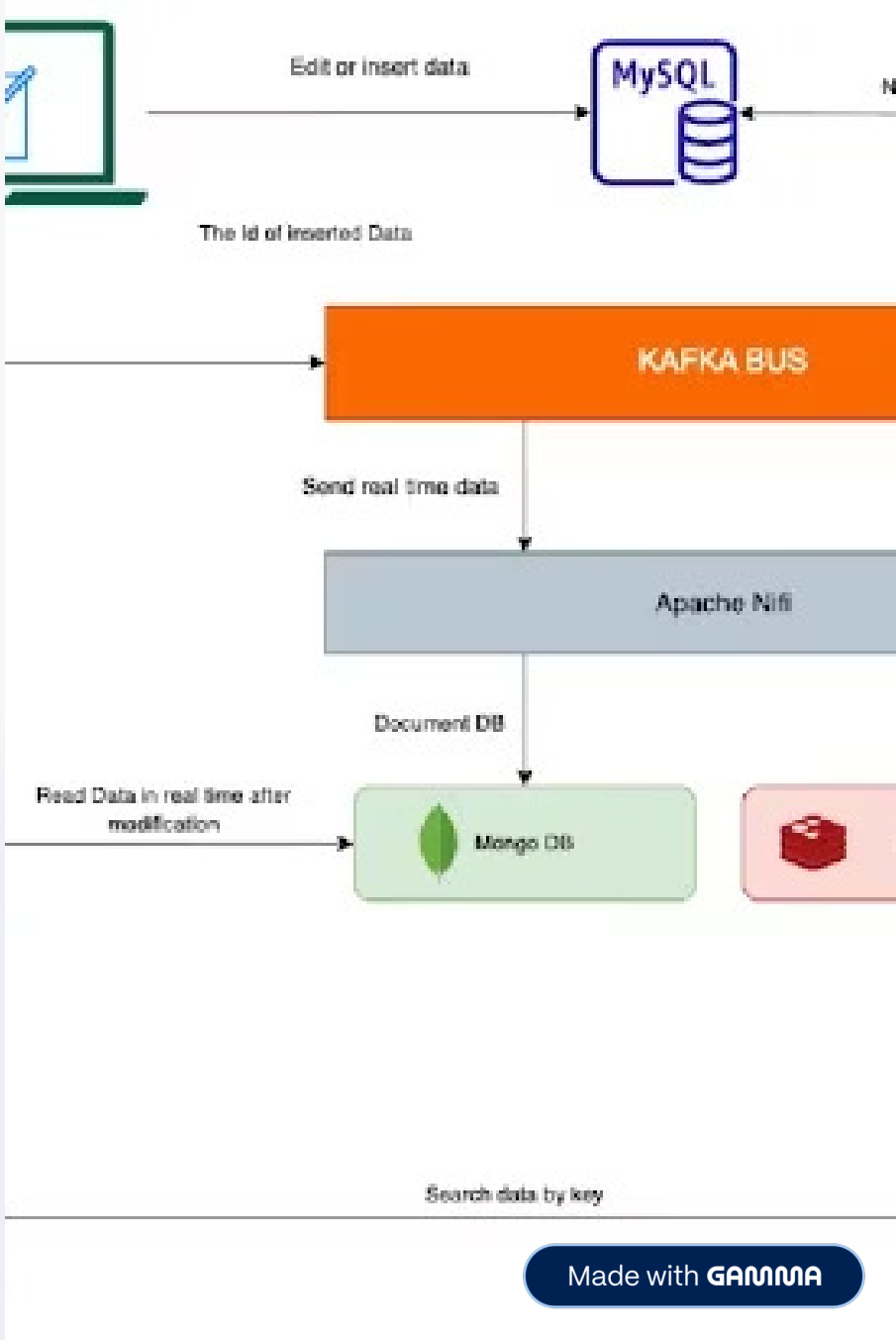
Système de Cache Distribué en Temps Réel

Orchestration de données événementielle via Kafka, NiFi, Redis et MongoDB.

Présenté par : G. Data Master

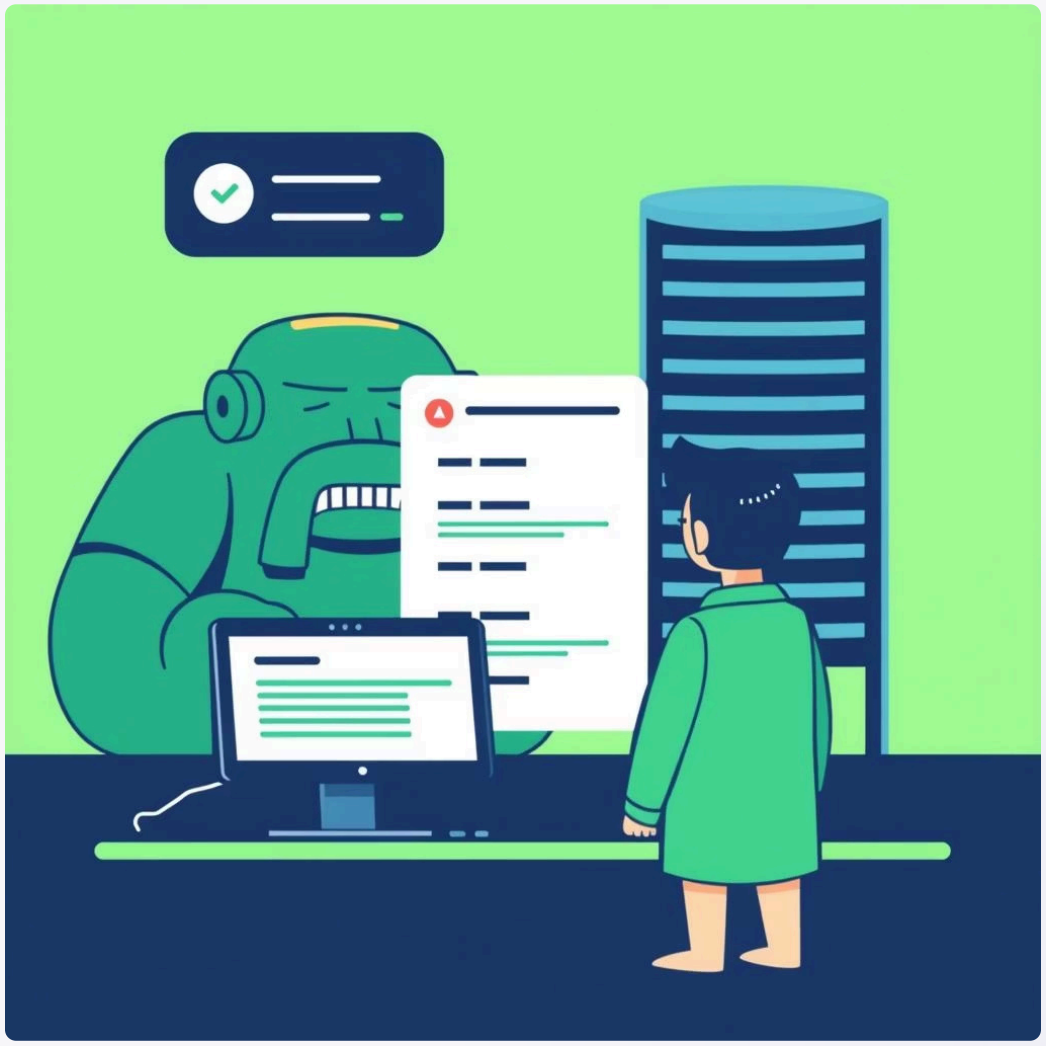
- **Brahim DARGUI** • **Mourad Benamre**
- **Mohamed Machlou** • **Mounir jaouhari**



Introduction & Problématique

Le Problème: La Lenteur des Applications Monolithiques

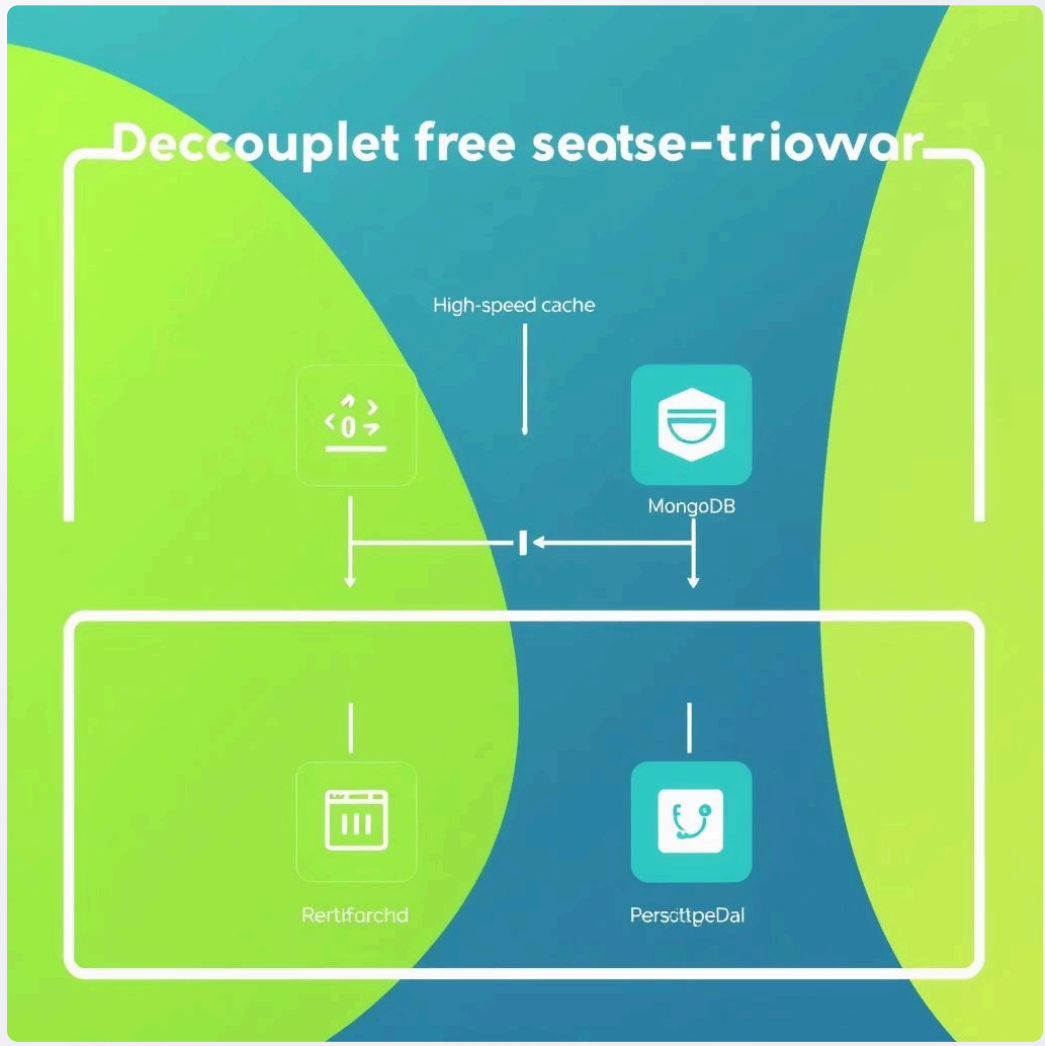
Les applications monolithiques traditionnelles rencontrent des difficultés significatives lorsqu'il s'agit d'interroger directement des bases de données volumineuses. Le temps de réponse est souvent insatisfaisant pour l'utilisateur, qui attend aujourd'hui un feedback instantané sans rechargement de page. Cette latence impacte l'expérience utilisateur et l'efficacité opérationnelle.



La Solution: Une Architecture Découplée et Réactive

Pour surmonter ces défis, nous proposons une architecture découplée et événementielle, conçue pour la vitesse et la réactivité :

- Décharger le trafic de lecture vers un cache ultra-rapide (Redis).
- Conserver un historique détaillé et persistant des données (MongoDB).
- Mettre à jour les interfaces utilisateur en temps réel (via WebSockets/Pusher) pour une expérience fluide.



Infrastructure : Le Cluster Docker

Notre système est entièrement conteneurisé, garantissant ainsi la reproductibilité, la portabilité et la scalabilité de l'environnement. L'orchestration via Docker Compose permet une gestion harmonisée de l'ensemble des services.

MySQL

Base de données relationnelle pour les données transactionnelles primaires, assurant l'intégrité et la persistance des commandes.

Zookeeper & Kafka

Un broker de messages distribué, formant l'épine dorsale de notre architecture événementielle pour une communication asynchrone.

Apache NiFi

Un outil puissant pour l'ingestion, la transformation et le routage des données, essentiel pour notre pipeline ETL.

MongoDB Replica Set (rs0)

Base de données NoSQL pour l'historisation des documents et l'activation des "Change Streams" pour la réactivité.

Redis

Un cache en mémoire ultra-rapide (In-memory data store) pour un accès instantané aux données fréquemment sollicitées.

Note : L'environnement complet, composé de plus de 10 conteneurs, a été orchestré et géré via Docker Compose, assurant une interconnexion réseau fluide entre tous les services.

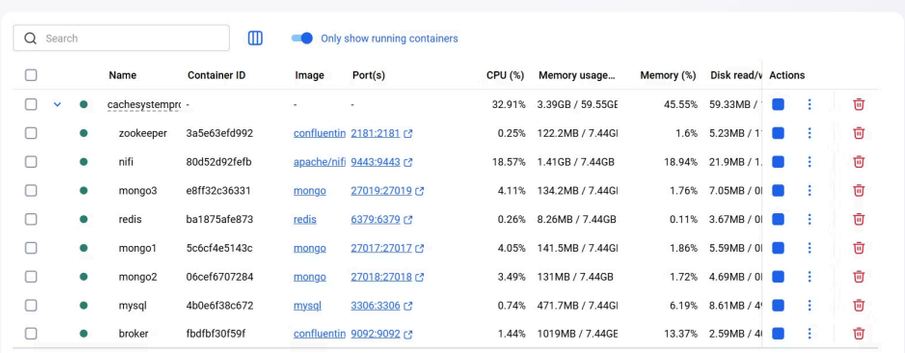
Phase 1 : Le Producteur (Laravel & Kafka)

Le Déclencheur et le Mécanisme

- **Le Déclencheur :** La création ou la modification d'une commande par un employé dans l'application Laravel.
- **Le Mécanisme :** Un Observer Laravel est configuré pour écouter spécifiquement les événements `created` sur les modèles de commande. Dès qu'un nouvel enregistrement est sauvegardé, cet observer exécute un script Python dédié.

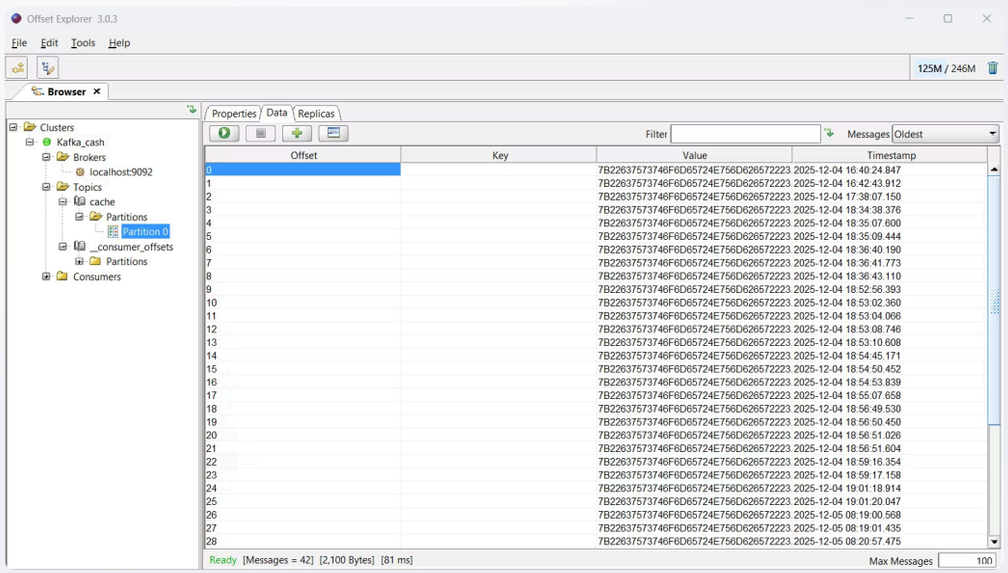
Le Transport : Kafka au Cœur de l'Événementiel

Le script Python, une fois activé, extrait les informations clés de la commande (notamment l'ID de la commande et l'ID du client). Il pousse ensuite ces données sous forme de messages dans un Topic Apache Kafka nommé `cache`. Kafka assure une transmission fiable et asynchrone des événements.



Visualisation des Messages Kafka

Les messages sont sérialisés et stockés temporairement dans le topic Kafka, prêts à être consommés par les systèmes en aval. Cette approche garantit que la création de la commande est découplée de son traitement, augmentant ainsi la réactivité de l'application Laravel.



Phase 2 : Orchestration des Données (Apache NiFi)

Apache NiFi agit comme le cerveau central de notre pipeline de données, consommant les messages de Kafka et orchestrant leur acheminement vers Redis et MongoDB avec des transformations spécifiques.

Consommation Kafka

NiFi se connecte au topic `cache` de Kafka pour extraire les messages entrants, initiant ainsi le flux de données.

Extraction et Transformation (ETL)

Utilisation de processeurs NiFi pour lire le JSON brut. Le processeur `JoltTransformJSON` est appliqué pour restructurer et optimiser les données.

Division du Flux

Le flux est ensuite divisé en deux chemins distincts : un pour l'alimentation du cache (Redis) et un autre pour l'historisation (MongoDB).

Chargement dans les Bases

Les données transformées sont chargées dans Redis pour un accès rapide et dans MongoDB pour l'archivage et l'analyse historique.

Ce flux complexe garantit que la base de données primaire n'est jamais surchargée par les lectures, tout en assurant la fraîcheur des données en cache et la persistance de l'historique.

Phase 3 : Stratégie de Transformation (JOLT)

La transformation Jolt est une étape cruciale pour optimiser la structure de nos données, en éliminant les informations superflues ou sensibles et en remodelant le JSON pour une efficacité maximale dans Redis et MongoDB.

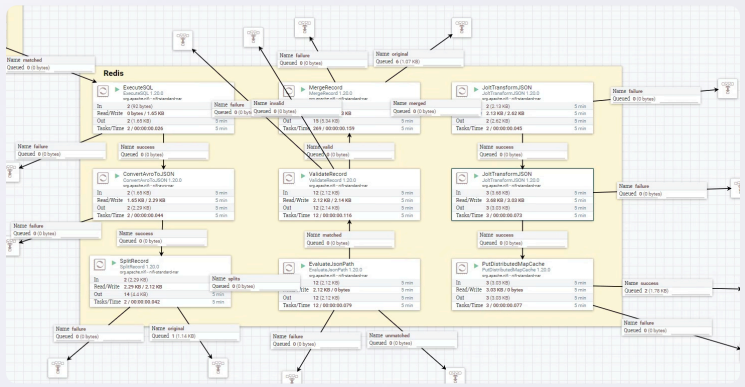
- **Le Défi : Données Brutes vs. Données Optimisées**

  Les données brutes provenant de la base de données transactionnelle contiennent souvent des champs redondants ou des informations sensibles qui ne sont pas pertinentes pour le cache ou l'historique simplifié. Le stockage de ces données non optimisées gaspille des ressources et ralentit les opérations de lecture.

- **La Solution : Transformations Jolt "Shift" et "Remove"**

  Nous utilisons la spécification Jolt pour restructurer le JSON de manière déclarative. Les transformations clés sont :

  - **"Shift" :** Permet de regrouper et de réorganiser les champs pertinents dans une nouvelle structure.
  - **"Remove" :** Permet de nettoyer le JSON en supprimant explicitement les champs non nécessaires.



- **Résultat : Objets JSON Optimisés**

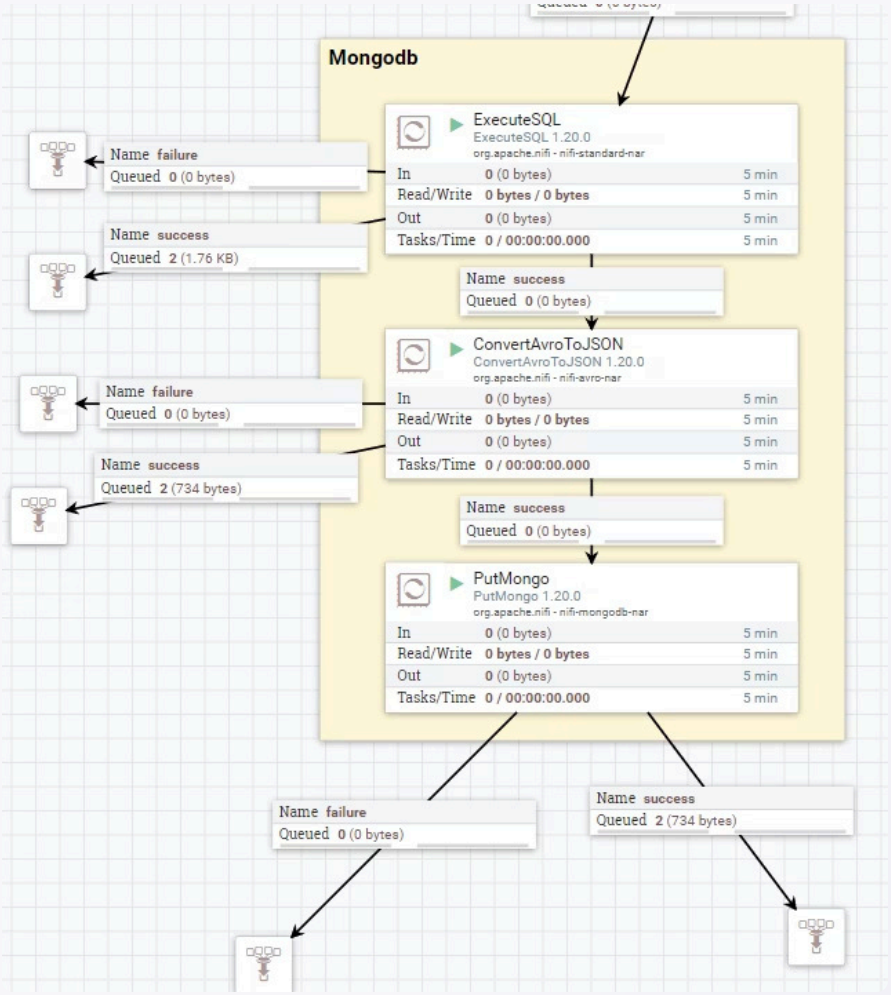
  Le processus Jolt produit des objets JSON épurés et optimisés, parfaitement adaptés pour une insertion rapide dans Redis et une consultation performante dans MongoDB. Cela garantit une lecture rapide et une empreinte mémoire réduite.

Phase 4 : Stockage Polyglotte (Redis & MongoDB)

Notre architecture tire parti de la force des bases de données polyglottes, utilisant Redis pour un cache ultra-rapide et MongoDB pour un historique riche et persistant, chacun optimisé pour son cas d'usage.

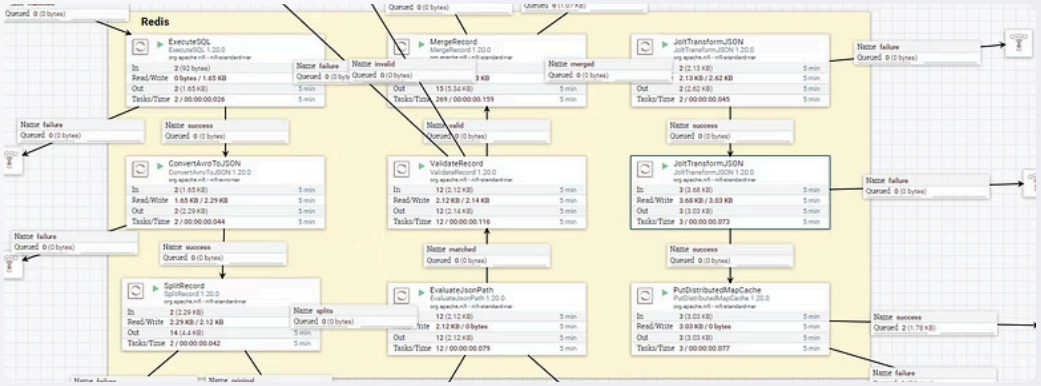
MongoDB (Replica Set)

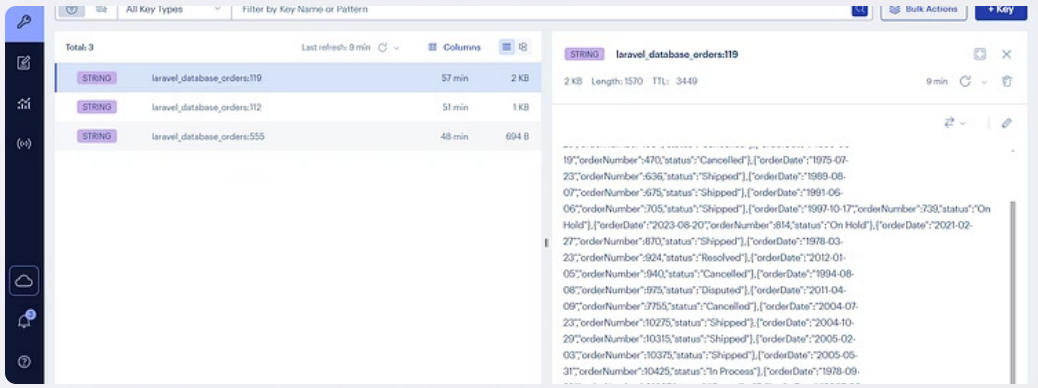
- **Rôle :** Stockage de l'historique complet des documents, offrant une flexibilité pour des requêtes complexes et l'analyse de données.
- **Configuration :** Déployé en tant que "Replica Set" (rs0). Cette configuration est cruciale car elle permet d'activer les "Change Streams", une fonctionnalité essentielle pour détecter les changements en temps réel dans la base de données et les propager au frontend.



Redis (Cache)

- **Rôle :** Stockage clé-valeur en mémoire pour un accès ultra-rapide aux données les plus fréquemment consultées. Cela décharge la base de données principale et réduit la latence.
- **Clé d'accès :** Nous utilisons le `customerNumber` comme clé unique pour chaque client, permettant une récupération directe et efficace de ses commandes.
- **Avantage Clé :** Récupération des commandes d'un client en quelques millisecondes, avec une complexité temporelle O(1), ce qui est idéal pour les applications temps réel.





Phase 5 : Diffusion Temps Réel (Le Watcher)

Pour garantir une expérience utilisateur dynamique, nous avons implémenté un mécanisme de "Watcher" basé sur les Change Streams de MongoDB, permettant des mises à jour instantanées de l'interface.

Boucle Infinie du Watcher

Une commande console Laravel est lancée en arrière-plan et s'exécute indéfiniment, surveillant activement les modifications.

MongoDB Change Streams

Le watcher utilise la fonctionnalité `$collection->watch()` de MongoDB pour s'abonner aux événements d'insertion, de mise à jour ou de suppression.

Détection et Événement Laravel

Dès qu'un nouvel insert est détecté dans MongoDB, le watcher déclenche un événement personnalisé dans Laravel, tel que `NewOrder`.

Diffusion via Pusher

Cet événement Laravel est ensuite diffusé en temps réel vers le frontend via Pusher (utilisant des WebSockets), assurant une mise à jour immédiate de l'interface.

Ce mécanisme assure que toute nouvelle commande enregistrée est immédiatement visible sur le tableau de bord du manager sans nécessiter un rafraîchissement manuel de la page.

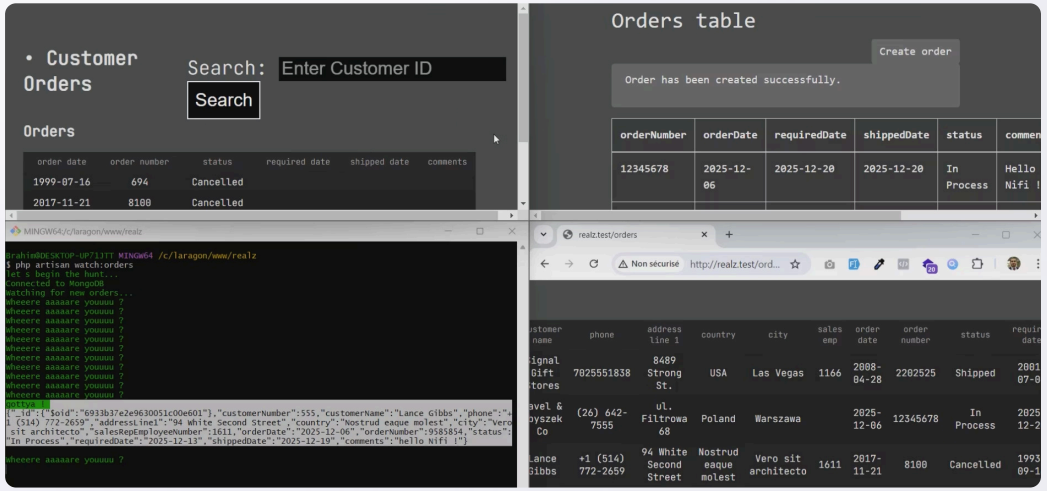
Le Résultat : Démonstration Live

La démonstration met en lumière l'efficacité et la réactivité de notre système de cache distribué en temps réel.

Workflow en Action

- **Création de Commande :** Un employé crée la commande #1005 via l'application Laravel.
- **Transit des Données :** La donnée transite de manière fluide et rapide à travers notre pipeline événementiel : Kafka → NiFi → MongoDB (pour l'historique) et Redis (pour le cache).
- **Mise à Jour Instantanée :** Le tableau de bord du manager se met à jour instantanément avec la nouvelle commande, sans aucun rafraîchissement manuel de la page. Cette réactivité est la preuve de la puissance de notre architecture temps réel.

La vidéo ci-dessus illustre en direct le fonctionnement du système, de la création de la commande à l'affichage instantané sur le tableau de bord. C'est la preuve ultime de l'efficacité de notre solution.



Défis & Acquis

Ce projet a été riche en apprentissages et a permis de surmonter des défis techniques complexes, consolidant ainsi de solides compétences en architecture de données distribuée.

Difficultés Surmontées

- **Configuration du MongoDB Replica Set (rs0) :** Un enjeu majeur a été la mise en place du Replica Set dans un environnement Docker, indispensable pour les Change Streams.
- **Intégration PHP/Laravel & Python :** Assurer la bonne communication et l'exécution des scripts Python depuis l'application Laravel, en tenant compte des spécificités des chemins sous Windows (Laragon).
- **Maîtrise des Transformations Jolt :** Développer des spécifications Jolt complexes pour gérer les tableaux imbriqués et optimiser la structure des données.

Compétences Acquises

- **Architecture Orientée Événements (EDA) :** Conception et implémentation d'un système robuste basé sur des événements, avec Kafka comme pierre angulaire.
- **Réseau Docker (Networking) :** Maîtrise avancée de la gestion réseau entre conteneurs pour une interopérabilité sans faille des services.
- **Intégration de Systèmes Hétérogènes :** Capacité à faire collaborer différentes technologies (PHP, Python, Java/NiFi) au sein d'une même architecture.