

# Documentation Technique - JO E-Tickets

## Table des matières

1. [Introduction](#)
2. [Architecture système](#)
3. [Modèles de données](#)
4. [Sécurité](#)
5. [Processus de développement](#)
6. [Infrastructure et déploiement](#)
7. [API et interfaces](#)
8. [Évolutions futures](#)
9. [Annexes](#)

## 1. Introduction

### 1.1 Objectif du document

Cette documentation technique décrit les aspects techniques de l'application JO E-Tickets, un système de billetterie électronique développé pour les Jeux Olympiques de Paris 2024. Elle est destinée aux développeurs, administrateurs système et responsables de la sécurité impliqués dans le développement, la maintenance et l'évolution future du système.

### 1.2 Périmètre fonctionnel

L'application JO E-Tickets permet aux visiteurs de consulter et d'acheter des billets pour les événements des Jeux Olympiques 2024, avec des offres solo, duo et familiales. Le système génère des billets électroniques sécurisés qui peuvent être vérifiés par les employés le jour de l'événement.

### 1.3 Technologies utilisées

- **Backend:** Python 3.9+, Flask 2.2+
- **Frontend:** HTML5, CSS3 (Bootstrap 5), JavaScript
- **Base de données:** SQLite (dev), PostgreSQL (prod)
- **ORM:** SQLAlchemy
- **Sécurité:** Bcrypt, PyOTP, JWT
- **Génération de PDF:** ReportLab
- **QR Codes:** qrcode
- **Déploiement:** Docker, Nginx, Gunicorn

## 2. Architecture système

### 2.1 Vue d'ensemble

L'application JO E-Tickets suit une architecture MVC (Modèle-Vue-Contrôleur) modulaire utilisant le framework Flask. Cette architecture est organisée autour de différents modules (blueprints) correspondant aux fonctionnalités principales du système.



Afficher l'image

### 2.2 Composants principaux

#### 2.2.1 Frontend

Le frontend est développé en HTML5/CSS3 avec Bootstrap 5 pour la mise en page responsive. Les templates utilisent le moteur Jinja2 intégré à Flask. L'interactivité est assurée par JavaScript (vanilla) et quelques bibliothèques spécifiques (Chart.js pour les statistiques administrateur).

#### 2.2.2 Backend

Le backend est développé en Python avec le framework Flask. Il est structuré en plusieurs modules (blueprints) :

- **auth:** Gestion de l'authentification et des comptes utilisateurs
- **offers:** Gestion des offres de billets
- **cart:** Gestion du panier d'achat
- **orders:** Gestion des commandes
- **tickets:** Génération et vérification des billets
- **admin:** Interface d'administration
- **main:** Pages principales du site

#### 2.2.3 Services

Des services métier transversaux sont implémentés pour encapsuler la logique métier complexe :

- **AuthService:** Services d'authentification et de sécurité
- **PaymentService:** Simulation de paiement
- **QRCodeService:** Génération et vérification des QR codes
- **KeyService:** Gestion des clés de sécurité

## **2.2.4 Base de données**

Le schéma relationnel comporte les entités principales suivantes :

- User
- Offer
- Cart/CartItem
- Order/OrderItem
- Ticket

Les relations entre ces entités sont détaillées dans la section 3 (Modèles de données).

## **2.3 Flux de données**

### **2.3.1 Flux d'achat de billet**

1. L'utilisateur s'authentifie
2. L'utilisateur ajoute des offres à son panier
3. L'utilisateur procède au paiement
4. Le système crée une commande et génère une clé d'achat
5. Le système combine la clé utilisateur et la clé d'achat pour créer la clé finale
6. Le système génère un QR code à partir de la clé finale
7. Le système génère les billets électroniques

### **2.3.2 Flux de vérification de billet**

1. L'employé scanne le QR code d'un billet
2. Le système décide les informations du QR code
3. Le système vérifie l'authenticité du billet via les clés
4. Le système vérifie si le billet n'a pas déjà été utilisé
5. Le système marque le billet comme utilisé s'il est valide

## **3. Modèles de données**

### **3.1 Schéma de la base de données**

Le schéma relationnel complet est disponible en annexe. Voici les principaux modèles et leurs relations :

#### **3.1.1 User**

```
python
```

```
class User(db.Model, UserMixin):  
    __tablename__ = 'users'  
  
    id = db.Column(db.Integer, primary_key=True)  
    username = db.Column(db.String(50), unique=True, nullable=False)  
    email = db.Column(db.String(120), unique=True, nullable=False)  
    nom = db.Column(db.String(50), nullable=False)  
    prenom = db.Column(db.String(50), nullable=False)  
    password = db.Column(db.String(120), nullable=False)  
    cle_securite = db.Column(db.String(255), unique=True, nullable=False)  
    date_creation = db.Column(db.DateTime, default=datetime.utcnow)  
    derniere_connexion = db.Column(db.DateTime, nullable=True)  
    role = db.Column(db.String(20), default='utilisateur')  
    est_verifie = db.Column(db.Boolean, default=False)  
    code_verification = db.Column(db.String(100), nullable=True)  
    code_2fa_secret = db.Column(db.String(32), nullable=True)  
    est_2fa_active = db.Column(db.Boolean, default=False)  
  
    # Relations  
    carts = db.relationship('Cart', backref='user', lazy=True)  
    orders = db.relationship('Order', backref='user', lazy=True)  
    tickets = db.relationship('Ticket', backref='user', lazy=True)
```

### 3.1.2 Offer

```
python
```

```
class Offer(db.Model):
    __tablename__ = 'offers'

    id = db.Column(db.Integer, primary_key=True)
    titre = db.Column(db.String(100), nullable=False)
    description = db.Column(db.Text, nullable=False)
    type = db.Column(db.String(20), nullable=False) # 'solo', 'duo', 'familiale'
    nombre_personnes = db.Column(db.Integer, nullable=False) # 1, 2 ou 4
    prix = db.Column(db.Float, nullable=False)
    disponibilite = db.Column(db.Integer, nullable=False, default=100)
    date_evenement = db.Column(db.DateTime, nullable=False)
    image = db.Column(db.String(255), nullable=True)
    est_publie = db.Column(db.Boolean, default=True)
    date_creation = db.Column(db.DateTime, default=datetime.utcnow)
    date_modification = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    # Relations
    tickets = db.relationship('Ticket', backref='offer', lazy=True)
```

### 3.1.3 Order et Ticket

```
python
```

```
class Order(db.Model):
    __tablename__ = 'orders'

    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
    reference = db.Column(db.String(50), unique=True, nullable=False)
    total = db.Column(db.Float, nullable=False)
    statut = db.Column(db.String(20), default='en attente') # 'en attente', 'payée', 'annulée'
    date_commande = db.Column(db.DateTime, default=datetime.utcnow)
    date_paiement = db.Column(db.DateTime, nullable=True)
    cle_achat = db.Column(db.String(255), unique=True, nullable=False)
    adresse_email = db.Column(db.String(120), nullable=False)

    # Relations
    items = db.relationship('OrderItem', backref='order', lazy=True, cascade="all, delete-orphan")
    tickets = db.relationship('Ticket', backref='order', lazy=True)

class Ticket(db.Model):
    __tablename__ = 'tickets'

    id = db.Column(db.Integer, primary_key=True)
    order_id = db.Column(db.Integer, db.ForeignKey('orders.id'), nullable=False)
    offer_id = db.Column(db.Integer, db.ForeignKey('offers.id'), nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
    cle_utilisateur = db.Column(db.String(255), nullable=False)
    cle_achat = db.Column(db.String(255), nullable=False)
    cle_billet = db.Column(db.String(255), unique=True, nullable=False)
    qr_code = db.Column(db.Text, nullable=True) # Stockage en base64
    est_valide = db.Column(db.Boolean, default=True)
    date_generation = db.Column(db.DateTime, default=datetime.utcnow)
    date_utilisation = db.Column(db.DateTime, nullable=True)
```

## 3.2 Migrations de base de données

Les migrations de base de données sont gérées via Flask-Migrate (basé sur Alembic). Toutes les modifications du schéma sont versionnées et peuvent être appliquées/annulées via les commandes suivantes :

```
bash
```

```
# Générer une migration
flask db migrate -m "Description de la migration"

# Appliquer Les migrations
flask db upgrade

# Annuler La dernière migration
flask db downgrade
```

## 4. Sécurité

### 4.1 Authentification et autorisation

#### 4.1.1 Authentification utilisateur

- **Hachage des mots de passe** : Utilisation de bcrypt pour stocker les mots de passe de manière sécurisée
- **Politique de mot de passe** : Exigence d'au moins 8 caractères, avec majuscules, minuscules, chiffres et caractères spéciaux
- **Authentification à deux facteurs (2FA)** : Support de l'authentification TOTP via PyOTP
- **Protection anti-force brute** : Limitation du nombre de tentatives de connexion
- **Sessions sécurisées** : Utilisation de Flask-Login avec sessions cryptées et expiration automatique

#### 4.1.2 Contrôle d'accès

- **RBAC (Role-Based Access Control)** : Différenciation des accès selon le rôle (utilisateur, employé, administrateur)
- **Middleware d'autorisation** : Vérification systématique des permissions pour chaque route protégée
- **Vérification de propriété** : Contrôle que l'utilisateur accède uniquement à ses propres ressources

```

python

@tickets_bp.route('/<int:ticket_id>')
@login_required
def detail(ticket_id):
    ticket = Ticket.query.get_or_404(ticket_id)

    # Vérifier que l'utilisateur est bien le propriétaire du billet
    if ticket.user_id != current_user.id:
        flash('Vous n\'êtes pas autorisé à accéder à ce billet.', 'danger')
        return redirect(url_for('tickets.index'))

    ...
    return render_template('tickets/detail.html', ticket=ticket)

```

## 4.2 Sécurité des billets

### 4.2.1 Système de double clé

Le système utilise un mécanisme de double clé pour sécuriser les billets :

1. **Clé utilisateur** : Générée à l'inscription, unique par utilisateur
2. **Clé d'achat** : Générée lors de l'achat, unique par commande
3. **Clé billet** : Combinaison des deux précédentes, unique par billet

python

```

def generate_user_key(user_id, email):
    key_material = f"{uuid.uuid4()}{email}{user_id}{datetime.utcnow().timestamp()}{SECRET_KEY}"
    return hashlib.sha256(key_material.encode()).hexdigest()

def generate_purchase_key(user_id, order_id):
    key_material = f"{uuid.uuid4()}{user_id}{order_id}{datetime.utcnow().timestamp()}{SECRET_KEY}"
    return hashlib.sha256(key_material.encode()).hexdigest()

def combine_keys(user_key, purchase_key):
    combined = f"{user_key}|{purchase_key}|{uuid.uuid4()}|{SECRET_KEY}"
    return hashlib.sha256(combined.encode()).hexdigest()

```

### 4.2.2 QR Codes sécurisés

Les QR codes générés incluent :

- L'identifiant du billet
- La clé combinée (clé billet)
- Un horodatage de génération

- Un identifiant unique de QR code

python

```
def generate_qrcode_data(ticket):  
    data = {  
        'qr_id': str(uuid.uuid4()),  
        'ticket_id': str(ticket.id),  
        'key': ticket.cle_billet,  
        'offer_id': ticket.offer_id,  
        'user_id': ticket.user_id,  
        'generated_at': datetime.utcnow().isoformat()  
    }  
  
    return json.dumps(data)
```

#### 4.2.3 Validation des billets

La validation des billets comprend plusieurs vérifications :

- Authenticité via la clé billet
- Validité du billet (non expiré, non utilisé)
- Marquage immédiat comme utilisé après validation

```
python
```

```
def verify_qrcode(qr_data):
    try:
        # Décoder Les données JSON
        data = json.loads(qr_data)

        # Extraire Les informations du billet
        ticket_id = data.get('ticket_id')
        key = data.get('key')

        if not ticket_id or not key:
            return False, "QR code invalide"

        # Récupérer Le billet
        ticket = Ticket.query.get(int(ticket_id))

        if not ticket:
            return False, "Billet introuvable"

        # Vérifier que le billet est valide
        if not ticket.est_valide:
            return False, "Billet déjà utilisé ou annulé"

        # Vérifier que la clé correspond
        if ticket.cle_billet != key:
            return False, "Clé de billet invalide"

        return True, ticket

    except json.JSONDecodeError:
        return False, "Format de QR code invalide"
    except Exception as e:
        return False, f"Erreur lors de la vérification: {str(e)}"
```

## 4.3 Protection des données

### 4.3.1 RGPD

L'application est conforme au RGPD avec les mesures suivantes :

- **Minimisation des données** : Collecte uniquement des données nécessaires
- **Droit à l'effacement** : Possibilité de supprimer les comptes et d'anonymiser les données
- **Consentement explicite** : Recueil du consentement lors de l'inscription
- **Sécurité des données** : Chiffrement et protection contre les accès non autorisés

### 4.3.2 Protection contre les attaques web

- **Protection CSRF** : Tokens CSRF dans tous les formulaires via Flask-WTF
- **Protection XSS** : Échappement automatique dans les templates Jinja2
- **Protection contre l'injection SQL** : Utilisation de l'ORM SQLAlchemy avec requêtes paramétrées
- **En-têtes de sécurité HTTP** : CSP, X-Frame-Options, etc.

python

```
@app.after_request
def add_security_headers(response):
    ... response.headers['Content-Security-Policy'] = "default-src 'self'"
    ... response.headers['X-Content-Type-Options'] = 'nosniff'
    ... response.headers['X-Frame-Options'] = 'SAMEORIGIN'
    ... response.headers['X-XSS-Protection'] = '1; mode=block'
    ...
    return response
```

### 4.4 Audits et journalisation

- **Journalisation des événements de sécurité** : Connexions, modifications sensibles, validations de billets
- **Détection d'anomalies** : Surveillance des comportements suspects (tentatives multiples, accès inhabituels)
- **Audit administrateur** : Enregistrement de toutes les actions administrateur

## 5. Processus de développement

### 5.1 Environnements

- **Développement** : Environnement local avec SQLite
- **Test** : Environnement isolé avec base de données de test
- **Préproduction** : Miroir de la production pour tests finaux
- **Production** : Environnement sécurisé et optimisé pour les performances

### 5.2 Gestion du code source

- **Versionnement** : Git avec GitLab
- **Branches** :
  - `main` : Version stable (production)
  - `develop` : Branche d'intégration
  - `feature/*` : Fonctionnalités en développement
  - `bugfix/*` : Corrections de bugs

- `[release/*]` : Préparation des releases

## 5.3 Tests

- **Tests unitaires** : Pytest pour les fonctions individuelles
- **Tests d'intégration** : Tests de bout en bout
- **Tests de sécurité** : Tests de pénétration et analyse statique de code

`python`

```
# Exemple de test unitaire
def test_generate_user_key():
    key1 = generate_user_key(1, "user1@example.com")
    key2 = generate_user_key(2, "user2@example.com")

    # Vérifier que les clés sont différentes
    assert key1 != key2

    # Vérifier que la même entrée génère la même clé
    assert key1 == generate_user_key(1, "user1@example.com")

    # Vérifier la longueur de la clé (SHA-256)
    assert len(key1) == 64
```

## 5.4 Intégration et déploiement continus

- **CI/CD** : Pipeline GitLab CI
- **Étapes** :
  - Build
  - Tests unitaires
  - Tests d'intégration
  - Analyse de code
  - Déploiement en préproduction
  - Tests de validation
  - Déploiement en production (manuel)

# 6. Infrastructure et déploiement

## 6.1 Architecture de déploiement

- **Containerisation** : Docker avec docker-compose
- **Serveur web** : Nginx comme reverse proxy
- **Serveur d'application** : Gunicorn (WSGI)

- **Base de données** : PostgreSQL

Afficher l'image

## 6.2 Configuration Docker

dockerfile

```
# Dockerfile
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY ..

CMD ["gunicorn", "run:app", "--bind", "0.0.0.0:5000", "--workers", "4"]
```

yaml

```
# docker-compose.yml
version: '3'

services:
  web:
    build: .
    restart: always
    environment:
      - FLASK_APP=run.py
      - FLASK_ENV=production
      - DATABASE_URL=postgresql://user:password@db:5432/jo_etickets
      - SECRET_KEY=${SECRET_KEY}
    depends_on:
      - db
    networks:
      - app-network

  nginx:
    image: nginx:alpine
    restart: always
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/conf.d:/etc/nginx/conf.d
      - ./nginx/ssl:/etc/nginx/ssl
      - ./app/static:/app/static
    depends_on:
      - web
    networks:
      - app-network

  db:
    image: postgres:13
    restart: always
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=jo_etickets
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - app-network

networks:
  app-network:
```

**volumes:**

... postgres\_data:

## 6.3 Scaling et haute disponibilité

- **Load balancing** : Nginx pour répartir la charge
- **RéPLICATION de la base de données** : PostgreSQL avec réPLICATION
- **Redondance** : Déploiement sur plusieurs nœuds
- **Sauvegarde** : Backup journalier de la base de données

## 6.4 Monitoring

- **Logs** : Centralisation avec ELK Stack (Elasticsearch, Logstash, Kibana)
- **Métriques** : Prometheus pour la collecte, Grafana pour la visualisation
- **Alerting** : Notifications en cas de problème détecté

# 7. API et interfaces

## 7.1 API internes

Le système expose plusieurs endpoints API JSON pour les fonctionnalités internes :

### 7.1.1 API Offres

`GET /offers/api/offers`

Retourne la liste des offres disponibles. Paramètres de filtrage disponibles :

- `[type]` : Type d'offre (solo, duo, familiale)

### 7.1.2 API Panier

`GET /cart/api/count`

Retourne le nombre d'articles dans le panier de l'utilisateur connecté.

### 7.1.3 API Administration

`GET /admin/api/stats/offers`

Retourne les statistiques de vente par type d'offre.

GET /admin/api/stats/sales

Retourne l'évolution des ventes par jour.

## 7.2 Interface employé pour la validation des billets

L'interface employé permet de scanner et valider les billets le jour de l'événement :

- **Scan** : Utilisation de la caméra du smartphone/tablette pour scanner le QR code
- **Vérification** : Contrôle d'authenticité et de validité du billet
- **Validation** : Marquage du billet comme utilisé

Cette interface est accessible uniquement aux utilisateurs ayant le rôle 'employé' ou 'administrateur'.

## 8. Évolutions futures

### 8.1 Améliorations techniques

#### 8.1.1 Architecture microservices

Une évolution majeure envisagée est la migration vers une architecture microservices pour améliorer la scalabilité et la modularité :

- **Service d'authentification** : Gestion des utilisateurs et des sessions
- **Service de catalogue** : Gestion des offres
- **Service de commande** : Gestion des paniers et commandes
- **Service de billetterie** : Génération et validation des billets
- **Service d'administration** : Statistiques et gestion

Cette architecture permettrait un scaling indépendant de chaque service selon les besoins.

#### 8.1.2 API publique

Développement d'une API REST complète pour permettre l'intégration avec d'autres services :

- Authentification OAuth2
- Documentation OpenAPI (Swagger)
- Versionnement de l'API
- Rate limiting et quotas

#### 8.1.3 Amélioration des performances

- **Mise en cache** : Implémentation de Redis pour le caching
- **Optimisation base de données** : Indexation avancée, réplicas en lecture

- **CDN** : Utilisation d'un CDN pour les ressources statiques

## 8.2 Nouvelles fonctionnalités

### 8.2.1 Billetterie mobile

- Application mobile native (iOS/Android) pour l'achat et la gestion des billets
- Wallet intégration (Apple Wallet, Google Pay)
- Notifications push pour les événements

### 8.2.2 Personnalisation avancée

- Offres personnalisées basées sur les préférences des utilisateurs
- Système de recommandation basé sur les achats précédents
- Possibilité de créer des packs d'événements personnalisés

### 8.2.3 Intégration sociale

- Partage de billets sur les réseaux sociaux
- Fonctionnalités de groupe pour coordonner les achats entre amis
- Intégration avec les calendriers pour rappels d'événements

## 8.3 Évolution de la sécurité

### 8.3.1 Blockchain pour les billets

Utilisation de la blockchain pour renforcer la sécurité et l'authenticité des billets :

- Enregistrement immuable des transactions
- Prévention de la fraude et de la duplication
- Traçabilité complète du cycle de vie des billets

### 8.3.2 Biométrie

- Authentification biométrique (empreinte digitale, reconnaissance faciale)
- Association facultative du billet avec données biométriques pour vérification à l'entrée
- Respect strict des normes de confidentialité et consentement explicite

### 8.3.3 Intelligence artificielle pour la détection de fraude

- Analyse comportementale pour détecter les activités suspectes
- Détection des anomalies dans les achats et les utilisations de billets
- Apprentissage continu pour améliorer la détection

## 9. Annexes

### 9.1 Dépendances

Liste complète des dépendances Python avec leurs versions :

```
Flask==2.2.3
Flask-Bcrypt==1.0.1
Flask-Login==0.6.2
Flask-Mail==0.9.1
Flask-Migrate==4.0.0
Flask-SQLAlchemy==3.0.3
Flask-WTF==1.1.1
Jinja2==3.1.2
Pillow==9.4.0
PyJWT==2.6.0
PyQRCode==1.2.1
SQLAlchemy==2.0.4
Werkzeug==2.2.3
WTForms==3.0.1
gunicorn==20.1.0
psycopg2-binary==2.9.5
pyotp==2.8.0
python-dotenv==1.0.0
qrcode==7.4.2
reportlab==3.6.12
```

### 9.2 Schéma complet de la base de données



Afficher l'image

### 9.3 Guide de déploiement

Guide détaillé pour déployer l'application en production :

1. Configuration des variables d'environnement
2. Préparation de la base de données
3. Construction des conteneurs Docker
4. Configuration de Nginx
5. Mise en place des certificats SSL
6. Démarrage des services

## 7. Vérification du déploiement

### 9.4 Documentation API

Documentation complète de l'API disponible au format OpenAPI (Swagger).

---

**Note :** Cette documentation est un document vivant qui sera mis à jour à mesure que le projet évolue. Veuillez vous référer à la dernière version pour les informations les plus récentes.

**Version :** 1.0.0

**Dernière mise à jour :** 19 avril 2025

**Auteurs :** Équipe InfoEvent