1. Contexte et Choix Initiaux

1.1. Domaine d'Application Choisi

Conformément aux directives du TP, J’ai choisi de développer une **application de réservation de places pour des événements**. Ce domaine inclut la gestion d'événements (concerts, conférences, etc.), la visualisation des places disponibles, le processus de réservation et la confirmation. Ce domaine est suffisamment riche pour explorer différentes problématiques architecturales sans être excessivement complexe pour une démonstration.

1.2. Contrainte Métier Particulière

La contrainte métier particulière que nous souhaitons adresser est la **gestion en temps réel de la disponibilité des places et la prévention des surréservations (surbooking) lors de pics de demande.** Cette contrainte rend le projet intéressant car :

- Elle nécessite une forte cohérence des données concernant l'état des places.
- Elle implique de gérer potentiellement un grand nombre de requêtes concurrentes, notamment à l'ouverture des ventes pour un événement populaire.
- Elle justifie l'utilisation d'un système de base de données robuste et potentiellement distribué pour assurer la disponibilité et la cohérence.

1.3. Exigence de Cluster MariaDB (Galera)

L'exigence d'utiliser un cluster MariaDB, spécifiquement avec Galera, s'intègre bien avec la contrainte métier choisie. Un

cluster Galera offrira :

- **Haute disponibilité :** Si un nœud de la base de données tombe, les autres peuvent prendre le relais, minimisant l'interruption de service, ce qui est crucial lors d'un processus de réservation.
- **Cohérence des données (synchronous replication) :** Toutes les écritures sont répliquées sur tous les nœuds avant d'être validées, garantissant que tous les services accédant à la base de données voient le même état des places disponibles, évitant ainsi les surréservations.
- **Scalabilité en lecture :** Les lectures peuvent être distribuées sur les différents nœuds.

2. Choix d'Architecture

2.1. Architecture Proposée : Microservices

Nous proposons d'adopter une architecture basée sur les **microservices**. Pour cette démonstration, nous nous concentrerons sur un ensemble minimal mais représentatif de services.

Services envisagés (pour une version simple) :
1. Service Événements (Event Service) : Responsable de la création, lecture, mise à jour, suppression (CRUD) des informations sur les événements (nom, date, lieu, capacité totale, plan de salle simplifié).

2. Service Réservations (Booking Service) : Responsable de la gestion des réservations. Il vérifiera la disponibilité des places (en interaction potentielle avec le Service Événements

ou sa propre vue matérialisée des disponibilités), créera des réservations, et gérera leur état.

3. Service Utilisateurs (User Service) (Optionnel pour la démo, pourrait être simplifié) : Gérerait les informations des utilisateurs effectuant les réservations. Pour simplifier, on pourrait initialement se contenter d'un `userId` sans gestion complète.

4. API Gateway: Point d'entrée unique pour les requêtes clientes, routage vers les services appropriés, et potentiellement gestion de l'authentification/autorisation (simplifiée pour la démo).

2.2. Justification du Choix Architectural (Microservices)

1. Alignement avec la Contrainte Métier :
    * Le Booking Service, critique pour la gestion des places en temps réel, peut être isolé et optimisé spécifiquement pour la performance et la cohérence.
    * D'autres services, comme la gestion du catalogue d'événements, ont des exigences différentes et peuvent évoluer indépendamment.

2. Scalabilité Indépendante : Si le Booking Service subit une charge élevée, il peut être scalé indépendamment des autres services.

3. Résilience : Une défaillance dans un service (ex: Service Événements) ne devrait pas impacter la capacité de prendre des réservations pour des événements déjà chargés, si les

informations nécessaires sont disponibles ou mises en cache dans le Booking Service.

4. Modularité et Maintenabilité : Chaque service est plus petit, plus facile à comprendre, à développer, à tester et à maintenir.

5. Adéquation avec le Cluster DB : Chaque microservice pourrait interagir avec le cluster MariaDB. Le Booking Service bénéficierait particulièrement de la cohérence forte de Galera pour éviter les surréservations.

6. Démonstration des Concepts : Une architecture microservices permet de bien illustrer les problématiques de communication inter-services, de découverte de services, et de gestion distribuée des données, même de manière simplifiée.

2.3. Diagrammes

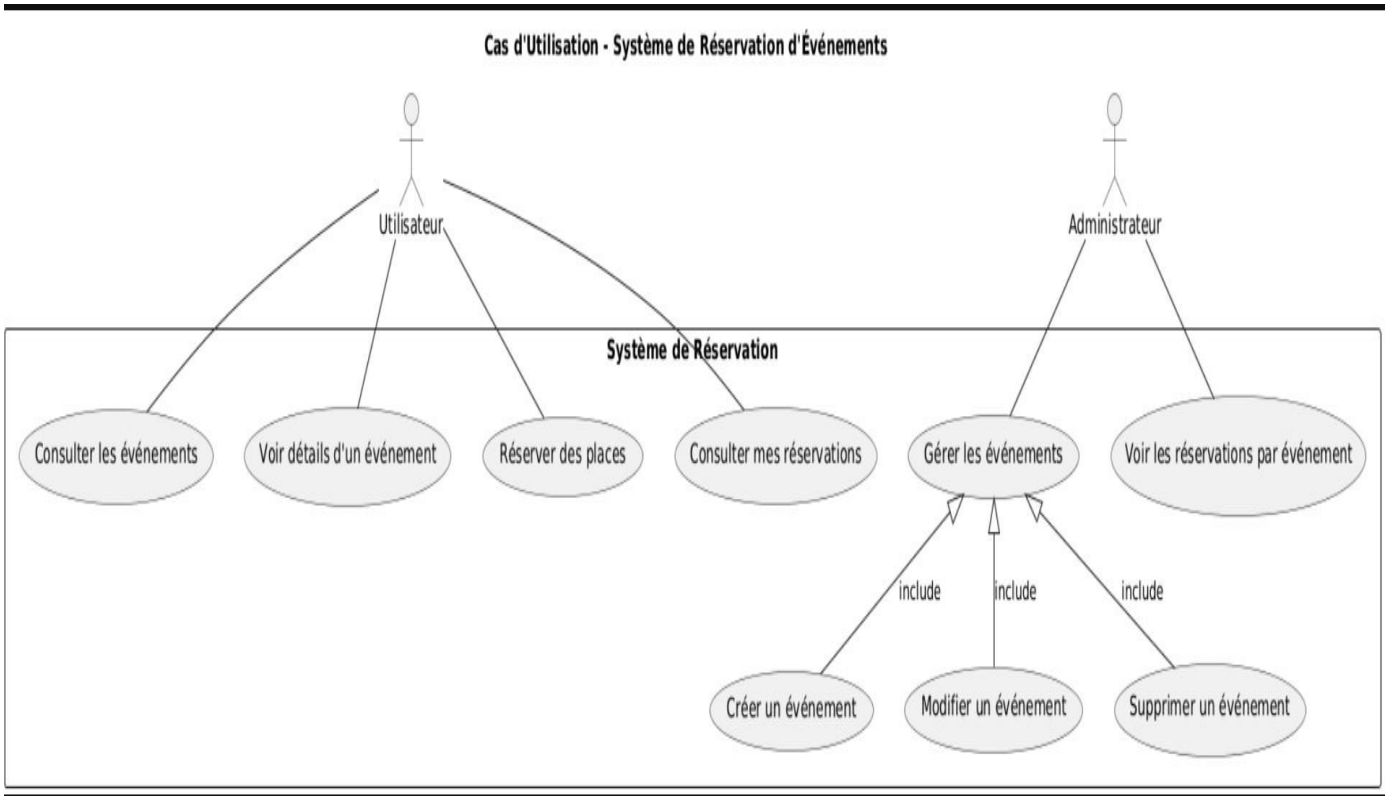
1. Diagramme de Cas d'Utilisation (Use Case Diagram)

Objectif : Identifier les principales fonctionnalités offertes par le système du point de vue des utilisateurs (acteurs).

Acteurs :

- Utilisateur : Peut consulter la liste des événements, voir les détails d'un événement spécifique, réserver des places, et consulter ses propres réservations.
- Administrateur : Gère le cycle de vie des événements (création, modification, suppression) et peut visualiser les réservations effectuées pour chaque événement.

Fonctionnalités Clés du Système : Elles sont représentées par les ellipses à l'intérieur du rectangle "Système de Réservation". Ce diagramme aide à définir le périmètre fonctionnel de l'application.



2. Diagramme de Classes (Domain Model - Simplifié)

Objectif : Représenter les concepts métiers clés (entités), leurs attributs, et les relations entre eux. C'est une vue statique de la structure des données et de la logique métier fondamentale.

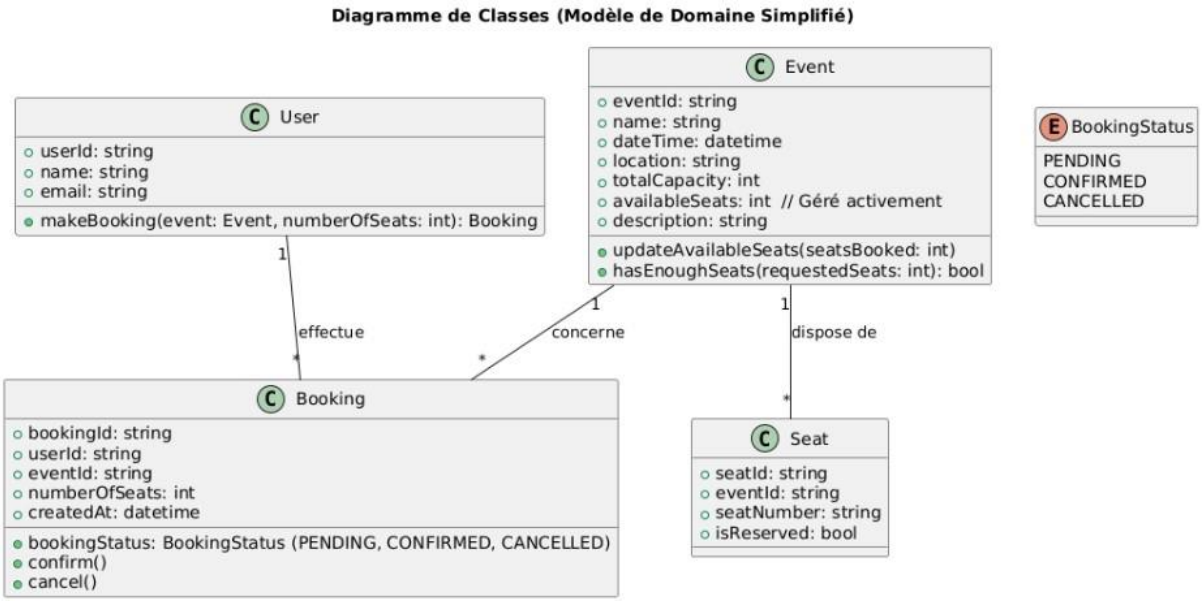
Classes Principales :

- `User` : Représente un utilisateur du système.
- `Event` : Représente un événement pour lequel des places peuvent être réservées. Contient des informations comme le nom, la date, la capacité totale, et un compteur `availableSeats` qui sera crucial pour notre contrainte métier.
- `Booking` : Représente une réservation effectuée par un utilisateur pour un événement donné, incluant le nombre de places et le statut de la réservation.
- `BookingStatus` (Enum) : Définit les états possibles d'une réservation.
- `Seat` (Optionnelle pour la démo) : Si l'on souhaitait une gestion individuelle des sièges (ex: A1, A2), cette classe serait pertinente.

Pour la démo, nous nous concentrerons probablement sur un nombre total de places.

Relations : Montre comment les utilisateurs effectuent des réservations, et comment les réservations et les sièges sont liés aux événements.

Note sur les Microservices : Dans une architecture microservices, ces classes ne seraient pas toutes dans le même "pot". L'`EventService` se concentrerait sur `Event` (et `Seat`), le `BookingService` sur `Booking` (et aurait besoin d'informations sur la disponibilité des `Event`), et le `UserService` sur `User`.



3. Diagramme de Séquence

Objectif : Illustrer les interactions entre les différents composants (acteurs, services, base de données) au fil du temps pour réaliser un cas d'utilisation spécifique. C'est une vue dynamique.

Scénario Décrit : Le processus de réservation de places par un utilisateur.

Flux d'Interactions :

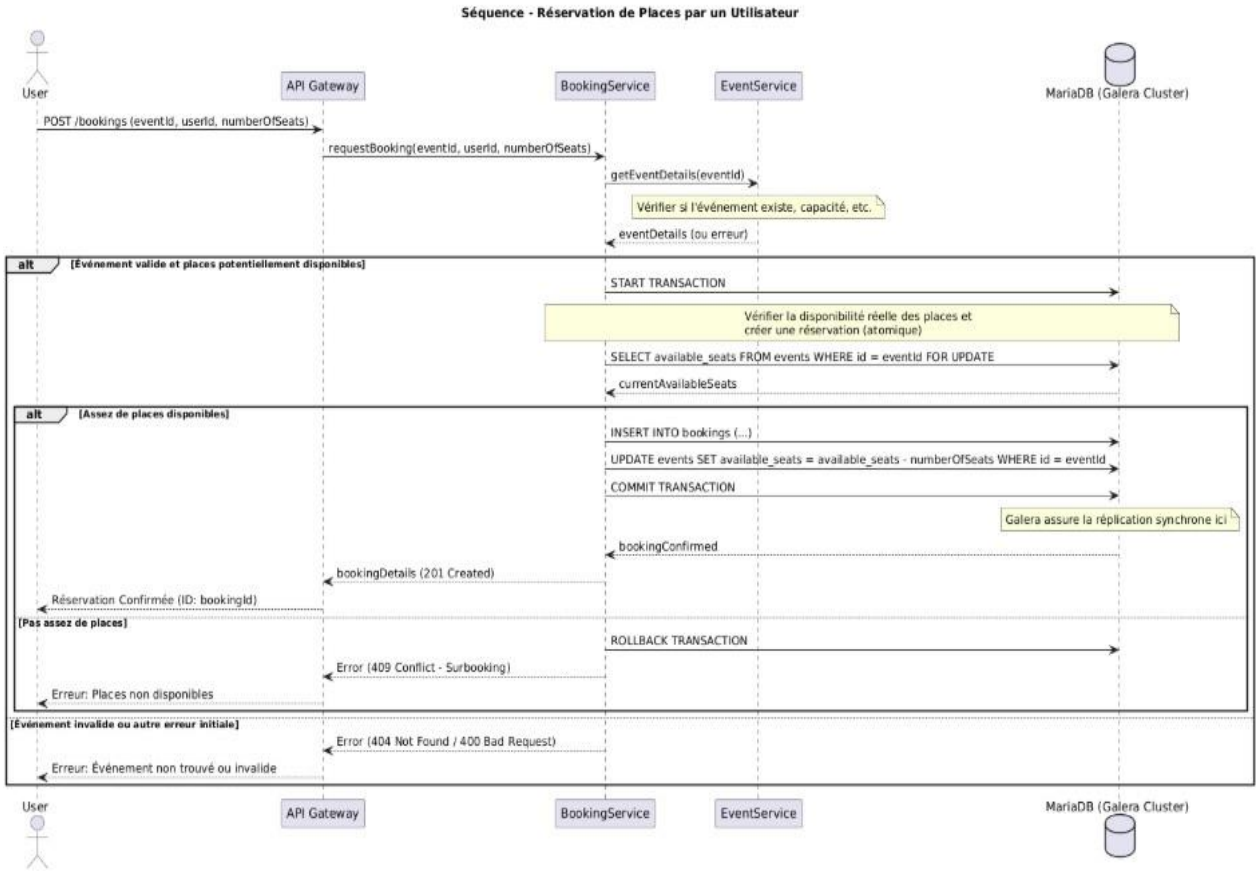
1. L'User initie une requête de réservation via l'API Gateway.
2. L'API Gateway route la requête vers le BookingService.
3. Le BookingService vérifie d'abord les détails de l'événement (existence, validité) auprès de l'EventService.

4. Si l'événement est valide, le BookingService interagit avec le MariaDB (Galera Cluster) pour :

- Démarrer une transaction.
- Vérifier la disponibilité réelle des places (potentiellement en verrouillant la ligne de l'événement pour éviter les conditions de concurrence – SELECT FOR UPDATE).
- Si des places sont disponibles : insérer la réservation et décrémenter le nombre de places disponibles pour l'événement.
- Valider (COMMIT) la transaction. Galera s'assure que cette écriture est répliquée de manière synchrone sur tous les nœuds du cluster avant de retourner le succès.
- Si pas assez de places : annuler (ROLLBACK) la transaction.

5. La réponse (succès ou échec) remonte ensuite à l'utilisateur via l'API Gateway.

Importance de Galera : La réplication synchrone de Galera est cruciale à l'étape de COMMIT pour garantir que l'état des available_seats est cohérent sur tout le cluster, évitant ainsi les surréservations.



2.4. Compromis et Limites

* **Complexité Opérationnelle Accrue :** Comparé à un monolithe, déployer et gérer plusieurs services est plus complexe (même pour une démo). Nous utiliserons Docker et Docker Compose pour simplifier.
* **Latence Réseau :** Les appels inter-services introduisent une latence.
* **Cohérence des Données Distribuées :** Bien que chaque service puisse avoir sa propre base de données (ou schéma), ici nous utilisons un cluster MariaDB partagé pour simplifier et mettre l'accent sur Galera. La gestion des transactions distribuées (saga pattern, etc.) est hors de portée pour cette démo simple, mais le besoin de cohérence forte pour les réservations est géré par Galera

au niveau de la base de données pour le Booking Service.

* **Simplicité de la Démo :** De nombreux aspects d'une architecture microservices robuste (service discovery, circuit breakers, logging centralisé avancé, tracing distribué) seront mentionnés mais implémentés de manière minimale ou pas du tout pour se concentrer sur les objectifs pédagogiques clés.

3. Application des Principes Pédagogiques

3.1. KISS (Keep It Simple, Stupid)

* **Fonctionnalités limitées :** Se concentrer sur le flux principal de création d'événement et de réservation de places. Pas de paiement, pas de gestion de compte utilisateur complexe, pas de notifications avancées.
* **Code concis et clair :** Privilégier la lisibilité et la simplicité dans chaque microservice.
* **Nomenclature claire :** Utiliser des noms de variables, fonctions, classes et services explicites.
* **Complexité cyclomatique raisonnable :** Éviter les fonctions et méthodes trop longues ou avec trop de branches conditionnelles.
* **Simplification des dépendances :** Utiliser un minimum de bibliothèques externes.

3.2. DDD (Domain-Driven Design)

Nous appliquerons les principes DDD principalement au sein du **booking Service** et de l'**Event Service**.

3.2.1. Bounded Contexts envisagés

Même avec des microservices, chaque service peut être vu comme opérant dans son propre Bounded Context (BC) ou contribuant à un BC

plus large.

* **BC Gestion des Événements :** Géré par l'Event Service. Concerne la définition, la programmation et les détails des événements.
* **BC Gestion des Réservations :** Géré par le Booking Service. Concerne le processus de réservation, la disponibilité des places, la confirmation.

3.2.2. Agrégats et Entités clés

* Dans le BC Gestion des Événements (Event Service):
    * `Event` (Agrégat) :
        * `EventId` (Identifiant unique)
        * `Name`
        * `DateTime`
        * `Location`
        * `TotalCapacity`
        * `SeatMap` (Simplifié, ex: juste un nombre de places, ou une liste de `Seat` si on veut aller plus loin)

* Dans le BC Gestion des Réservations (Booking Service):
    * `Booking` (Agrégat) :
        * `BookingId` (Identifiant unique)
        * `EventId` (Référence à l'événement)
        * `UserId` (Référence à l'utilisateur)
        * `ReservedSeats` (Liste de `SeatId` ou nombre de places)
        * `BookingStatus` (ex: PENDING, CONFIRMED, CANCELLED)

    * `EventAvailability` (Potentiel Agrégat ou Entité dans Booking Service) :
        * `EventId`
        * `AvailableSeats` (Nombre, ou liste de `SeatId` disponibles)
        * Ce pourrait être une vue matérialisée des informations de l'Event Service, mise à jour pour refléter les réservations.

3.2.3. Événements de Domaine potentiels

* `EventCreated` (publié par Event Service)
* `EventCapacityUpdated` (publié par Event Service)
* `BookingRequested` (interne au Booking Service)
* `SeatsReserved` (publié par Booking Service)
* `BookingConfirmed` (publié par Booking Service)
* `BookingFailedDueToNoAvailability` (publié par Booking Service)

Pour la démo, la publication et la consommation explicite d'événements de domaine via un bus de messages pourraient être simplifiées en appels directs ou en logique interne au service, mais leur identification reste importante.

3.2.4. Vocabulaire Ubiquitaire (Exemples)

* Événement (Event)
* Place (Seat)
* Capacité (Capacity)
* Réservation (Booking, Reservation)
* Disponible (Available)
* Confirmé (Confirmed)

Ce vocabulaire sera utilisé dans le code, les discussions, la base de données et l'API.

3.3. TDD (Test-Driven Development)

* **Tests Unitaires :** Pour chaque microservice, tester la logique métier des classes et fonctions. Exemples :
    * `EventService`: Valider la création d'un événement avec des données correctes/incorrectes.
    * `BookingService`:
        * Un utilisateur peut-il réserver des places si elles sont

disponibles ?
    *   Une réservation est-elle refusée si aucune place n'est disponible ?
    *   Le nombre de places disponibles est-il correctement mis à jour après une réservation ?

*   **Tests d'Intégration :**
    *   Tester l'interaction de chaque service avec la base de données (MariaDB).
    *   Tester les interactions entre services (ex: Booking Service appelant Event Service, si applicable). Pour la démo, cela pourrait être des tests au niveau de l'API Gateway.

*   **Tests d'Acceptation (ou E2E simplifiés) :**
    *   Simuler un flux utilisateur complet :
        1. Créer un événement.
        2. Lister les événements.
        3. Réserver une place pour cet événement.
        4. Vérifier que la réservation est confirmée et que la disponibilité a changé.
    *   Ces tests se feront via des appels à l'API Gateway.

3.4. SOLID

Nous nous efforcerons de montrer comment ces principes sont appliqués :
*   **S (Single Responsibility Principle) :**
    *   Chaque microservice a une responsabilité principale (gestion des événements, gestion des réservations).
    *   Au sein des services, les classes auront également des responsabilités uniques (ex: `EventRepository` pour la persistance des événements, `BookingPolicy` pour les règles de réservation).
*   **O (Open/Closed Principle) :**

    *   Concevoir les services et classes pour qu'ils soient extensibles (ex: ajouter de nouveaux types de stratégies de tarification sans modifier le code de réservation existant) mais fermés à la modification. Ceci sera plus conceptuel pour la démo.
    *   Ex: Utilisation d'interfaces pour les services de règles métier, permettant d'injecter différentes implémentations.
*   **L (Liskov Substitution Principle) :**
    *   Si nous utilisons de l'héritage (ex: différents types d'événements ou de stratégies de réservation), les sous-types devront être substituables à leurs types de base.
*   **I (Interface Segregation Principle) :**
    *   Définir des interfaces granulaires pour les services. Les clients ne devraient pas dépendre d'interfaces qu'ils n'utilisent pas.
    *   Ex: L'API Gateway n'exposera que les endpoints nécessaires pour les clients, et chaque service n'exposera que les opérations relatives à sa responsabilité.
*   **D (Dependency Inversion Principle) :**
    *   Les modules de haut niveau (logique métier) ne dépendront pas des modules de bas niveau (accès aux données), mais des deux dépendront d'abstractions (interfaces).
    *   Ex: Le `BookingService` dépendra d'une interface `IBookingRepository` plutôt que d'une implémentation concrète de l'accès à MariaDB. L'implémentation sera injectée (Injection de Dépendances).

4. Cluster MariaDB (Galera)

4.1. Justification de l'utilisation de Galera Cluster

Tel que mentionné en 1.3, Galera est choisi pour :

*   Haute Disponibilité : Essentiel pour un service de réservation

qui doit être accessible en permanence, surtout lors de pics de demande.

* Cohérence Forte (Synchronous Replication) : Crucial pour la contrainte métier de gestion en temps réel des places et prévention du surbooking. Chaque transaction est validée sur tous les nœuds avant d'être confirmée au client.
* Prévention des Conflits : Galera gère la certification des transactions pour éviter les conflits d'écriture.
* Scalabilité en Lecture : Les requêtes de lecture peuvent être distribuées sur les nœuds du cluster.

4.2. Plan de Mise en Place et Tests de Haute Disponibilité

* **Mise en Place :**
    1. Utiliser Docker et Docker Compose pour configurer un cluster MariaDB Galera avec au moins 3 nœuds.
    2. Configurer les microservices pour se connecter au cluster (via un load balancer ou en listant les nœuds, selon la simplicité recherchée pour la démo). `HAProxy` pourrait être un bon choix simple pour répartir la charge devant le cluster Galera.
* **Tests de Haute Disponibilité (Failover) :**
    1. **Scénario 1 : Arrêt d'un nœud DB**
        * Lancer l'application, effectuer des réservations.
        * Arrêter (simuler une panne) l'un des nœuds MariaDB du cluster.
        * Vérifier que l'application continue de fonctionner (les nouvelles réservations sont possibles, la lecture des données est correcte).
        * Observer les logs du cluster pour voir la réélection/reconfiguration.
    2. **Scénario 2 : Réintégration d'un nœud DB**
        * Redémarrer le nœud arrêté.
        * Observer sa resynchronisation automatique avec le

cluster (State Transfer).

3. **Scénario 3 (Optionnel) : Coupure réseau partielle**

* Simuler une partition réseau où un nœud est isolé. Observer comment le cluster gère le quorum pour éviter le "split-brain". (Peut être plus complexe à démontrer simplement).

5. Architectural Decision Records (ADR)

Nous utiliserons un format simple pour les ADRs.

* ADR-001 : Choix de l'Architecture Microservices**

* **Contexte**: Nécessité de modularité, scalabilité indépendante pour la gestion des réservations, et alignement avec des pratiques modernes pour un projet démonstratif.

* **Décision**: Adopter une architecture microservices.

* **Conséquences**: Complexité opérationnelle accrue (mitigée par Docker), besoin de communication inter-services, mais meilleure isolation des fautes et scalabilité ciblée. Alternative (Monolithe Modulaire) considérée mais rejetée pour moins bien démontrer la scalabilité sous contrainte.

* ADR-002 : Choix de MariaDB Galera Cluster pour la Persistance**

* **Contexte**: Exigence du TP et besoin de haute disponibilité et de cohérence forte pour la réservation de places.

* **Décision**: Utiliser MariaDB avec Galera Cluster.

* **Conséquences**: Configuration plus complexe qu'une base de données unique, mais répond aux besoins de résilience et de cohérence. Performance en écriture peut être impactée par la réplication synchrone, mais acceptable pour le volume attendu.

6. Documentation des Doutes et Renoncements

* **Doute/Renoncement 1 :** Bus de Messages Asynchrone

    *   Option envisagée : Utiliser un bus de messages (RabbitMQ/Kafka) pour la communication inter-services (ex: `EventCreated` -> Booking Service).
    *   Renoncement (pour la v1 de la démo) : Afin de respecter KISS et de se concentrer sur les objectifs principaux, la communication inter-services sera initialement synchrone (REST). Un bus de messages ajoute une dépendance et une complexité supplémentaires.
    *   Justification : Les bénéfices (découplage, résilience accrue aux pannes temporaires) sont importants mais peuvent être démontrés ultérieurement ou discutés comme une amélioration.

*   **Doute/Renoncement 2 :** Gestion des Utilisateurs Complète**
    *   Option envisagée : Un User Service complet avec authentification, rôles, etc.
    *   Renoncement : Pour la démo, le User Service sera minimaliste ou simulé (simple `userId`).
    *   Justification : La gestion complète des utilisateurs est un domaine complexe qui détournerait l'attention des objectifs principaux d'architecture applicative et de cluster DB.

7. Plan de Démonstration et Livrables Attendus

Livrables:
1. **Dépôt Git public (ou accès privé)** : Code source complet des microservices, fichiers Dockerfile, fichier docker-compose.yml, scripts de test, et ce document de conception (ou sa version finale).
2. **Rapport Technique (PDF)** : Ce document, mis à jour et complété, incluant :
    *   Instructions pour lancer l'application (build, run avec Docker Compose).
    *   Instructions pour exécuter les tests (unitaires,

intégration, acceptation).
    *   Instructions pour "casser" le système (tests de failover DB).
    *   Explication des migrations de base de données (si des évolutions de schéma sont nécessaires et gérées, par ex. avec Flyway/Liquibase, bien que pour une démo simple cela puisse être manuel).
3. **Diaporama / Démo** :
    *   **Storytelling et Contexte** : Présentation du domaine, de la contrainte métier.
    *   **Pourquoi cette architecture ?** : Justification des microservices et de Galera. Présentation du diagramme.
    *   **Matérialisation des Principes** :
        *   Montrer concrètement dans le code des exemples d'application de KISS, DDD (Bounded Contexts, agrégats dans le code), SOLID (exemples spécifiques de chaque principe), TDD (montrer des tests et leur exécution).
    *   **Démonstration Live du Failover des Bases de Données** :
        1. Lancer l'application et le cluster DB.
        2. Effectuer quelques opérations (ex: réservations).
        3. Arrêter un nœud MariaDB.
        4. Montrer que l'application continue de fonctionner.
        5. Redémarrer le nœud et montrer sa resynchronisation.
    *   **Conclusion et Limites**.

Prochaines étapes pour la création de l'application (nous allons y aller pas à pas) :

Mise en place de l'environnement de développement :

Choix du langage/framework pour les microservices (ex: Java/Spring Boot, Python/Flask, Node.js/Express). *Je suggère quelque chose avec lequel vous êtes à l'aise et qui se prête bien aux microservices et à REST. Python/Flask ou Node.js/Express sont rapides à prototyper. Java/Spring Boot est plus robuste mais un peu plus verbeux pour une démo simple.*

Installation de Docker et Docker Compose.

Configuration initiale du cluster MariaDB Galera avec Docker Compose.

Développement du premier microservice (par exemple, Event Service) :

Définition de l'API REST (endpoints).

Implémentation de la logique métier simple.

Connexion à la base de données.

Dockerfile pour le service.

Ajout au Docker Compose.

Développement des autres services (Booking Service, puis l'API Gateway).

Implémentation des tests (TDD approche).

Quel langage/framework préférez-vous utiliser pour les microservices ? Cela guidera les exemples de code. Pour une démo simple, Python avec Flask ou FastAPI, ou Node.js avec Express, sont souvent de bons choix pour leur rapidité de développement.