Page 1 ：  
Titre « Architecture Polyglotte : mix de MongoDB, PostgreSQL et Neo4j »

Sous titre« trip management »

Yang YANG – Binh Minh NGUYEN

# Page 2 ：

Stack de Bases de Données Polyglotte

**1. PostgreSQL (relationnelle)**

* Version : PostgreSQL 17.5
* Gestion via pgAdmin 4 (interface graphique)

**2. MongoDB (base de données orientée document)**

* Client : mongosh 2.5.1
* Serveur : mongod v8.0.9
* Outils : MongoDB Compass pour la gestion locale + MongoDB Atlas pour le cloud

**3. Neo4j (base de données orientée graphe)**

* Version : Neo4j Enterprise 5.24.0
* Gestion via Neo4j Desktop

Page 3 :   
A diagram of a computer

AI-generated content may be incorrect.  
Le **micro-service info-api**, basé sur **MongoDB**, gère les données CRUD et la recherche sur les ressources de contenu : villes, hébergements, points d’intérêt et activités. Grâce à son modèle documentaire, il convient parfaitement aux données hiérarchiques ou semi-structurées.

Le **trip‑api**, quant à lui, utilise **PostgreSQL** pour créer et gérer les itinéraires jour par jour. Les itinéraires imposent des contraintes relationnelles fortes et nécessitent des transactions ACID, ce que PostgreSQL assure parfaitement.

Enfin, le **route-api** exploite **Neo4j** pour modéliser le réseau routier en graphe : les villes et points d’intérêt comme nœuds, les trajets comme arêtes. Neo4j nous permet de calculer les distances, durées et itinéraires optimisés via des algorithmes de graphe.

Ces trois services communiquent entre eux par l’échange d’identifiants : par exemple, trip-api stocke seulement les ID d’un hébergement ou d’une activité, puis interroge info-api pour obtenir les détails à afficher. Cela garantit une séparation claire des responsabilités, mais nécessite de gérer la cohérence inter-services et la latence induite par les appels réseau.

# Page 4 :

A diagram of a computer

AI-generated content may be incorrect.  
**MongoDB : structure et gestion des relations**

* Dans notre base MongoDB, nous avons quatre collections principales : **Accommodation**, **POI**, **Activities**, et **City**.
* Contrairement aux bases SQL, MongoDB n’utilise pas de **clés étrangères** : nous établissons les liens avec des **références**.  
  Par exemple, **POI** et **Accommodation** contiennent l’**\_id** d’un **City**, et **Activities** contient l’**\_id** d’un **POI**.

**Amélioration des performances via l’embedding partiel**

Pour les données fréquemment lues mais rarement mises à jour, nous utilisons un **embedding** partiel :

1. Dans **City**, on **embed** { id, name } des POIs et { id, name, price, available } des Accommodations.
2. Dans **Activity**, on **embed** le pointOfInterest.name.
3. Dans **POI** et **Accommodation**, on **embed** le city.name.

**Consistance des données et problème**

* Inconvénient : si une information est modifiée dans une collection (par exemple, le nom d’un POI), les champs **embed** dans d’autres collections risquent de devenir **obsolètes**, causant **des incohérences**.

# Page 5 :

**Solution : triggers Atlas Cloud**

Pour résoudre ce problème, nous utilisons les **triggers** de MongoDB Atlas, qui assurent la **synchronisation automatique** entre les collections lorsqu’on **insère**, **met à jour** ou **supprime** des documents.

**Nos 6 triggers**

*(Vous pouvez ici introduire la liste ou le schéma de vos six triggers, en précisant pour chaque trigger : l’événement déclencheur (insert/update/delete), les collections concernées, et l’action effectuée — mise à jour de l’embed lié.)*

A screenshot of a computer

AI-generated content may be incorrect.

💡 **En résumé** :

1. On utilise des **références** pour modéliser les relations et On améliore les performances avec un **embedding partiel** et On maintient la **consistance des données** grâce à des **triggers Atlas Cloud**.

# Page 6 :

Création de collections avec Validators et gestion de l’intégrité

| **Collection** | **But du Validator** | **Champs clés et contraintes** |
| --- | --- | --- |
| **city** | Assurer que chaque ville inclut des POI et hébergements dénormalisés. | \_id (objectId) · name (string) · photos (tableau d’URL HTTP/HTTPS) · geoInfo (lat∈[-90,90], lon∈[-180,180]) · accommodations (liste d’objets avec id, nom, prix ≥0, dispo booléen) · pointsOfInterest (liste d’objets avec id et nom) |
| **pointOfInterest** | Vérifier les POI détaillés, avec infos géographiques et référence à la ville. | \_id, name, photos (URL) · address (string) · geoInfo (lat/lon valides) · city (objet avec id et nom) |
| **activity** | Garantir que chaque activité est liée à un POI, avec saisons et tarification. | \_id, name · pointOfInterest (id + nom) · photos (URL) · seasons (un ou plusieurs mois parmi 12 valeurs autorisées) · price (objet { adult ≥ 0, child ≥ 0 }) |

Validators à la création de la collection – > Usage des triggers pour la cohérence des données

# Page 7 :

A diagram of a function

AI-generated content may be incorrect.

**Image d’une base de données relationnelle traditionnelle :**

1. Nous utilisons des **triggers** pour garantir la cohérence des données entre plusieurs tables après mise à jour. Nous avons également mis en place des opérations en cascade (cascade delete/update) pour assurer l’intégrité des données.
2. Nous appliquons des **contraintes CHECK** et **UNIQUE** pour garantir la validité des données — par exemple, s’assurer que la **date de fin** n’est pas antérieure à la **date de début**.
3. Cependant, pour des validations de formats de données plus complexes, les **constraints SQL** ne sont pas aussi puissantes ou simples à écrire que les **validators MongoDB**. Nous avons donc choisi d’effectuer ces vérifications — et de renvoyer des erreurs — au niveau de l’application, c’est-à-dire dans **Spring Boot**.

# Page 8 ：

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* Chaque ville est un nœud **City**, relié par une relation unidirectionnelle **LOCATED\_AT** dans la base de données.
* Lors des requêtes, nous traitons ces relations comme **bidirectionnelles** pour naviguer librement entre les villes.
* Nous avons mis en place un **constraint UNIQUE** pour garantir que chaque cityId reste cohérent et correspond au même CityId dans MongoDB.
* Nous exploitons le **type géospatial** de Neo4j (point) et la fonction point.distance pour :
  1. **Calculer en temps réel** la distance à vol d’oiseau entre deux villes,
  2. Estimer automatiquement le **temps de trajet**, sans nécessiter une saisie manuelle.
* Avec la fonction shortestPath(), nous trouvons le **chemin minimal** (en nombre de sauts) dans le graphe **LOCATED\_AT**, ce qui nous permet de déterminer le **plus court itinéraire** entre villes.

A computer screen shot of a black screen

AI-generated content may be incorrect.

# Page 9 : conclusion