RAPPORT DU TP DE PROGRAMMATION ORIENTÉE OBJET

TITRE : APPLICATION DE GESTION DES ÉVÈNEMENTS

AUTEUR : AHMED JALIL TADIDA D.

SUPERVISEUR : DR KUNGNE

Table des matières

INTRODUCTION

La programmation orientée objet (POO) est un pilier fondamental du développement logiciel moderne, permettant de concevoir des applications robustes, modulaires et évolutives. Dans le cadre de ce travail pratique, nous avons développé une application de gestion des événements en Java, en exploitant le framework Spring Boot pour une architecture RESTful. Cette application vise à simplifier la gestion des événements à travers des opérations CRUD (Create, Read, Update, Delete), la gestion des participants, et la persistance des données dans un fichier JSON, utilisé comme une base de données légère. Ce projet est une opportunité unique de mettre en pratique les concepts clés de la POO, tels que l'encapsulation, l'héritage, le polymorphisme et l'abstraction, tout en intégrant des techniques avancées comme la sérialisation JSON et le design pattern Observer. L'objectif est de créer une solution intuitive, performante et extensible, capable de répondre aux besoins des organisateurs et des participants tout en gérant efficacement les erreurs et les notifications. Ce rapport détaille le cahier des charges, la conception, l'implémentation, et les leçons apprises, tout en offrant une analyse critique des diagrammes utilisés.

I- CAHIER DE CHARGES

1- Objectifs Fonctionnels

L'application répond aux besoins suivants :

— Gestion des inscriptions : Permettre aux participants de s'inscrire ou de se désinscrire à des événements.

— Gestion des organisateurs : Offrir aux organisateurs la possibilité de créer, modifier ou supprimer des événements.

— Notifications automatiques : Informer les participants en cas de modification ou d'annulation d'un événement.

— Persistance des données : Sauvegarder les informations dans des fichiers JSON ou XML.

2- Contraintes Techniques

Les outils et technologies utilisés sont :

— Langage : Java (version 23)

— IDE : IntelliJ IDEA

— Framework : Spring Boot (version 3.2.0)

— Tests : Postman pour les endpoints REST,

JUnit pour les tests unitaires

Les dépendances Maven incluent :

— spring-boot-starter-web (3.2.0) : Pour les fonctionnalités REST

— jackson-databind (2.15.2) : Pour la sérialisation JSON

— junit-jupiter (5.10.0) : Pour les tests unitaires

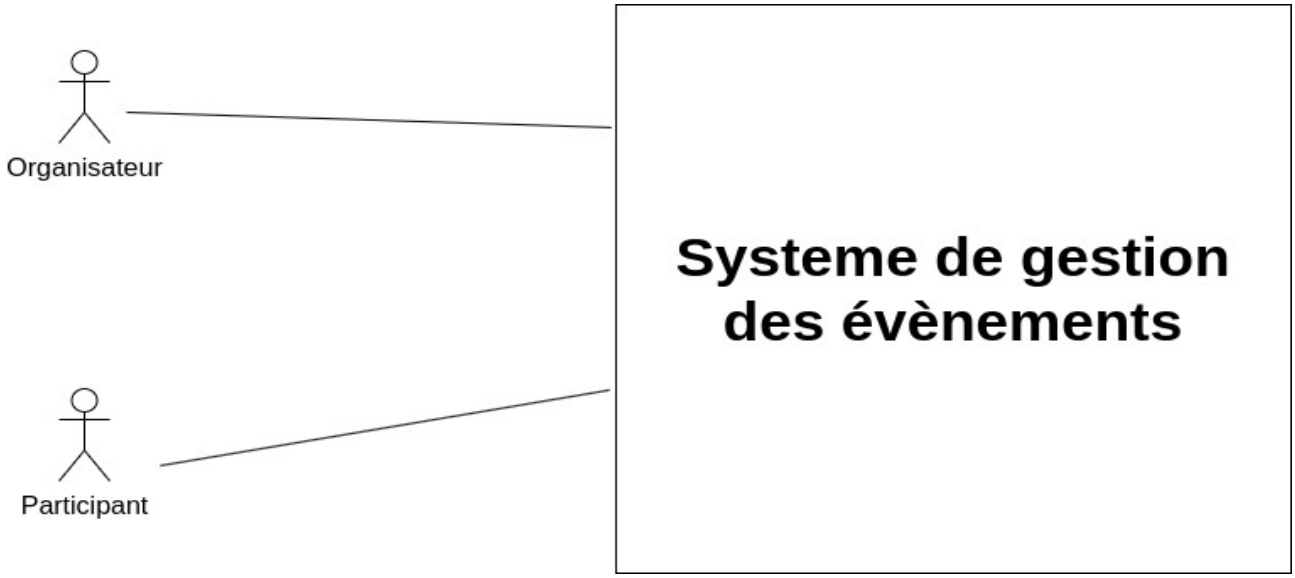
— lombok (1.18.30) : Pour réduire le code boilerplate

II- CONCEPTION

1- Diagrammes UML

1.1 -Diagramme de contexte

Le diagramme de contexte illustre les interactions entre les acteurs (Organisateur, Participant) et le système. L'organisateur crée, modifie ou supprime des événements, tandis que le participant s'inscrit ou se désinscrit. Le système envoie des confirmations et des notifications.



1-2- Diagramme de classe

Les classes principales sont :

— Evenement : Classe abstraite avec nom : String, date : LocalDate, lieu : String, capacite : int, ajouterParticipant().

— Concert : Hérite de Evenement, ajoute artiste : String.

— Conference : Hérite de Evenement, ajoute theme : String, orateur : String.

— Participant : Attributs nom : String, email : String, méthode sInscrire().

— Organisateur : Hérite de Participant, ajoute creerEvenement().

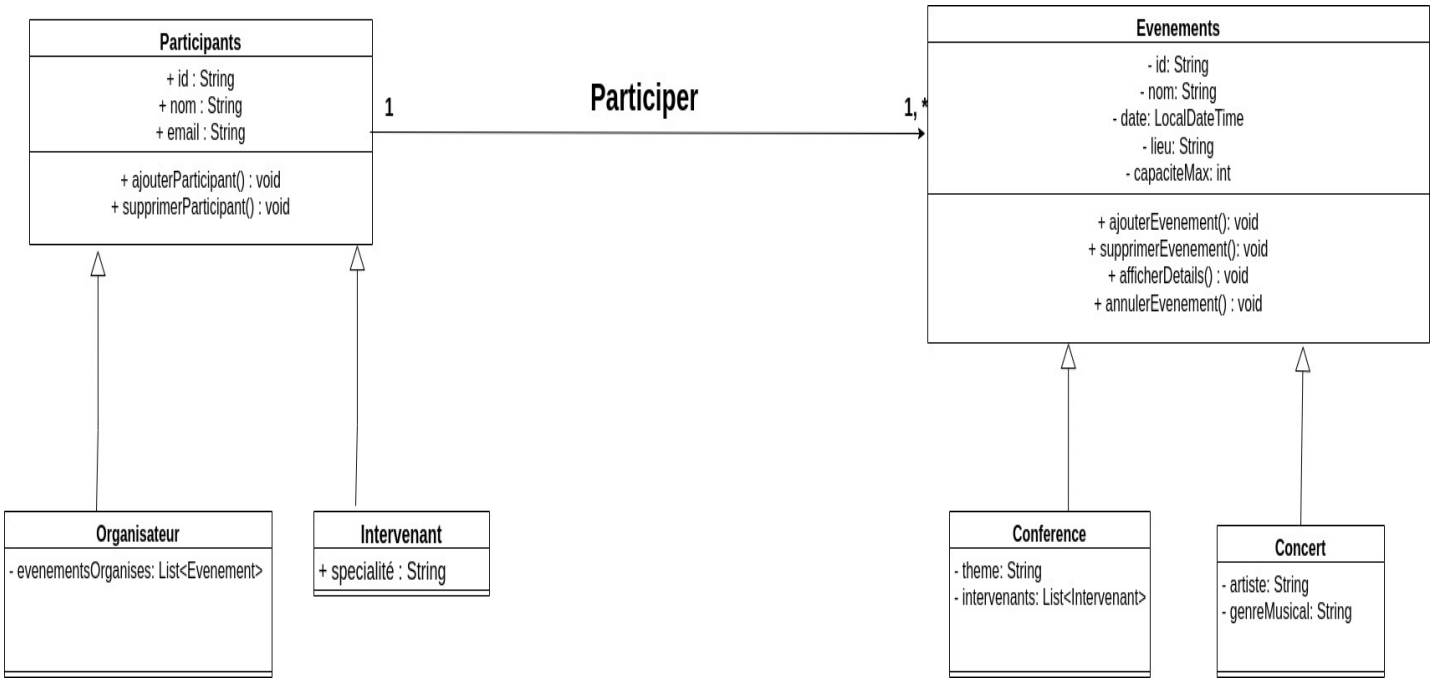
— Intervenant : Attributs nom : String, specialite : String.

Les relations incluent :

— Héritage : Concert et Conference héritent de Evenement. Organisateur et intervenant héritent de participant

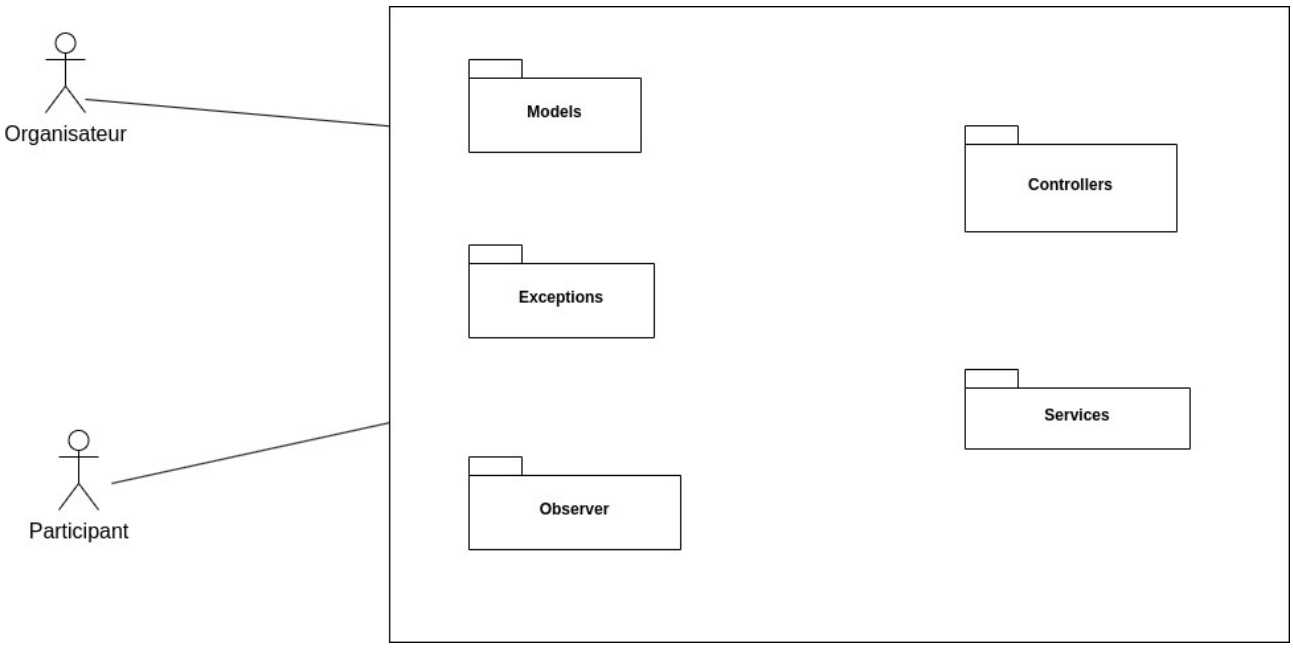
— Composition : Evenement contient une liste de Participant (inscriptions).

— Association : Conference est associée à Intervenant.



1-3- Diagramme de paquetage

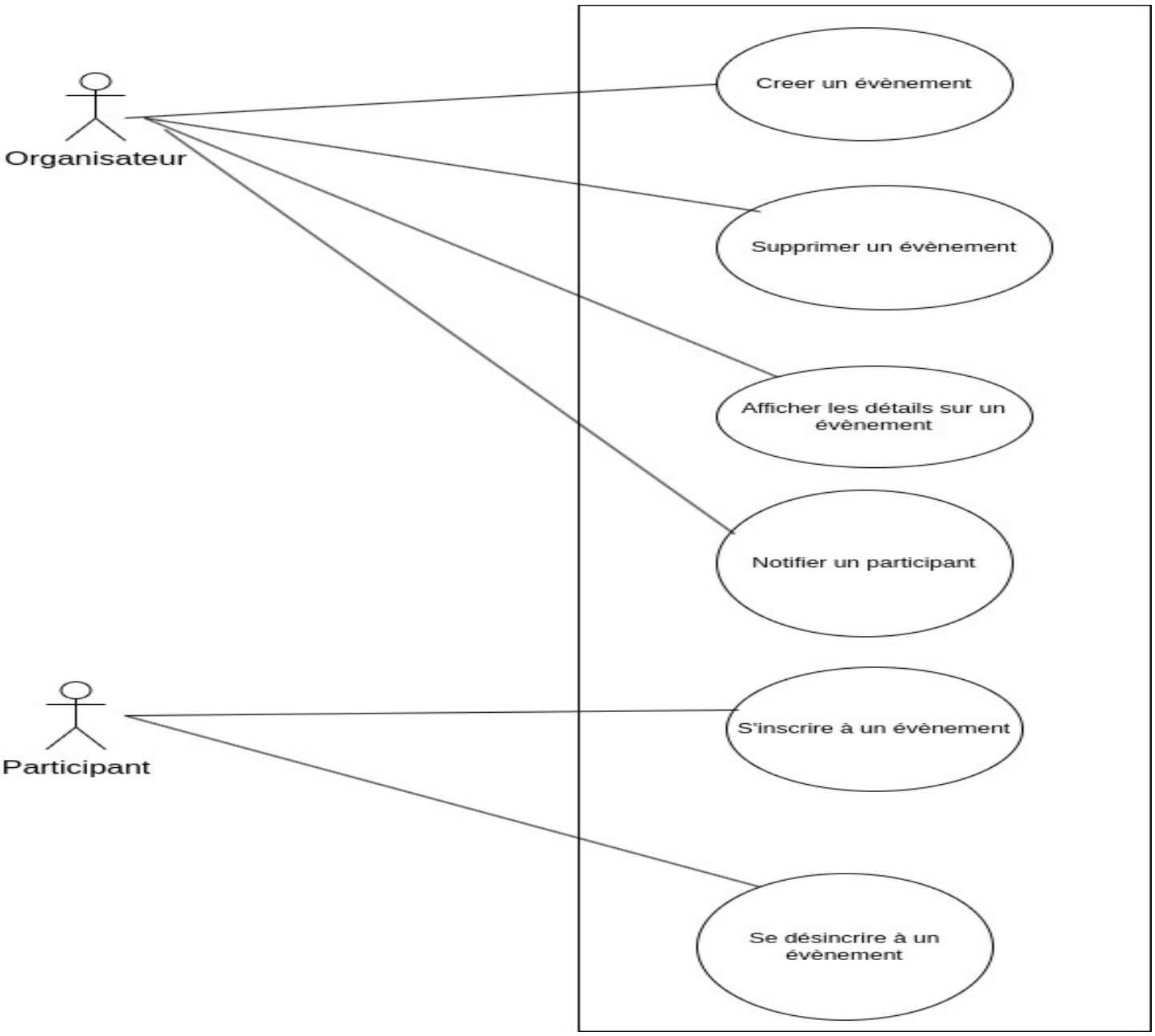
Les packages de l'application sont :
— Models : Contient les classes Evenement, Concert, Conference, Participant, Organisateur, Intervenant, GestionEvenement.
— Exceptions : Inclut EvenementDejaExistant, EvenementInvalide, CapaciteMaxAtteinte, ParticipantNonTrouve, etc.
— Observer : Interfaces EvenementObserver et ParticipantObservable.
— Controller : Contrôleurs REST pour les endpoints.
— Services : Services métiers comme EvenementService, JsonDataService, NotificationService, ParticipantService.
Les dépendances sont :
— Controller dépend de Services.
— Services utilise Models, Exceptions, et Observer.



2- Diagramme de conception

2-1 Diagramme de cas d’utilisation

Les cas d’utilisation incluent :
— Organisateur : Créer/modifier/supprimer un événement, consulter la liste des participants, envoyer des notifications.
— Participant : S’inscrire/se désinscrire, consulter les événements disponibles.


Organisateur
Creer un évènement
Supprimer un évènement
Afficher les détails sur un évènement
Notifier un participant
Participant
S'inscrire à un évènement
Se désincrire à un évènement

III- NOTIONS SUR LE DESIGN PATTERN

Les design patterns, ou patrons de conception, sont des solutions standardisées à des problèmes récurrents en programmation orientée objet (POO). Ils permettent de structurer le code de manière modulaire, maintenable et extensible, tout en respectant les principes fondamentaux comme l'encapsulation et le polymorphisme. Dans notre application de gestion des événements, développée en Java avec Spring Boot, deux patterns ont été essentiels : le Singleton et l'Observer. Ces patterns répondent respectivement aux besoins de centralisation des ressources et de communication dynamique.

1. Le Pattern Singleton

Définition et rôle

Le Singleton garantit qu'une classe n'a qu'une seule instance dans l'application et fournit un point d'accès global à cette instance. Il est particulièrement adapté aux ressources partagées, comme un gestionnaire de configuration ou un service de persistance de données.

Structure

- Un **constructeur privé** pour empêcher l'instanciation directe.
- Une **instance statique** de la classe, souvent initialisée de manière paresseuse.
- Une **méthode statique publique** (getInstance()) pour accéder à l'instance unique.

Application dans le projet

Dans notre application, le Singleton est utilisé pour la classe JsonDataService, qui gère la sauvegarde des événements dans un fichier JSON. Une instance unique est cruciale pour éviter les conflits d'accès au fichier et garantir la cohérence des données, par exemple lors de la création ou de la modification d'un événement.

Avantages

- **Cohérence** : Une seule instance évite les problèmes d'accès concurrent au fichier JSON.
- **Accessibilité** : La méthode getInstance() permet un accès facile depuis n'importe quel composant.
- **Efficacité** : Réduit l'utilisation des ressources en évitant la création de multiples instances.

Limites

- **Testabilité** : L'état global du Singleton peut compliquer les tests unitaires.
- **Concurrence** : Dans un environnement multi-threadé, comme notre application, une synchronisation (ex. : double-checked locking) est nécessaire.
- **Couplage** : Les classes dépendantes du Singleton sont moins flexibles pour les modifications.

Analyse dans le projet

Le Singleton est idéal pour JsonDataService, car il centralise la gestion des données JSON. Pour gérer les accès concurrents, nous avons implémenté une initialisation paresseuse avec double vérification, garantissant la sécurité dans un contexte multi-threadé. Cependant, pour les tests, une alternative comme l'injection de dépendances pourrait être envisagée pour réduire le couplage.

2. Le Pattern Observer

Définition et rôle

Le pattern Observer permet à un objet (le sujet) de notifier automatiquement un ensemble d'objets dépendants (les observateurs) lorsqu'un changement d'état survient. Il est parfait pour les systèmes nécessitant des notifications dynamiques, comme les mises à jour en temps réel.

Structure

- **Sujet** : Maintient une liste d'observateurs et notifie les changements via une méthode comme notifyObservers().
- **Observateur** : Interface avec une méthode update() pour recevoir les notifications.
- **Observateurs concrets** : Implémentent la logique de mise à jour.

Application dans le projet

Dans notre application, le pattern Observer est utilisé pour envoyer des notifications automatiques aux participants lorsqu'un événement est modifié ou annulé. L'événement (Evenement) est le sujet, et les participants (Participant) sont les observateurs, notifiés via le service NotificationService.

Avantages

- **Flexibilité** : Les observateurs peuvent être ajoutés ou supprimés dynamiquement.
- **Découplage** : Le sujet et les observateurs sont indépendants, facilitant la maintenance.
- **Réactivité** : Les notifications sont envoyées en temps réel, améliorant l'expérience utilisateur.

Limites

- **Performance** : Une grande liste d'observateurs peut ralentir les notifications.
- **Complexité** : La gestion des abonnements peut devenir lourde pour les systèmes complexes.

Analyse dans le projet

Le pattern Observer est parfait pour notre application, car il permet de notifier les participants en cas de changement (par exemple, annulation d'un concert). Pour les événements avec de nombreux participants, une optimisation, comme l'utilisation d'une file d'attente pour les notifications, pourrait être envisagée pour éviter les problèmes de performance.

3. Étude de Cas : Application des Patterns

Imaginons un scénario où un organisateur modifie un concert :

- **Singleton** : L’organisateur utilise JsonDataService.getInstance() pour sauvegarder la modification dans le fichier JSON, garantissant une gestion cohérente des données.
- **Observer** : La modification déclenche notifyObservers("Le concert a été modifié"), et tous les participants inscrits reçoivent une notification via leur méthode update().

Ce scénario montre comment les deux patterns collaborent pour assurer une gestion fluide et une communication efficace.

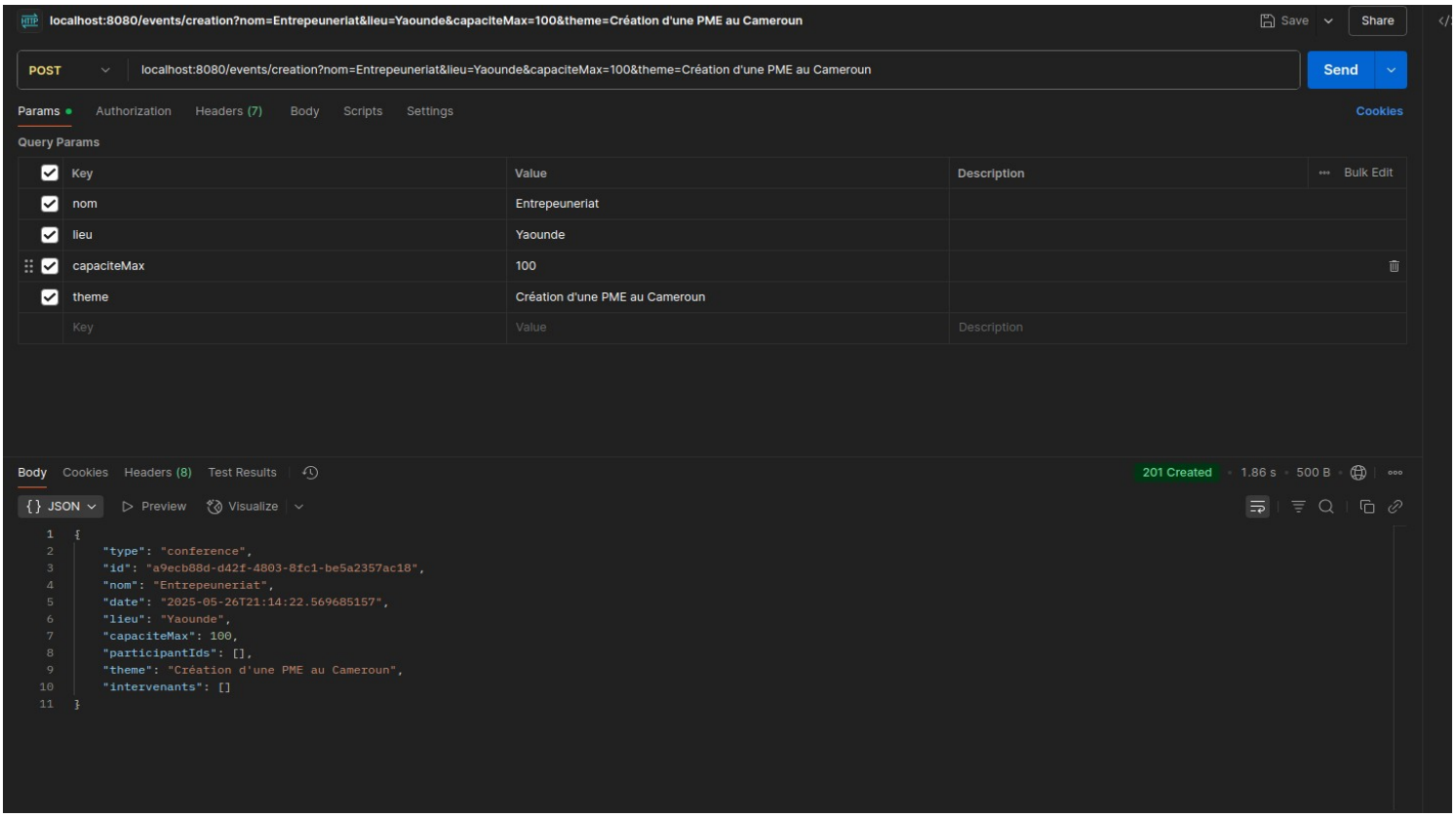
4. Importance des Design Patterns

Les patterns Singleton et Observer ont été essentiels pour notre application. Le Singleton a garanti une gestion centralisée des données, tandis que l’Observer a permis une communication dynamique avec les participants. Ces solutions standardisées ont renforcé la modularité, la fiabilité et l’extensibilité du code. À l’avenir, des patterns comme le Factory pourraient être explorés pour simplifier la création d’événements (par exemple, instancier Concert ou Conference).

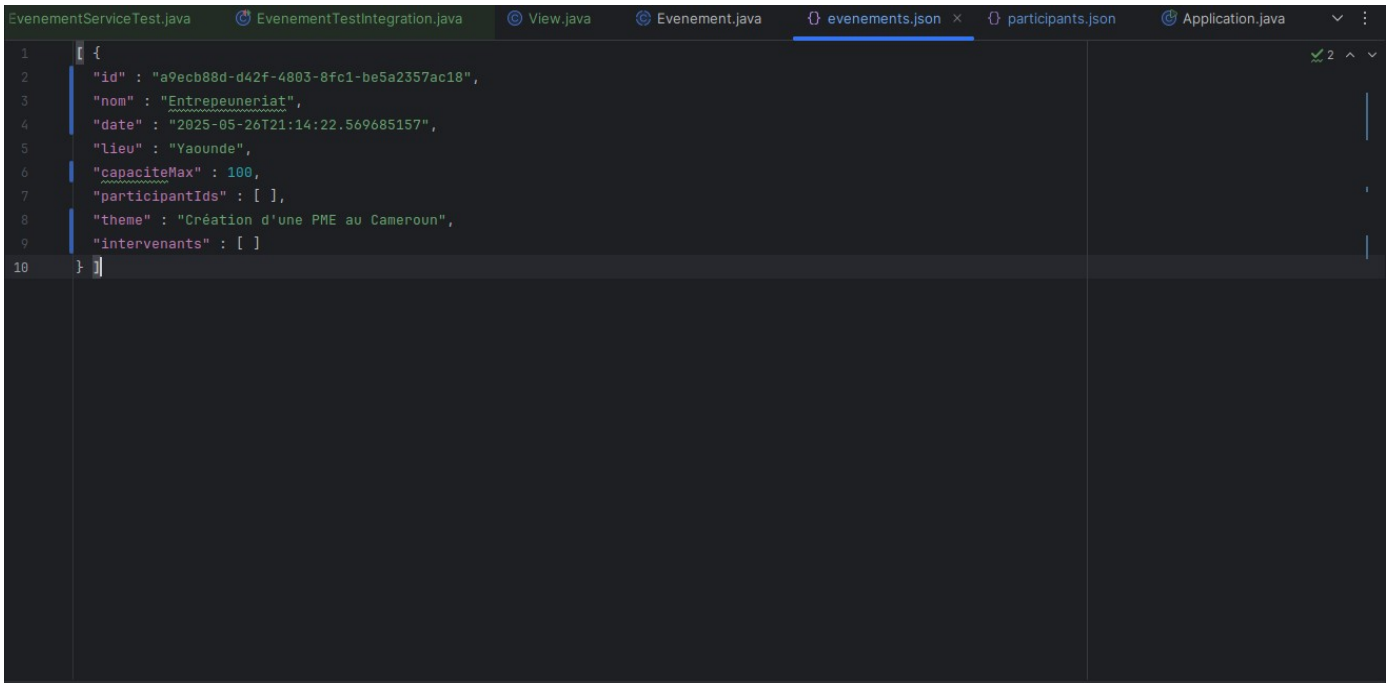
IV- IMPLÉMENTATION

Dans cette partie du travail, nous allons montrer étape par étape comment l’application fonctionne. Le code source se trouve sur le dépôt github https://github.com/AHMED-X18/TP-POO-Gestion-evenementiel. Nous allons juste montré le résultat de l’exécution de quelques fonctions comme Créer evenement, Créer participant, Inscrire Participant, Notifier Participant.
Nous avons utilisé l’approche des endpoints. Nous montrerons donc les résultats produits par Postman

Création d’un évènement

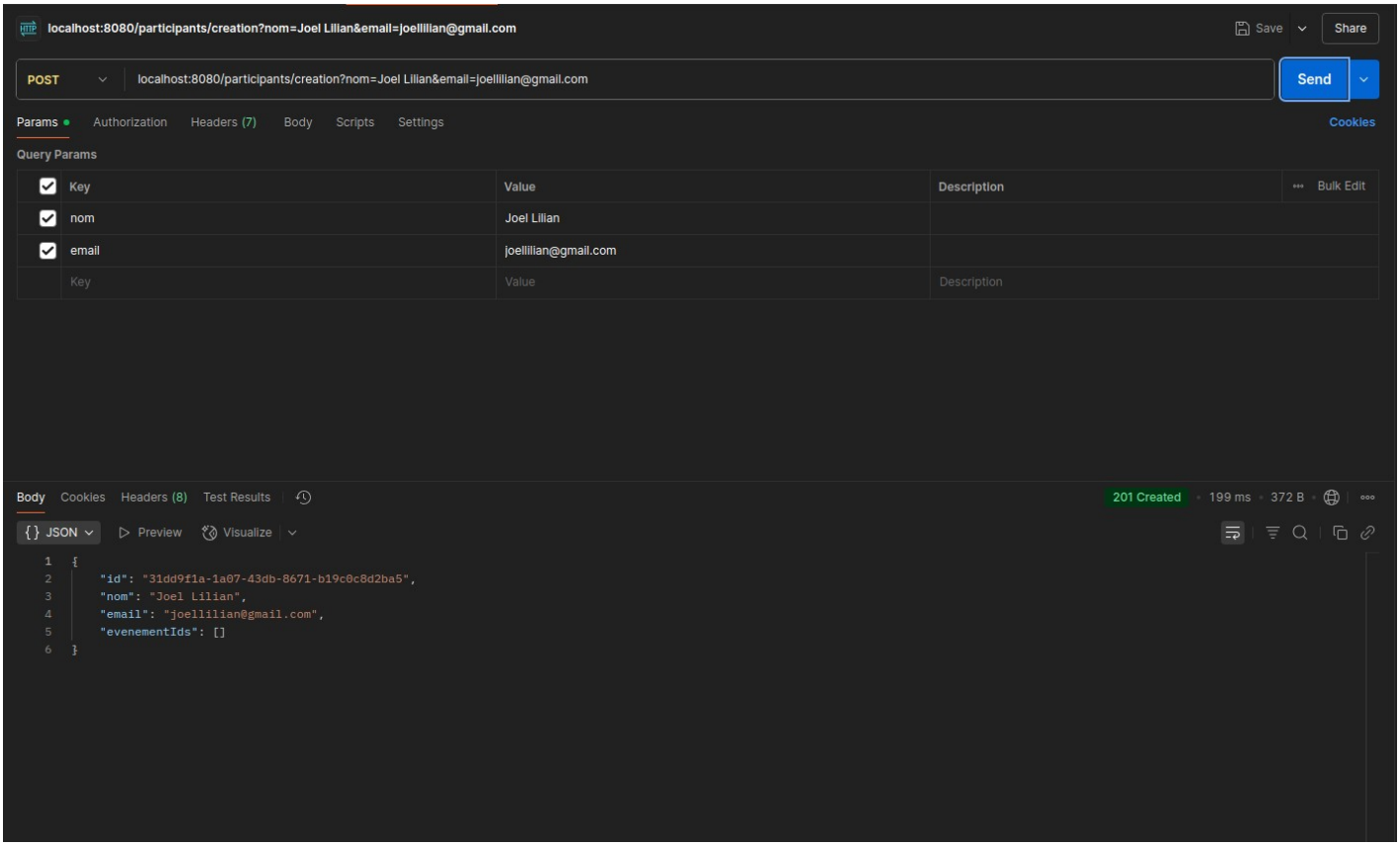


Test avec Postman

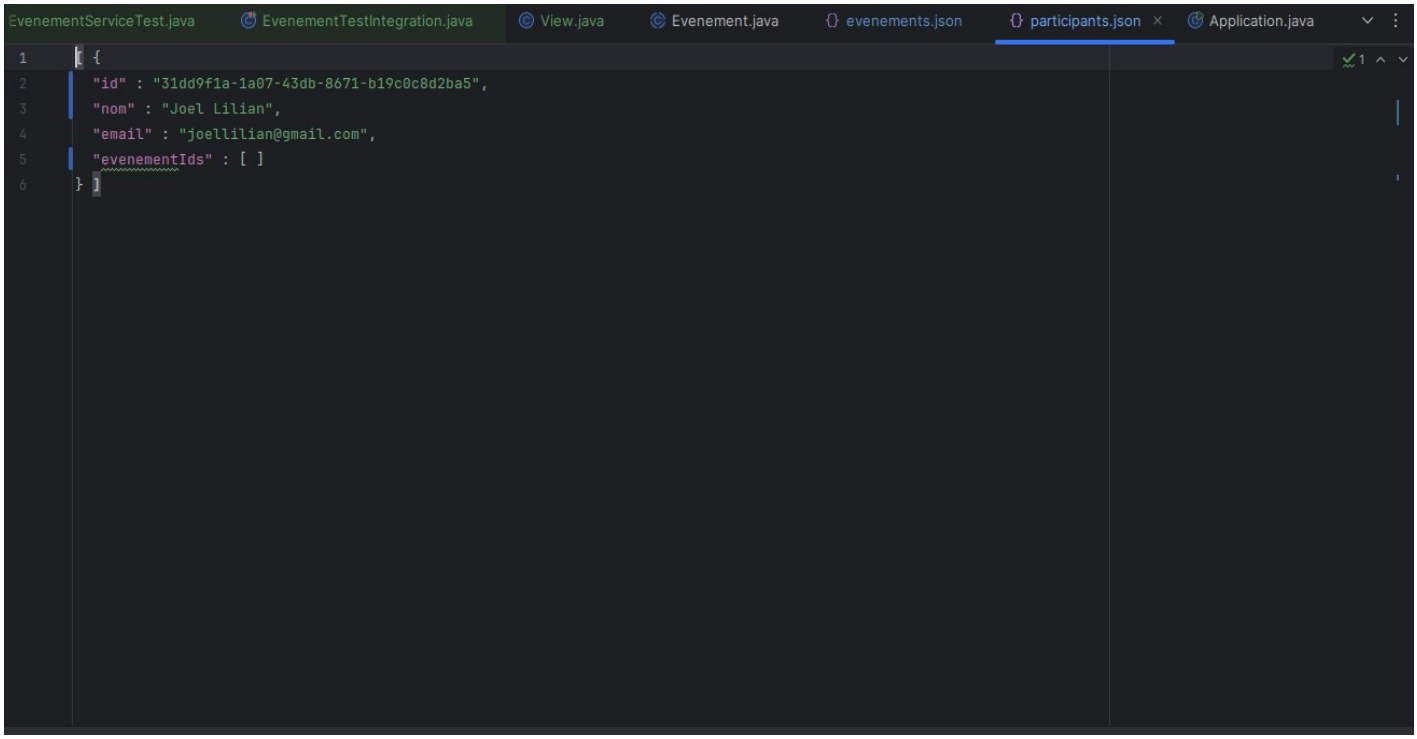


Écriture dans le fichier evenements.json

Création d'un participant

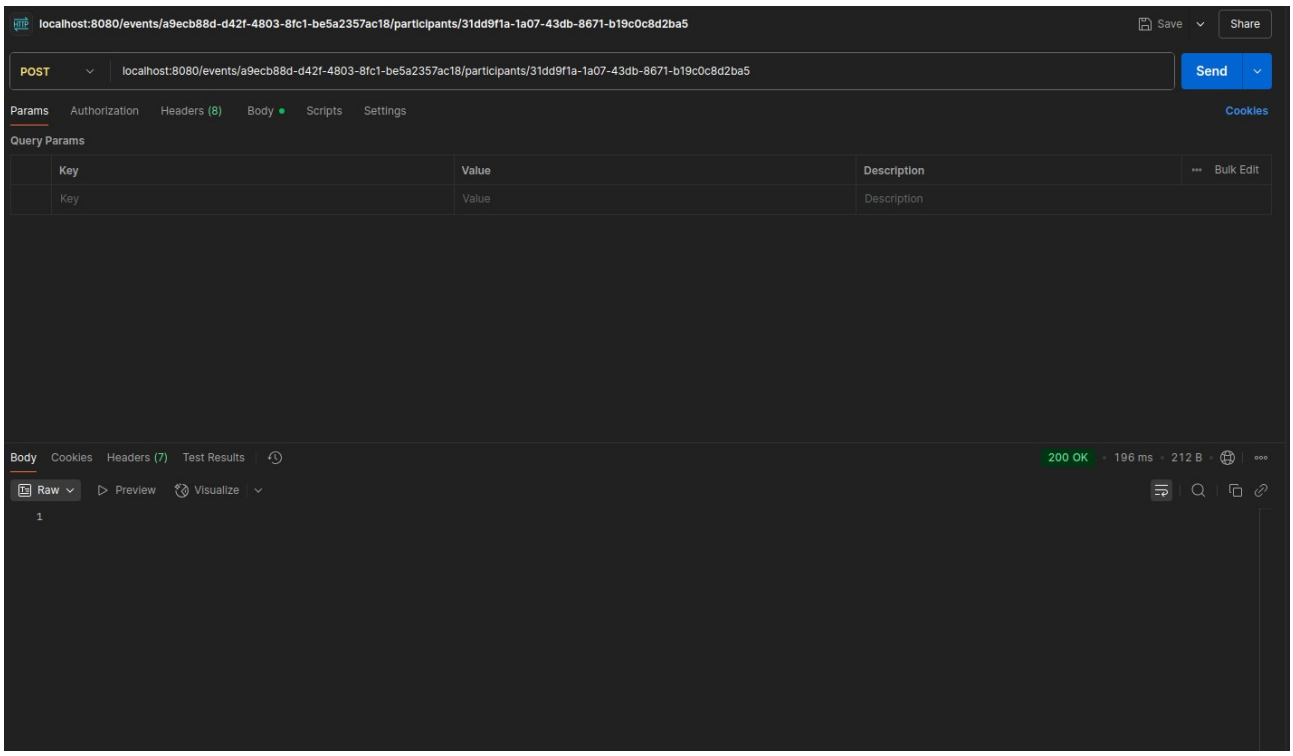


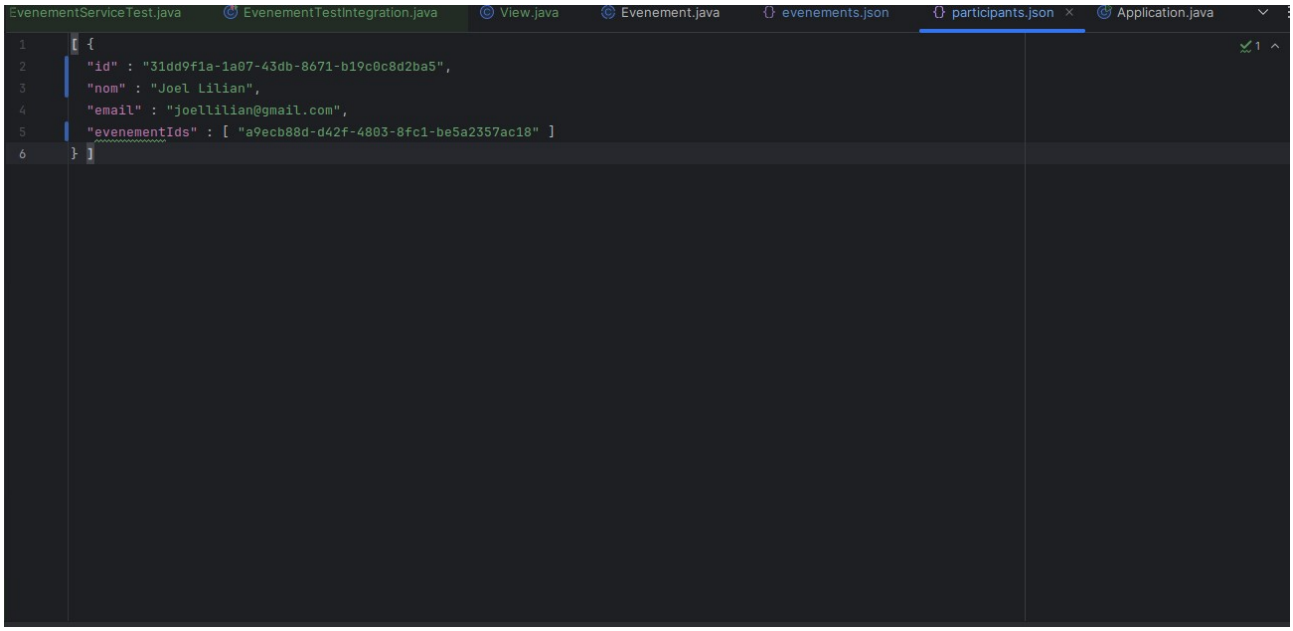
Test avec Postman

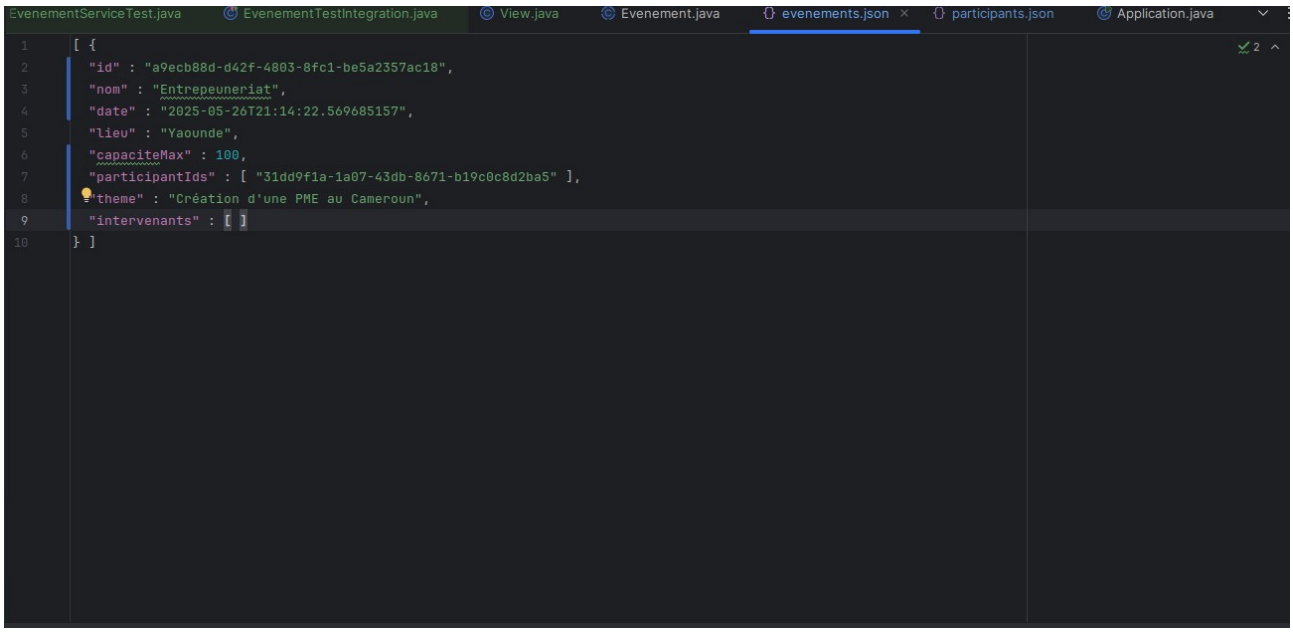


Écriture dans le fichier participants.json

Inscription à un évènement

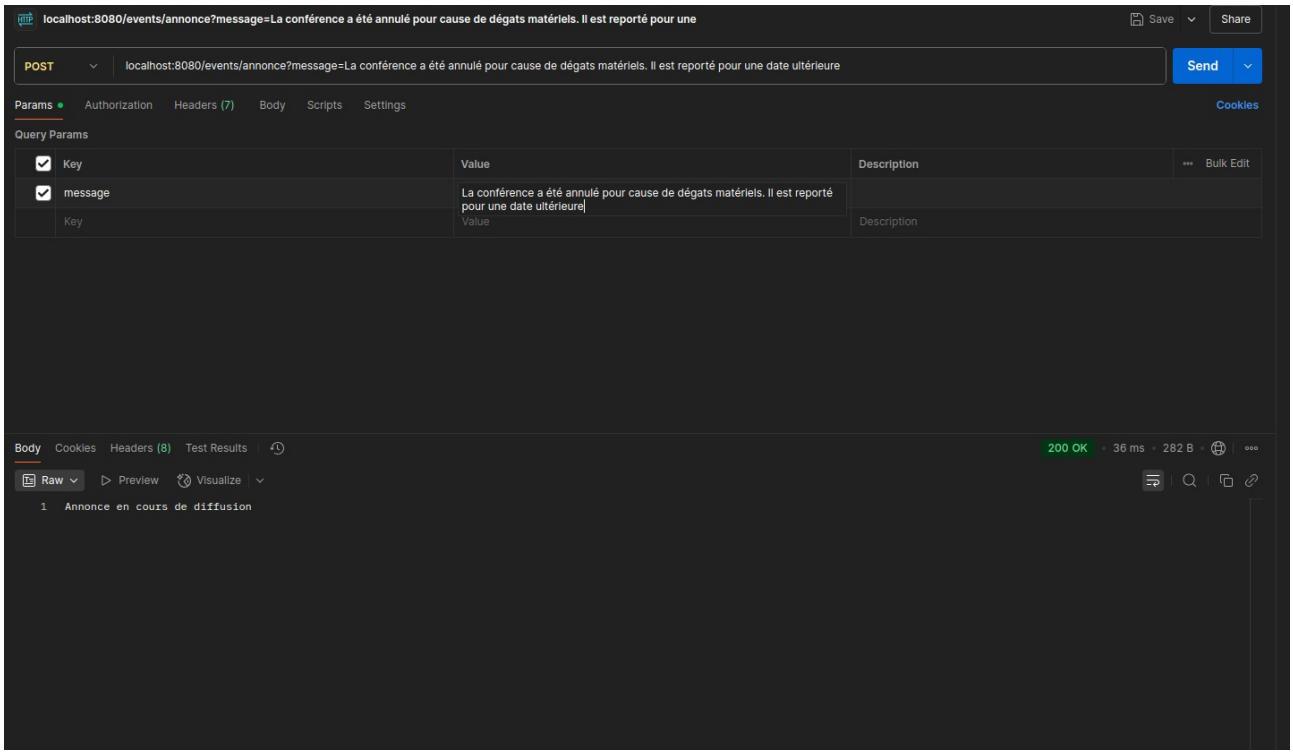

localhost:8080/events/a9ecb88d-d42f-4803-8fc1-be5a2357ac18/participants/31dd9f1a-1a07-43db-8671-b19c0c8d2ba5
Save
Share
POST
localhost:8080/events/a9ecb88d-d42f-4803-8fc1-be5a2357ac18/participants/31dd9f1a-1a07-43db-8671-b19c0c8d2ba5
Send
Params
Authorization
Headers (8)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (7)
Test Results
200 OK
196 ms
212 B
Raw
Preview
Visualize
1

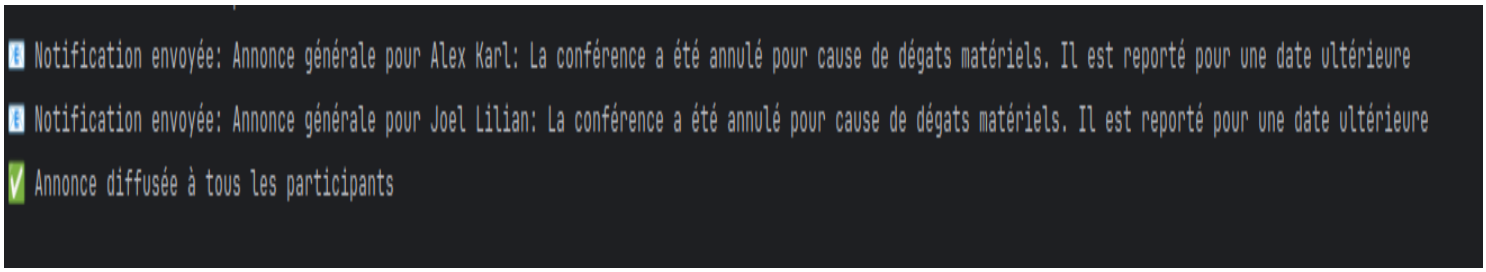

EvenementServiceTest.java
EvenementTestIntegration.java
View.java
Evenement.java
evenements.json
participants.json
Application.java
1
[{
"id" : "31dd9f1a-1a07-43db-8671-b19c0c8d2ba5",
"nom" : "Joel Lilian",
"email" : "joellilian@gmail.com",
"evenementIds" : ["a9ecb88d-d42f-4803-8fc1-be5a2357ac18"]
}]



Remarque : Les tables participantIds (respectivement evenementsIds) dans les différents fichiers json prennent la valeur de l'évènement auquel est inscrit le participant (respectivement les participants inscrits à l'évènement donné)

Notifier Participant





CONCLUSION

Il a été question ici pour nous de mettre sur pied une application de gestion des évènements avec différents objectifs fonctionnels : CRUD sur un évènement, inscrire et désinscrire des participants, sérialisation et désérialisation des données. Nous avons utilisé le langage Java qui est de type orientée objet ; ainsi, il nous a fallut d'abord peaufiner notre phase de conception avant de se lancer dans l'écriture du code. Nous avons énuméré les diagrammes d'analyse (diagramme de contexte, de classe, de paquetage) et les diagrammes de conception (diagramme de cas d'utilisation). Ensuite, nous avons expliqué les notions du design pattern avec en particulier Singleton et Observer que nous avons utilisé dans le programme. Et enfin, nous avons présenté quelques résultats des tests effectués.

Ce travail pratique est un exercice bénéfique dans le cadre d'apprentissage afin de mieux respecter les principes de la programmation orientée objet notamment le SOLID sans quoi la maintenabilité d'une application est vouée à l'échec.