**Pourquoi j’ai choisi une architecture monolithique modulaire pour mon application ?**

**3 raisons principales : simplicité, clarté, et capacité d’évolution**

**1. Simplicité de mise en place**

Application pas très complexe fonctionnellement : une architecture distribuée aurait ajouté une complexité technique inutile.

Avec une architecture monolithique, je peux :

• Avancer plus vite dans le développement,

• Me concentrer sur la logique métier sans avoir à gérer les échanges entre services,

• Et éviter les surcoûts liés à la configuration d’un environnement distribué (comme les microservices : securité et résilience plus compliqué à gérer, communication entre les services, implication d'un service d'orchestraction).

**2. Modularité = organisation + maintenabilité**

Je structure mon monolithe en modules métier bien séparés (utilisateur, événement, réservation).

Cela me permet de :

• Travailler de façon plus claire, avec un code bien organisé

• Appliquer plus facilement les concepts de DDD (bounded contexts)

**3. Pertinence pédagogique**

Cette architecture me permet de respecter les exigences pédagogiques du projet :

• En appliquant les principes SOLID dans chaque module,

• En gardant une structure KISS, simple à lire et à maintenir,

• Et en intégrant progressivement une démarche TDD avec des tests unitaires ciblés sur les modules.

**Présentation des Use Cases principaux de l’application de réservation**

1. **Réserver une place à un événement**
   * **Acteur principal : Utilisateur (Client)**
   * **Description : Le client consulte la liste des événements, choisit un événement, sélectionne le nombre de places, et valide sa réservation. L’application vérifie la disponibilité en temps réel et confirme la réservation si les places sont suffisantes.**
   * **Objectif métier : Assurer une réservation fiable et rapide, avec gestion dynamique du taux de remplissage.**
2. **Consulter les détails d’un événement**
   * **Acteur principal : Utilisateur (Client)**
   * **Description : Permet à l’utilisateur de voir les informations détaillées d’un événement (date, lieu, description, nombre de places restantes, prix).**
   * **Objectif métier : Fournir une information claire et à jour pour aider à la décision d’achat.**
3. **Gérer un événement (création / modification / suppression) côté admin**
   * **Acteur principal : Administrateur (Organisation)**
   * **Description : L’administrateur peut créer de nouveaux événements, modifier leurs informations ou annuler un événement.**
   * **Objectif métier : Assurer que l’offre événementielle est à jour et conforme aux attentes.**
4. **Annuler une réservation**
   * **Acteur principal : Utilisateur (Client)**
   * **Description : Le client peut annuler sa réservation avant une certaine date limite.**
   * **Objectif métier : Offrir de la flexibilité au client et libérer des places pour d’autres clients.**

**Principes architecturaux dans l'application de réservation d'événements**

**KISS (Keep It Simple, Stupid)**

Le projet respecte le principe KISS grâce à une structure claire et intuitive. Chaque composant a une responsabilité unique et bien définie. Par exemple, les services API sont séparés par domaine (events, auth, bookings) avec des méthodes explicites. L'architecture backend/frontend distincte simplifie la maintenance et l'évolution du code.

**DDD (Domain-Driven Design)**

L'application est structurée autour des domaines métier (événements, réservations, utilisateurs). Les modèles reflètent les entités du domaine avec leurs relations (User, Event, Booking) et encapsulent la logique métier. Les contrôleurs organisent les opérations par domaine fonctionnel, et les routes API expriment clairement l'intention des endpoints (/events, /bookings, /auth).

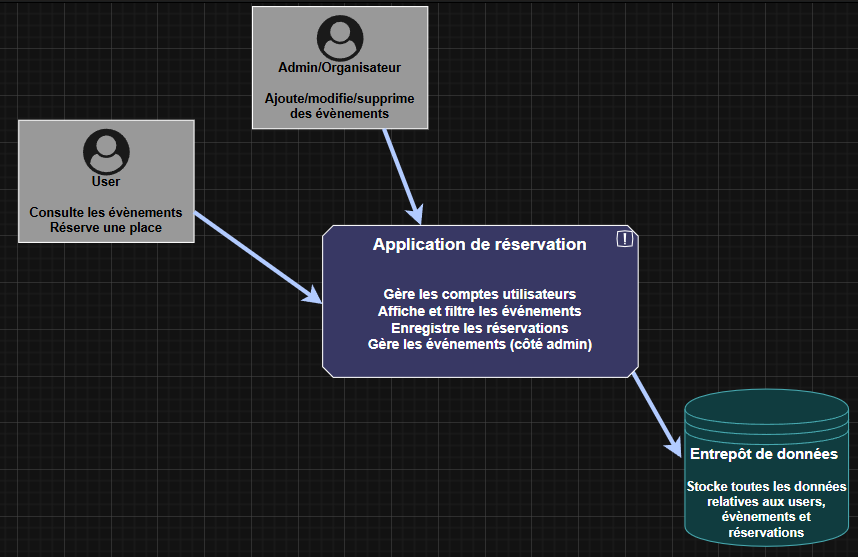
**SOLID**

* **Single Responsibility**: Chaque composant a une responsabilité unique (AuthContext gère l'authentification, EventList affiche les événements).
* **Open/Closed**: L'architecture permet d'étendre les fonctionnalités sans modifier le code existant (ajout facile de nouvelles routes).
* **Liskov Substitution**: Les interfaces sont cohérentes (même structure pour tous les contrôleurs API).
* **Interface Segregation**: Les interfaces sont spécifiques et focalisées (séparation claire entre les services d'API).
* **Dependency Inversion**: Les dépendances sont injectées plutôt que créées en interne (comme le contexte d'authentification).

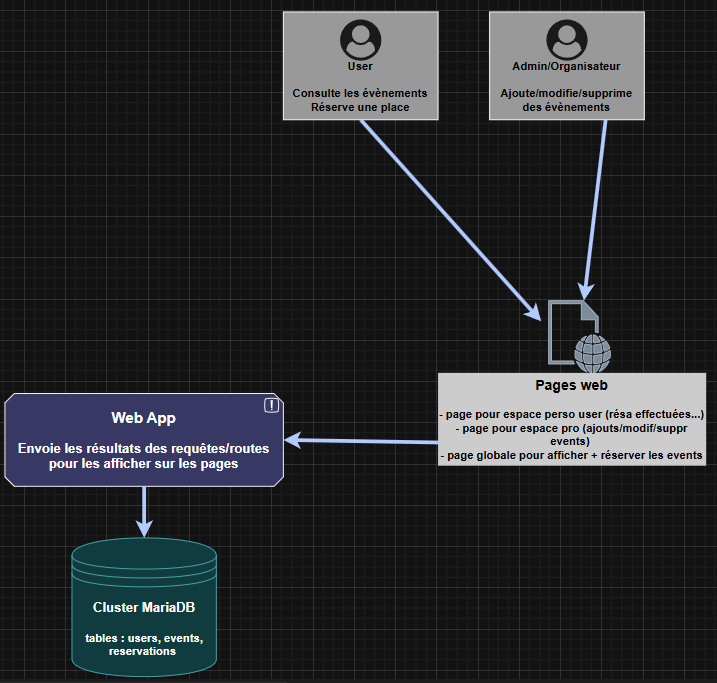
**Représentations graphiques de l’architecture et des interactions**

Afin de mieux illustrer la conception de l’application, plusieurs diagrammes ont été réalisés. Ils permettent de visualiser à la fois l’architecture globale du système, la structure de ses entités principales ainsi que les interactions entre les composants ou les acteurs du système. Ces représentations facilitent la compréhension des choix techniques et fonctionnels effectués tout au long du projet.

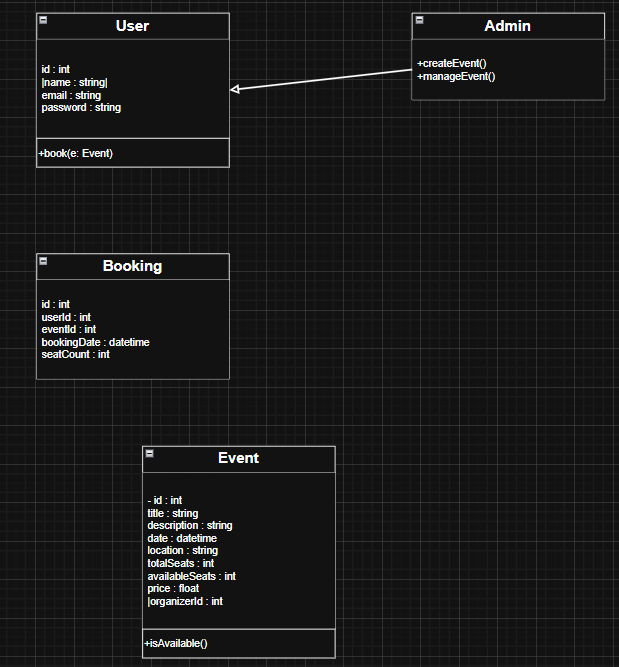
**C4 niveau 1**



**C4 niveau 2**



**Diagramme de classe**



**Diagramme de séquence**

