CQRS / ES

Nos heuristiques après plusieurs années de production

Aurélien BOUDOUX

boudoux.fr

aboudoux

Romain BERTHON

romaintrm.bsky.social

RomainTrm

superindep.fr

CQRS / ES : Rappel des bases

Connaissez vous le fonctionnement de la justice ?

Demandeur

Demande

Justice

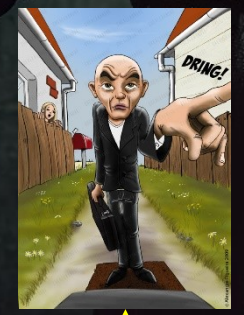
Lois

Juge

décision

Jugement(s)

Casier judiciaire



Greffe

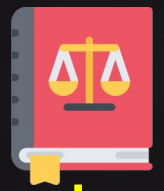
Application de la décision

Command and Query Responsibility Segregation / Event Sourcing

Utilisateur

Commande

Command Handler



Business Rules

Aggregate

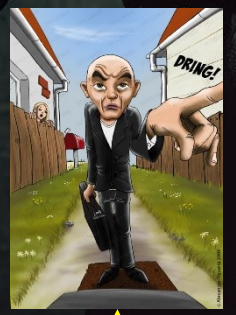
Décision

Event(s)

Stream

Event Store

Event Dispatcher



Event Handlers

Il n’est pas rare que des architectes confondent Event sourcing avec les architectures Event Driven et veulent l'utiliser pour faire des systèmes distribués, et vice-versa.

Il y a des similitudes entre l'event sourcing et l'architecture event-driven, mais aussi des différences importantes :

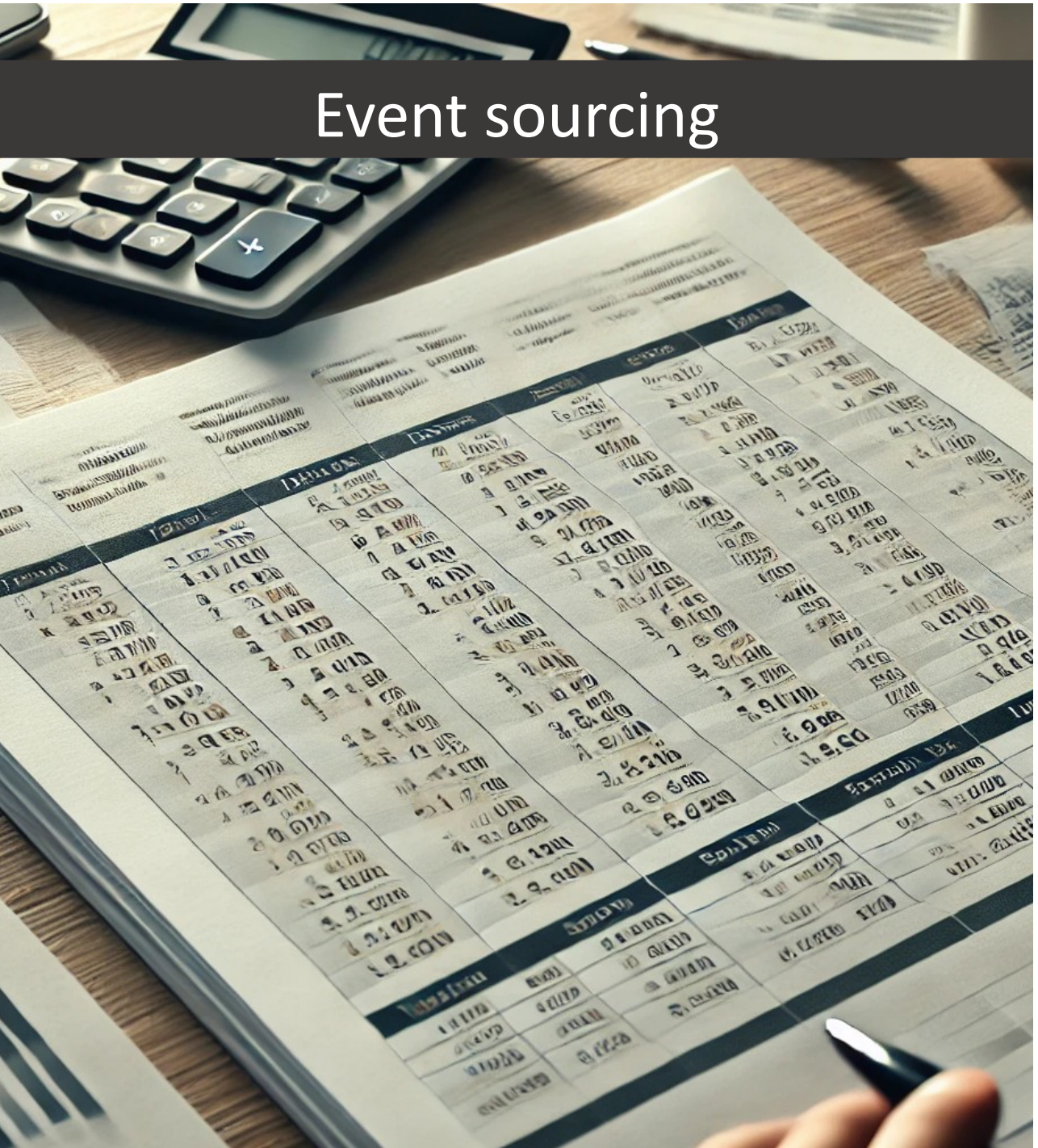
Event Driven Architecture

Modèle où les composants du système communiquent principalement via des évènements.

Les évènements déclenchent des actions dans d'autres parties du système.

Se concentre sur la communication et le découplage des composants.

Event sourcing



Approche de persistance des données où l'état du système est stocké comme une séquence d'évènements.

L'état actuel est reconstruit en rejouant les évènements.

Se concentre sur la capture et la persistance de l'historique complet des changements d'état.

Principales différences

Objectif : L'event-driven concerne la communication, l'event sourcing la persistance des données.

Portée : L'event-driven est une architecture globale, l'event sourcing une stratégie de stockage.

État : L'event-driven ne stocke pas nécessairement l'historique complet, contrairement à l'event sourcing.

Je prévois de faire du Micro-Services ou du SOA avec EventSourcing

Dans ce cas, vous ne devez surtout pas diffuser vos évènements directement sur l'ESB ou le Broker de votre système, car les évènements ES sont des détails d'implémentation alors que vous avez besoin d'un mécanisme de communication où les messages doivent être considérés comme des contrats inter services.

Je veux répartir la charge de travail sur plusieurs instances.

Dans ce cas EventSourcing est hors scope. Intéressez vous plutôt aux mécanismes de répartitions de charges traditionnels (Load balancing, Kafka, CQRS, ...)

J’ai un modulithe en EventSourcing qui doit pouvoir fonctionner en multi-sites via de la réplication

Dans ce cas vous pouvez diffuser les évènements issus d'ES car c'est bien votre persistance que vous voulez répliquer.

Pour les problèmes de conflit, vous pouvez opter pour des mécanismes d'appropriation pour vous assurer qu'un agrégat ne peut être modifié que par une seule personne et sur un seul site à la fois, soit choisir une stratégie de réconciliation basée sur les différences de stream.


Premier contact

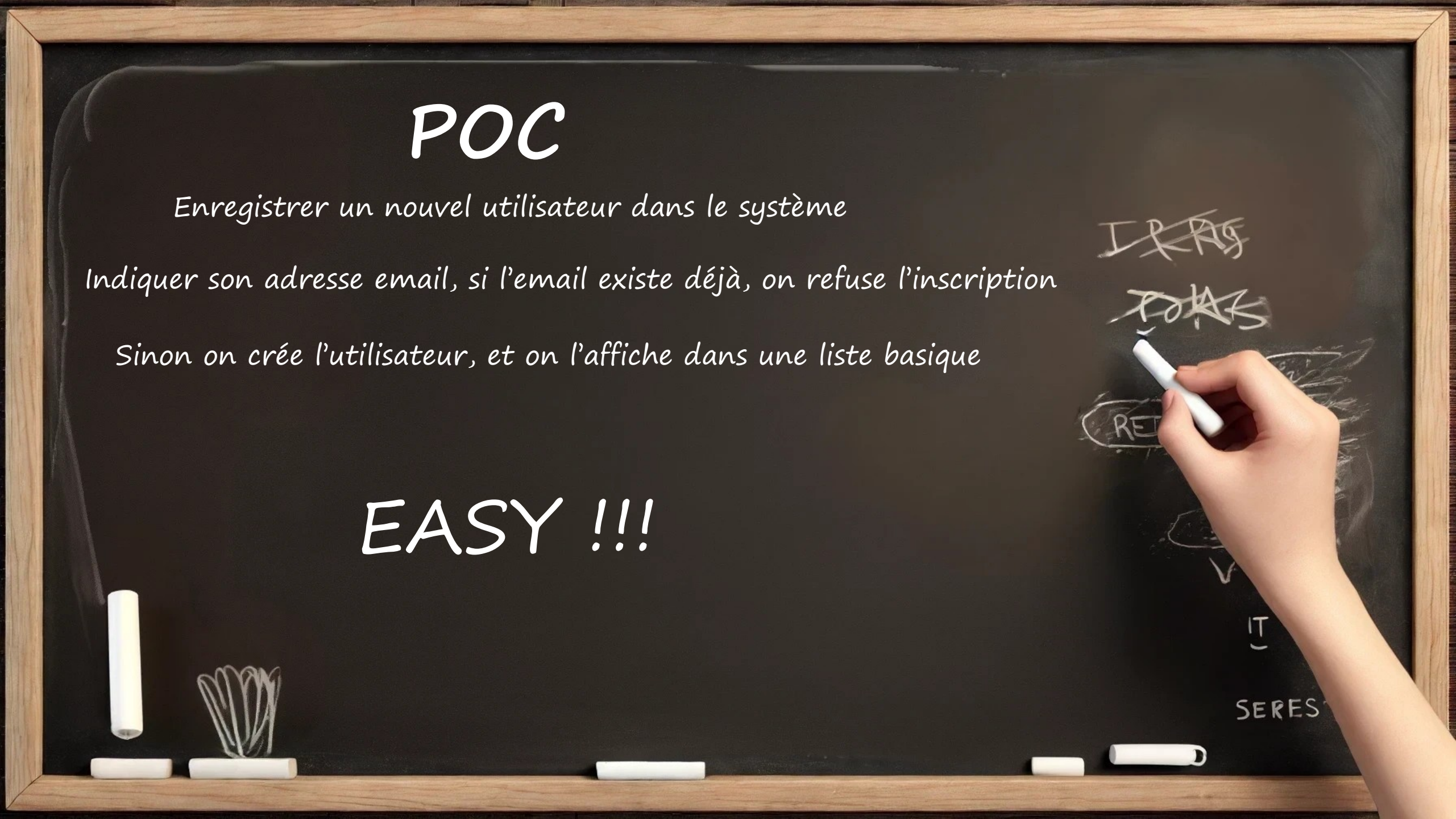
POC

Enregistrer un nouvel utilisateur dans le système

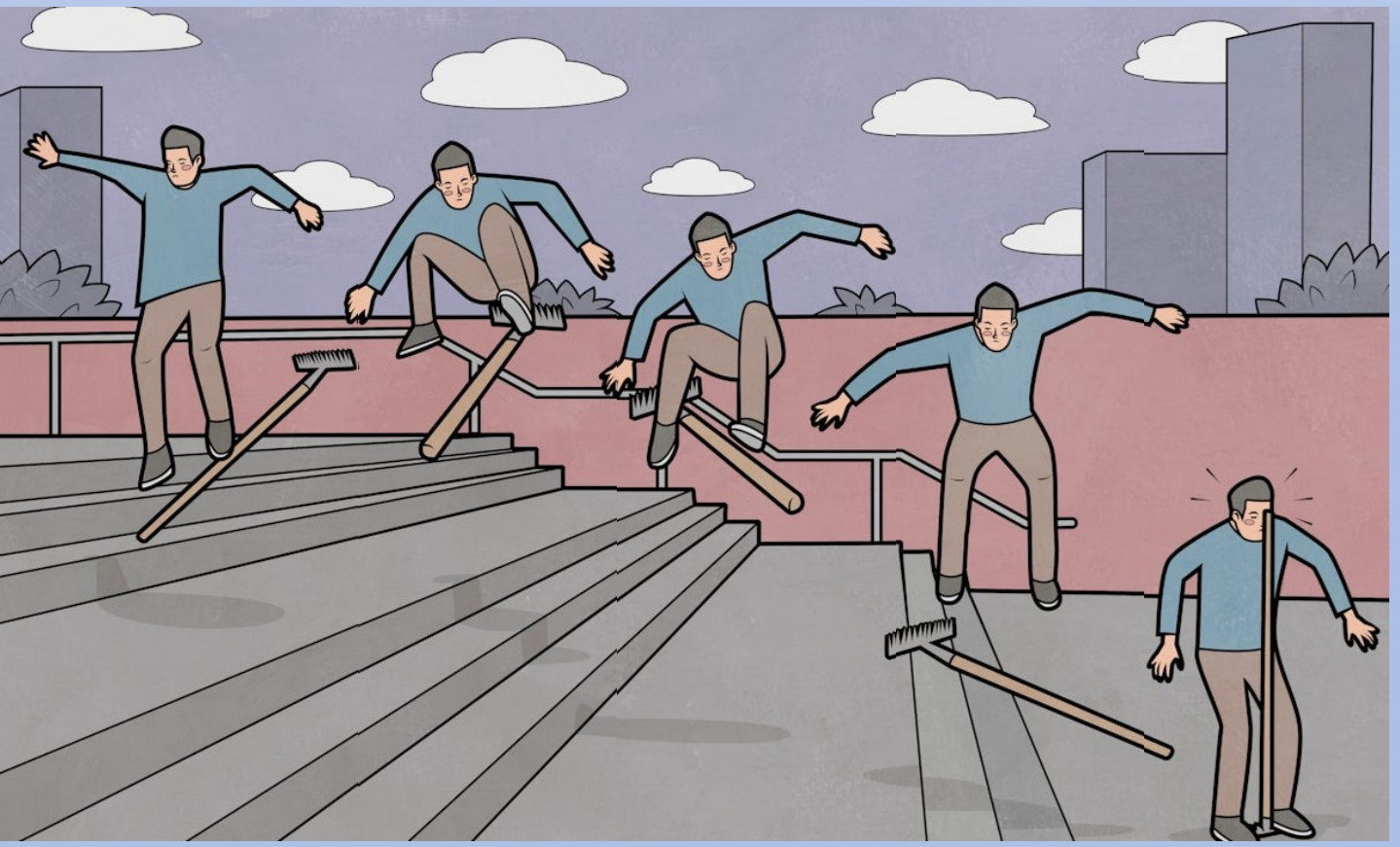
Indiquer son adresse email, si l'email existe déjà, on refuse l'inscription

Sinon on crée l'utilisateur, et on l'affiche dans une liste basique

EASY !!!



Let’s code !



Mise en place EventStore

Conception des Events

Event Handler

Command Handler

Vérification de l’unicité de l’@mail

Il est interdit de se baser sur le ReadModel pour prendre une décision

Il faut reconstruire un état en rejouant les évènements

Comment ça va se passer quand le nombre d’utilisateurs va augmenter ?

Si j’autorise la suppression du compte ou sa désactivation, comment je reconstruit un état cohérent ?

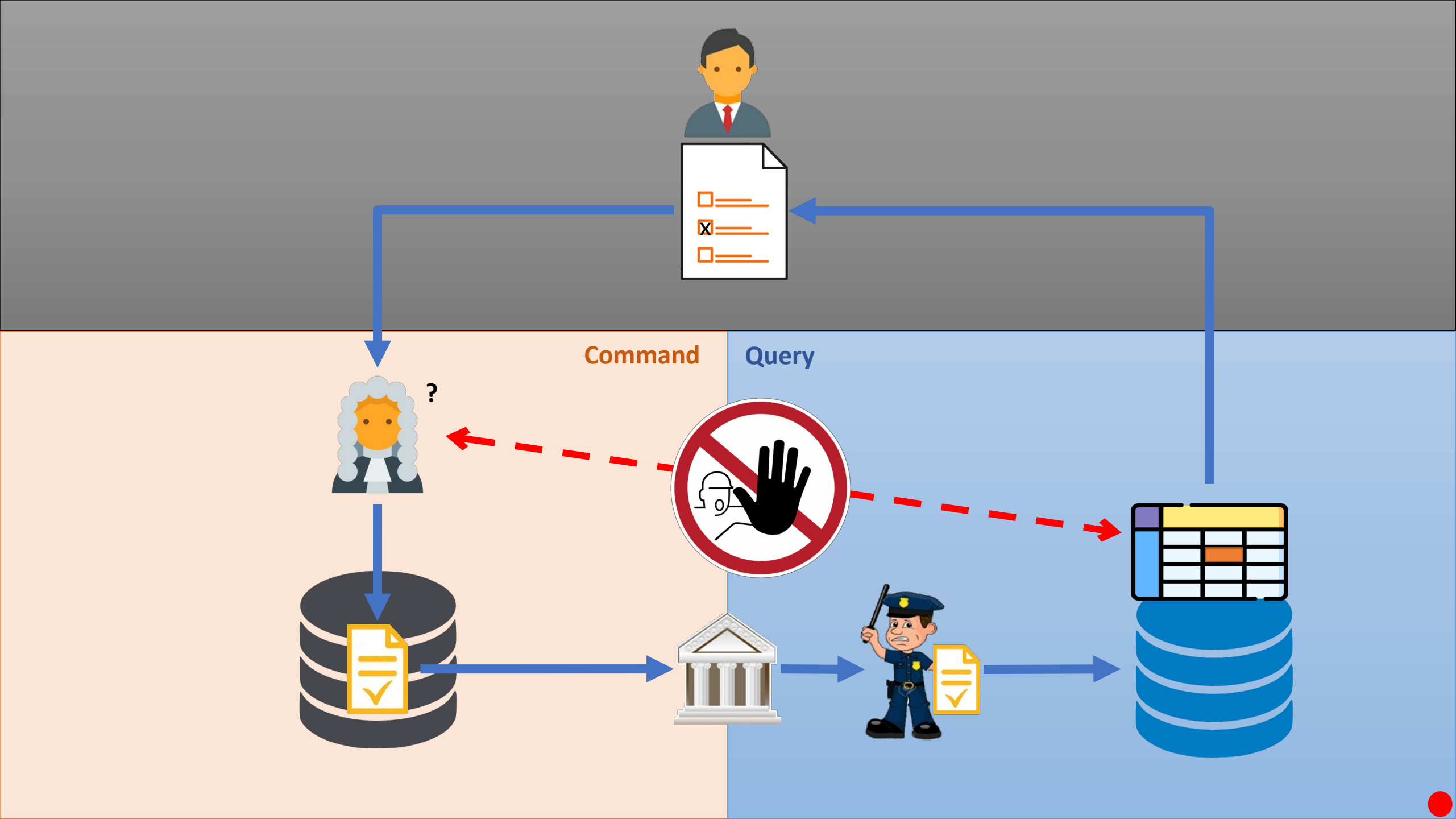
Alors qu’un bon vieux :

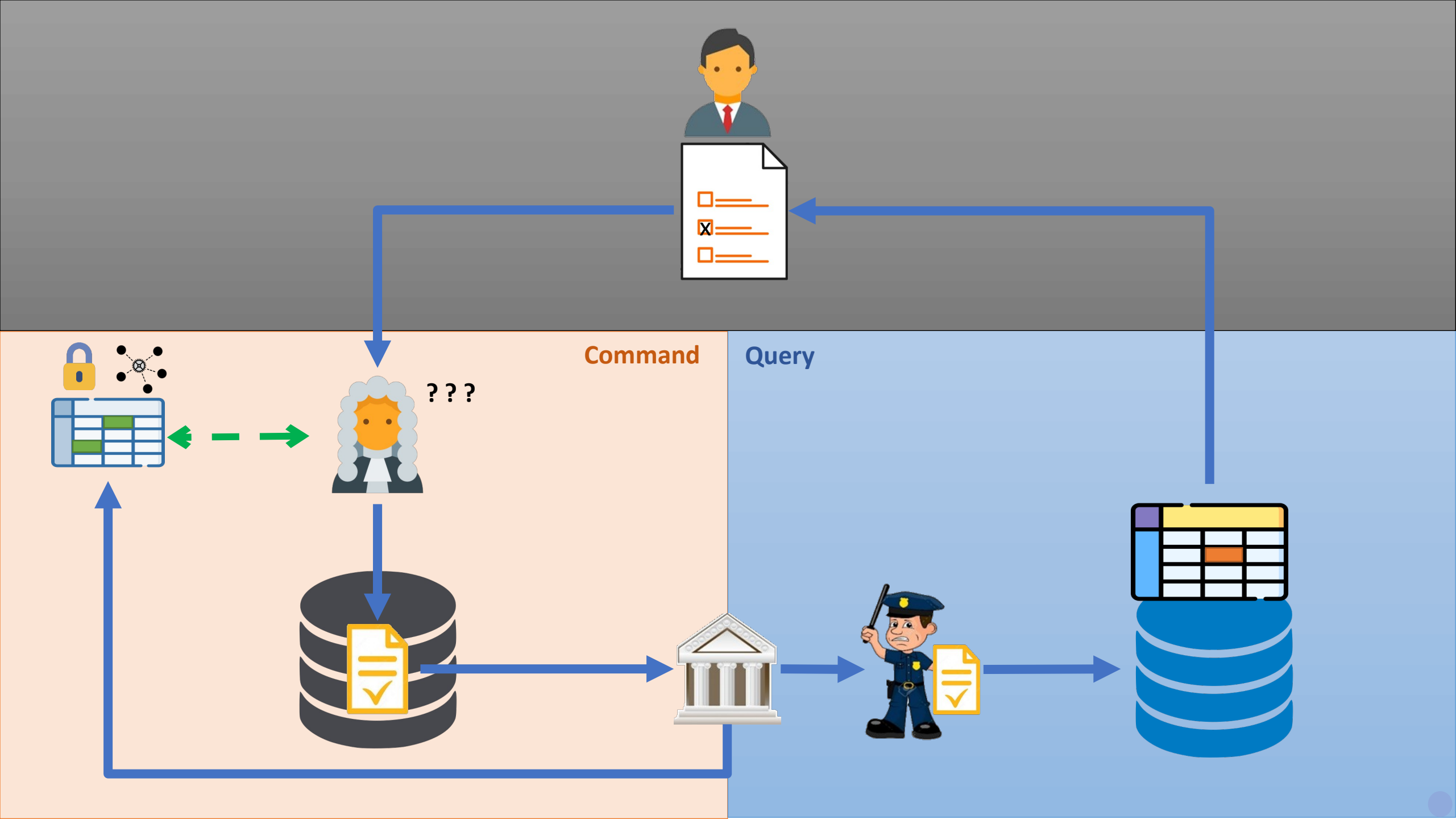
```
SELECT 1
FROM Users
WHERE email = ‘test@email.com'
LIMIT 1;
```

C’est quand même plus simple.


FAIL


Penser au-delà des dogmes.
Pourquoi vouloir à tout prix se baser sur le modèle de lecture ?
Personne n’a jamais interdit de créer des projections dédiées aux commandes ?


Command
Query
?


Command
Query
? ? ?


VOILA,

Nos divergences

Nos divergences

Usage là où l’usage semble pertinent

Nos divergences

Usage en mode exploratoire


Conséquence
Des solutions techniques parfois différentes

Stockage des évènements

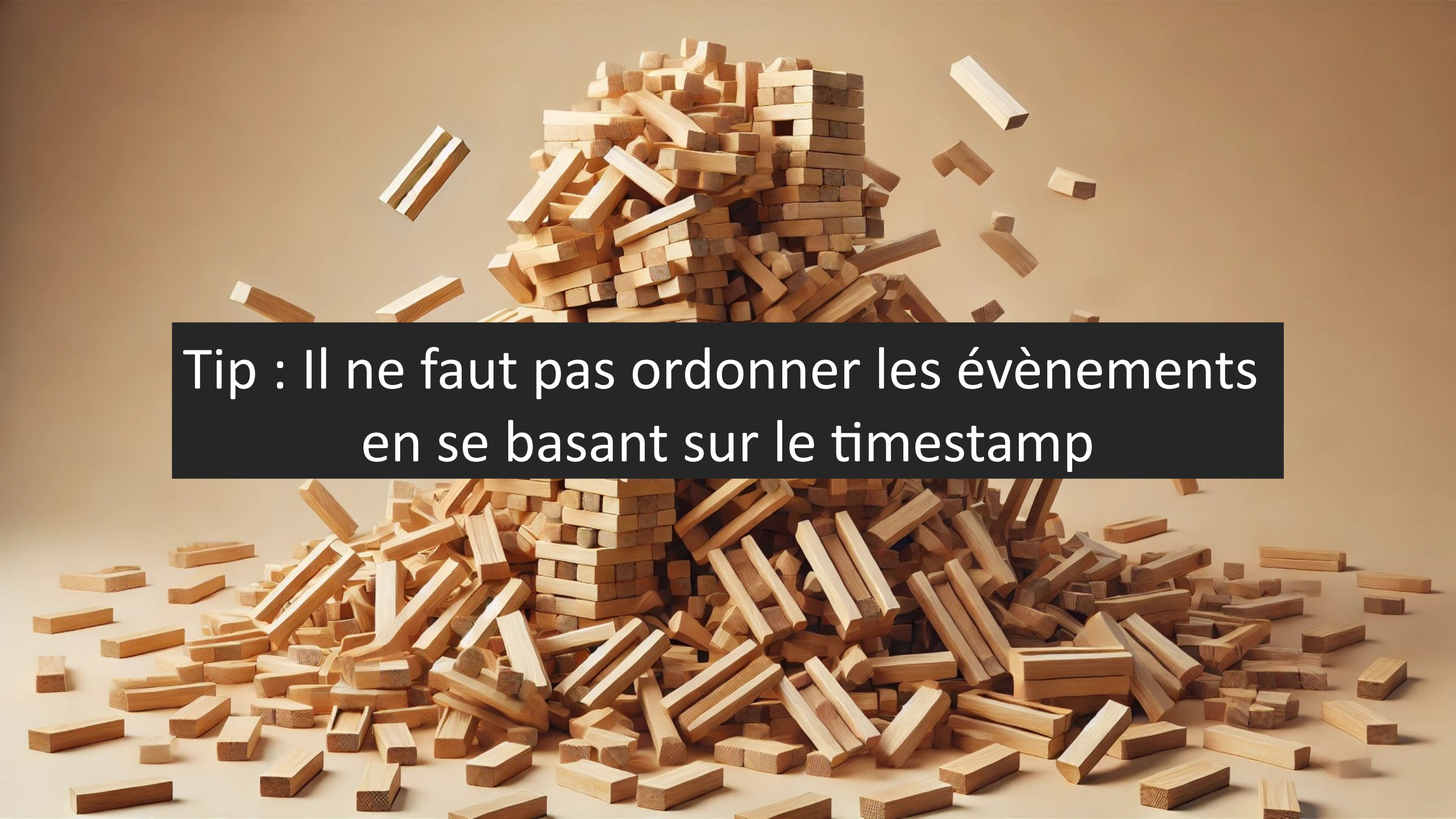
Casier judiciaire


Quel choix pour l'event store ?
Aucune importance !
Stock ses évènements dans PostgreSQL
Stock ses évènements dans un fichier plat

Structure minimale

```
CREATE TABLE events (
    aggregate_id VARCHAR NOT NULL,
    sequence_number INTEGER NOT NULL,
    data JSONB NOT NULL,
    timestamp TIMESTAMPZ NOT NULL DEFAULT NOW()
);

CREATE UNIQUE INDEX "events_unique" ON "events" USING btree
    ("aggregate_id", "sequence_number");
```


Tip : Il ne faut pas ordonner les évènements
en se basant sur le timestamp

Structure Pratique

```sql
CREATE TABLE events (
    id INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    context VARCHAR NOT NULL, -- with or without tenant
    aggregate_name VARCHAR NOT NULL,
    aggregate_id VARCHAR NOT NULL,
    sequence_number INTEGER NOT NULL,
    data JSONB NOT NULL,
    timestamp TIMESTAMPZ NOT NULL DEFAULT NOW(),
    creator_login VARCHAR NOT NULL
);

CREATE UNIQUE INDEX "events_unique" ON "events" USING btree
    ("context", "aggregate_name", "aggregate_id", "sequence_number");
```

Tip : encodez le type dans le payload

```
{
    "Type":"UserRegistered",
    "Fields":[{...}]
}
```

Casier judiciaire

Stockage des évènements

Stratégies de sérialisation

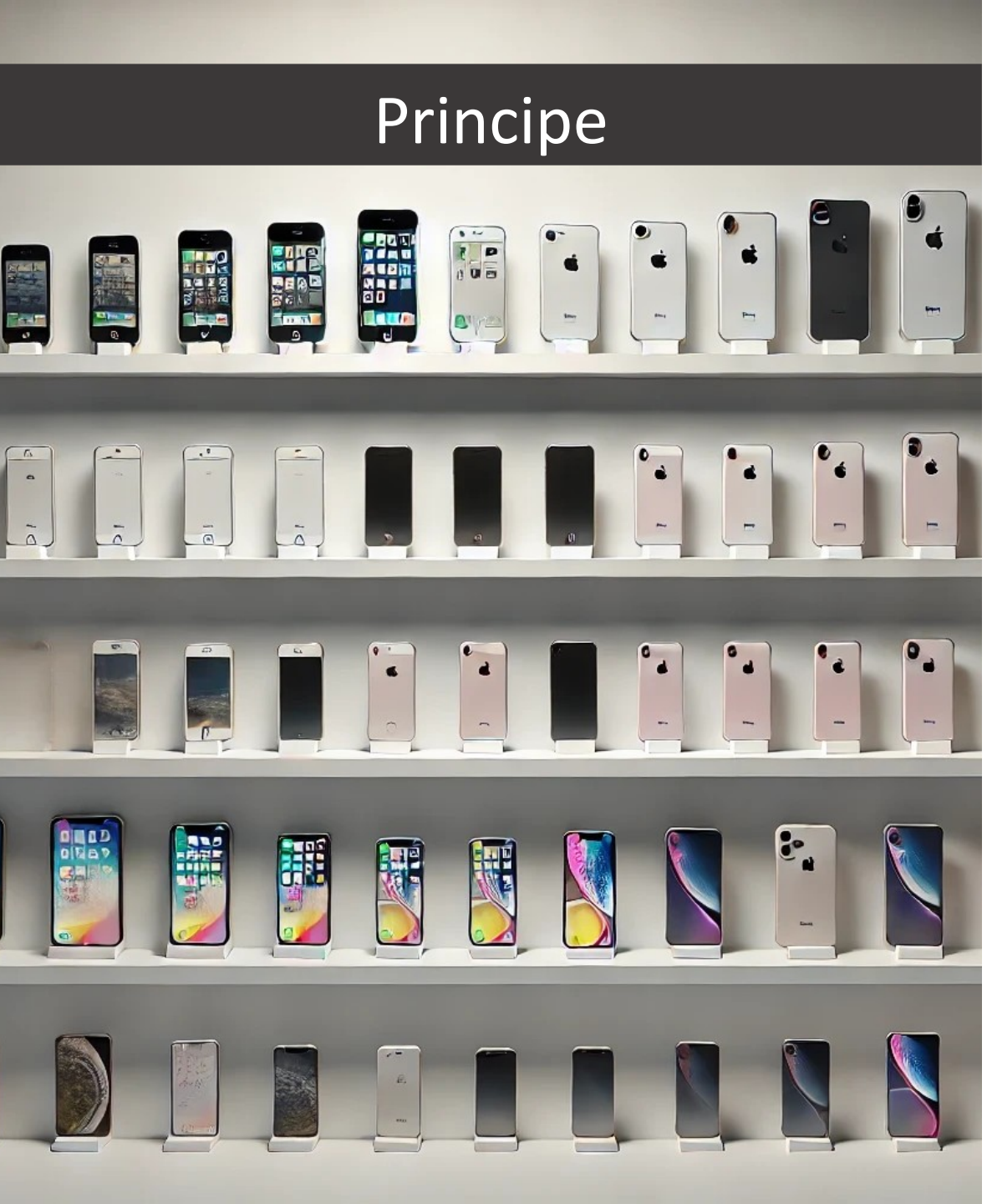
Ou comment gérer la migration

des évènements (renommage, ajout de champs, …)

Anecdote

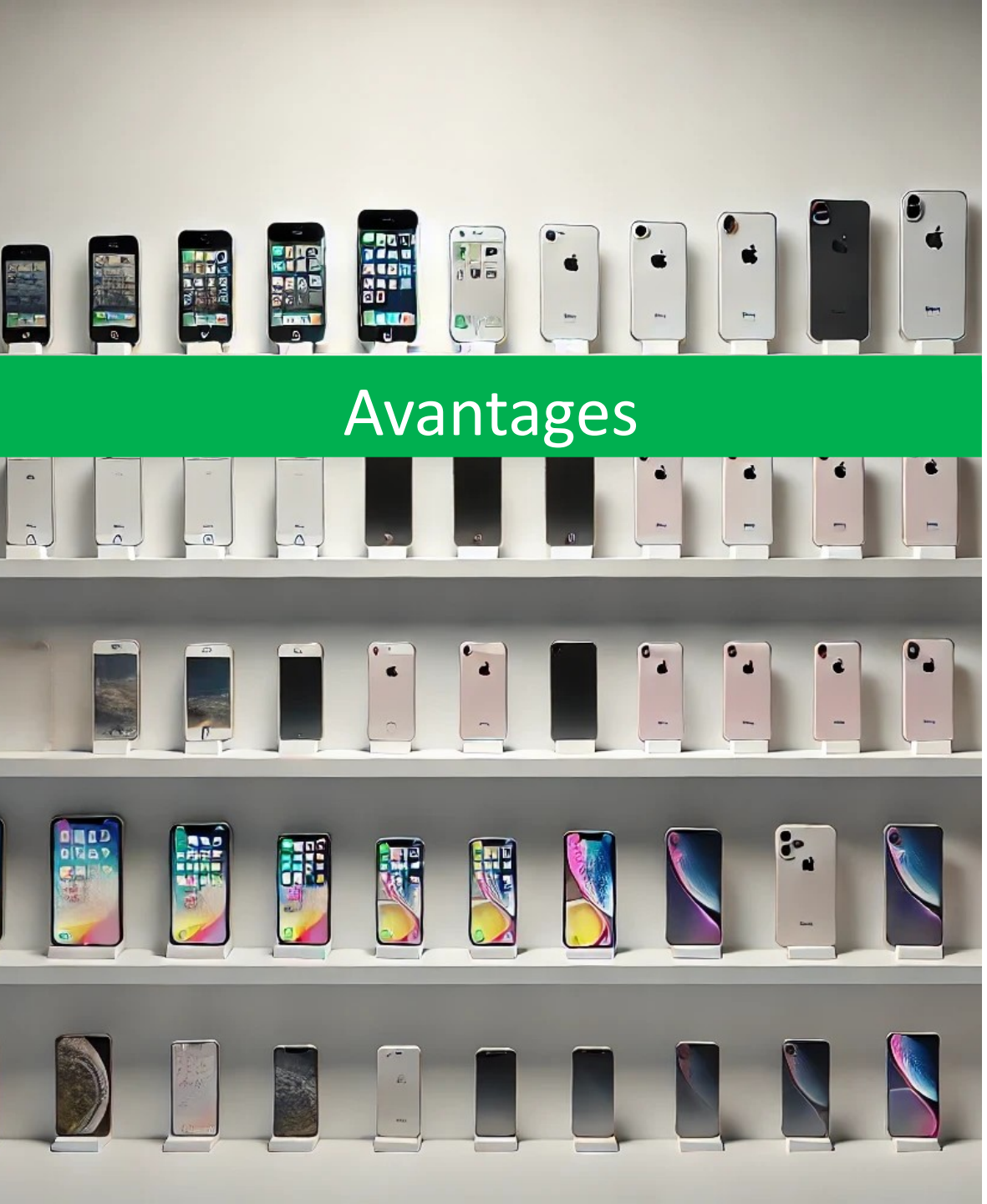
Stratégie du versioning

Principe



On donne une version par évènement. La modification d'un évènement implique la création et l'utilisation d'une nouvelle version.

L’application doit toujours savoir lire les anciennes versions.



Avantages

Pas de migration.

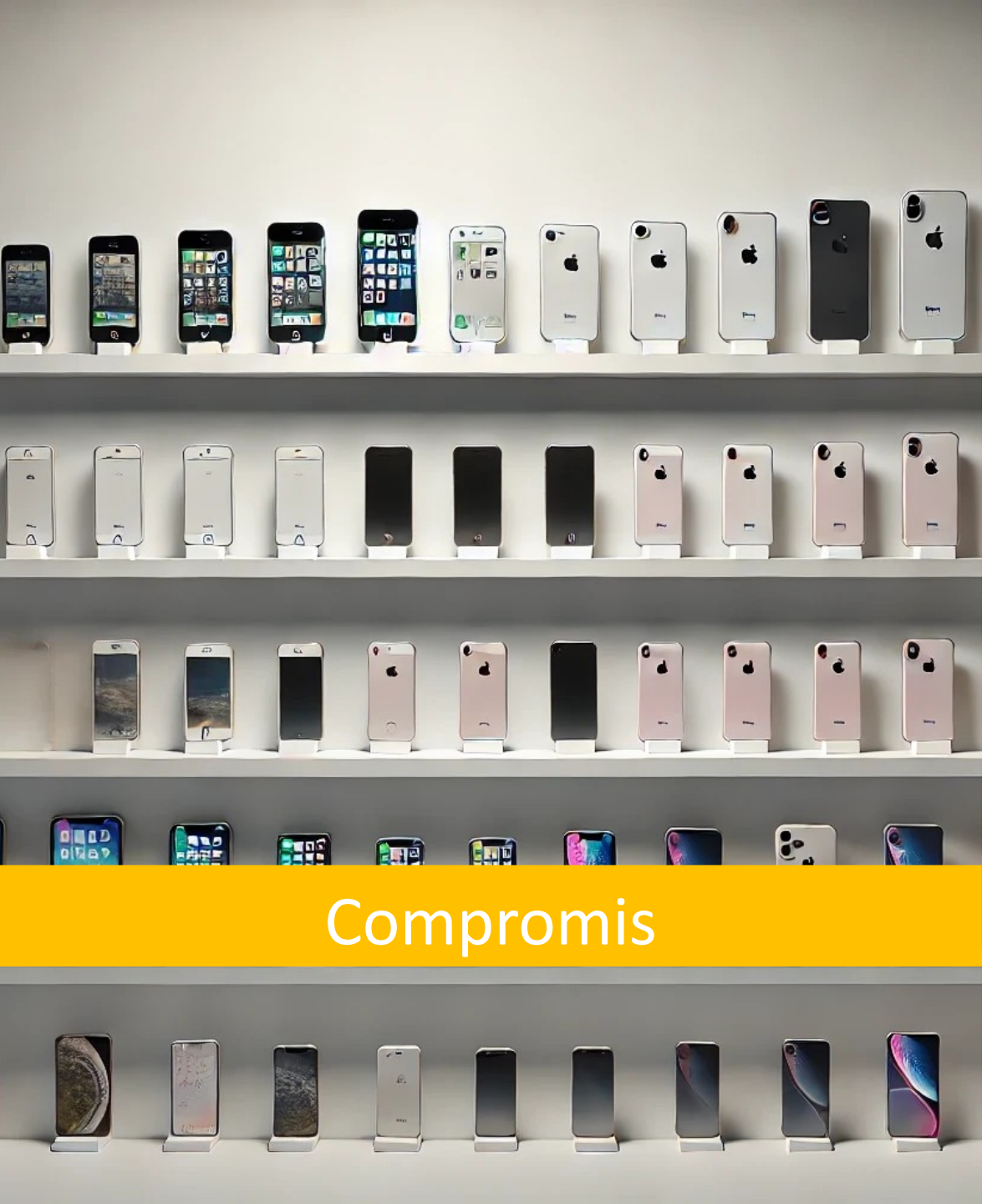
Pas de nécessité de sérialisation custom.

Inconvénients

Le code doit rester compatible. Il faut appliquer de la logique dans toutes les couches de l'application.

Il faut garder toutes les classes, même celles obsolètes.

Complexité croissante.



Compromis

L’ajout de complexité est-il acceptable en vue du nombre de versions que l’on va produire ?

Adapté dans un contexte où le métier est stable.

Stratégie de la migration

Principe

On ne garde dans le code que la dernière version des évènements. Tout changement nécessite une migration de la base.

Avantages

Code clair

Complexité contenue

Pas de nécessité de sérialisation Custom

Inconvénients

Payload pas toujours simple à manipuler.

Ajout d’un nouveau champ : plusieurs choix possibles au cas par cas.

Nécessite de réécrire l’histoire dans l’event-store : pas très en phase avec l’esprit EventSourcing.

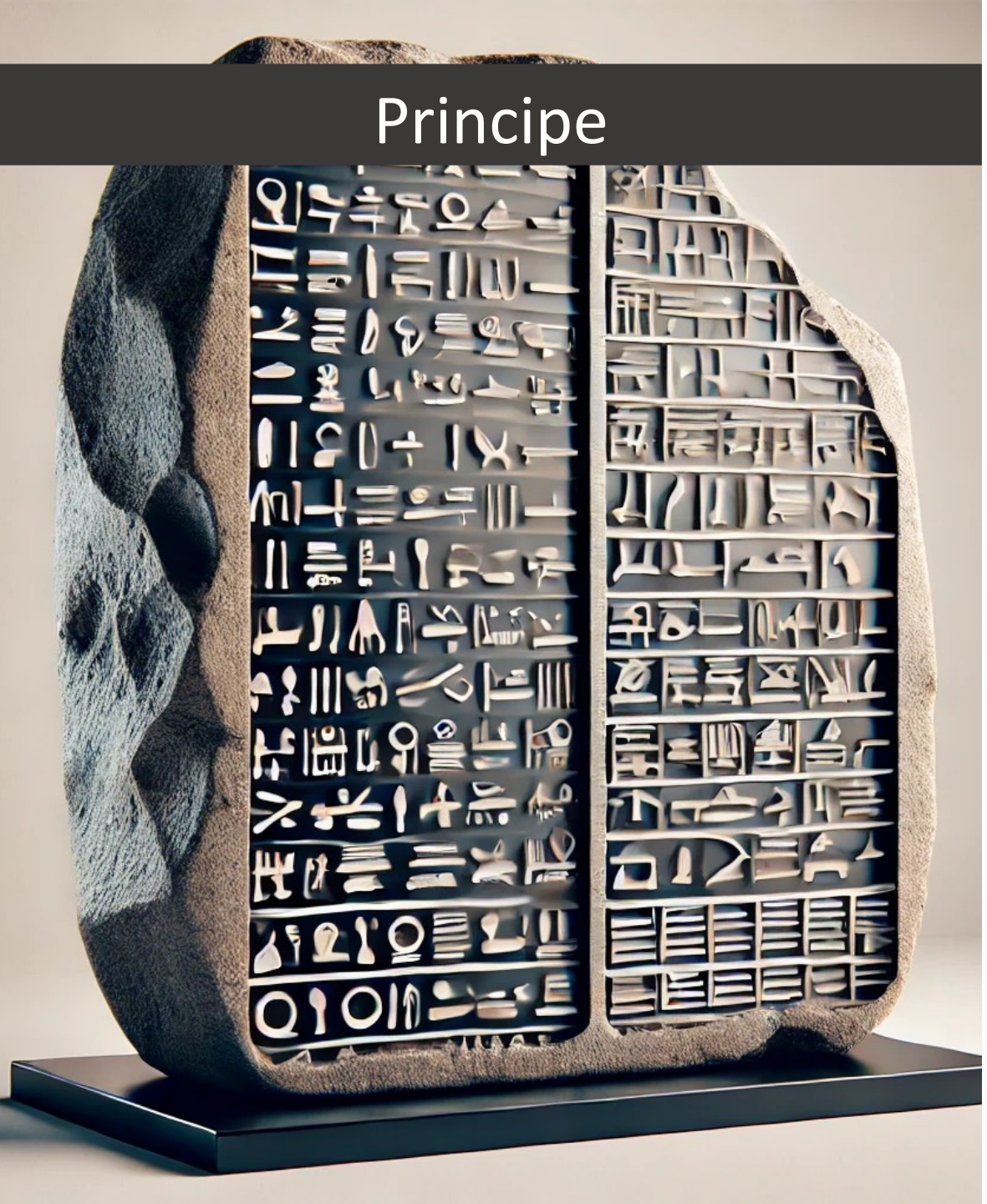
Compromis

Le surcoût des migrations est-il compensé par leur nombre relativement faible ?

Adapté dans un contexte où le métier a émergé mais peut encore évoluer.

Stratégie du mapping

Principe



On ne stock pas directement les types dans le payload mais un alias (le plus souvent un numéro) afin de découpler le nom du code de celui du payload.

Avantages

Aucune migration nécessaires en cas de modification simple (renommage de classes ou de champs, ajout ou suppression de champ), donc pas de charge mentale pour refactoriser son code.

Permet de garder l'historique des events intact.

```
[SerializableTypeIdentifier("149")]
41 references
public record FactoriesDailyPrice(ParameterLineId LineId, DailyPriceMs1 DailyPriceMs1,
    [JsonProperty("DailyPriceMaiche")] DailyPriceCompany2 DailyPriceCompany2,
    DailyPriceCompany3 DailyPriceCompany3, DailyPriceCompany4 DailyPriceCompany4)
{

    5 references
    public DailyPriceCompany3 DailyPriceCompany3 { get; init; } = DailyPriceCompany3 ?? new(0);
    5 references
    public DailyPriceCompany4 DailyPriceCompany4 { get; init; } = DailyPriceCompany4 ?? new(0);
}
```

Inconvénients

Payload pas lisible en l’état.

Nécessite un sérialiseur customisé pour le mapping.

Évènements pollués par des attributs de sérialisation.

Si la structure d’un event change totalement, il faut rebasculer en stratégie du versioning.

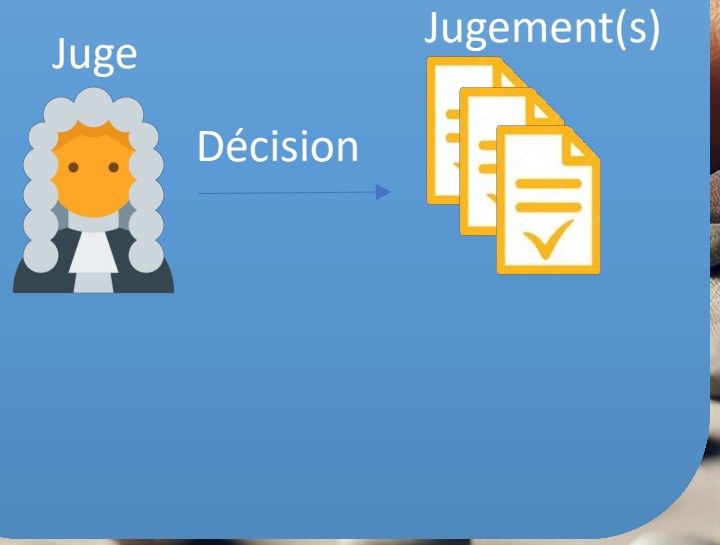
Compromis

Le coût de mise en place des mécanismes de sérialisation et de mapping sont-ils compensés par le nombre de migrations évités ?

Adapté dans un contexte exploratoire (où les events vont beaucoup bouger).

Stratégies de sérialisation

Granularité des évènements



Une des choses les plus difficiles avec Event Sourcing

Un évènement :

- est émis par un agrégat

MAIS

- appartient à un contexte métier

On raisonne en terme d’effets :

- état de l’agrégat.
- Mise à jour de read models.
- Autres event handlers (Envoi de mail, création de fichier, communication inter-context, …).

On veut des évènements autonomes.

Ils contiennent toutes les informations nécessaires :

- pour réhydrater un agrégat.
- au déclenchement d’un effet (idéalement)

Une donnée peut être encodée dans plusieurs évènements.

Un même effet avec la même intention est représenté par le même évènement.

Plusieurs commandes peuvent lever cet évènement.

Si l’effet est le même, mais l’intention est différente, celle-ci est portée :

- par une propriété
OU
- par des évènements différents

Le métier n’a pas encore émergé

L’approche classique

L’approche classique

Partir sur un CRUD traditionnel au départ.

On migre vers CQRS/ES quand le métier émerge ou que le besoin est criant.

Pattern : Brouillon – Workflow – Archive.


CQRS/ES est un pattern
d’architecture local et non global.

L’approche confiante

L’approche confiante

Au départ les events doivent être les plus petits possibles.

Ne pas avoir peur au début de faire des Event orientés CRUD (xxxCreated, xxxValueChanged, xxxDeleted) contrairement à ce que peut dire la communauté.

Faire plusieurs types d’event lorsque les intentions sont différentes, même s’ils font la même chose.

Evitez de calquer un events avec un formulaire entier, surtout s’il a plus de 10 champs.

Quand les vrais besoins émergent, on peut renommer nos évènements ou en créer de nouveaux plus complets et moins CRUD (xxxValidated, xxxRegistered,…).


CQRS/ES n’est pas réservé qu’aux
gros projets très complexes

Granularité des évènements

Les évènements de compensation

Évènements de compensation

Comme le dit Greg Young : Si vous avez une question sur EventSourcing, demandez à votre comptable !

En cas d'erreur métier :

- ne modifiez pas les évènements déjà écrits.
- ne modifiez pas le stream d’évènements.

Utilisez des événements de compensation : "ok, voici l'état à appliquer".

PARENTHÈSE SUR LES SNAPSHOTS

LES SNAPSHOTS

Sujet récurrent dans la littérature.

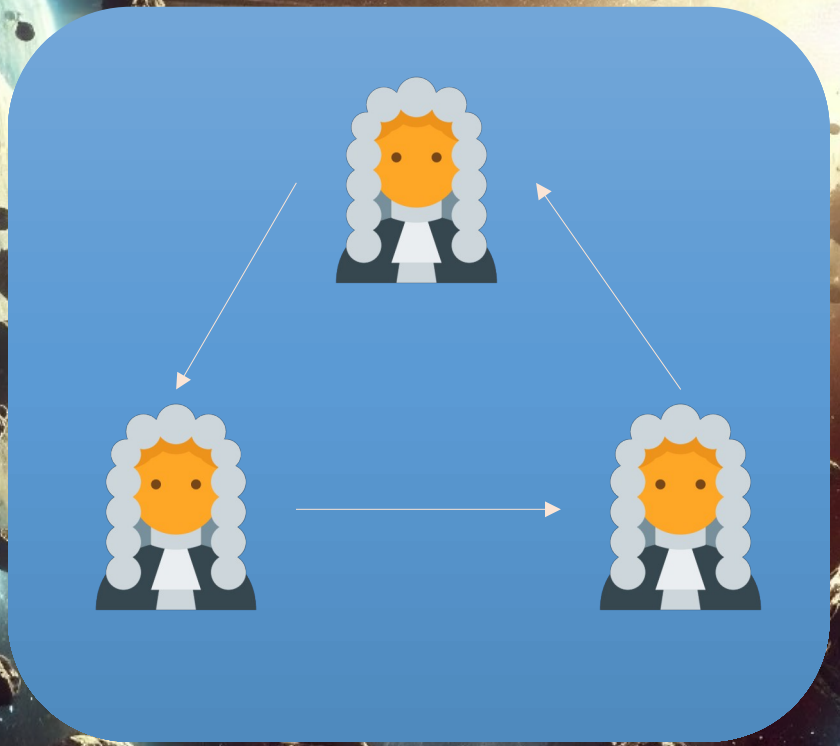
Aucun de nous n’en n’a jamais eu besoin.

Potentiel code-smell sur un mauvais découpage de stream.

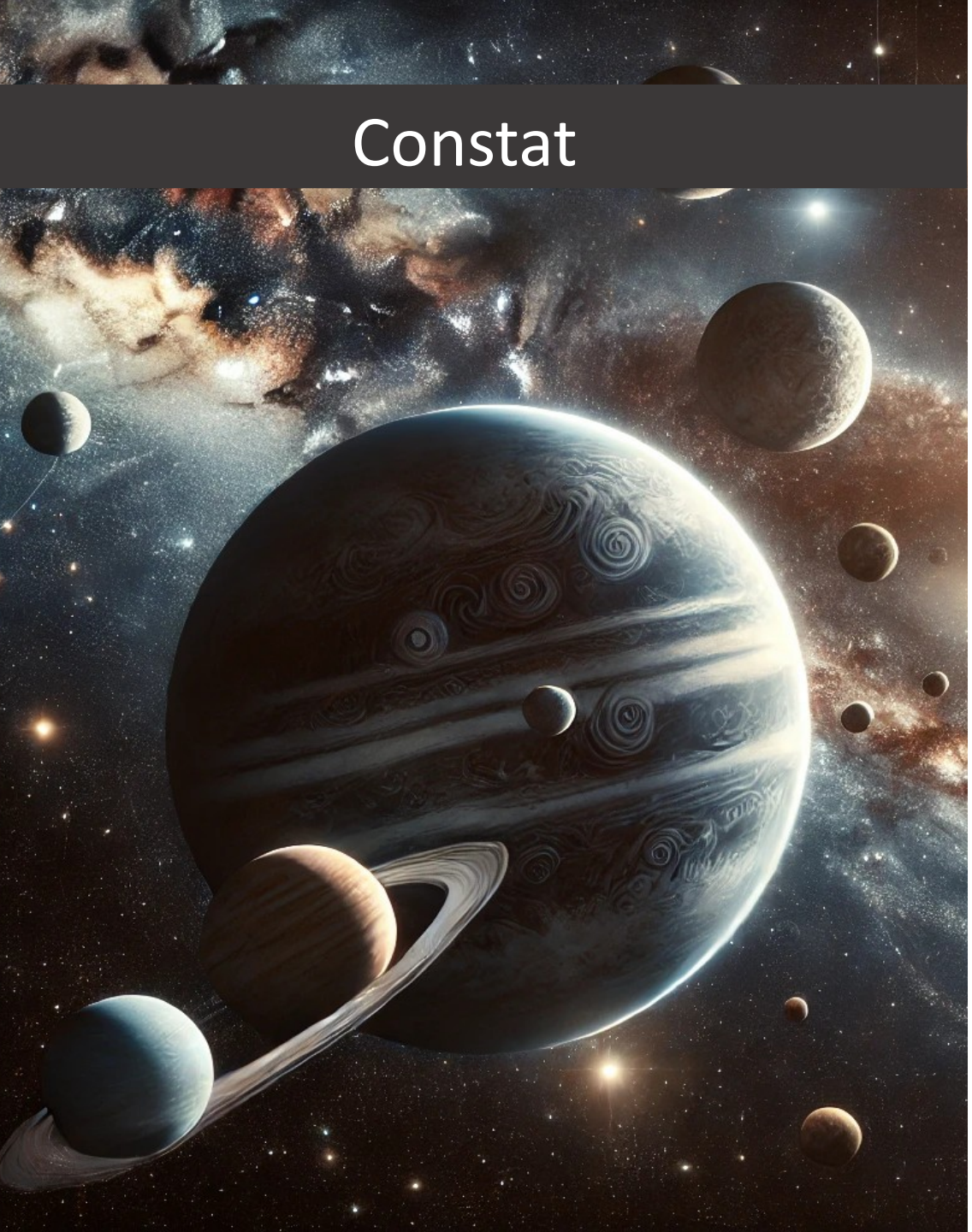
Anecdote

Granularité des évènements

Les arborescences d’agrégats

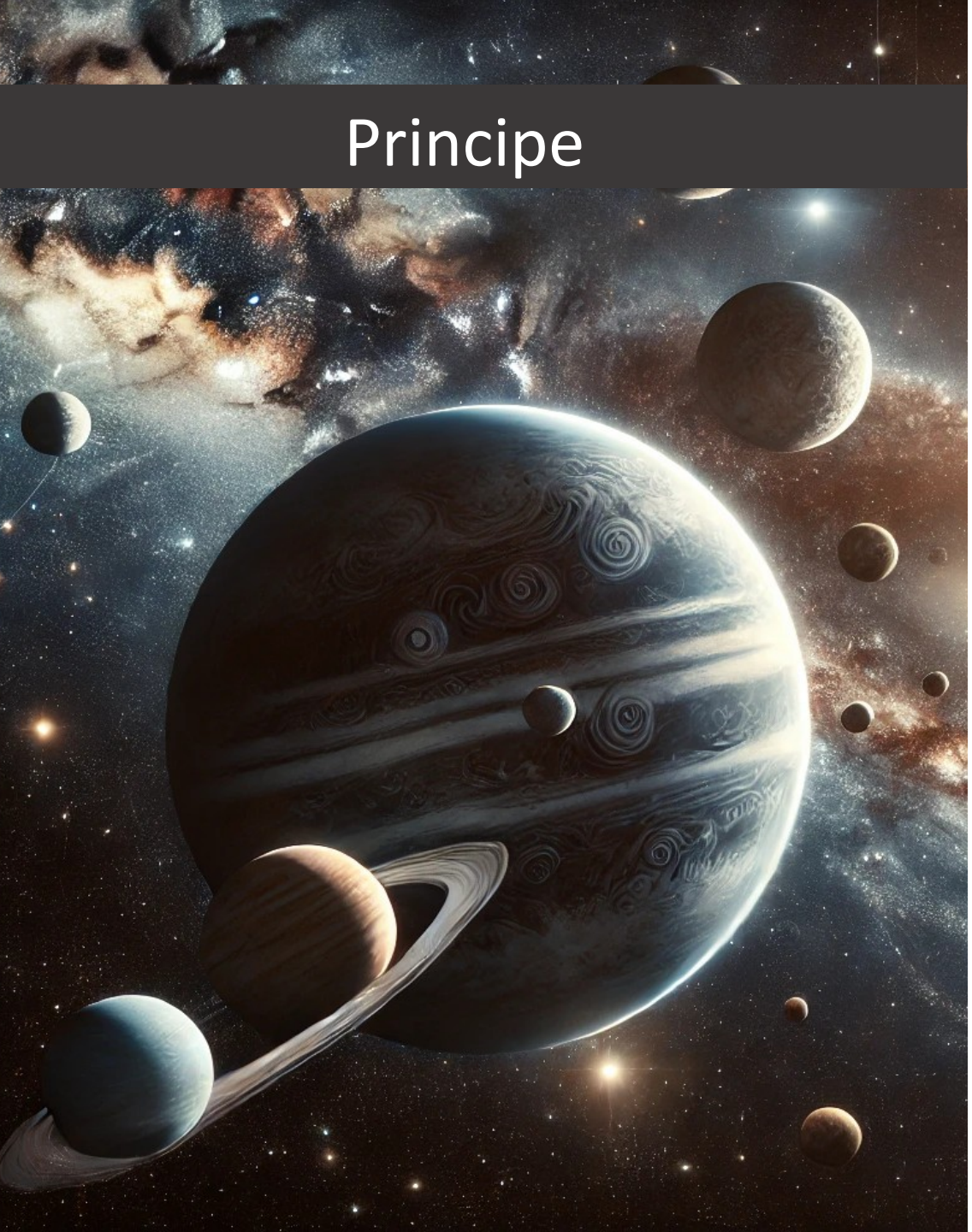


Constat



Dans la plupart des cas, une commande va venir manipuler un seul stream. Mais dans certains cas, on va devoir en manipuler plusieurs.

Il est préférable de ne pas se restreindre à une seule stratégie, mais plutôt de choisir au cas par cas en fonction des contraintes.



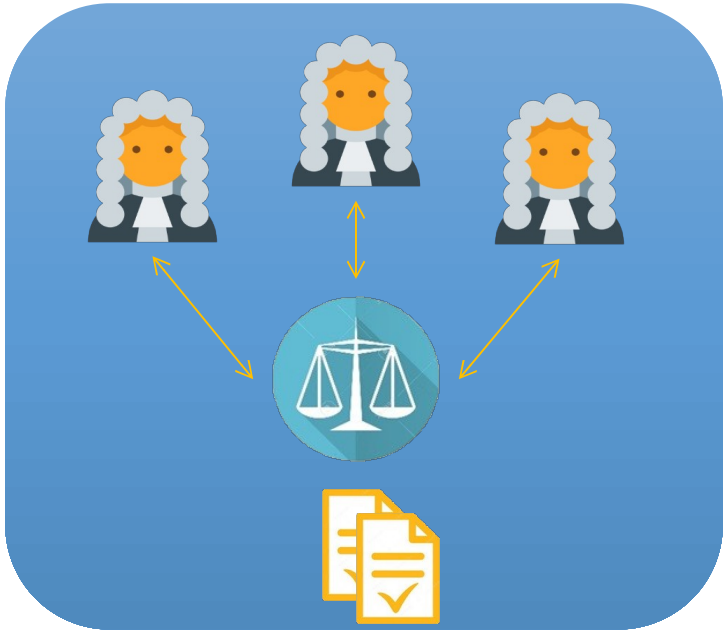
Principe

Un agrégat ES ne se conçoit pas comme un agrégat traditionnel, à cause de la notion de stream.

Ainsi, il faut éviter d'embarquer des sous-objets, mais plutôt mettre des références.

```
{
    "Case":"TissuAjoute",
    "Fields":[{"TissuId":"..."}]
}
```

ORCHESTRATION



Un service qui manipule les agrégats dans une même transaction.

1. On charge les agrégats.
2. On appelle les méthodes des agrégats.
3. On sauvegarde les nouveaux évènements.

Avantages

Transactionnalité.

Pas de nécessité de compensation.

Tout le code est au même endroit.

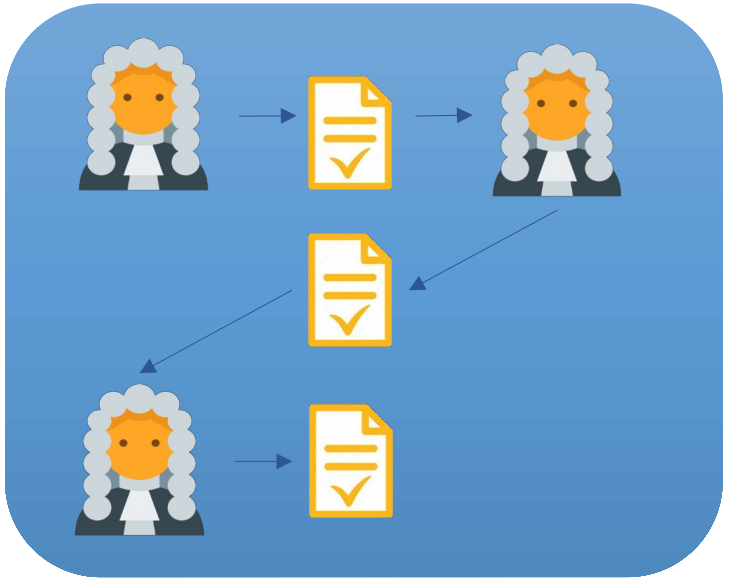
Inconvénients

Complexité concentrée en un seul endroit.

Potentielles difficultés de charges.

Pas adapté aux traitements asynchrones ou long (bloquant pour l’utilisateur).

CHOREGRAPHIE



Traitement en cascade :

1. Un premier agrégat lance des évènements.
2. Un event handler écoute ces évènements.
3. Il lance de nouvelles commandes en conséquence.
4. Ces commandes peuvent à leurs tours lancer des évènements.

Avantages

Facilité de venir ajouter des nouveaux traitements.

Possibilité de découper un traitement.

Adapté aux traitements asynchrones .

Inconvénients

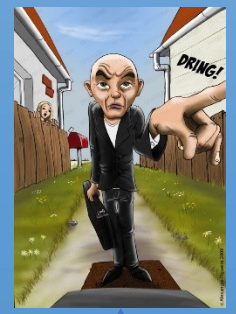
Flux des actions plus difficile à suivre.

Nécessite des mécanismes de compensation en cas d'erreur (saga, process manager).


Les arborescences d’agrégats

Organiser les modèles de lectures (Read models)

Jugement



Application de la décision

La base

Projetez l’état de vos évènements de manière optimale pour vos lectures (IHM, API, …).

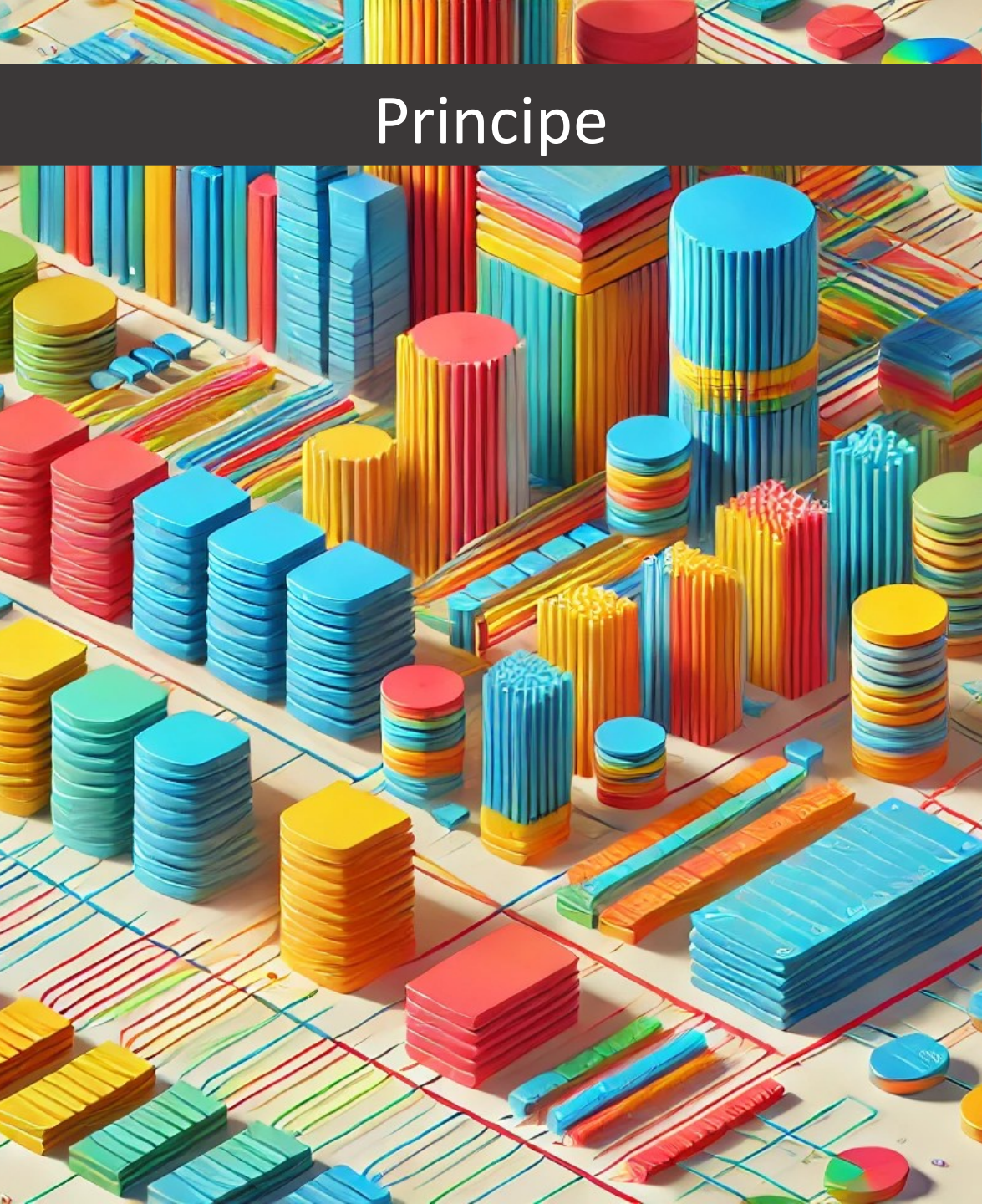
La projection ne porte aucune intelligence autre que du mapping : pas de déduction d’état.

Plusieurs solutions de stockages : SGBD, RAM, Fichier, Cloud, …

Prévoyez des mécanismes de reconstruction.

Stratégie des projections relationnelles

Principe



Les données sont organisées comme dans un SGBD relationnel, avec des tables et des jointures.

Il n’est pas obligé de stocker les données dans une base, on peut organiser ça en RAM sous forme de listes ou de dictionnaires.

Avantages

Pas besoin de reconstruire un nouveau modèle pour ajouter des données existantes.

Approche familière.

Inconvénients

Les données peuvent être complexe à projeter.

Pas adapté à tous les types de vues en fonction de la volumétrie, les performances, certaines représentations hiérarchiques ou très complexes.

Compromis

Le modèle répond il aux besoins de performance et de flexibilité ?

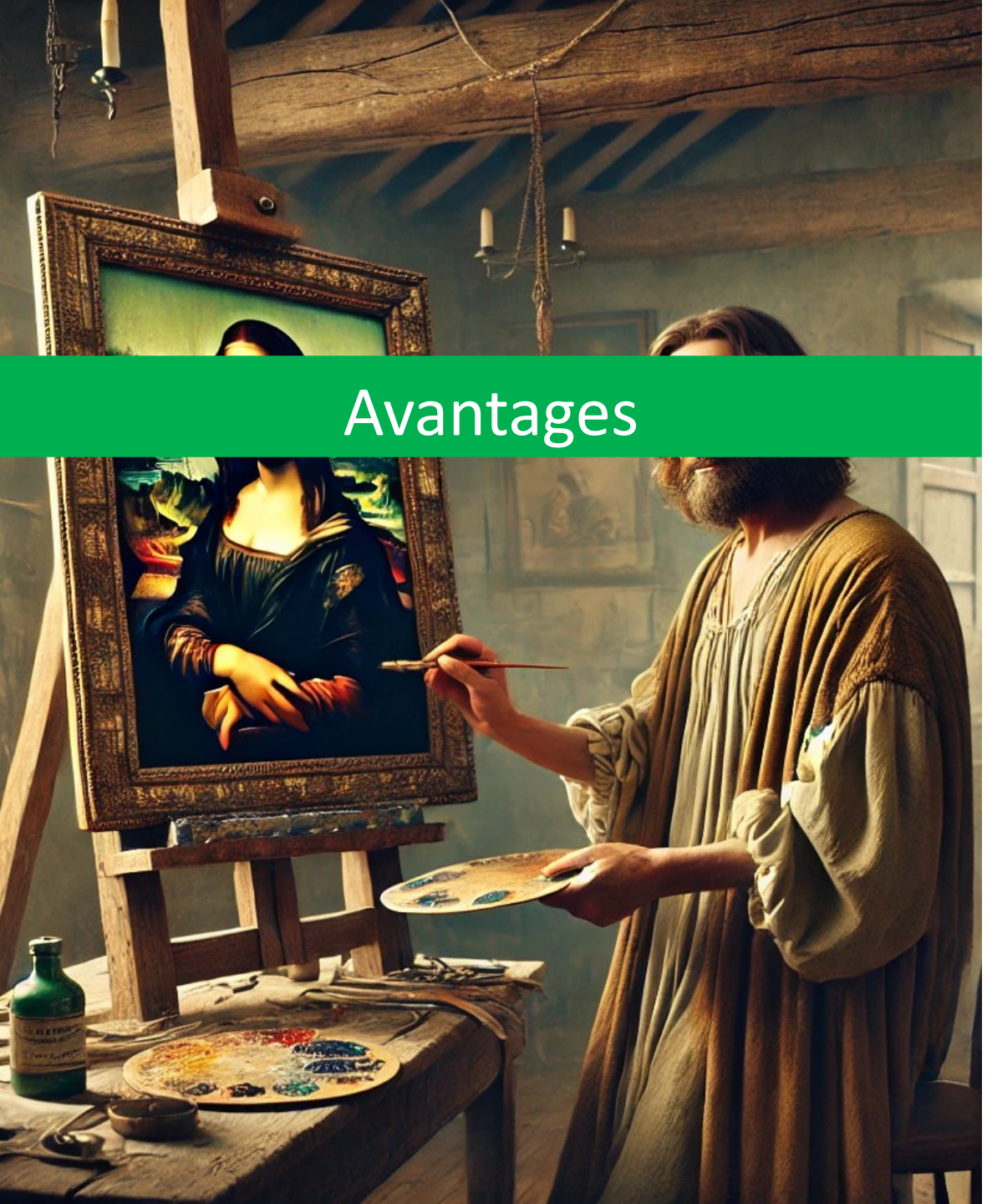
Adapté à un contexte dont les requêtes sont assez simple mais où les données à afficher peuvent changer.

Stratégie des projections spécialisées

Principe

On créé un nouveau readmodel à chaque fois que l'on a besoin de représenter une information d'une nouvelle façon.

On peut avoir plusieurs queries pour un même readmodel.



Avantages

Facile et rapide à requêter.

Représentation adaptée pour le consommateur.

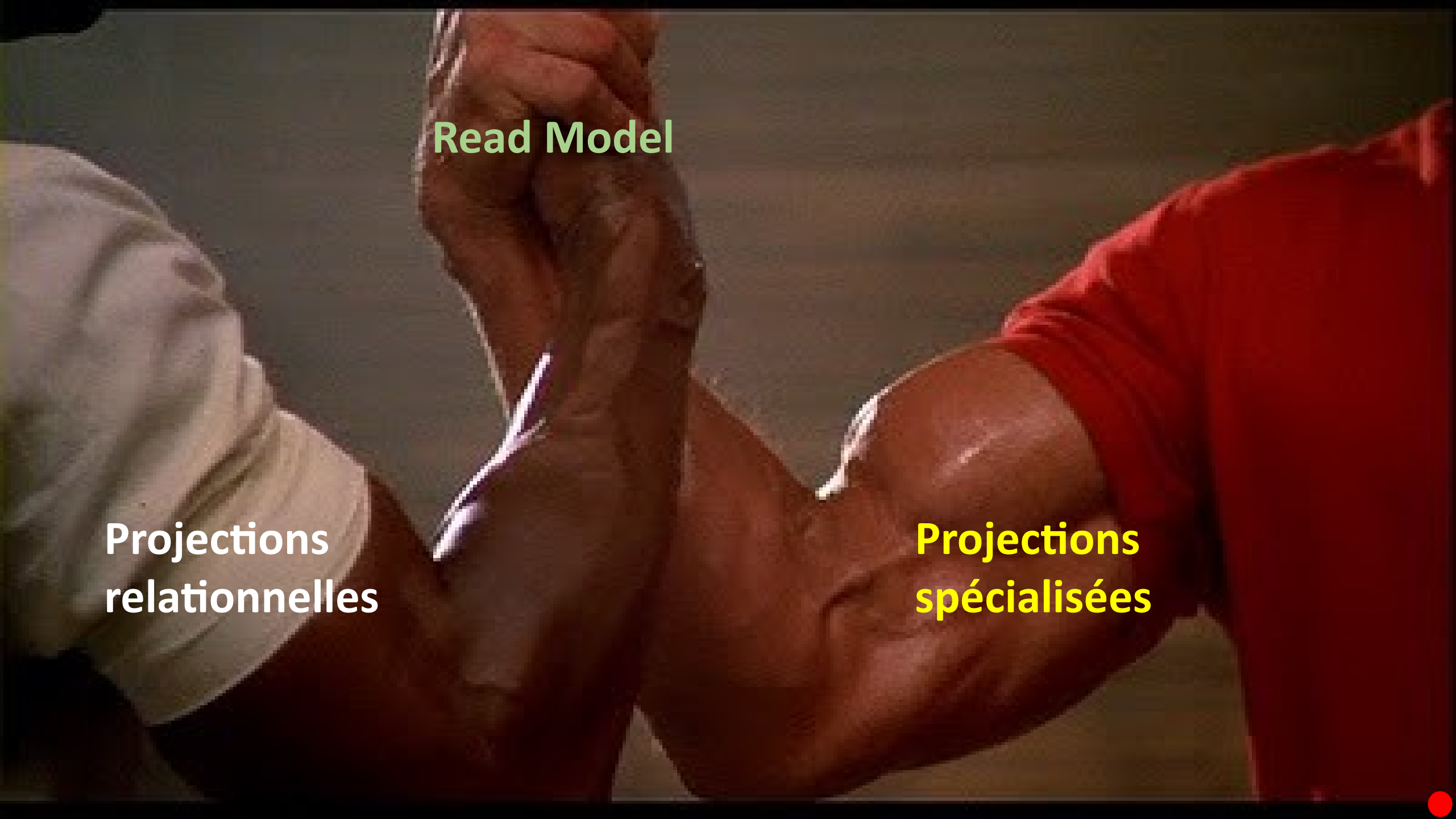
Inconvénients

Multiplication rapide du nombre de modèles.

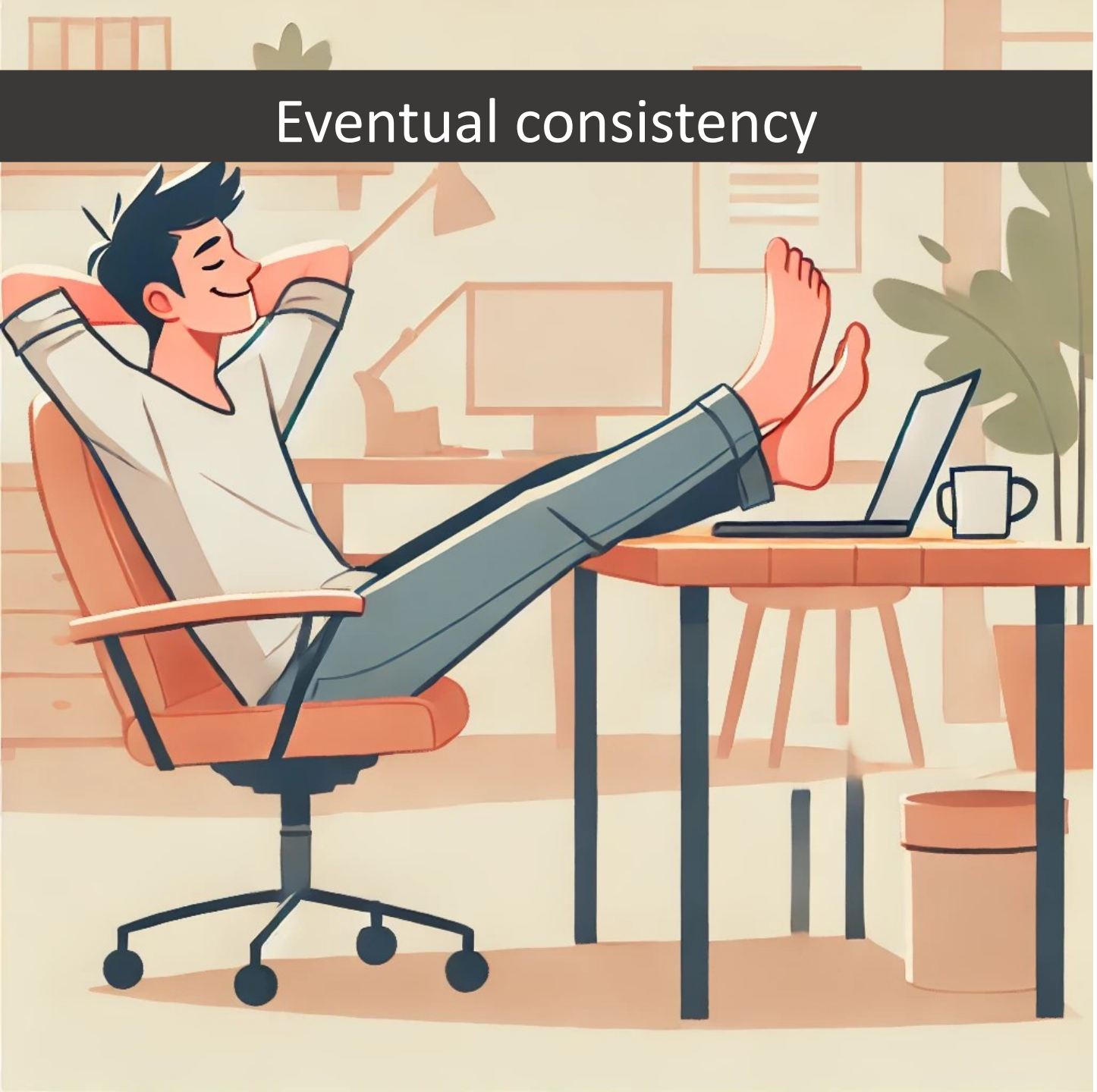
Chaque changement de représentation demande une reconstruction.

Compromis

La diversité des représentations justifie-t-elle plusieurs modèles dédiés ?


Read Model
Projections relationnelles
Projections spécialisées

Eventual consistency

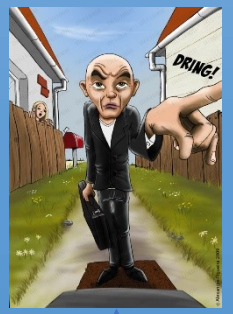


À ne faire que sous la torture.

Faites des projections synchrones par défaut.

Le surcoût de temps à la commande vaut elle le coup de gérer des comportements asynchrones en terme d’UX (données pas à jour) ?

Jugement

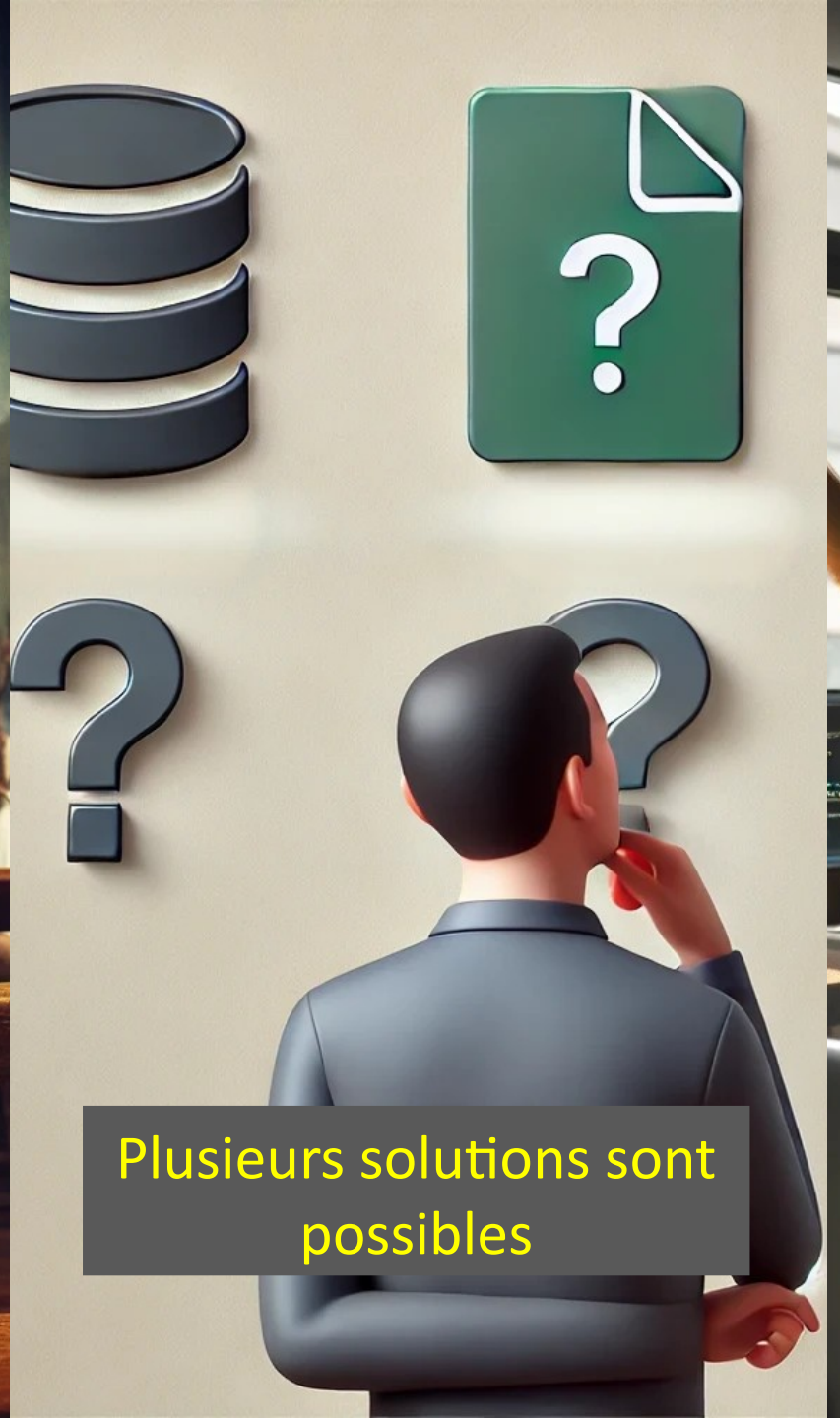


Application de la décision

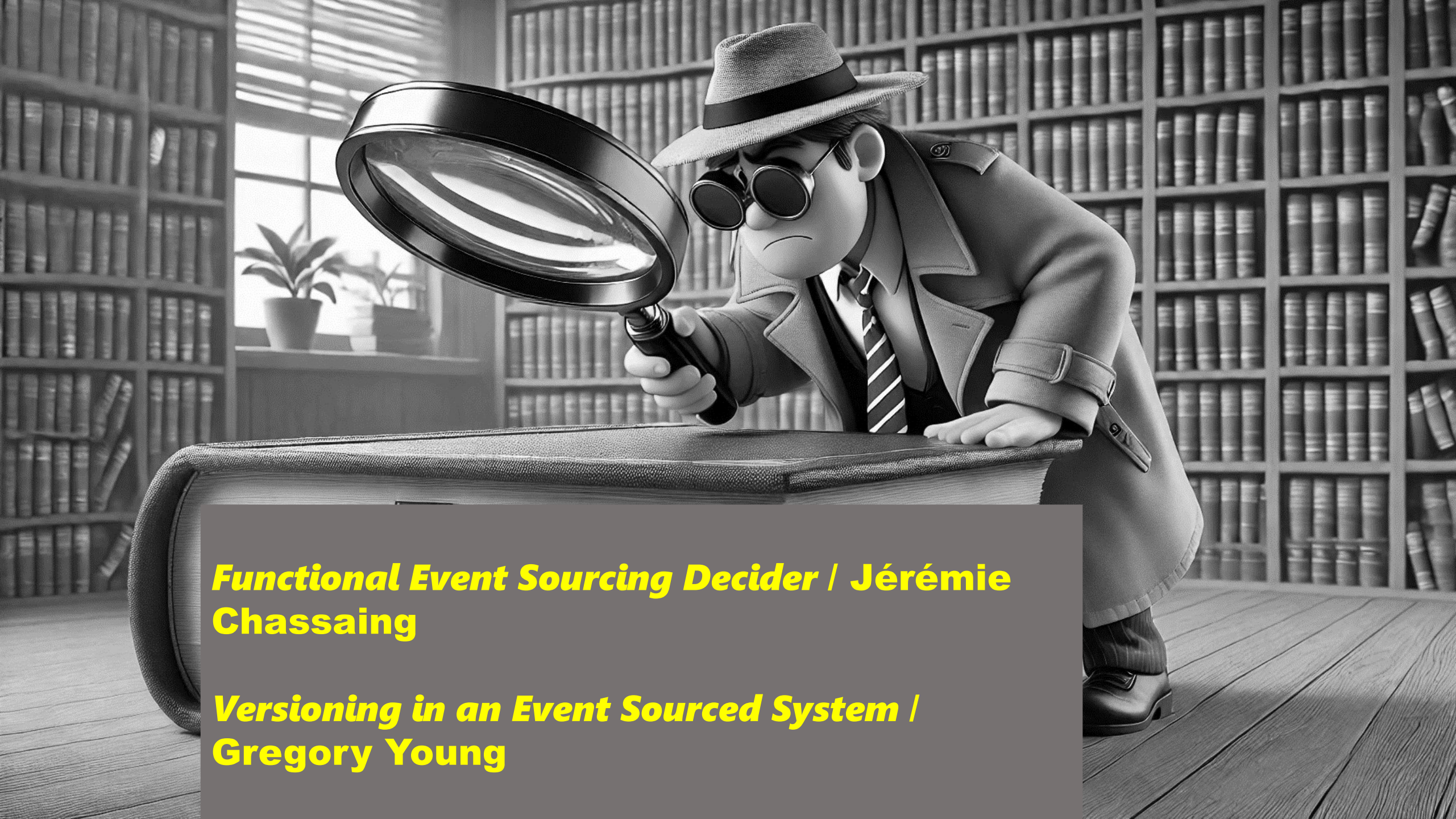
Organiser les modèles de lectures (Read models)

Le mot de la fin


Méfiez vous des dogmes


Plusieurs solutions sont possibles


Faites votre propre expérience



Functional Event Sourcing Decider / Jérémie Chassaing

Versioning in an Event Sourced System / Gregory Young