Rapport Technique du Projet

Application de gestion d'événements Java (Spring Boot)

Réalisé par Toulepi Jordan

26 mai 2025

Table des matières

Chapitre 1

Introduction

Ce rapport présente les choix techniques et conceptuels réalisés dans le cadre du développement d'une application de gestion d'événements. L'application repose sur Java avec Spring Boot et suit des principes modernes de conception logicielle.

Chapitre 2

Architecture générale (MVC)

Le projet adopte l'architecture **MVC (Model - View - Controller)** :

— **Model** : contient les entités métiers (ex : Événement, Participant).

— **View** : dans le cas d'une API, elle est représentée par les données retournées (JSON).

— **Controller** : gère les requêtes utilisateur, appelle les services et retourne la réponse.

Cette séparation permet une meilleure organisation, une testabilité accrue et un couplage faible entre les composants.

Chapitre 3

Design Patterns utilisés

3.1 Pattern Strategy

Utilisé pour permettre un changement dynamique de la stratégie de validation ou de gestion des événements (ex : différents modes d'inscription).

Avantages :

— Facilite l'extension de comportements sans modifier le code existant.

— Réduit le couplage.

3.2 Pattern Observer

Appliqué pour notifier automatiquement certains modules (ex : notification d'un participant quand un événement change d'état).

Avantages :

— Synchronisation des données.

— Réactivité accrue.

3.3 Pattern Singleton

Utilisé pour certaines classes utilitaires (ex : gestion centralisée de configuration ou d'ID unique).

Avantages :

— Réduction de la consommation mémoire.

— Garantie d'une instance unique.

Chapitre 4

Principes SOLID appliqués

4.1 S – Responsabilité unique

Chaque classe (Participant, Evenement, etc.) a une responsabilité bien définie.

4.2 O – Ouvert/Fermé

Le code est conçu pour permettre l'ajout de nouveaux comportements (ex : nouvelles règles de validation) sans modification des existants.

4.3 L – Substitution de Liskov

Les services et exceptions peuvent être remplacés par des sous-classes sans briser le comportement.

4.4 I – Ségrégation des interfaces

Les interfaces sont spécifiques et évitent de forcer les classes à implémenter des méthodes inutiles.

4.5 D – Inversion des dépendances

Les dépendances sont injectées via Spring (Injection de dépendances).

Chapitre 5

Sérialisation/Désérialisation avec Jackson

5.1 Utilisation

Les entités sont automatiquement converties en JSON grâce à la bibliothèque `Jackson` via les annotations `@JsonProperty`, `@JsonIgnore`, etc.

5.2 Avantages

— Simplicité et rapidité de conversion.
— Haute compatibilité avec les formats REST.
— Permet de manipuler aisément les flux d'entrée/sortie.

Chapitre 6

Exceptions personnalisées

6.1 Justification

Des classes d'exception spécifiques permettent une gestion fine des erreurs :

— `CapaciteMaxAtteinteException`

— `EvenementDejaExistantException`

— `ParticipantNonExistantException`

6.2 Avantages

— Meilleure lisibilité et traçabilité des erreurs.

— Possibilité de retourner des messages d'erreur personnalisés à l'utilisateur.

— Facilite le debugging et les tests.

Chapitre 7

Développement orienté API (REST)

7.1 Choix

L'API expose des endpoints RESTful pour gérer les entités (CRUD), facilitant l'intégration avec des front-ends modernes (mobile ou web).

7.2 Avantages

— Requêtes HTTP simples (GET, POST, PUT, DELETE).
— Architecture scalable et légère.
— Facilité d'intégration avec des clients externes.

Chapitre 8

Conclusion

L'application repose sur des choix de conception solides et éprouvés. L'utilisation de patterns et de principes modernes a permis de produire une solution maintenable, modulaire et évolutive.

Ces bonnes pratiques garantissent une meilleure gestion de la complexité, facilitent les tests unitaires et rendent le projet plus professionnel.