Rapport de réalisation – Projet Points d'Intérêt Touristiques (PIT)

1. Architecture implémentée

1.1 Vue d'ensemble

Le projet suit une architecture par couches classique Spring Boot. Les entités et énumérations sont regroupées dans `src/main/java/com/pit/domain`, les accès aux données sont délégués aux interfaces Spring Data JPA du répertoire `repository`, la logique métier réside dans `service` / `service/impl`, et l'exposition (REST + vues Thymeleaf) est concentrée dans `web`. Cette séparation, couplée à l'injection de dépendances automatique de Spring, m'a permis d'appliquer le cahier des charges « architecture découplée » sans recourir à du code passe-plat.

1.2 Couche présentation

Pour répondre à la double cible API / interface utilisateur, deux familles de contrôleurs cohabitent :

- `PlaceController`, `RatingController` et `AuthController` exposent une API REST structurée sous `/api/**`. Les DTO (`web/dto`) sont mappés depuis les entités via MapStruct (`web/mapper`) ce qui fiabilise la sérialisation et évite le couplage aux entités JPA.
- `PageController`, `AdminPlaceController` et `AuthPageController` orchestrent les écrans Thymeleaf (`templates/home.html`, `templates/places/detail.html`, `templates/admin/places.html`, etc.). L'interface permet d'inscrire un compte, de soumettre un lieu et de voter sans recourir à l'API brute, conformément à la demande fonctionnelle.

La pagination est gérée partout avec les objets `Pageable` et `PageRequest` (ex. `PageController.home` pour les lieux publics et `AdminPlaceController.dashboard` pour les files d'attente). Les coordonnées GPS sont manipulées dans toute la stack en doubles (6 décimales), validées côté DTO (`CreatePlaceRequest`) et côté formulaires (contraintes Bean Validation).

1.2.1 API REST (modèle, style et usage)

L'API est une API REST JSON exposée sous `/api/**`. Elle suit un style ressource (ex. `/api/places`, `/api/places/{id}`, `/api/places/{id}/ratings`) et retourne des DTO dédiés (`PlaceDto`, `RatingDto`) pour isoler le modèle JPA du modèle d'échange. Les statuts HTTP sont utilisés pour refléter les règles métiers (ex. 401/403 pour accès interdit, 404 pour ressource introuvable, 409 pour conflit sur la notation). L'API est versionnée implicitement via le contexte applicatif (pas de `/v1`), ce qui reste conforme au périmètre du projet. Les vues sont réalisées en Thymeleaf (pas de JSP), ce qui assure une séparation claire entre UI serveur et API REST.

1.3 Sécurité et gestion des utilisateurs

`SecurityConfig` définit deux chaînes de filtres indépendantes : une pour les pages HTML avec session et formulaire de connexion, une pour l'API REST stateless sécurisée par JWT (`JwtAuthFilter`, `JwtService`). Cette configuration permet à un visiteur de créer un compte via `/register`, de s'authentifier via `/login` (formulaire) et de générer des tokens via `/api/auth/**`. Les rôles `USER` et `ADMIN` sont portés par l'entité `User`, alimentés par Flyway (`V2__seed_admin.sql` pour l'admin par défaut).

`AuthService` centralise l'encodage bcrypt des mots de passe, l'authentification et la récupération de l'utilisateur courant. Cela simplifie l'autorisation fine appliquée par `PlaceController` et `RatingController` (visibilité restreinte pour les lieux en attente, notation réservée aux comptes authentifiés).

1.4 Persistance et base de données

L'ensemble s'appuie sur une base relationnelle H2 en mémoire pour le développement et les tests (`application.yml` + `application-dev.yml`). Les migrations Flyway (`db/migration/V1__init_schema.sql` à `V4__place_created_at.sql`) créent les tables `users`, `places`, `ratings`, ajoutent les colonnes de métriques et horodatages. Chaque insertion de note déclenche `RatingServiceImpl.refreshPlaceMetrics` qui met à jour la moyenne et le compteur associés au lieu, garantissant une lecture instantanée côté UI.

1.5 Documentation et observabilité

`OpenApiConfig` publie la documentation Swagger sur `/swagger-ui/index.html`. Swagger UI sert à explorer l'API (liste des endpoints, modèles, paramètres), tester les routes directement depuis le navigateur, et vérifier rapidement les statuts de réponse sans écrire de scripts. C'est utile pour la validation fonctionnelle (démonstrations, recettes) et pour partager une documentation interactive aux évaluateurs.
Les endpoints d'activité (`/health`, `/actuator/health`) sont exposés pour l'intégration continue

et l'observabilité. Le logging SQL est activé dans les profils par défaut pour faciliter le diagnostic en développement.

1.6 Notifications temps réel (WebSocket)

Pour compléter l'expérience utilisateur, des notifications temps réel sont implémentées avec WebSocket + STOMP. Le serveur expose un endpoint `/ws` (SockJS activé) et un broker simple (`/topic`, `/queue`). Deux canaux sont utilisés :

- `/topic/admin/places` : informe les administrateurs lorsqu'un utilisateur propose un nouveau lieu.
- `/user/queue/places` : informe l'auteur lorsqu'un lieu est approuvé ou refusé.

L'implémentation est volontairement légère : `PlaceServiceImpl` envoie les messages au moment de la création (statut `PENDING`) et lors des transitions `APPROVED/REJECTED`. Côté UI, un script STOMP (`ws-notifications.js`) affiche un toast et rafraîchit la page pour refléter immédiatement l'état des listes.

2. Réalisation du cahier des charges

- **Application Spring Web / API REST :** l'API REST couvre l'intégralité du CRUD utile : soumission, récupération détaillée, modération, notation. Les contrôleurs HTML offrent l'expérience utilisateur demandée (création de compte, soumission, vote).
- **Documentation des API :** Springdoc fournit la documentation interactive. Chaque route REST est annotée (`@Tag`) pour apparaître clairement dans Swagger.
- **WebSocket (en complément) :** des notifications temps réel sont présentes pour les admins (nouvelle proposition) et les auteurs (lieu approuvé/refusé).
- **Structure claire et injection de dépendances :** la hiérarchie de packages répond aux responsabilités métier, et l'injection via `@RequiredArgsConstructor` supprime la configuration manuelle.
- **Base de données relationnelle et opérations CRUD :** `PlaceServiceImpl` et `RatingServiceImpl` encapsulent les opérations insert/update/read. Les statuts `PENDING/APPROVED/REJECTED` implémentent le workflow de validation.
- **Suppression administrative des lieux :** un administrateur peut supprimer définitivement un lieu via l'API REST (`DELETE /api/places/{id}`) ou depuis l'interface d'administration.
- **Pagination :** toutes les listes exposées (`/`, `/admin/places`, `/api/places`, `/api/places/{id}/ratings`) sont paginées. Les tests vérifient notamment que la liste publique exclut les lieux en attente.

- **Gestion utilisateurs (public/admin) :** la séparation des rôles est gérée via Spring Security (`@PreAuthorize` , restrictions dans `SecurityConfig`). Les vues affichent dynamiquement les menus en fonction du rôle.
- **Tests unitaires / d'intégration sur le périmètre public :** la couverture des contrôleurs REST et HTML (voir section 5) garantit que les comportements attendus sur les endpoints accessibles au public sont validés.
- **Fonctionnalité métier « points d'intérêt touristiques » :** la fiche lieu (`places/detail.html`) affiche description, coordonnées GPS, notes agrégées, et propose un formulaire de notation une fois le lieu approuvé – conforme au sujet imposé.

3. Problèmes rencontrés, résolutions et choix assumés

1. **Compatibilité Java 21** – Les premières exécutions Maven échouaient sur l'erreur « release version 21 not supported ». Le poste utilisait un JDK plus ancien. J'ai uniformisé l'environnement en pointant `JAVA_HOME` vers le JDK 21 et en verrouillant la propriété `<java.version>` dans `pom.xml` . Cette étape était indispensable pour bénéficier de Spring Boot 3.3 et des records Java utilisés dans les DTO.
2. **Dépendances de tests sécurité** – Les tests `PlaceControllerTest` utilisaient `@WithMockUser` , mais la dépendance `spring-security-test` n'était pas déclarée. L'ajout de cette dépendance (scope test) dans le POM a rétabli la compilation et permis le mocking du contexte de sécurité.
3. **Exception Thymeleaf sur** `System.currentTimeMillis()` – Les templates utilisaient `T(java.lang.System).currentTimeMillis()` pour afficher l'année courante, bloqué par la sandbox Spring EL. J'ai remplacé cet appel par `T(java.time.Instant).now()` déjà autorisé, supprimant l'erreur `Access is forbidden for type 'java.lang.System'` .
4. **Statuts HTTP incohérents sur les appels anonymes** – L'API devait retourner 401 sur les routes nécessitant un token mais renvoyait 403. Après inspection de `SecurityConfig` , j'ai explicitement aligné l'ordre des chaînes de filtres et laissé Spring Security gérer la réponse 401. Les tests (`PlaceControllerTest.unauthenticatedUserCannotCreatePlace`) valident désormais le comportement attendu.
5. **Localisation des décimales dans les tests JSON** – Les tests d'API renvoyaient des latitudes formatées avec des virgules lorsque la locale système n'était pas US. Le correctif consiste à forcer `Locale.setDefault(Locale.US)` dans les méthodes `@BeforeEach` , garantissant des assertions stables sur les points GPS.
6. **Choix d'architecture pour la métrique de notation** – Plutôt qu'une vue SQL matérielle, la mise à jour des moyennes est effectuée côté service via `refreshPlaceMetrics` . Ce choix simplifie le déploiement (pas de dépendance SGBD), au prix d'une transaction supplémentaire – acceptable au vu du volume attendu.

4. Planning initial vs effectif

- **Semaine 1 – Cadrage et socle technique** : mise en place du projet Spring Boot, configuration Maven, intégration Flyway et premiers templates. Cette phase s'est déroulée comme prévu.
- **Semaine 2 – Fonctionnalités principales** : implémentation de la soumission de lieux, listing public, modération admin et authentification. Cette phase s'est déroulée comme prévu.
- **Semaine 3 – Notation, UI enrichie et tests** : ajout du module de notes, mise à jour des métriques en temps réel, pagination avancée et écriture des tests MockMvc. J'avais prévu deux jours, il en a fallu trois pour stabiliser les scénarios de tests et gérer les détails d'UX (messages flash, pré-remplissage des formulaires).
- **Semaine 4 – Documentation et travail sur les websockets** : finalisation de Swagger, préparation du guide d'installation, rédaction de ce rapport.Implémentation des websockets.

(Vu la charge du travail le long du semestre , les semaines de travail étaient séparées , donc le planning est approximatif)

5. Stratégie de tests et validation

J'ai combiné tests unitaires (services + contrôleurs via `@WebMvcTest`) et tests d'intégration MockMvc (API/UI via `@SpringBootTest`) afin de couvrir la logique métier et les endpoints publics exigés.

- `src/test/java/com/pit/service/AuthServiceTest.java` vérifie l'encodage des mots de passe, la création d'un compte (email unique) et la génération de JWT à l'inscription/connexion.
- `src/test/java/com/pit/service/PlaceServiceImplTest.java` valide la création d'un lieu (statut `PENDING`, auteur), la gestion des erreurs (utilisateur introuvable), la validation des transitions (`approve`) et la suppression.
- `src/test/java/com/pit/service/RatingServiceImplTest.java` couvre les règles de notation (score 1..5, refus si lieu non approuvé) et le recalcul des métriques.
- `src/test/java/com/pit/web/AuthControllerTest.java` teste les routes publiques `/api/auth/register` et `/api/auth/login`, y compris les validations d'email/mot de passe (test unitaire WebMvcTest).
- `src/test/java/com/pit/web/PlaceControllerTest.java` explore la matrice d'accès REST : filtrage des lieux approuvés, accès interdit aux listes `PENDING` pour les non-admins, visibilité d'un lieu en attente par son auteur, pagination, validation des coordonnées, suppression interdite pour un utilisateur standard, et suppression autorisée pour un admin.
- `src/test/java/com/pit/web/RatingControllerTest.java` vérifie la notation via l'API REST (authentifié vs anonyme), le comportement 409 sur lieux non publiés et les validations sur le score.

- `src/test/java/com/pit/web/AdminPlaceControllerTest.java` couvre la sécurisation de l'interface admin (redirection login, 403 pour user, accès admin) et l'action de suppression via la vue.
- `src/test/java/com/pit/web/PageControllerTest.java` couvre la page d'accueil, la page login et le détail d'un lieu approuvé (test unitaire WebMvcTest).
- `src/test/java/com/pit/web/PlaceControllerWebMvcTest.java` valide les statuts de liste (paramètre `status` invalide, accès `ALL`, liste approuvée) côté API publique.
- `src/test/java/com/pit/web/RatingControllerWebMvcTest.java` valide la consultation des notes pour un lieu approuvé et le 404 sur lieu inexistant.
- `src/test/java/com/pit/web/AuthPageControllerWebMvcTest.java` vérifie l'accès à la page d'inscription (test unitaire WebMvcTest).
- `src/test/java/com/pit/security/JwtServiceTest.java` vérifie la génération de JWT (claims, expiration) et l'extraction d'identité.
- `src/test/java/com/pit/security/JwtAuthFilterTest.java` vérifie le filtrage JWT (absence de header, token invalide, authentification réussie).
- `src/test/java/com/pit/web/HealthControllerTest.java` valide `/health`, utile pour les probes d'environnement.

L'intégralité de la suite s'exécute avec `mvn test`. Les tests reposent sur H2 en mode PostgreSQL pour refléter les contraintes SQL (notamment l'unicité `uk_rating_user_place`). Un run complet confirme que tous les tests passent et couvre les points sensibles du cahier des charges (API publique, modération, notation, pagination, sécurité).

6. Bilan et perspectives d'évolution

6.1 Bilan

Le produit respecte l'ensemble des contraintes fixées : interface riche permettant l'inscription et la soumission, API REST documentée, workflow administrateur pour la validation, calcul en temps réel des moyennes, pagination sur tous les flux et finalement les websockets . L'usage de Spring (Security, Data JPA, Validation) est maximisé, et les tests automatisés donnent confiance dans les comportements critiques.

6.2 Pistes d'évolution

1. **Recherche géographique** : index spatial (PostGIS) et filtrage par rayon autour d'une position GPS.
2. **Modération collaborative** : historique des décisions et commentaires d'examinateurs.
3. **Exposition GraphQL** : suggérée dans le cahier des charges comme option, pourrait compléter l'API REST.

4. **Accessibilité et internationalisation** : traductions et support RTL pour élargir la base utilisateur.

7. Démarrage rapide

1. **Prérequis** : JDK 21 (par exemple `C:\Soft\java\jdk`), Maven 3.8+.
2. **Variables d'environnement** : sur Windows/WSL, exporter `JAVA_HOME=/mnt/c/Soft/java/jdk` puis ajouter `PATH="$JAVA_HOME/bin:$PATH"`. Ce point élimine l'erreur « JAVA_HOME is not defined correctly ».
3. **Cloner et installer** : `git clone ...`, puis depuis la racine du projet `mvn clean install`. Cette commande compile, lance les tests et exécute les migrations Flyway sur l'H2 embarquée.
4. **Accéder à l'application** : http://localhost:8080 pour l'UI. Créer un compte via « Inscription » (aucune intervention API n'est nécessaire). Un administrateur par défaut (`admin@pit.local` / mot de passe `admin`) est disponible pour la modération. La documentation API est sur http://localhost:8080/swagger-ui/index.html.
5. **Exécuter les tests** : `mvn test`.