
Cahier des Charges – QRAPI

1. Contexte et Objectifs

1.1 Contexte

La **QRAPI** est une API REST développée en Spring Boot destinée à la génération et au scan de QR codes dans le cadre d'un système d'authentification par handshake et de gestion de courses. L'API s'inscrit dans un écosystème plus large dans lequel un endpoint d'authentification (ex. /api/reserve) permet d'obtenir un token JWT pour sécuriser les échanges avec les autres endpoints.

1.2 Objectifs

- **Génération de QR codes** : Créer des QR codes intégrant un token JWT signé (contenant des informations sur une course) pour garantir leur intégrité et leur validité dans le temps.
- **Scan et validation de QR codes** : Permettre le scan d'un QR code pour valider une course ou enregistrer une action, avec vérification de la signature.
- **Sécurisation** : Protéger l'accès aux endpoints via un mécanisme d'authentification par token JWT.
- **Traçabilité** : Enregistrer les historiques de scan pour le suivi des actions et la gestion de la course.

---

2. Périmètre Fonctionnel

2.1 Endpoints

a) Authentification – Réservation du Token

- **URL** : /api/reserve
- **Méthode** : POST
- **Paramètres** :
    - fournisseur (corps de la requête ou query string)
- **But** : Obtenir un token JWT nécessaire pour accéder aux autres endpoints.
- **Réponse attendue** :
- {
- "token": "TOKEN_JWT"
- }
- **Remarque** : Cet endpoint est supposé exister dans l'architecture globale et n'est pas détaillé dans le contrôleur actuel.

b) Génération du QR Code

- **URL** : /api/qr/generate

- **Méthode** : POST
- **Paramètres** :
    - **Query Parameters** :
        - secret : Clé utilisée pour signer le QR code (min. 32 caractères pour HS256).
        - expirationMillis : Durée de validité en millisecondes du token.
    - **Corps (JSON)** : Objet QRData contenant les informations de course, par exemple :
    - {
    - "clientId": 123456,
    - "chauffeurId": 654321,
    - "courseId": 987654,
    - "lieu": "Gare Centrale",
    - "heure": "14:30",
    - "date": "2025-02-07",
    - "ville": "Paris",
    - "pays": "France",
    - "fournisseur": "NomDuFournisseur"
    - }
- **Authentification** : Le header HTTP doit inclure
- Authorization: Bearer <TOKEN>
- **Processus** :

1. Attribution d'un UUID à l'objet QRData et récupération du fournisseur via le Principal.
2. Transformation de QRData en chaîne et génération d'un hash (SHA-256).
3. Signature du hash via un token JWT avec la clé secret et la durée définie.
4. Conversion du token signé en image QR (format PNG) grâce à ZXing.

- **Réponse** : Image PNG contenant le QR code.

c) Scan du QR Code

- **URL** : /api/qr/scan
- **Méthode** : POST
- **Paramètres** :
    - **Query Parameters** :
        - qrCodeData : Token JWT issu du QR code (contenant le hash signé).

        - secret : Clé utilisée pour vérifier la signature.
    - **Corps (JSON)** : Objet History avec des informations complémentaires sur l'opération de scan.
- **Authentification** :
- Authorization: Bearer <TOKEN>
- **Processus** :

1. Décodage et vérification du token contenu dans qrCodeData avec la clé secret.
    - En cas d'invalidité, renvoi d'un HTTP 401 (« QR Code invalide ! »).
2. Si la signature est valide, récupération de l'objet QRHash correspondant.
3. Recherche dans la base des données complètes associées (table QRData).
4. Complétion et enregistrement d'un objet History avec les informations de QRData.

- **Réponses** :
    - **Succès** : Renvoi de l'objet QRData au format JSON, status HTTP 200.
    - **Erreur** : HTTP 401 en cas de signature invalide, ou HTTP 400 avec message en cas d'exception.

---

3. Spécifications Techniques et Conception

3.1 Architecture Globale

L'API repose sur une architecture **REST** basée sur Spring Boot. Les composants clés incluent :

- **Contrôleur** : QRCodeController qui gère les endpoints de génération et de scan.
- **Services** :
    - ScanService pour la logique métier relative à la validation des QR codes.
    - JwtUtil (ou équivalent) pour la création et la vérification des tokens JWT.
- **Repositories** : Accès aux tables Cassandra (QRData, QRHash, History).

3.2 Modèle de Données

La base de données (Cassandra) contient les tables suivantes :

- **Table QRData** : Stocke les informations de course et les données intégrées dans le QR code.
- **Table QRHash** : Enregistre le hash signé et la référence à l'UUID de QRData.
- **Table History** : Historique des scans avec les informations de la course et du scan.

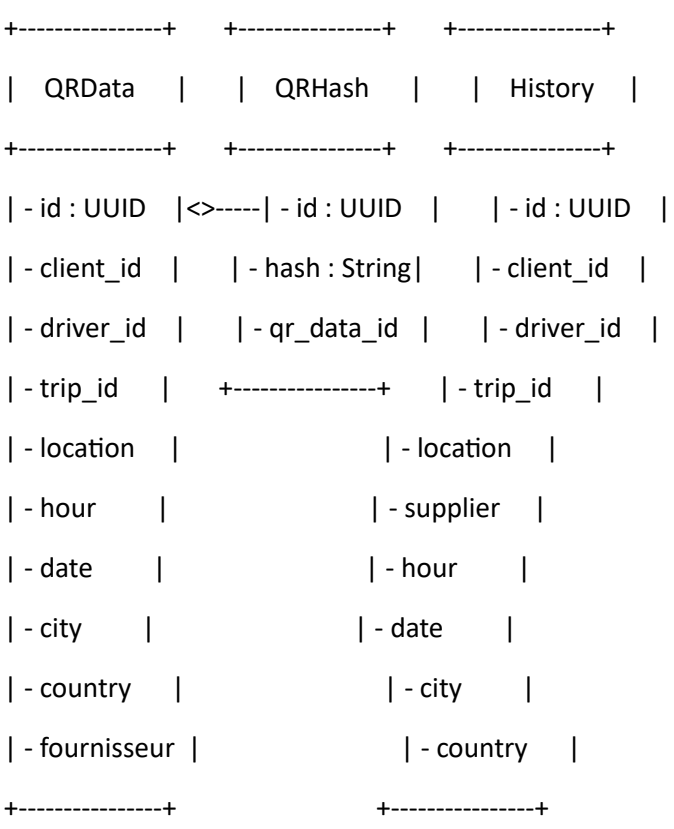
Schéma simplifié en CQL :

```
CREATE KEYSPACE IF NOT EXISTS transportapp
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

```
USE transportapp;

CREATE TABLE IF NOT EXISTS qr_data (
    id uuid PRIMARY KEY,
    client_id bigint,
    driver_id bigint,
    trip_id bigint,
    location text,
    hour text,
    date text,
    city text,
    country text,
    fournisseur text
);

CREATE TABLE IF NOT EXISTS qr_hash (
    id uuid PRIMARY KEY,
    hash text,
    qr_data_id uuid
);

CREATE TABLE IF NOT EXISTS history (
    id uuid PRIMARY KEY,
    client_id bigint,
    driver_id bigint,
    trip_id bigint,
    location text,
    supplier text,
    hour text,
    date text,
```

```
    city text,
    country text
);
```

3.3 Diagramme de Classes (UML Simplifié)



+--------------------+

| (Logique de scan) |

+--------------------+

---

4. Spécifications Non Fonctionnelles

4.1 Sécurité

- **Authentification** : Obligation de transmettre un token JWT pour accéder aux endpoints de génération et de scan.
- **Intégrité des données** : Utilisation de hash SHA-256 et de signature JWT pour éviter toute falsification des informations.
- **Expiration** : Les tokens intègrent une durée de validité configurable (expirationMillis) pour limiter les risques liés aux QR codes expirés.

4.2 Performance et Scalabilité

- **Performance** : Optimisation de la génération et de la vérification des QR codes pour assurer une réponse rapide même en cas de trafic important.
- **Scalabilité** : Utilisation de Cassandra comme base NoSQL pour supporter une montée en charge horizontale et garantir la haute disponibilité.

4.3 Fiabilité et Traçabilité

- **Journalisation** : Enregistrement des actions de scan dans la table History pour un suivi détaillé.
- **Tolérance aux pannes** : Déploiement en cluster (possibilité d'utilisation de Docker et orchestration avec Kubernetes par exemple) pour assurer la continuité de service.

---

5. Plan de Déploiement

5.1 Prérequis Techniques

- **Java** : Version 11 minimum (idéalement Java 17 ou 23)
- **Gestionnaire de Build** : Maven ou Gradle
- **Base de données** : Cassandra (localement, en Docker ou sur un cluster)
- **Conteneurisation (optionnelle)** : Docker

5.2 Configuration de l'Environnement

Cassandra

1. **Installation**
    - Installation locale ou via Docker :
    - docker run --name cassandra -d -p 9042:9042 cassandra:latest

2. **Création du Keyspace et Tables**
    - Se connecter via cqlsh et exécuter le script de création du keyspace et des tables (voir section 3.2).

Application

1. **Fichier de configuration**
    - Modifier le fichier src/main/resources/application.properties ou application.yml :
    - spring.application.name=QRAPI
    - spring.data.cassandra.contact-points=127.0.0.1
    - spring.data.cassandra.port=9042
    - spring.data.cassandra.keyspace-name=transportapp
    - spring.data.cassandra.local-datacenter=datacenter1
    - spring.data.cassandra.schema-action=create-if-not-exists
    - server.port=8080
2. **Dépendances Maven**
    - Vérifier que le pom.xml inclut les dépendances nécessaires (Spring Boot, Spring Data Cassandra, Spring Security, JJWT, ZXing, etc.).

5.3 Construction et Lancement

1. **Compilation et Packaging**
    - Avec Maven :
    - mvn clean install
2. **Démarrage de l'application**
    - Exécuter le jar généré :
    - java -jar target/QRAPI-0.0.1-SNAPSHOT.jar
    - L'application sera accessible sur http://localhost:8080.
3. **Conteneurisation (optionnelle)**
    - Création d'un Dockerfile pour containeriser l'application et éventuellement la base Cassandra.

---

6. Cas d'Utilisation et Scénarios

6.1 Cas d'Utilisation : Génération d'un QR Code

- **Acteur** : Fournisseur (ex. chauffeur)
- **Préconditions** :

- L’utilisateur est authentifié (token JWT obtenu via /api/reserve).
- Les données de la course sont préalablement enregistrées ou validées.

- **Flux Principal** :

1. Envoi d’une requête POST /api/qr/generate avec les paramètres requis (secret, expirationMillis) et un objet QRData.
2. Attribution d’un UUID, création du hash, signature du token JWT, et génération du QR code.
3. Renvoi d’une image PNG contenant le QR code généré.

- **Postconditions** :
  - Enregistrement des informations dans les tables QRData et QRHash.

6.2 Cas d’Utilisation : Scan et Validation d’un QR Code

- **Acteur** : Fournisseur (ex. chauffeur, agent de vérification)
- **Préconditions** :
  - Le QR code a été généré et est en possession de l’utilisateur.
  - L’utilisateur possède un token JWT valide.
- **Flux Principal** :

1. Scan du QR code et envoi d’une requête POST /api/qr/scan avec qrCodeData et le secret.
2. Vérification de la signature du token.
3. Récupération des informations associées à partir de QRHash et QRData.
4. Enregistrement d’un objet History avec les données complémentaires.
5. Renvoi des informations de course (objet QRData) en réponse.

- **Flux Alternatif** :
  - Signature invalide : renvoi d’un HTTP 401 avec message d’erreur.
  - Erreur de traitement : renvoi d’un HTTP 400 avec message d’erreur.
- **Postconditions** :
  - Mise à jour de l’historique dans la table History.

---

7. Suivi, Tests et Maintenance

7.1 Stratégie de Tests

- **Tests Unitaires** : Couvrir la logique de génération de QR codes, la signature et la vérification des tokens.
- **Tests d’Intégration** : Valider l’interaction entre les services (ScanService, contrôleur, et accès à Cassandra).

- **Tests de Sécurité** : Vérifier que les endpoints protègent correctement l'accès via le token JWT et renvoient les statuts HTTP appropriés en cas d'erreur.
- **Tests de Performance** : Assurer la réactivité de l'API sous une charge simulée importante.

7.2 Suivi et Maintenance

- **Journalisation** : Mettre en place une stratégie de log pour suivre les accès et les erreurs.
- **Monitoring** : Utilisation d'outils (ex. Prometheus, Grafana) pour surveiller la santé de l'API et des ressources Cassandra.
- **Mises à jour** : Prévoir un plan de mise à jour régulier pour les dépendances (Spring Boot, sécurité, bibliothèques tierces).

---

8. Livrables et Documentation

- **Code Source** : Organisation du code en packages (contrôleurs, services, modèles, repositories).
- **Documentation Technique** : Documentation des endpoints, des schémas de données et des procédures de déploiement.
- **Rapport de Travail** : Le dossier "RAPPORT" contiendra l'analyse détaillée du développement, les choix techniques et les tests réalisés.
- **Diagrammes UML** : Diagramme de classes et diagrammes de séquence pour illustrer les interactions principales.

---

9. Conclusion

Ce cahier des charges définit l'ensemble des aspects de conception, de déploiement et d'utilisation de la QRAPI. L'API se veut sécurisée, performante et traçable, répondant aux besoins d'un système d'authentification par QR code dans un contexte de gestion de courses. La mise en œuvre s'appuie sur des technologies éprouvées (Spring Boot, Cassandra, JWT, ZXing) et suit des standards de développement et de sécurité adaptés aux environnements de production.
