionne en environnement local, certaines contraintes liées à l'hébergement et aux connexions réseau ont nécessité un réarrangement.

5.4. Solutions apportées

Pour résoudre ces difficultés, plusieurs solutions ont été mises en place. Une phase d'apprentissage a été réalisée afin de maîtriser Python, Flask et les bibliothèques utilisées dans le projet. La documentation officielle ainsi que différents tutoriels ont été consultés pour que je comprenne les mécanismes nécessaires au développement.

Concernant Stripe, j'ai fait des tests dans un environnement sécurisé afin de comprendre le fonctionnement des paiements et des webhooks. Cela m'a permis d'intégrer progressivement les fonctionnalités que je voulais tout en limitant les risques d'erreurs.

Pour la base de données, j'ai créé plusieurs schémas avant de choisir la structure finale. Cette étape m'a permis d'identifier les relations entre les différentes entités et d'éviter certains problèmes de conception.

La génération des QR Codes a été automatisée grâce à une bibliothèque Python spécialisée. Chaque billet reçoit ainsi un identifiant unique permettant son contrôle lors de l'accès au stade.

Enfin, les différents problèmes rencontrés lors du déploiement ont été résolus grâce à l'utilisation de services adaptés tels que Render pour l'hébergement de l'application, Railway pour la base de données PostgreSQL et Stripe pour les paiements sécurisés. Les erreurs rencontrées ont été corrigées progressivement grâce à l'analyse des journaux d'exécution et à de nombreux tests réalisés tout au long du développement.

6. Tests et validation

6.1. Tests réalisés

Afin de vérifier le bon fonctionnement de l'application, plusieurs tests ont été réalisés tout au long du développement.

Un premier test a concerné l'authentification des utilisateurs. J'ai été vérifié qu'un utilisateur puisse créer un compte, se connecter et accéder uniquement aux fonctionnalités correspondant à son rôle. Les administrateurs ont accès à la gestion des matchs et des tribunes tandis que les utilisateurs classiques peuvent uniquement consulter les matchs et effectuer des réservations.

Des tests ont également été réalisés sur la gestion des matchs. L'objectif était de vérifier que les administrateurs puissent créer, modifier et supprimer des matchs sans provoquer d'erreurs dans la base de données.

Le système de réservation a également été testé. Plusieurs réservations ont été effectuées afin de vérifier que le nombre de places disponibles soit correctement mis à jour après chaque achat et qu'il soit impossible de dépasser la capacité maximale d'une tribune.

Une autre série de tests a été consacrée à l'intégration de Stripe. Des paiements fictifs ont été réalisés dans l'environnement de test afin de vérifier que les transactions soient correctement validées et que les billets soient créés uniquement après confirmation du paiement.

Enfin, la génération automatique des QR Codes a été testée afin de vérifier que chaque billet possède un identifiant unique et qu'aucun doublon ne soit créé.

6.2. Résultats obtenus

Les différents tests réalisés ont permis de valider le bon fonctionnement général de l'application.

L'inscription et la connexion des utilisateurs fonctionnent correctement. La gestion des rôles administrateur et utilisateur est également opérationnelle.

La création des matchs et des tribunes est correctement enregistrée dans la base de données PostgreSQL. Les informations sont conservées même après redémarrage du serveur.

Le système de réservation permet aux utilisateurs d'acheter des billets de manière simple et sécurisée. Les paiements Stripe sont correctement pris en compte et les billets sont générés automatiquement après validation de la transaction.

Les QR Codes sont créés sans erreur et sont associés au billet correspondant.

L'application est également accessible en ligne grâce à son déploiement sur Render, ce qui permet son utilisation depuis n'importe quel appareil connecté à Internet.

6.3. Bugs rencontrés

Au cours du développement il y a eu plusieurs bugs. D'abord, des erreurs sont apparues lors de la connexion entre l'application et la base de données PostgreSQL hébergée sur Railway. Certaines connexions étaient interrompues de manière inattendue et nécessitaient une reconfiguration des paramètres SQLAlchemy.

Ensuite, l'intégration du système de paiement Stripe a également provoqué plusieurs erreurs liées à la gestion des webhooks et à la validation des paiements.

Par ailleurs, il y a eu d'autre erreur pour l'envoi automatique des billets par courrier électronique. Bien que le système fonctionne en environnement local, certaines limitations liées à l'hébergement distant ont entraîné des difficultés de communication avec le serveur SMTP.

Enfin, certains problèmes de cohérence des données ont été observés lors des premiers tests de réservation, notamment lorsque plusieurs achats étaient réalisés successivement.

6.4. Corrections apportées

Pour résoudre ces problèmes, j'ai fait plusieurs correction. Premièrement, les paramètres de connexion à PostgreSQL ont été optimisés afin de garantir une meilleure stabilité entre l'application et la base de données. Deuxièmement, le système Stripe a été progressivement corrigé grâce à l'analyse des journaux d'exécution et à de nombreux tests réalisés dans l'environnement de développement fourni par Stripe. Des vérifications supplémentaires ont été ajoutées afin d'empêcher la création de réservations dépassant la capacité disponible des tribunes. De plus, une limite maximale de dix billets par réservation a également été mise en place afin d'éviter certains comportements imprévus et de simplifier la gestion des commandes.

Les différents problèmes détectés lors des phases de test ont permis d'améliorer la fiabilité générale de l'application avant sa version finale.

5.1. Coût du projet

Équipement / Service	Prix approximatif	Lien
Raspberry Pi 4	117€	Raspberry Pi
Capteur ultrason HC-SR04	8€ (pour trois)	ultrason
Servomoteur SG90	7.26€	Servo
Hébergement Render	0€	render
Base de données PostgreSQL (Railway)	0€	Railway
Stripe	0€ (hors commission)	Stripe
Total	132.26€	/

6. Conclusion

Ce travail de fin d'études m'a permis de développer une plateforme de billetterie en ligne destinée aux clubs de football amateur. L'objectif principal était de proposer une solution simple et accessible permettant aux clubs de créer leurs matchs, gérer leurs tribunes et vendre des billets en ligne de manière sécurisée. Pour les supporters, le système permet de consulter les rencontres, réserver des places et effectuer un paiement en ligne. Une partie physique a également été réalisée avec une maquette basée sur un Raspberry Pi afin de simuler le contrôle d'accès à l'entrée d'un stade.

Dans l'ensemble, les objectifs fixés au début du projet ont été atteints. Le site est fonctionnel, les matchs peuvent être créés, les réservations sont enregistrées dans la base de données et les paiements sont traités via Stripe. Le système de QR code et la maquette physique permettent également de montrer comment un billet numérique peut être utilisé dans un contexte réel.

La réalisation de ce projet m'a permis d'acquérir de nombreuses compétences. J'ai appris à développer une application web complète avec Flask, à utiliser une base de données PostgreSQL, à intégrer un système de paiement en ligne avec Stripe et à connecter une application web à une partie physique . Ce projet m'a également appris à mieux organiser mon travail sur une longue période, à rechercher des solutions de manière autonome et surtout à trouver du temps pour avancer sur mon TFE tout en continuant à travailler sur les autres cours de l'année.

Même si le projet est fonctionnel, plusieurs améliorations pourraient être ajoutées à l'avenir. Par exemple, il serait intéressant de mettre en place un système d'abonnement permettant aux supporters de suivre leur club. Un système de notifications pourrait également être ajouter pour informer automatiquement les supporters lorsqu'un nouveau match est créé. D'autres fonctionnalités que j'ai imaginées comme une application mobile, des statistiques de fréquentation ou encore un système de réduction pour les abonnés pourraient également être intéressent .

Pour conclure, ce projet a été une expérience très enrichissante. Il m'a permis de mettre en pratique les connaissances acquises pendant ma formation tout en réalisant un projet concret répondant à un besoin réel. Malgré les difficultés rencontrées au cours du développement, le résultat final est une application complète qui combine développement web, base de données, paiement en ligne et électronique embarquée.

7. Bibliographie / Webographie

- Eventbrite. (s.d.). *Eventbrite*. Consulté le 24 février 2026, à l'adresse : https://www.eventbrite.com/
- Pallets Projects. (s.d.). *Flask Documentation*. Consulté le 12 mars 2026, à l'adresse : https://flask.palletsprojects.com/
- Python Software Foundation. (s.d.). *Python Documentation*. Consulté le 15 avril 2026, à l'adresse : https://docs.python.org/3/
- Raspberry Pi Foundation. (s.d.). *Raspberry Pi Documentation*. Consulté le 5 avril 2026, à l'adresse : https://www.raspberrypi.com/documentation/
- Railway Corp. (s.d.). *Railway Documentation*. Consulté le 28 mars 2026, à l'adresse : https://docs.railway.com/
- Render Services, Inc. (s.d.). *Render Documentation*. Consulté le 2 avril 2026, à l'adresse : https://render.com/docs
- PostgreSQL Global Development Group. (s.d.). *PostgreSQL Documentation*. Consulté le 18 mars 2026, à l'adresse : https://www.postgresql.org/docs/
- SQLAlchemy Authors. (s.d.). *SQLAlchemy Documentation*. Consulté le 15 mars 2026, à l'adresse : https://www.sqlalchemy.org/
- Stripe, Inc. (s.d.). *Stripe Documentation*. Consulté le 25 mars 2026, à l'adresse : https://docs.stripe.com/
- Ticketmaster. (s.d.). *Ticketmaster*. Consulté le 20 février 2026, à l'adresse : https://www.ticketmaster.com/
- Weezevent. (s.d.). *Weezevent*. Consulté le 22 février 2026, à l'adresse : https://weezevent.com/

8. Annexes

8.1. Application principale (app.py)

Cette annexe contient le fichier principal de l'application. Il regroupe les routes Flask, la gestion des utilisateurs, les réservations, les paiements Stripe, la génération des billets ainsi que la communication avec la maquette physique.

```python
from flask import Flask, render_template, request, redirect, url_for, session, flash
from models import db, User, Club, Tribune, Match, Billet, Competition
from config import Config
from datetime import datetime
from flask_mail import Mail, Message
import os
import socket


from flask import request



app = Flask(__name__)
app.config.from_object(Config)
db.init_app(app)
mail = Mail(app)

import stripe
stripe.api_key = app.config['STRIPE_SECRET_KEY']
endpoint_secret = os.environ.get("STRIPE_WEBHOOK_SECRET")


# Création de la base et d'un club par défaut
with app.app_context():
    db.create_all()
    # Ajout de quelques compétitions par défaut
    if not Competition.query.first():
        competitions = [
            Competition(nom="Ligue 1", pays="France"),
            Competition(nom="Coupe de France", pays="France"),
            Competition(nom="Premier League", pays="Angleterre"),
            Competition(nom="La Liga", pays="Espagne"),
            Competition(nom="Ligue des Champions", pays="International"),
            Competition(nom="Europa League", pays="International")
        ]
        db.session.add_all(competitions)

    if not Club.query.first():
        db.session.add(Club(nom="Paris Saint-Germain", pays="France"))
        db.session.commit()
```

```python
@app.route('/')
def home():

    user_id = session.get('user_id')

    if not user_id:
        return redirect(url_for('login'))

    user = User.query.get(user_id)

    if not user:
        return redirect(url_for('login'))

    # 🔥 ADMIN
    if user.is_admin == True:
        return render_template(
            'admin_dashboard.html',
            user=user
        )

    # 🔥 CLIENT
    maintenant = datetime.now().strftime('%Y-%m-%dT%H:%M')

    club_search = request.args.get("club_search", "")
    competition_id = request.args.get("competition_id", "")

    query = Match.query.filter(Match.date >= maintenant)

    # 🔥 filtre club
    if club_search:
        query = query.join(Club).filter(
            Club.nom.ilike(f"%{club_search}%")
        )

    # 🔥 filtre compétition
    if competition_id:
        query = query.filter(
            Match.competition_id == int(competition_id)
        )

    tous_les_matchs = query.all()
    competitions = Competition.query.all()

```

```
    return render_template(
        'client_dashboard.html',
        user=user,
        matchs=tous_les_matchs,
        competitions=competitions,
        selected_club_search=club_search,
        selected_competition=competition_id
    )


@app.route('/login', methods=['GET', 'POST'])
def login():

    if request.method == 'POST':

        nom = request.form.get('username')
        mdp = request.form.get('password')

        user = User.query.filter_by(username=nom).first()

        if user and user.password == mdp:

            session['user_id'] = user.id

            # 🔥 ADMIN
            if user.is_admin == True:
                return render_template(
                    'admin_dashboard.html',
                    user=user
                )

            # 🔥 CLIENT
            maintenant = datetime.now().strftime('%Y-%m-%dT%H:%M')

            tous_les_matchs = Match.query.filter(
                Match.date >= maintenant
            ).all()

            competitions = Competition.query.all()
```

```
            return render_template(
                'client_dashboard.html',
                user=user,
                matchs=tous_les_matchs,
                competitions=competitions,
                selected_club_search="",
                selected_competition=""
            )

        flash("Identifiants incorrects", "error")

    return render_template('login.html')

@app.route('/register/client', methods=['GET', 'POST'])
def register_client():
    if request.method == 'POST':
        nom = request.form.get('username')
        mail = request.form.get('email')
        mdp = request.form.get('password')

        existe = User.query.filter(
            (User.username == nom) | (User.email == mail)
        ).first()

        if existe:
            flash("Username ou email déjà utilisé", "error")
            return redirect(url_for('register_client'))

        nouveau = User(username=nom, email=mail, password=mdp, is_admin=False)
        db.session.add(nouveau)
        db.session.commit()

        # ✅ AJOUT ICI (IMPORTANT)
        envoyer_email(nouveau.email, nouveau.username)

        flash("Compte créé avec succès ! Connectez-vous.", "success")
        return redirect(url_for('login'))

    return render_template('register_client.html')




    return render_template('register_client.html')
    print("EMAIL TEST EN COURS")
```

```python
@app.route('/register/admin', methods=['GET', 'POST'])
def register_admin():
    if request.method == 'POST':
        nom = request.form.get('username')
        mail = request.form.get('email')
        mdp = request.form.get('password')
        nom_club = request.form.get('club_name')
        pays_club = request.form.get('pays')

        club = Club.query.filter_by(nom=nom_club).first()
        if not club:
            club = Club(nom=nom_club, pays=pays_club)
            db.session.add(club)
            db.session.flush()

        # 🔴 CREATE STRIPE ACCOUNT
        compte = stripe.Account.create(
            type="express",
            country="FR",
            email=mail
        )

        # 🟢 CREATE ONBOARDING LINK (TON CODE ICI)
        link = stripe.AccountLink.create(
            account=compte.id,
            refresh_url=url_for('home', _external=True),
            return_url=url_for('home', _external=True),
            type="account_onboarding",
        )

        nouveau = User(
            username=nom,
            email=mail,
            password=mdp,
            is_admin=True,
            club_id=club.id,
            stripe_account_id=compte.id
        )

        db.session.add(nouveau)
        db.session.commit()

        return redirect(link.url)

    return render_template('register_admin.html')
```

```python
def envoyer_email(destinataire, username):
    msg = Message(
        "Bienvenue sur Stadium Manager",
        recipients=[destinataire]
    )
    msg.body = f"Bonjour {username}, merci pour votre inscription !"


    mail.send(msg)


# 🔥 AJOUTE ICI ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

import qrcode
import os

def generate_qr(data, filename):
    base_dir = os.path.dirname(os.path.abspath(__file__))

    folder = os.path.join(base_dir, "static", "qrcodes")

    # 🔥 crée le dossier SI il existe pas
    os.makedirs(folder, exist_ok=True)

    path = os.path.join(folder, filename)

    qr = qrcode.make(data)
    qr.save(path)

    return path


def envoyer_billet_email(destinataire, qr_filename):
    try:
        msg = Message(
            "🎟️ Ton billet",
            recipients=[destinataire]
        )

        msg.body = "Voici ton billet avec ton QR code"

        base_dir = os.path.dirname(os.path.abspath(__file__))
        path = os.path.join(base_dir, "static", "qrcodes", qr_filename)

        with open(path, "rb") as fp:
            msg.attach(qr_filename, "image/png", fp.read())

        socket.setdefaulttimeout(10)
```

```
        mail.send(msg)
        print("✅ EMAIL ENVOYÉ :", destinataire)

    except Exception as e:
        print("❌ ERREUR EMAIL :", e)

@app.route('/add_tribune', methods=['POST'])
def add_tribune():
    if 'user_id' not in session:
        return redirect(url_for('login'))

    user = User.query.get(session['user_id'])
    if not user or not user.is_admin:
        return redirect(url_for('home'))

    nom = request.form.get('nom')
    capa = request.form.get('capacite')
    prix = request.form.get('prix')

    if nom and capa and prix:
        nouvelle_tribune = Tribune(
            nom=nom,
            capacite=int(capa),
            prix=float(prix),
            club_id=user.club_id
        )
        db.session.add(nouvelle_tribune)
        db.session.commit()
        flash("Tribune ajoutée avec succès !", "success")

    return redirect(url_for('home'))

@app.route('/delete_tribune/<int:id>')
def delete_tribune(id):
    if 'user_id' not in session:
        return redirect(url_for('login'))

    user = User.query.get(session['user_id'])
    tribune = Tribune.query.get(id)

    # Sécurité : On vérifie que la tribune existe ET qu'elle appartient bien au club de l'admin
    if tribune and user and user.is_admin and tribune.club_id == user.club_id:
        db.session.delete(tribune)
        db.session.commit()
        flash("Tribune supprimée !", "success")

    return redirect(url_for('home'))
```

```
@app.route('/create_match')
def create_match():
    if 'user_id' not in session:
        return redirect(url_for('login'))
    user = User.query.get(session['user_id'])
    if not user or not user.is_admin:
        return redirect(url_for('home'))

    # On récupère tous les clubs pour la barre de recherche (sauf le nôtre)
    tous_les_clubs = Club.query.filter(Club.id != user.club_id).all()
    # On récupère les tribunes de l'admin
    ses_tribunes = Tribune.query.filter_by(club_id=user.club_id).all()

    # On récupère les compétitions éligibles (Pays du club + International)
    competitions = Competition.query.filter(
        (Competition.pays == user.club.pays) | (Competition.pays == "International")
    ).all()

    return render_template('create_match.html', user=user, clubs=tous_les_clubs, tribunes=ses_tribunes, competitions=competitions)

@app.route('/add_match', methods=['POST'])
def add_match():
    if 'user_id' not in session:
        return redirect(url_for('login'))

    user = User.query.get(session['user_id'])
    if not user or not user.is_admin:
        return redirect(url_for('home'))

    adversaire = request.form.get('adversaire')
    date_str = request.form.get('date')
    lieu = request.form.get('lieu')
    tribunes_ids = request.form.getlist('tribunes')
    competition_id = request.form.get('competition_id')

    # CONDITION 1 : On vérifie que TOUT est rempli
    if not adversaire or not date_str or not lieu or not tribunes_ids or not competition_id:
        flash("Erreur : Tous les champs doivent être remplis (n'oubliez pas la compétition et les tribunes !)", "error")
        return redirect(url_for('create_match'))

    # CONDITION 2 : On vérifie que la date n'est pas dans le passé
    date_objet = datetime.strptime(date_str, '%Y-%m-%dT%H:%M')
    maintenant = datetime.now()

    if date_objet < maintenant:
        flash("Erreur : Vous ne pouvez pas programmer un match dans le passé !", "error")
        return redirect(url_for('create_match'))

```

```
    nouveau_match = Match(
        adversaire=adversaire,
        date=date_str,
        lieu=lieu,
        club_id=user.club_id,
        competition_id=competition_id
    )

    for t_id in tribunes_ids:
        tribune = Tribune.query.get(t_id)
        if tribune:
            nouveau_match.tribunes_ouvertes.append(tribune)

    db.session.add(nouveau_match)
    db.session.commit()
    flash("Match programmé avec succès !", "success")
    return redirect(url_for('home'))

@app.route('/delete_match/<int:id>')
def delete_match(id):
    if 'user_id' not in session:
        return redirect(url_for('login'))

    user = User.query.get(session['user_id'])
    match = Match.query.get(id)

    if match and user and user.is_admin and match.club_id == user.club_id:
        db.session.delete(match)
        db.session.commit()
        flash("Match supprimé !", "success")

    return redirect(url_for('home'))

@app.route('/reserver/<int:match_id>')
def reserver(match_id):
    if 'user_id' not in session:
        return redirect(url_for('login'))

    user = User.query.get(session['user_id'])
    match = Match.query.get(match_id)

    if not match:
        flash("Match introuvable", "error")
        return redirect(url_for('home'))

    places_restantes = {
        tribune.id: tribune.capacite - Billet.query.filter_by(
```

```
            match_id=match.id,
            tribune_id=tribune.id
        ).count()
        for tribune in match.tribunes_ouvertes
    }

    return render_template(
        'reserver.html',
        user=user,
        match=match,
        places=places_restantes
    )


@app.route('/confirmer_achat', methods=['POST'])
def confirmer_achat():

    if 'user_id' not in session:
        return redirect(url_for('login'))

    match = Match.query.get(request.form.get('match_id'))

    if not match:
        return redirect(url_for('home'))

    admin = User.query.filter_by(
        club_id=match.club_id,
        is_admin=True
    ).first()

    if not admin or not admin.stripe_account_id:
        return "Erreur : compte Stripe non configuré pour ce club", 400

    tribunes_data = {}
    line_items = []
    total_centimes = 0
    total_billets = 0

    for tribune in match.tribunes_ouvertes:

        qty = int(request.form.get(f'qty_{tribune.id}') or 0)

        if qty <= 0:
            continue

        total_billets += qty
```

```
        # 🔥 LIMITE
        if total_billets > 10:

            flash(
                "Vous ne pouvez pas acheter plus de 10 billets.",
                "error"
            )

            return redirect(
                url_for('reserver', match_id=match.id)
            )

        prix_centimes = int(tribune.prix * 100)

        tribunes_data[tribune.id] = qty

        total_centimes += prix_centimes * qty

        line_items.append({
            'price_data': {
                'currency': 'eur',
                'product_data': {
                    'name': f"{match.adversaire} - {tribune.nom}",
                },
                'unit_amount': prix_centimes,
            },
            'quantity': qty,
        })

    if not line_items:

        flash("Sélectionne au moins un billet", "error")

        return redirect(
            url_for('reserver', match_id=match.id)
        )

    session_stripe = stripe.checkout.Session.create(

        payment_method_types=['card'],

        line_items=line_items,

        mode='payment',

        payment_intent_data={
```

```
        payment_intent_data={
            "application_fee_amount": int(total_centimes * 0.10),
            "transfer_data": {
                "destination": admin.stripe_account_id
            }
        },

        success_url=url_for(
            'success',
            _external=True
        ) + f"?match_id={match.id}",

        cancel_url=url_for(
            'reserver',
            match_id=match.id,
            _external=True
        ),

        metadata={
            "match_id": str(match.id),
            "user_id": str(session['user_id']),
            "tribunes": str(tribunes_data)
        }
    )

    return redirect(session_stripe.url)


@app.route('/stripe/webhook', methods=['POST'])
def stripe_webhook():

    import uuid
    import ast

    try:
        event = stripe.Webhook.construct_event(
            request.data,
            request.headers.get('Stripe-Signature'),
            endpoint_secret
        )

    except Exception as e:
        print("❌ ERROR WEBHOOK:", e)
        return "error", 400

    if event['type'] != 'checkout.session.completed':
        return "ok", 200
```

```
548         return "ok", 200
549
550     session_data = event['data']['object']
551     session_id = session_data['id']
552
553     # 🔥 anti doublon
554     already_done = Billet.query.filter_by(
555         stripe_session_id=session_id
556     ).first()
557
558     if already_done:
559         print("⚠️ Déjà traité")
560         return "ok", 200
561
562     match = Match.query.get(
563         int(session_data['metadata']['match_id'])
564     )
565
566     user = User.query.get(
567         int(session_data['metadata']['user_id'])
568     )
569
570     tribunes = ast.literal_eval(
571         session_data['metadata']['tribunes']
572     )
573
574     if not match or not user:
575         return "error", 400
576
577     emails_a_envoyer = []
578
579     for tribune_id, qty in tribunes.items():
580
581         tribune = Tribune.query.get(int(tribune_id))
582
583         if not tribune:
584             continue
585
586         for _ in range(qty):
587
588             qr_token = str(uuid.uuid4())
589             filename = f"{qr_token}.png"
590
591             generate_qr(qr_token, filename)
592
593             billet = Billet(
594                 user_id=user.id,
```

```
                user_id=user.id,
                match_id=match.id,
                tribune_id=tribune.id,
                stripe_session_id=session_id,
                qr_code=qr_token,
                is_used=False
            )

            db.session.add(billet)

            emails_a_envoyer.append(filename)

    # 🔥 commit AVANT emails
    db.session.commit()

    # 🔥 emails après commit
    print("AVANT EMAIL")
    for filename in emails_a_envoyer:
        envoyer_billet_email(user.email, filename)
    print("APRES EMAIL")

    print("✅ OK PAYEMENT + QR + EMAIL")

    return "ok", 200

@app.route('/success')
def success():
    match_id = request.args.get('match_id')

    match = Match.query.get(match_id)

    if not match:
        return redirect(url_for('home'))

    return render_template("success.html", match=match)









import requests

```

```python

@app.route('/scan/<qr_token>')
def scan_qr(qr_token):
    billet = Billet.query.filter_by(qr_code=qr_token).first()

    if not billet:
        return "❌ QR invalide"

    if billet.is_used:
        return "⚠️ Déjà utilisé"

    # 🔥 MARQUE UTILISÉ
    billet.is_used = True
    db.session.commit()

    # 🔥 OUVRIR LE PORTIQUE (RASPBERRY)
    try:
        requests.get("http://192.168.0.189:5001/open")
    except:
        print("Erreur connexion Raspberry")

    return "✅ Accès autorisé"



@app.route('/scanner')
def scanner():
    return render_template("scanner.html")

@app.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('login'))



@app.route('/check/<qr_token>/<int:club_id>')
def check(qr_token, club_id):

    billet = Billet.query.filter_by(qr_code=qr_token).first()

    if not billet:
        return {"status": "invalid"}

    if billet.is_used:
        return {"status": "used"}

```

```python

        # 🔥 vérif club
        if billet.match.club_id != club_id:
            return {"status": "wrong_club"}

        # ✅ OK → on valide le billet
        billet.is_used = True
        db.session.commit()

        return {"status": "ok"}

    if __name__ == '__main__':
        app.run(host="0.0.0.0", port=3000, debug=True)
```

8.2. Modèles de la base de données (models.py)

Cette annexe contient les modèles SQLAlchemy utilisés pour créer la base de données. Les tables permettent de gérer les utilisateurs, les clubs, les matchs, les tribunes, les compétitions et les billets.

```python
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class Club(db.Model):
    __tablename__ = 'clubs' # On donne un nom clair à la table
    id = db.Column(db.Integer, primary_key=True)
    nom = db.Column(db.String(100), nullable=False)
    pays = db.Column(db.String(50), default="France") # Pays du club

class Competition(db.Model):
    __tablename__ = 'competitions'
    id = db.Column(db.Integer, primary_key=True)
    nom = db.Column(db.String(100), nullable=False)
    pays = db.Column(db.String(50), nullable=False) # Nom du pays ou "International"

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(80), nullable=False)
    is_admin = db.Column(db.Boolean, default=False)
    club_id = db.Column(db.Integer, db.ForeignKey('clubs.id'), nullable=True)
    club = db.relationship('Club', backref='membres')
    stripe_account_id = db.Column(db.String(255), nullable=True)

class Tribune(db.Model):
    __tablename__ = 'tribunes'
    id = db.Column(db.Integer, primary_key=True)
    nom = db.Column(db.String(100), nullable=False)
    capacite = db.Column(db.Integer, nullable=False)
    prix = db.Column(db.Float, nullable=False)
    club_id = db.Column(db.Integer, db.ForeignKey('clubs.id'), nullable=False)
    club = db.relationship('Club', backref='tribunes')

# Table d'association pour l'accès aux tribunes par match
match_tribunes = db.Table('match_tribunes',
    db.Column('match_id', db.Integer, db.ForeignKey('matchs.id'), primary_key=True),
    db.Column('tribune_id', db.Integer, db.ForeignKey('tribunes.id'), primary_key=True)
)
```

```python
class Match(db.Model):
    __tablename__ = 'matchs'
    id = db.Column(db.Integer, primary_key=True)
    adversaire = db.Column(db.String(100), nullable=False)
    date = db.Column(db.String(50), nullable=False)
    lieu = db.Column(db.String(20), default="Domicile") # Domicile ou Extérieur
    club_id = db.Column(db.Integer, db.ForeignKey('clubs.id'), nullable=False)
    club = db.relationship('Club', backref='matchs')

    # Lien avec la compétition
    competition_id = db.Column(db.Integer, db.ForeignKey('competitions.id'), nullable=True)
    competition = db.relationship('Competition', backref='matchs')

    # Relation avec les tribunes ouvertes pour ce match
    tribunes_ouvertes = db.relationship('Tribune', secondary=match_tribunes, backref='matchs_prevus')

class Billet(db.Model):
    __tablename__ = 'billets'
    id = db.Column(db.Integer, primary_key=True)

    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
    match_id = db.Column(db.Integer, db.ForeignKey('matchs.id'), nullable=False)
    tribune_id = db.Column(db.Integer, db.ForeignKey('tribunes.id'), nullable=False)

    stripe_session_id = db.Column(db.String) # 🔥 AJOUT IMPORTANT

    user = db.relationship('User', backref='mes_billets')
    match = db.relationship('Match', backref='billets_vendus')
    tribune = db.relationship('Tribune', backref='reservations')
    qr_code = db.Column(db.String, unique=True)  # 🔥 IMPORTANT
    is_used = db.Column(db.Boolean, default=False)
```

8.3. Configuration du projet (config.py)

Cette annexe contient les paramètres de configuration de l'application, notamment la connexion à PostgreSQL, les paramètres d'envoi d'e-mails et les clés nécessaires à l'intégration de Stripe.

```python
import os

class Config:
    SQLALCHEMY_DATABASE_URI = os.environ.get("DATABASE_URL")
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    SQLALCHEMY_ENGINE_OPTIONS = {
        "pool_pre_ping": True,
        "pool_recycle": 300,
        "connect_args": {
            "sslmode": "require"
        }
    }

    SECRET_KEY = os.environ.get("SECRET_KEY")

    MAIL_SERVER = "smtp.gmail.com"
    MAIL_PORT = 465
    MAIL_USE_TLS = False
    MAIL_USE_SSL = True
    MAIL_USERNAME = os.environ.get("MAIL_USERNAME")
    MAIL_PASSWORD = os.environ.get("MAIL_PASSWORD")
    MAIL_DEFAULT_SENDER = os.environ.get("MAIL_USERNAME")

    STRIPE_SECRET_KEY = os.environ.get("STRIPE_SECRET_KEY")
    STRIPE_PUBLIC_KEY = os.environ.get("STRIPE_PUBLIC_KEY")
```