

Choix techniques - Projet JO E-Tickets

Table des matières

1. [Architecture globale](#)
2. [Langage et framework](#)
3. [Base de données](#)
4. [Authentification et sécurité](#)
5. [Génération et vérification des billets](#)
6. [Interface utilisateur](#)
7. [Gestion des paiements](#)
8. [Déploiement et hébergement](#)
9. [Tests et qualité](#)
10. [Conclusion](#)

1. Architecture globale

Choix technique

Architecture MVC (Modèle-Vue-Contrôleur) modulaire basée sur des blueprints.

Justification

- **Besoin client:** Le client a besoin d'un système évolutif capable de gérer plusieurs types d'offres (solo, duo, familiale) avec possibilité d'en ajouter d'autres à l'avenir.
- **Modularité:** L'architecture en blueprints permet de séparer clairement les fonctionnalités (authentification, offres, billets, paiement) facilitant la maintenance et l'évolution du système.
- **Séparation des responsabilités:** Le pattern MVC offre une distinction claire entre les données (modèles), la présentation (vues) et la logique métier (contrôleurs), ce qui facilite le développement en équipe.
- **Évolutivité:** Cette architecture permet d'ajouter facilement de nouvelles fonctionnalités sans toucher au code existant, répondant à la possibilité que les Jeux Olympiques souhaitent publier d'autres offres à l'avenir.

Implémentation

```
app/
├── __init__.py ..... # Initialisation de l'application
├── config.py ..... # Configuration centralisée
├── models/..... # Modèles de données
├── routes/..... # Contrôleurs par fonctionnalité
│   ├── auth.py ..... # Authentification
│   ├── offers.py ..... # Gestion des offres
│   ├── cart.py ..... # Panier
│   ├── orders.py ..... # Commandes
│   ├── tickets.py ..... # Billets
│   └── admin.py ..... # Administration
        └── main.py ..... # Pages principales
└── templates/..... # Vues (interface utilisateur)
└── static/..... # Ressources statiques
└── forms/..... # Formulaires
└── services/..... # Services métier réutilisables
    ├── auth_service.py ..... # Services d'authentification
    ├── payment_service.py ..... # Services de paiement
    └── qrcode_service.py ..... # Services de génération de QR codes
```

2. Langage et framework

Choix technique

- **Langage:** Python 3.9+
- **Framework principal:** Flask
- **Extensions:**
 - Flask-SQLAlchemy (ORM)
 - Flask-Login (Gestion de sessions)
 - Flask-WTF (Formulaires sécurisés)
 - Flask-Bcrypt (Hachage de mots de passe)
 - Flask-Mail (Envoi d'emails)

Justification

- **Besoin client:** Le client a besoin d'une application web sécurisée, capable de gérer l'authentification, la génération de billets électroniques et de QR codes, ainsi que l'administration.
- **Python:** Langage lisible, maintenable et avec un excellent écosystème de bibliothèques.
 - Grande communauté et documentation abondante
 - Facilité de développement et de maintenance
 - Excellentes bibliothèques pour la génération de PDF et de QR codes (reportlab, qrcode)

- **Flask:** Framework léger mais puissant, idéal pour cette application.
 - Approche minimaliste permettant de n'inclure que ce qui est nécessaire
 - Architecture modulaire via les blueprints
 - Grande flexibilité pour répondre aux besoins spécifiques
 - Parfaitement adapté pour construire des applications web sécurisées
 - Communauté active et mature

Implémentation

```
python
```

```
# app/__init__.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager
from flask_bcrypt import Bcrypt
from flask_mail import Mail

# Initialisation des extensions
db = SQLAlchemy()
login_manager = LoginManager()
bcrypt = Bcrypt()
mail = Mail()

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # Initialisation des extensions avec l'application
    db.init_app(app)
    login_manager.init_app(app)
    bcrypt.init_app(app)
    mail.init_app(app)

    # Enregistrement des blueprints
    from app.routes.auth import auth_bp
    from app.routes.offers import offers_bp
    from app.routes.cart import cart_bp
    from app.routes.orders import orders_bp
    from app.routes.tickets import tickets_bp
    from app.routes.admin import admin_bp
    from app.routes.main import main_bp

    app.register_blueprint(auth_bp)
    app.register_blueprint(offers_bp)
    app.register_blueprint(cart_bp)
    app.register_blueprint(orders_bp)
    app.register_blueprint(tickets_bp)
    app.register_blueprint(admin_bp)
    app.register_blueprint(main_bp)

    return app
```

3. Base de données

Choix technique

- **SGBD:** SQLite en développement, SQLAlchemy comme ORM pour permettre une migration facile vers PostgreSQL/MySQL en production
- **Modèles:** Relationnels avec clés étrangères et contraintes d'intégrité

Justification

- **Besoin client:** Le système doit gérer des utilisateurs, des offres, des commandes et des billets avec des relations complexes entre ces entités.
- **SQLAlchemy:**
 - Abstraction de la base de données permettant de changer facilement de SGBD sans modifier le code
 - Sécurité améliorée contre les injections SQL
 - Gestion efficace des relations complexes entre entités
 - Facilité de requêtage et de migration
- **Modèle relationnel:**
 - Parfaitement adapté pour représenter les relations entre utilisateurs, offres, commandes et billets
 - Garantie de l'intégrité des données (contraintes d'intégrité référentielle)
 - Permet des requêtes analytiques complexes pour les statistiques administrateur

Implémentation - Exemple de modèle

```
python
```

```
# app/models/user.py
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)
    role = db.Column(db.String(20), default='utilisateur')
    est_verifie = db.Column(db.Boolean, default=False)
    code_2fa_secret = db.Column(db.String(32), nullable=True)
    est_2fa_active = db.Column(db.Boolean, default=False)

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

4. Authentification et sécurité

Choix technique

- **Système d'authentification:** Flask-Login avec sessions sécurisées
- **Hachage des mots de passe:** Bcrypt (algorithme moderne avec salage automatique)
- **Authentification à deux facteurs (2FA):** TOTP via pyotp (compatible Google Authenticator)
- **Protection CSRF:** Via Flask-WTF
- **Génération de clés de sécurité:** Algorithmes cryptographiques (UUID4 + SHA-256)

Justification

- **Besoin client:** "Le client insiste énormément sur la sécurité du compte de l'utilisateur" et a besoin d'un système de clés invisibles pour l'utilisateur.
- **Flask-Login:** Gestion robuste des sessions utilisateur et des permissions
- **Bcrypt:**
 - Résistant aux attaques par force brute (facteur de travail ajustable)
 - Salage automatique pour éviter les attaques par table rainbow
 - Standard de l'industrie pour le hachage sécurisé
- **2FA:** Couche supplémentaire de sécurité, particulièrement importante pour un événement de cette envergure

- **Protection CSRF:** Empêche les attaques par falsification de requêtes
- **Génération de clés:**
 - Répond directement à l'exigence: "Au moment de la création de son compte, une clef est générée. Cette clef n'est pas visible pour l'utilisateur"
 - Utilisation de techniques cryptographiques modernes pour garantir l'unicité et la sécurité

Implémentation - Service d'authentification

python

```
# app/services/auth_service.py
def register_user(username, email, password, nom, prenom):
    """
    Enregistre un nouvel utilisateur.
    """

    # Vérifier si L'email existe déjà
    if User.query.filter_by(email=email).first():
        return False, "Cette adresse email est déjà utilisée."

    # Vérifier si Le nom d'utilisateur existe déjà
    if User.query.filter_by(username=username).first():
        return False, "Ce nom d'utilisateur est déjà utilisé."

    # Valider Le mot de passe
    is_valid, message = validate_password(password)
    if not is_valid:
        return False, message

    # Hacher Le mot de passe
    hashed_password = bcrypt.generate_password_hash(password).decode('utf-8')

    # Créer L'utilisateur
    user = User(
        username=username,
        email=email,
        password=hashed_password,
        nom=nom,
        prenom=prenom,
        est_verifie=True # Dans une version finale, il faudrait une vérification par email
    )

    # Générer une clé de sécurité invisible pour L'utilisateur
    user.cle_securite = generate_user_key(username, email)

    # Enregistrer L'utilisateur
    db.session.add(user)
    db.session.commit()

    return True, user

def generate_user_key(username, email):
    """
    Génère une clé de sécurité unique pour l'utilisateur.
    """
```

```
... key_material = f"{{uuid.uuid4()}}{{email}}{{username}}{{datetime.utcnow().timestamp()}}{{current_app.config['SECRET_KEY']}"
... return hashlib.sha256(key_material.encode()).hexdigest()
```

5. Génération et vérification des billets

Choix technique

- **Système de double clé:** Clé utilisateur + Clé d'achat combinées pour les billets
- **Format des billets:** PDF généré dynamiquement via ReportLab
- **QR codes:** Bibliothèque qrcode avec haute correction d'erreur
- **Stockage du QR code:** Base64 dans la base de données

Justification

- **Besoin client:** Système de e-tickets avec QR codes sécurisés basé sur une double clé, "Lors de l'achat, une autre clef est générée. Elle sera utilisée avec la clef précédente pour sécuriser le billet acheté (Concaténation des deux clefs)."
- **Double clé:**
 - Implémentation directe de l'exigence du client
 - Sécurité renforcée: même si une clé est compromise, l'autre reste nécessaire
 - La clé utilisateur garantit que le billet appartient à l'utilisateur authentifié
 - La clé d'achat valide la transaction spécifique
- **PDF via ReportLab:**
 - Création de billets visuellement attractifs et professionnels
 - Intégration facile des QR codes et des informations
 - Format universellement reconnu, facilement imprimable
- **QR codes avec haute correction d'erreur:**
 - Lecture fiable même si le code est partiellement endommagé
 - Compatibilité avec tous les smartphones modernes
 - Idéal pour la vérification rapide lors de l'événement

Implémentation

python

```
# app/services/qrcode_service.py
def generate_ticket_qrcode(ticket_id):
    """
    Génère un QR code pour un billet spécifique.
    """
    ticket = Ticket.query.get(ticket_id)

    if not ticket:
        return None

    # Générer les données pour le QR code
    qr_data = generate_qrcode_data(ticket)

    # Créer l'image QR code
    qr_image = create_qrcode_image(qr_data)

    # Mettre à jour le billet avec le QR code
    ticket.qr_code = qr_image
    db.session.commit()

    return qr_image

def generate_qrcode_data(ticket):
    """
    Génère les données à encoder dans le QR code.
    """

    data = {
        'qr_id': str(uuid.uuid4()), # Identifiant unique pour ce QR code
        '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)

def create_qrcode_image(data):
    """
    Crée une image QR code à partir des données fournies.
    """

    qr = qrcode.QRCode(
        version=1,
        error_correction=qrcode.constants.ERROR_CORRECT_H, # Haute correction d'erreur
        box_size=10,
        border=4,
```

```

    ... )
    qr.add_data(data)
    qr.make(fit=True)

    ...

    img = qr.make_image(fill_color="black", back_color="white")

    ...

    # Conversion en base64
    buffered = BytesIO()
    img.save(buffered, format="PNG")
    img_str = base64.b64encode(buffered.getvalue()).decode('utf-8')

    ...

    return f"data:image/png;base64,{img_str}"

```

6. Interface utilisateur

Choix technique

- **Framework CSS:** Bootstrap 5
- **Moteur de template:** Jinja2 (intégré à Flask)
- **JavaScript:** Vanilla JS avec quelques bibliothèques légères (Chart.js pour les statistiques admin)
- **Responsive design:** Adaptation à tous les appareils (desktop, tablette, mobile)

Justification

- **Besoin client:** Interface intuitive pour "la réservation de ticket" accessible depuis le site des Jeux Olympiques.
- **Bootstrap 5:**
 - Framework CSS mature et bien documenté
 - Design responsive natif pour tous les appareils
 - Composants prêts à l'emploi (cartes, formulaires, navigation)
 - Thème personnalisable pour s'adapter à l'identité visuelle des JO
- **Jinja2:**
 - Intégration native avec Flask
 - Héritage de templates pour une structure cohérente
 - Échappement automatique contre les attaques XSS
- **Vanilla JS:**
 - Performance optimale sans surcharge de frameworks
 - Suffisant pour les besoins d'interactivité de l'application
- **Chart.js pour les statistiques admin:**
 - Visualisations attractives et interactives

- Léger et facile à intégrer
- Répond au besoin: "un administrateur peut depuis son espace visualiser le nombre de ventes par offre"

Implémentation

html

```
<!-- app/templates/base.html -->
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ block title }} JO E-Tickets{{ endblock }}</title>

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css"

    <!-- Font Awesome -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/

    <!-- Custom CSS -->
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}>

    {{ block styles }}{{ endblock }}
</head>
<body>
    <!-- Navigation -->
    <nav class="navbar navbar-expand-lg navbar-dark bg-primary">
        <!-- ... contenu de la navigation ... -->
    </nav>

    <!-- Messages flash -->
    <div class="container mt-3">
        {{ with messages = get_flashed_messages(with_categories=true) }}
            {% if messages %}
                {% for category, message in messages %}
                    <div class="alert alert-{{ category }} alert-dismissible fade show" role="alert">
                        {{ message }}
                        <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close">
                    </div>
                {% endfor %}
                {% endif %}
                {% endwith %}
        </div>

        <!-- Contenu principal -->
        <main class="container py-4">
            {{ block content }}{{ endblock }}
        </main>

        <!-- Pied de page -->
        <footer class="bg-dark text-white py-4 mt-5">
```

```

.... <!-- ... contenu du footer ... -->
.... </footer>

.... <!-- Bootstrap JS Bundle with Popper -->
.... <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.bundle.m

.... <!-- Custom JS -->
.... <script src="{{ url_for('static', filename='js/main.js') }}></script>

.... {% block scripts %}{% endblock %}
</body>
</html>

```

7. Gestion des paiements

Choix technique

- **Simulation de paiement:** Système interne sans intégration réelle de passerelle de paiement
- **Préparation pour intégration réelle:** Architecture conçue pour faciliter l'intégration future d'une passerelle de paiement

Justification

- **Besoin client:** "Le paiement n'est pas à effectuer dans l'application, vous pouvez effectuer un mock pour simuler un paiement."
- **Simulation:**
 - Respect direct de l'exigence du client
 - Permet de tester le flux complet sans infrastructure de paiement réelle
 - Inclut les validations de base des informations de carte
- **Architecture préparée:**
 - Service de paiement isolé facilitant le remplacement par une vraie intégration
 - Stockage sécurisé des résultats de transaction
 - Workflow complet de commande-paiement-génération de billets

Implémentation

python

```
# app/services/payment_service.py
class PaymentResult:
    """Classe pour représenter le résultat d'un paiement."""

    def __init__(self, success, transaction_id=None, message=None, error_code=None):
        self.success = success
        self.transaction_id = transaction_id
        self.message = message
        self.error_code = error_code
        self.timestamp = datetime.utcnow()

def process_payment(order_id, card_number, expiry_month, expiry_year, cvv, amount):
    """
    Traite un paiement (simulation).

    Cette fonction simule le traitement d'un paiement par carte de crédit.
    Dans un environnement de production, cette fonction ferait appel à un service de paiement réel.
    """

    # Récupérer La commande
    order = Order.query.get(order_id)

    if not order:
        return PaymentResult(False, message="Commande introuvable", error_code="ORDER_NOT_FOUND")

    # Vérifier Le montant
    if order.total != amount:
        return PaymentResult(False, message="Montant incorrect", error_code="INVALID_AMOUNT")

    # Vérifications basiques des informations de carte
    if not _validate_card(card_number, expiry_month, expiry_year, cvv):
        return PaymentResult(False, message="Informations de carte invalides", error_code="INVALID_CARD")

    # Simuler un taux de réussite de 90%
    if random.random() < 0.9:
        # Paiement réussi
        transaction_id = str(uuid.uuid4())

        # Mettre à jour Le statut de La commande
        order.set_paid()

    return PaymentResult(
        success=True,
        transaction_id=transaction_id,
        message="Paiement traité avec succès"
    )
else:
```

```

    .... # Paiement échoué
    .... error_codes = ["INSUFFICIENT_FUNDS", "CARD_DECLINED", "NETWORK_ERROR"]
    .... error_code = random.choice(error_codes)

    .... error_messages = {
        "INSUFFICIENT_FUNDS": "Fonds insuffisants",
        "CARD_DECLINED": "Carte refusée par l'émetteur",
        "NETWORK_ERROR": "Erreur réseau lors du traitement du paiement"
    }

    .... return PaymentResult(
        .... success=False,
        .... message=error_messages[error_code],
        .... error_code=error_code
    )

```

8. Déploiement et hébergement

Choix technique

- **Serveur d'application:** Gunicorn (WSGI)
- **Serveur web:** Nginx (reverse proxy)
- **Containerisation:** Docker avec docker-compose
- **CI/CD:** GitLab CI pour l'intégration et le déploiement continus

Justification

- **Besoin client:** L'événement est très important et la France ne peut pas se permettre une mauvaise image. Cela implique une solution robuste et fiable.
- **Gunicorn + Nginx:**
 - Configuration éprouvée et robuste pour les applications Flask
 - Nginx gère efficacement les connexions et les fichiers statiques
 - Gunicorn optimise l'exécution de l'application Python
- **Docker:**
 - Environnement cohérent entre développement et production
 - Facile à déployer et à mettre à l'échelle
 - Isolation pour améliorer la sécurité
- **CI/CD:**
 - Déploiement automatisé et fiable
 - Tests automatiques avant déploiement

- Réduction des risques d'erreurs humaines

Implémentation

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=${SECRET_KEY}
      - MAIL_SERVER=${MAIL_SERVER}
      - MAIL_PORT=${MAIL_PORT}
      - MAIL_USERNAME=${MAIL_USERNAME}
      - MAIL_PASSWORD=${MAIL_PASSWORD}
    volumes:
      - ./app:/app
    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:
```

9. Tests et qualité

Choix technique

- **Tests unitaires:** pytest
- **Tests d'intégration:** pytest-flask
- **Couverture de code:** pytest-cov
- **Analyse statique:** flake8, mypy
- **Continuous Integration:** GitLab CI

Justification

- **Besoin client:** "La France ne peut pas se permettre de montrer une mauvaise image ce jour-là" - nécessite une qualité irréprochable.
- **Tests complets:**
 - Garantie de la fiabilité des fonctionnalités critiques
 - Détection précoce des régressions
 - Focus particulier sur les fonctionnalités de sécurité
- **Couverture de code:**
 - S'assurer que tous les chemins de code sont testés
 - Identifier les parties non testées pour amélioration
- **Analyse statique:**
 - Maintien d'un code propre et conforme aux standards
 - Détection précoce des bugs potentiels
 - Vérification des types pour prévenir les erreurs courantes
- **CI:**
 - Exécution automatique des tests à chaque changement
 - Garantie que chaque version déployée passe tous les tests
 - Rapport automatique des problèmes détectés

Implémentation

python

```
# tests/test_auth.py
import pytest
from app import create_app, db
from app.models.user import User
from app.config import TestingConfig

@pytest.fixture
def client():
    app = create_app(TestingConfig)

    with app.test_client() as client:
        with app.app_context():
            db.create_all()
            yield client
            db.session.remove()
            db.drop_all()

def test_register(client):
    """Test d'inscription utilisateur."""
    response = client.post('/auth/register', data={
        'username': 'testuser',
        'email': 'test@example.com',
        'password': 'Test@1234',
        'confirm_password': 'Test@1234',
        'nom': 'Nom',
        'prenom': 'Prenom'
    }, follow_redirects=True)

    assert response.status_code == 200
    assert b'Votre compte a' in response.data

    # Vérifier que l'utilisateur a été créé
    user = User.query.filter_by(email='test@example.com').first()
    assert user is not None
    assert user.username == 'testuser'
    assert user.cle_securite is not None # Vérifier que la clé de sécurité a été générée
```

10. Conclusion

Les choix techniques pour le projet JO E-Tickets ont été soigneusement sélectionnés pour répondre aux exigences spécifiques du client tout en assurant une solution robuste, sécurisée et évolutive. L'utilisation de Python avec Flask offre la flexibilité nécessaire, tandis que l'architecture MVC modulaire facilite le

développement et la maintenance. Le système de double clé pour la sécurisation des billets répond directement à la demande du client, et l'interface utilisateur responsive assure une expérience cohérente sur tous les appareils.

La sécurité a été une préoccupation majeure, avec l'implémentation de hachage de mots de passe, d'authentification à deux facteurs et de protection contre les attaques courantes. Le système de simulation de paiement permet de tester l'application tout en préparant l'intégration future d'une vraie passerelle.

L'infrastructure de déploiement proposée, basée sur Docker, Nginx et Gunicorn, assure une solution robuste et évolutive, capable de gérer la charge attendue pour un événement de l'envergure des Jeux Olympiques. Les tests automatisés et l'intégration continue garantissent la qualité et la fiabilité du système.

Ces choix techniques forment une base solide pour répondre aux besoins du client et assurer le succès du système de billetterie électronique pour les Jeux Olympiques 2024.