

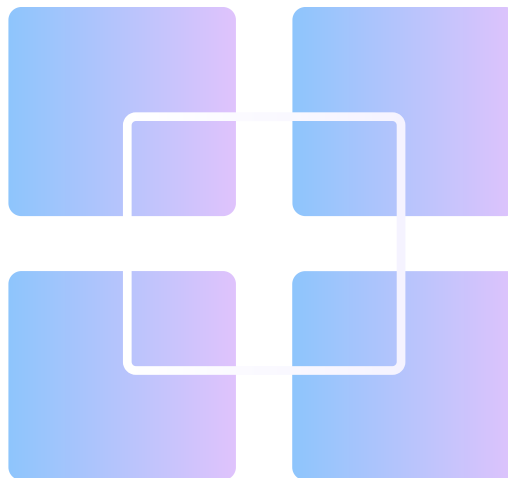
Architectures Back-end

Back-ends et API pour le Web, le Mobile et l'IA



Sommaire

- 📜 Historique des architectures de SI
- 🛠 Ecosystèmes de développement
- 🏗 Patterns d'Architecture
- 💾 Transactions
- 🚀 Microservices
- 📧 Messaging Asynchrone
- 🎯 Serverless
- 🚀 Caching Avancé
- 🗃 Database Sharding et Partitioning
- 🎯 Domain-Driven Design
- ✨ Clean Code & Architecture
- 🛡 Sécurité Avancée
- 🤖 MCP & Intégration IA
- 📖 Conclusion





Historique des architectures de SI



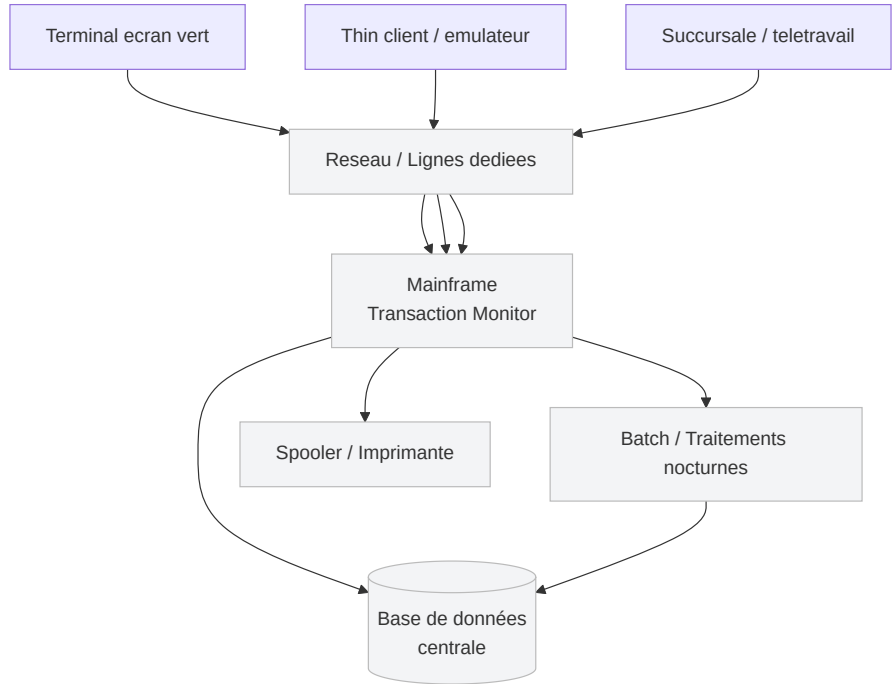
L'Ère des Mainframes (1960-1980)

Caractéristiques

- **Architecture** : Entièrement centralisée
- **Accès** : Via terminaux "bêtes" (pas de calcul local)
- **Coûts** : Énormes investissements initiaux
- **Fiabilité** : Uptime critique, équipes dédiées

Impact sur l'architecture moderne

✓ **Héritage** : Respect de la sécurité, transactions ACID, contrôle centralisé



<https://www.lacommunauteducobol.com/histoire-du-mainframe/>



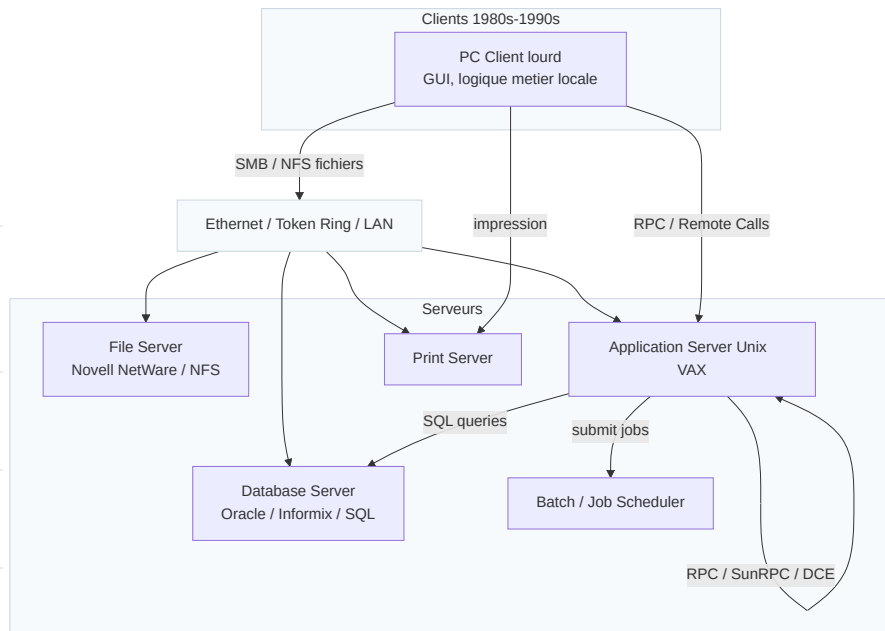
Révolution PC et Client-Serveur (1980-1990)

LA DÉCENTRALISATION COMMENCE

Aspect	Avant	Après
Calcul	Centralisé	Distribué (PC + Serveur)
Données	Mainframe	Débat : où stocker ?
Responsabilité	Unique entité	Partagée
Scalabilité	Verticale uniquement	Horizontale possible

🔑 **Innovation clé** : TCP/IP et protocoles réseau standardisés

ARCHITECTURE ÉMERGENTE



Ère Web et Serveurs d'Applications (1990-2000)

ARCHITECTURE 3-TIERS

MIDDLEWARES ET FRAMEWORKS

- **J2EE** (Java 2 Enterprise Edition)
- **.NET Framework** (Microsoft)
- **Application Servers** (WebLogic, JBoss, WebSphere)

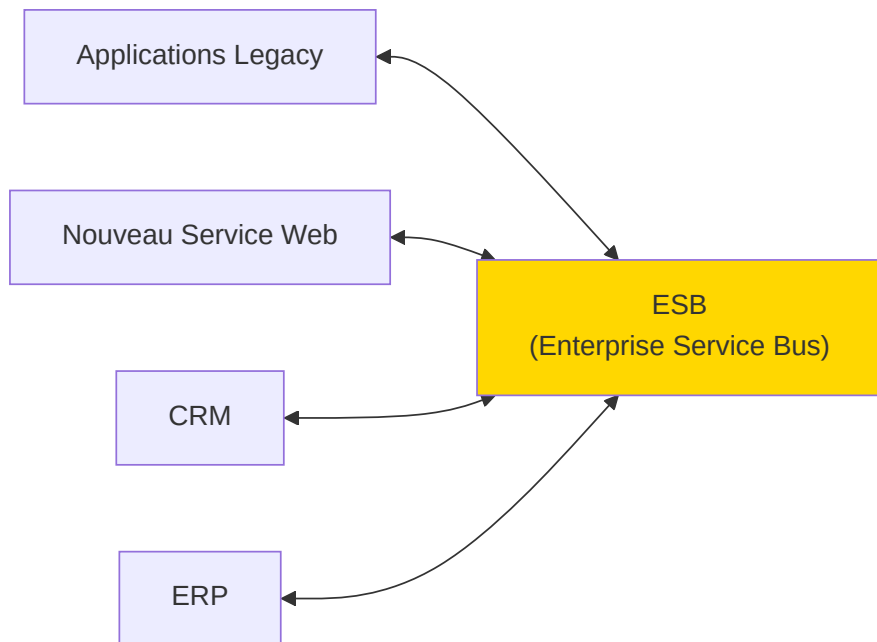
AVANTAGES & DÉFIS

- ✓ Séparation claire des couches
- ✓ Scalabilité horizontale du web tier
- ✗ Monolithes deviennent énormes
- ✗ Déploiement complexe



L'Ère des Services (2000-2010)

ÉVOLUTION VERS L'INTÉGRATION D'ENTREPRISE



PROTOCOLES & STANDARDS

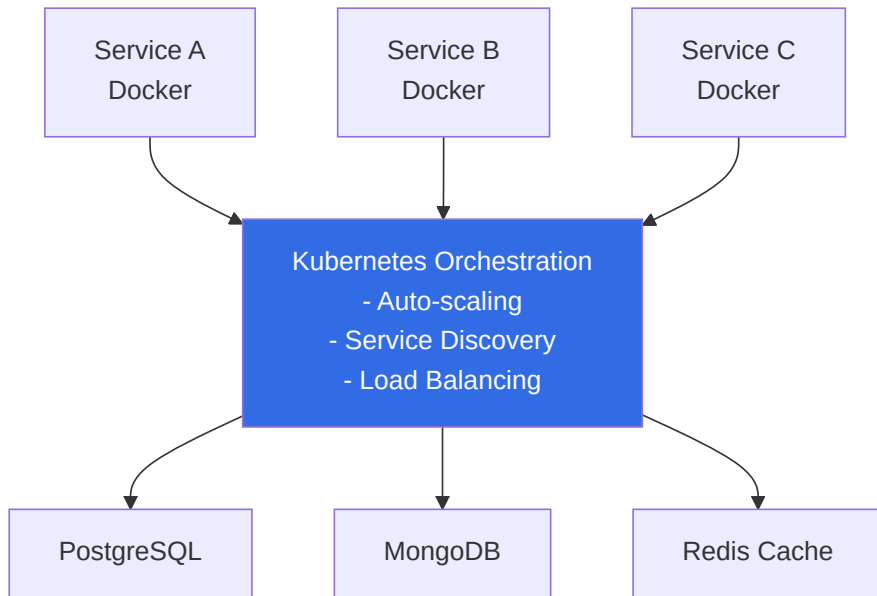
Standard	Usage	Problèmes
SOAP	Intégration complexe	Verbeux, lent
WSDL	Description de services	Difficile à maintenir
XML-RPC	RPC distribué	Pas de typage fort
REST (émergent)	Simplicité HTTP	Pas encore dominant

PROBLÈMES RÉCURRENTS

- ⚠ **Couplage fort** entre services
- ⚠ **Versioning** des APIs très complexe
- ⚠ **Monitoring** difficile à grande échelle

L'Ère Distribuée (2010-2020)

MICROSERVICES & CLOUD NATIVE



PARADIGME SHIFT

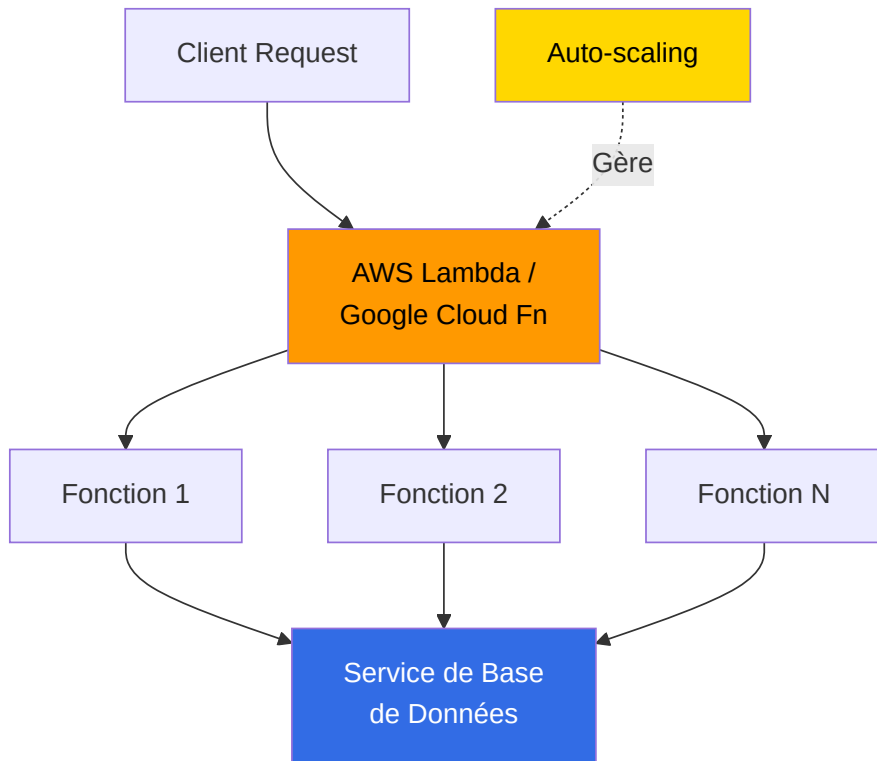
Aspect	Avant	Maintenant
Déploiement	Monolithe unique	Services indépendants
Langage	Homogène	Polyglotte
BD	Base centralisée	BD par service
Communication	RPC synchrone	REST/gRPC asynchrone

OUTILS RÉVOLUTIONNAIRES

 **Docker** - Containerisation  **Kubernetes** - Orchestration  **Prometheus** - Monitoring  **ELK** - Logging centralisé

⚡ Paradigmes Récents (2020+)





SERVERLESS & FUNCTIONS-AS-A-SERVICE



Caractéristiques

- ✓ Pas de gestion d'infrastructure
- ✓ Paiement à l'usage
- ✓ Scaling instantané
- ✗ Vendor lock-in
- ✗ Latence imprévisible
- ✗ Debugging compliqué

AUTRES TENDANCES

-  **IA/ML intégré** dans l'architecture (LLMs, embeddings)
-  **Edge Computing** (calcul près de l'utilisateur)
-  **5G & IoT** (milliards de devices)
-  **Web3 & Blockchain** (architectures décentralisées)



Matrice de Décision

Architecturale

CHOISIR LA BONNE ARCHITECTURE SELON
LE CONTEXTE

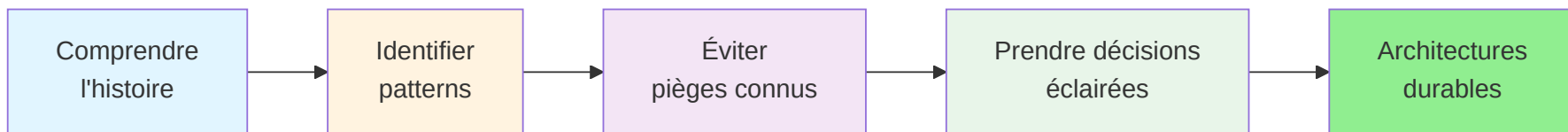
Architecture	Complexité	Coûts Infra	Scalabilité
Monolithe	★ Basse	★ Basse	★ Faible
Microservices	★★★★ Haute	★★★★ Moyenne	★★★★★★ Excellente
Serverless	★★ Moyenne	★★★★ Haute (variable)	★★★★★★ Instantanée

Peu de complexité métier → Monolithe simple
↓
Croissance rapide + équipes → Monolithe modulaire
↓
Scaling horizontal requis → Microservices
↓
Pas de gestion infra + coûts → Serverless
↓
Latence ultra-faible → Edge + On-Premise







De l'Histoire à la Pratique

APPLIQUER L'HISTORIQUE À VOS DÉCISIONS



EXEMPLES DE QUESTIONS À SE POSER

-  Quel stage de maturité pour mon système ?
-  Quelle taille et organisation d'équipe ?
-  Quel budget infrastructure ?
-  Quel taux de croissance prévu ?

L'histoire nous enseigne : il n'y a pas une seule bonne réponse, mais la bonne réponse pour VOTRE contexte.



**Écosystèmes de
développement**

Vue d'ensemble

Les principaux écosystèmes pour développer des applications backend robustes et scalables.

Spring Boot (Java)

CARACTÉRISTIQUES

- **Framework:** Spring Boot (Spring Framework)
- **Langage:** Java (écosystème JVM)
- **Popularité:** ★★★★★ Très populaire en entreprise
- **Courbe d'apprentissage:** Moyenne (concepts entreprise)

POINTS FORTS

- Écosystème riche et mature
- Excellente scalabilité
- Performance élevée
- Nombreuses intégrations
- Transactions ACID robustes

CAS D'USAGE

- Systèmes d'entreprise complexes
- Applications haute disponibilité
- Microservices à grande échelle
- Systèmes financiers



NestJS (Node.js/TypeScript)

CARACTÉRISTIQUES

- **Framework:** Node.js moderne (TypeScript par défaut)
- **Langages:** TypeScript/JavaScript
- **Popularité:** ★★★★★ En croissance rapide
- **Courbe d'apprentissage:** Facile (syntaxe proche d'Angular)

POINTS FORTS

- Développement très rapide
- Partage de code front/back (JavaScript/TS)
- Développement agile
- Excellent pour API REST et GraphQL
- Écosystème npm très riche

CAS D'USAGE

- APIs modernes et scalables
- Applications temps réel (WebSocket)
- Microservices légers
- Projets startup et agiles



Python (FastAPI & Django)

FASTAPI

- **Caractéristiques:** Framework ultra-moderne et performant
- **Avantages:** Développement rapide, documentation automatique
- **Cas d'usage:** APIs légères, Machine Learning, Data Science

DJANGO

- **Caractéristiques:** Framework complet "batteries included"
- **Avantages:** ORM puissant, admin générée, sécurité native
- **Cas d'usage:** Applications web complètes, startups, prototypage rapide

POINTS COMMUNS

- **Popularité:** ★★★★★ Excellente pour l'IA/ML
- **Courbe d'apprentissage:** Facile (syntaxe simple et claire)
- **Productivité:** Très élevée



.NET / ASP.NET Core (C#)

CARACTÉRISTIQUES

- **Framework:** ASP.NET Core (cross-platform)
- **Langage:** C# (langage moderne et puissant)
- **Popularité:** ★★★★★ Très utilisé en entreprise
- **Courbe d'apprentissage:** Moyenne (C# plus complexe que Python)

POINTS FORTS

- Performance exceptionnelle
- Typage fort et sécurité
- Excellent écosystème Microsoft
- Cross-platform (Windows, Linux, macOS)
- Intégration Azure native

CAS D'USAGE

- Applications d'entreprise Windows
- Systèmes critiques
- Solutions sur Azure
- Applications Windows Desktop + Backend



Comparaison Synthétique

Critère	Spring Boot	NestJS	Python	.NET
Vitesse dev	Moyenne	Rapide	Très rapide	Moyenne
Performance	★★★★★	★★★★	★★★	★★★★★
Scalabilité	★★★★★	★★★★	★★★	★★★★★
Courbe d'apprentissage	Moyenne	Facile	Facile	Moyenne
Écosystème	★★★★★	★★★★	★★★★	★★★★★
Entreprise	★★★★★	★★★★	★★★★	★★★★★
Startup/Agile	★★★	★★★★★	★★★★★	★★

Quelle pile choisir?

SPRING BOOT 🙌

- Vous avez une équipe Java expérimentée
- Vous développez un système critique d'entreprise
- Vous avez besoin d'une scalabilité extrême

NESTJS 🙌

- Vous voulez une pile moderne et unifiée (Front/Back en TypeScript)
- Vous développez des microservices
- Vous cherchez un bon équilibre productivité/performance

PYTHON 🙌

- Vous découvrez la programmation backend
- Vous travaillez avec l'IA/ML
- Vous voulez développer très rapidement

.NET 🙌

- Vous êtes dans un environnement Microsoft/Azure
- Vous avez besoin de performance extrême
- Vous développez pour Windows et le web

Défis de l'architecture moderne

Performance

- Latence réduite
- Caching efficace
- Scalabilité

Sécurité

- OAuth2, JWT
- HTTPS, TLS
- Validation des données

Scalabilité

- Horizontal scaling
- Load balancing
- Caching distribué

Maintainabilité

- Documentation
- Tests automatisés
- CI/CD pipeline



 **Patterns
d'Architecture**

Pourquoi utiliser des patterns ?

LES PATTERNS RÉSOLVENTS DES PROBLÈMES RÉCURRENTS

"The purpose of design patterns is to give a name and a context to design problems and their solutions." — **Gang of Four**, Design Patterns

SÉPARATION DES PRÉOCCUPATIONS

🇬🇧 Separation of concerns

Chaque couche a une responsabilité unique et bien définie.

Ce principe, introduit par Edsger Dijkstra en 1974, isole les "préoccupations" (comme la logique métier, l'interface utilisateur ou la persistance des données) pour réduire la complexité et les interdépendances. Cela permet de modifier une partie sans impacter les autres.

Avantages pratiques :

- Maintenabilité : Les changements sont localisés, facilitant la maintenance et le débogage.
- Réutilisabilité : Les modules isolés peuvent être réemployés ailleurs.
- Testabilité : Chaque composant se teste indépendamment, avec moins de cas à couvrir.

Présentation (UI)



Logique métier (Règles de gestion)



Accès aux données (Persistance)



Infrastructure (Serveurs, BD)

Pattern Dependency Injection (DI)

Injecter les dépendances plutôt que les créer soi-même.

SANS DEPENDENCY INJECTION (COUPLAGE FORT):

```
public class ContractService {  
    private DatabaseService db = new DatabaseService(); // Couplage fort  
  
    public void createContract(Contract c) {  
        db.save(c);  
    }  
}
```

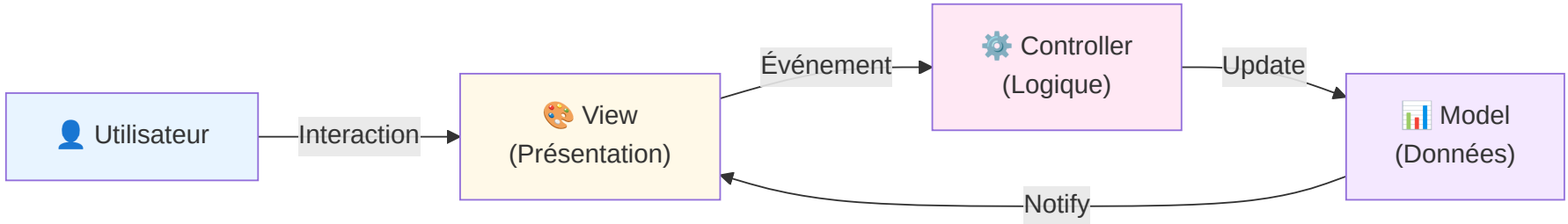
AVEC DEPENDENCY INJECTION (DÉCOUPLAGE):

```
public class ContractService {  
    private DatabaseService db; // Interface  
  
    @Inject // Spring  
    public ContractService(DatabaseService db) {  
        this.db = db;  
    }  
}
```

Pattern MVC (Model-View-Controller)

SÉPARATION DES RESPONSABILITÉS:

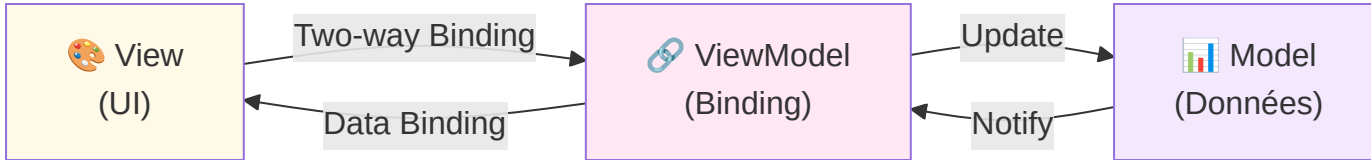
- Model: Données et logique métier
- View: Présentation et interface utilisateur
- Controller: Coordination et gestion des événements



Pattern MVVM (Model-View-ViewModel)

CARACTÉRISTIQUES:

- Binding bidirectionnel: Sync automatique View ↔ ViewModel
- Testabilité: ViewModel indépendant de la Vue
- Réactivité: Mises à jour temps réel



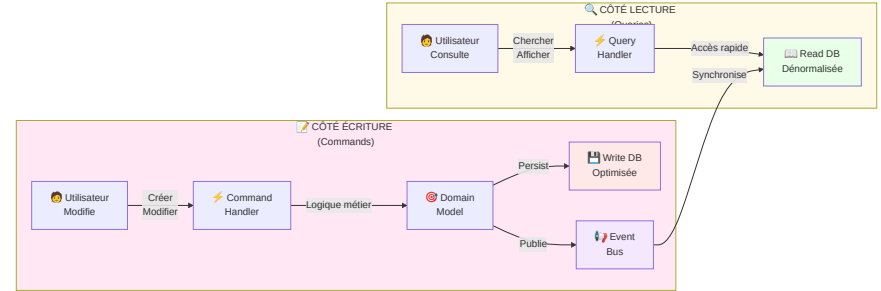
Pattern CQRS (Command Query Responsibility Segregation)

CONCEPT CLÉ

Séparer les modèles de lecture et écriture pour optimiser chacun indépendamment.

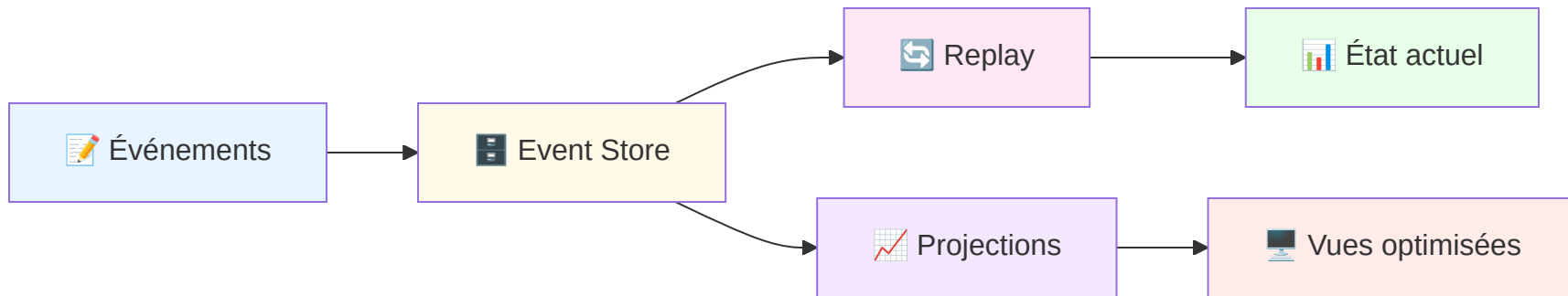
AVANTAGES

- o **✓ Optimisation indépendante:** Chaque modèle optimisé pour son usage
- o **✓ Scalabilité:** Lectures et écritures peuvent être déployées séparément
- o **✓ Performance:** Read DB peut être dénormalisée (cache, index spécifiques)
- o **✓ Clarté:** Séparation claire des responsabilités



Architecture Event-Sourcing

PRINCIPES FONDAMENTAUX



CONCEPTS CLÉS

- **Événements immutables:** Tous les changements sont stockés comme événements
- **Reconstruction d'état:** L'état actuel est reconstruit en replayant les événements
- **Projections:** Vues optimisées pour différents cas d'usage
- **Audit trail:** Historique complet de toutes les modifications

CAS D'USAGE

- **Finance**: Traçabilité complète des transactions
- **Assurance**: Historique des contrats et sinistres
- **Santé**: Dossiers patients avec historique complet

OUTILS POPULAIRES

- **EventStoreDB**: Base de données dédiée
- **Kafka**: Pour le streaming d'événements
- **Axoni**: Plateforme complète

Comparaison Event-Sourcing vs CRUD

Aspect	Event-Sourcing	CRUD Traditionnel
Historique	✓ Complet	✗ Partiel
Audit	✓ Natif	✗ Requieret logs
Performance lecture	✗ Replay nécessaire	✓ Direct
Complexité	⚠ Élevée	✓ Simple
Évolutivité	✓ Excellente	⚠ Limitée

Pattern Event-Driven Architecture

CAS D'USAGE ASSURANCE:

Services réactifs aux événements métiers asynchrones.

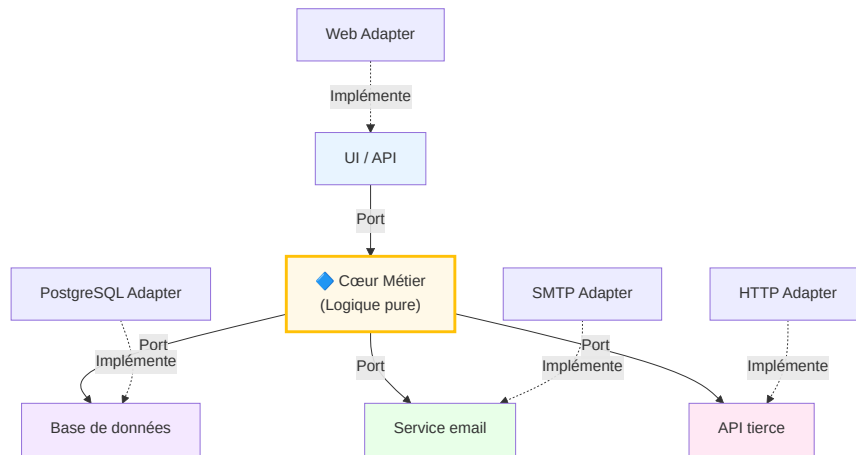
- Événement: "ContractCreated" - Un nouveau contrat est créé
- Consommateurs: Service email (notification), Service CRM (update), Service audit (logging)
- Avantage: Découplage complet entre services

Pattern Hexagonal (Ports & Adapters)

BÉNÉFICES:

Isoler le cœur métier des détails techniques.

- Cœur métier indépendant des frameworks
- Adaptation facile aux changements technologiques
- Tests unitaires sans dépendances externes



Pattern Repository

AVANTAGES:

Abstraction de la couche d'accès aux données.

- Logique métier indépendante du mécanisme de persistance
- Facile de basculer de PostgreSQL à MongoDB
- Tests unitaires avec implémentation mock

Récapitulatif: Quand utiliser quel pattern ?

Pattern	Problème	Quand l'utiliser
MVC	Séparation UI/logique	Web traditionnel, applications simples
MVVM	Binding bidirectionnel	Interfaces réactives, desktop/mobile
CQRS	Scalabilité lecture/écriture	Hauts volumes, complex queries
Event-Driven	Découplage asynchrone	Microservices, systèmes réactifs
Hexagonal	Isolation cœur métier	Logique métier complexe, DDD
DI	Gestion dépendances	Tous les projets modernes



Transactions



Transactions en Backend

Introduction aux Transactions

QU'EST-CE QU'UNE TRANSACTION?





Une transaction est une **séquence d'opérations** qui doit s'exécuter en totalité ou pas du tout.

"Un paiement est soit accepté complètement, soit rejeté en totalité - jamais partiellement."

PROPRIÉTÉS ACID (FONDAMENTALES)

Propriété	Signification	Assurance
A tomicité	Tout ou rien	Pas de paiement partiel
C ohérence	État valide avant/après	Soldes toujours corrects
I solation	Transactions indépendantes	Pas de lecture sale
D urabilité	Persistance garantie	Pas de perte de données





CAS D'USAGE ASSURANCE

-  Création de contrat + enregistrement prime
-  Sinistre + déblocage indemnisation
-  Transfert de fonds entre comptes
-  Mise à jour risque + calcul cotisation

Problèmes sans Transactions

SCÉNARIOS CATASTROPHIQUES

Scénario: Achat d'assurance avec paiement

1.  Prime débitée du compte client (-500€)
2.  ERREUR BASE DE DONNÉES
3.  Contrat NON créé
4.  Prime perdue (ou non enregistrée)

→ Client a payé mais pas de contrat!

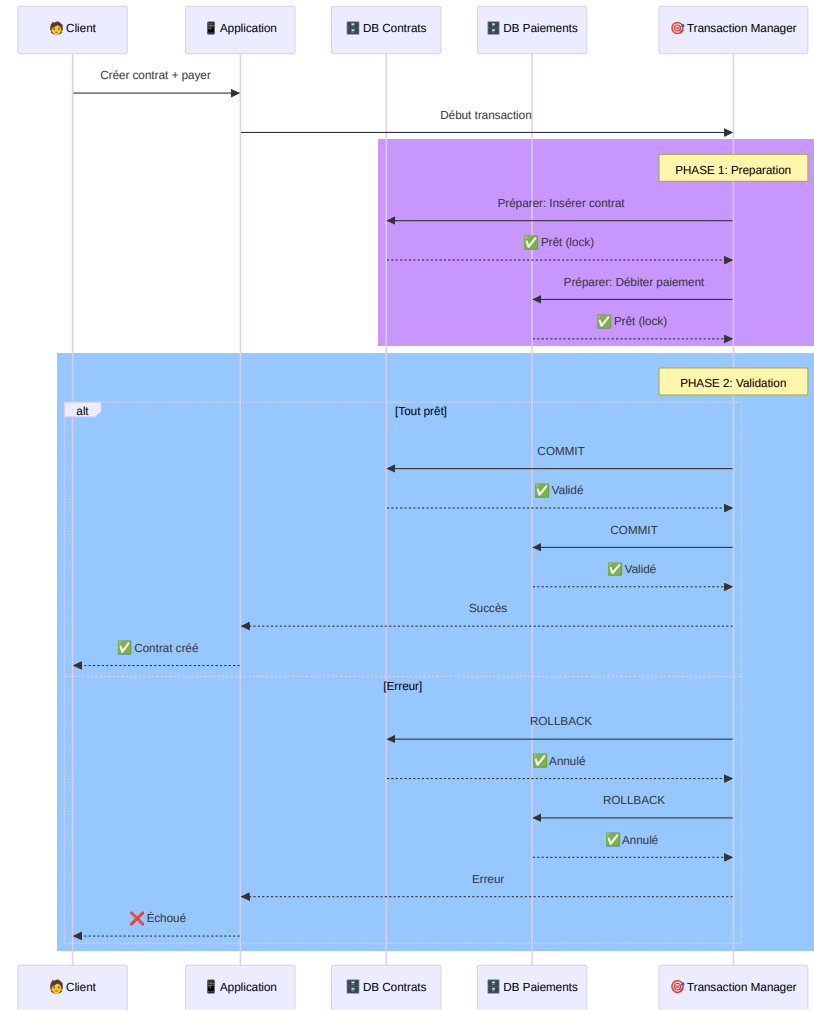
→ Risque juridique et financier énorme

SANS ACID (BASE DE DONNÉES SIMPLE)

- Lecture sale: Lire une donnée non validée
- Modification perdue: Deux écritures simultanées
- Violation de contrainte: Somme = 0, mais montants = -50 et 100
- Crash pendant mise à jour: État inconsistant













2-Phase Commit (2PC)

FONCTIONNEMENT SCHÉMATIQUE :



Niveaux d'Isolation

4 NIVEAUX D'ISOLATION SQL STANDARDISÉS (ANSI SQL) :

Niveau	Lecture sale	Non-Répétable	Fantôme
READ UNCOMMITTED	 Oui	 Oui	 Oui
READ COMMITTED	 Non	 Oui	 Oui
REPEATABLE READ	 Non	 Non	 Oui
SERIALIZABLE	 Non	 Non	 Non

DÉFINITIONS

- **Lecture sale**: Lire une donnée non commitée (peut être annulée)
- **Non-Répétable**: Deux lectures différentes de la même donnée
- **Fantôme**: Lignes qui apparaissent/disparaissent entre lectures

PostgreSQL par exemple supporte les quatre niveaux, et utilise Read Committed par défaut pour éviter les lectures sales sans trop de verrouillages.

Implémentation dans les frameworks

SPRING BOOT (JAVA)

```
@Service
@Transactional // ← Gère les transactions automatiquement
public class ContractService {
    @Transactional(propagation = Propagation.REQUIRED,
                    isolation = Isolation.REPEATABLE_READ)
    public void createContractWithPayment(Contract c, Payment p) {
        contractRepository.save(c);          // Insert contrat
        paymentRepository.save(p);           // Débiter paiement
        // ✅ COMMIT automatique si pas d'exception
        // ❌ ROLLBACK automatique si exception
    }
}

@Transactional // Gestion d'erreur
public void transfer(Account from, Account to, double amount) {
    try {
        from.withdraw(amount);    // -500
        to.deposit(amount);       // +500
        accountRepo.save(from);
        accountRepo.save(to);
    } catch (Exception e) {
        // Rollback automatique, soldes intacts
        throw new TransactionException("Transfert échoué");
    }
}
```

NESTJS (NODE.JS/TYPESCRIPT)

```
// Avec TypeORM
@Injectable()
export class ContractService {
  constructor(
    private dataSource: DataSource,
    private contractRepo: Repository<Contract>
  ) {}

  async createContractWithPayment(
    contract: Contract,
    payment: Payment
  ) {
    const queryRunner = this.dataSource.createQueryRunner();
    await queryRunner.connect();
    await queryRunner.startTransaction();

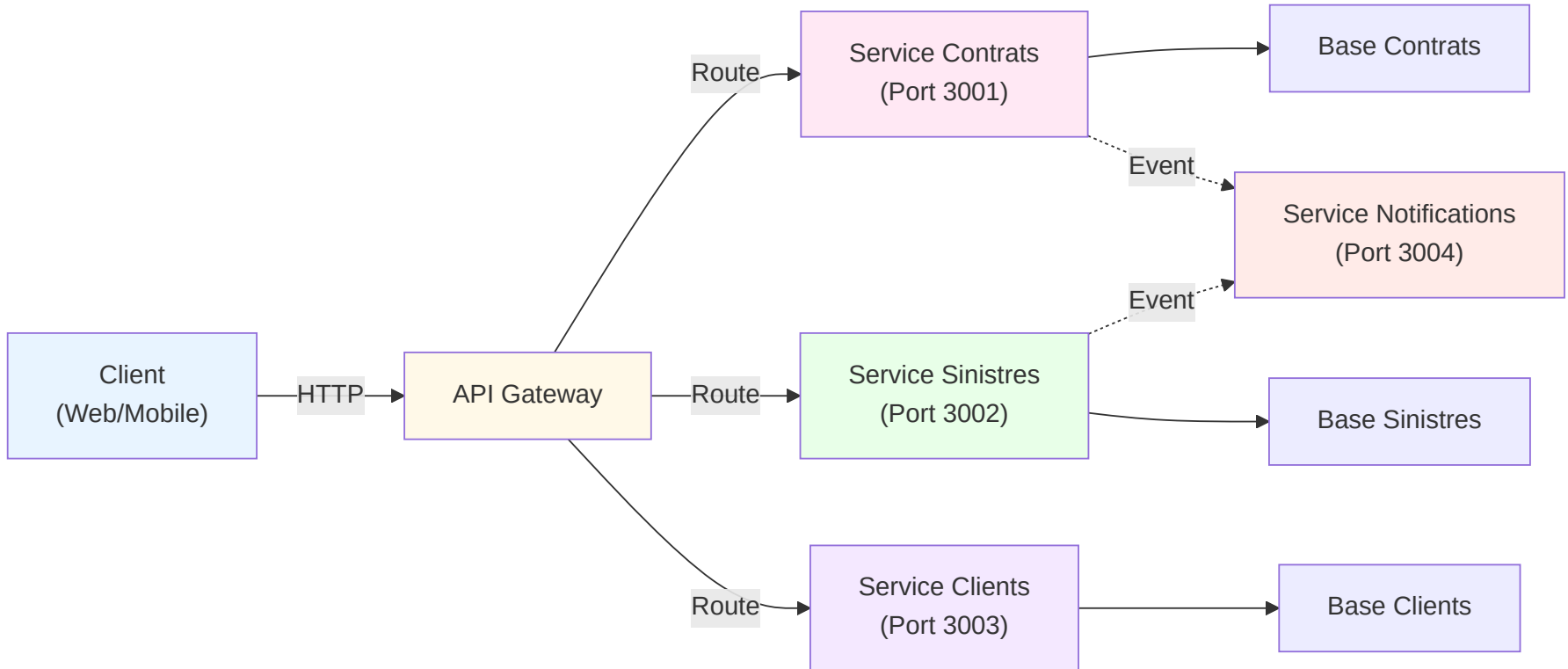
    try {
      await queryRunner.manager.save(contract);
      await queryRunner.manager.save(payment);
      await queryRunner.commitTransaction();
    } catch (err) {
      await queryRunner.rollbackTransaction();
      throw new Error('Transaction failed');
    } finally {
      await queryRunner.release();
    }
  }
}
```



Microservices

Microservices: Introduction

Architectures distribuées basées sur des services indépendants.



Caractéristiques des Microservices

PROPRIÉTÉS CLÉS:

Autonomie

- Services indépendants
- Déploiement indépendant
- BD dédiée
- Équipes autonomes

Communication

- API REST / gRPC
- Message brokers (Kafka)
- Events asynchrones
- Découverte de services

Résilience

- Circuit breaker
- Timeout
- Retry policy
- Health checks

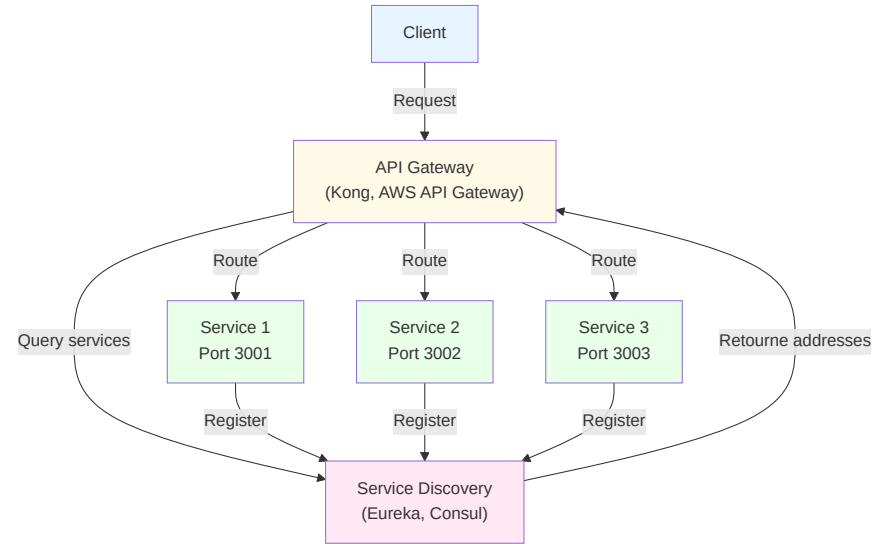
Observabilité

- Logging distribué
- Tracing
- Monitoring
- Alerting

API Gateway et Service Discovery

API GATEWAY (POINT D'ENTRÉE UNIQUE):

- Routage: Diriger requêtes aux services corrects
- Authentification: JWT validation
- Rate limiting: Protection DOS
- Caching: Réduire latence
- Load balancing: Distribuer charge



Communication inter-services

APPROCHES DE COMMUNICATION:

● Synchrone (REST/gRPC)

```
Service A
  ↓ (HTTP/gRPC)
Service B
  ↓ (attend réponse)
Service C
  ↓
Réponse retourne
```

Avantages:

- ✓ Cohérence immédiate
- ✓ Facile à déboguer

Inconvénients:

- ✗ Couplage fort
- ✗ Service lent = tout lent

● Asynchrone (Events)

```
Service A
  ↓ (Publie event)
Kafka/RabbitMQ
  ↓ (Message broker)
Service B (reçoit)
Service C (reçoit)
```

Avantages:

- ✓ Découplage complet
- ✓ Haute disponibilité
- ✓ Scalabilité

Inconvénients:

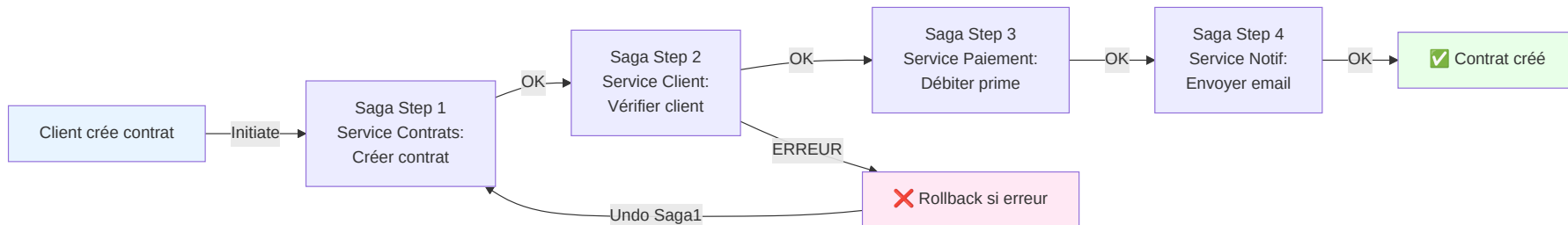
- ✗ Eventual consistency
- ✗ Plus complexe

Saga Pattern: Transactions distribuées

Deux approches:

Maintenir la cohérence des données sur plusieurs services

- Choreography: Services écoutent les events et réagissent (loose coupling)
- Orchestration: Service central coordonne les étapes (plus simple mais couplage)

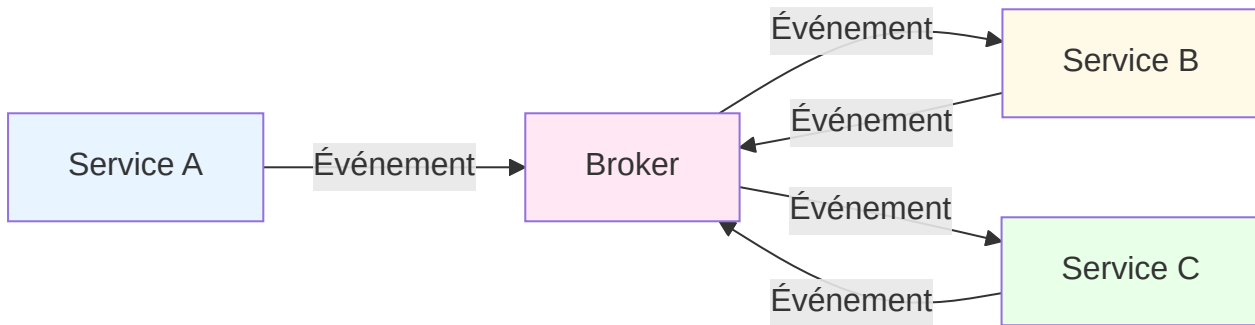




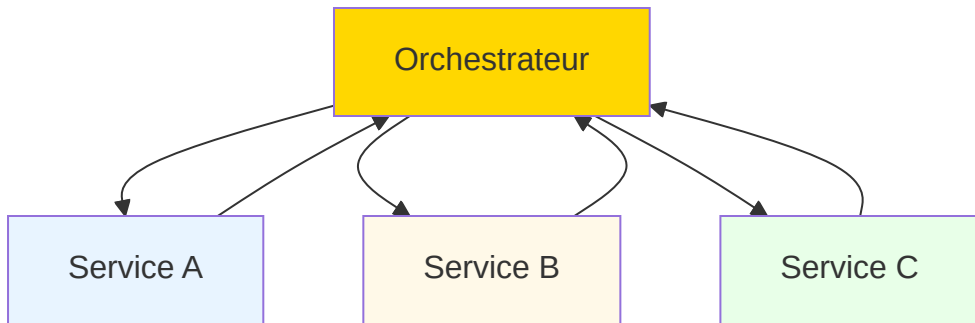
Microservices: Choreography vs Orchestration

Définitions et Comparaison

CHOREOGRAPHY



ORCHESTRATION



Critères de Choix

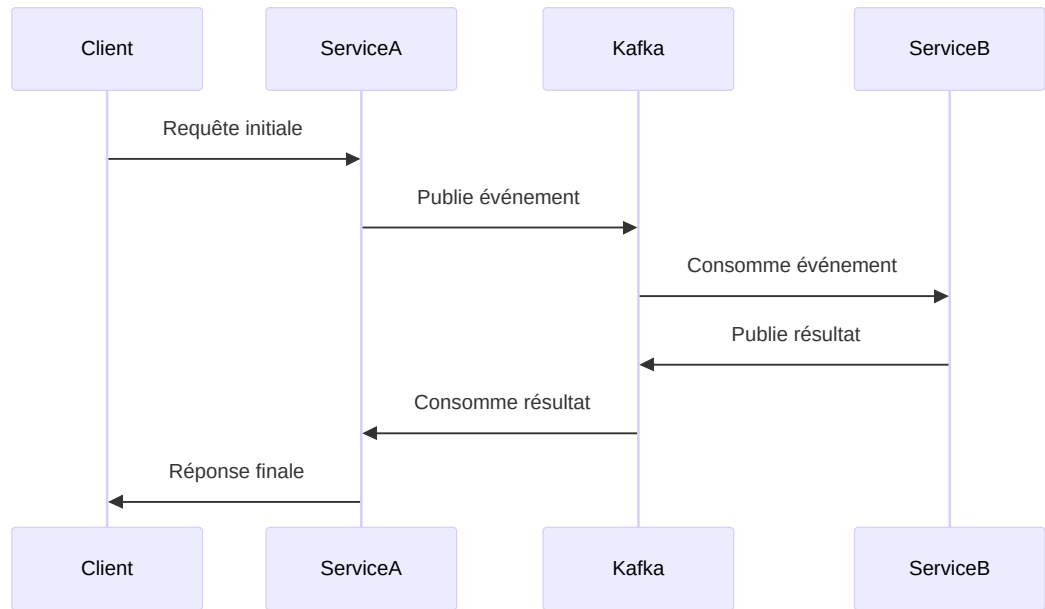
Critère	Choreography	Orchestration
Couplage	✓ Faible	✗ Fort
Complexité	⚠ Élevée	✓ Modérée
Flexibilité	✓ Élevée	⚠ Limitée
Visibilité	✗ Difficile	✓ Claire
Maintenance	✗ Complexe	✓ Simple

OUTILS POPULAIRES

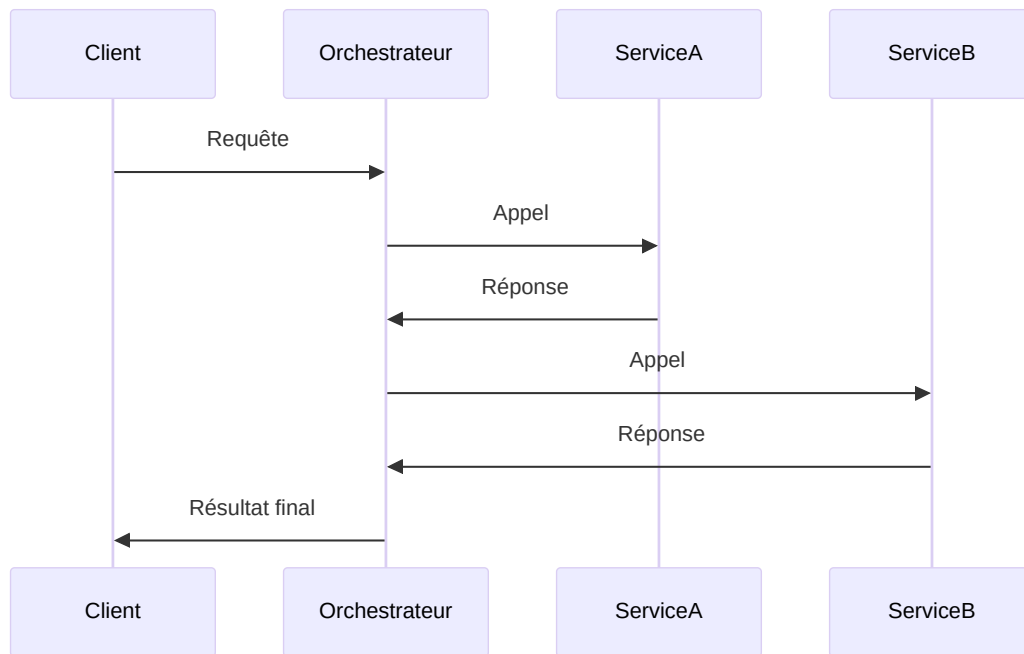
- **Choreography**: Kafka, RabbitMQ, AWS EventBridge
- **Orchestration**: Zeebe, Cadence, AWS Step Functions

Implémentation Pratique

CHOREOGRAPHY AVEC KAFKA



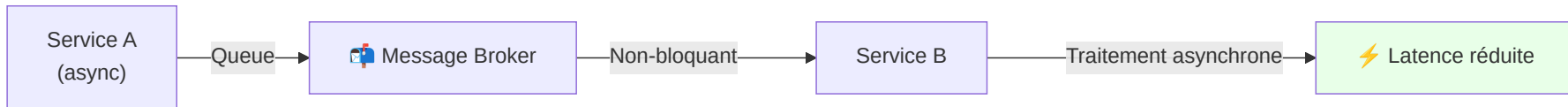
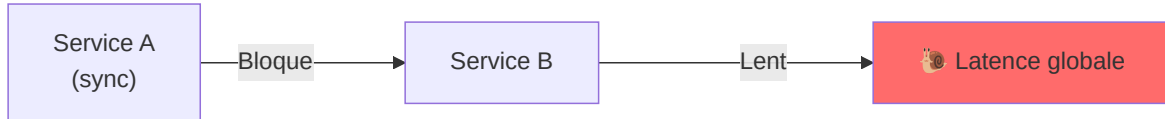
ORCHESTRATION AVEC ZEEBE









 **Messaging
Asynchrone**

Pourquoi Async Messaging ?



AVANTAGES

-  **Découplage**: Services indépendants, pas de dépendance directe
-  **Résilience**: Si consommateur down, messages en attente
-  **Scalabilité**: Producteur rapide, consommateurs à leur rythme
-  **Buffering**: Pics de charge absorbés par la queue

EXEMPLE SPRING: ENVOI & RÉCEPTION

```
@Configuration
@EnableJms
public class JmsConfig {

    @Bean
    public JmsTemplate.jmsTemplate(ConnectionFactory connectionFactory) {
        return new JmsTemplate(connectionFactory);
    }
}

// Producteur
@Component
public class OrderProducer {

    @Autowired
    private JmsTemplate.jmsTemplate;

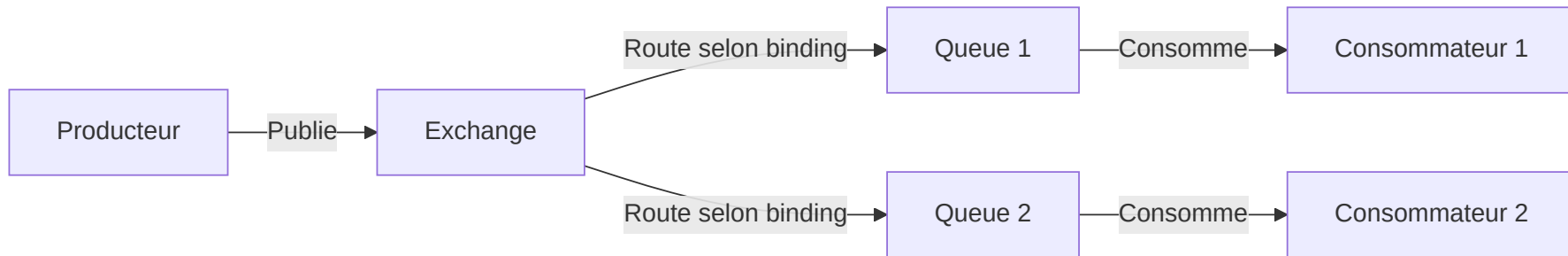
    public void sendOrder(Order order) {
       .jmsTemplate.convertAndSend("order-queue", order);
    }
}

// Consommateur
@Component
public class OrderConsumer {

    @JmsListener(destination = "order-queue")
```

AMQP (Advanced Message Queuing Protocol)

ARCHITECTURE



CONCEPTS CLÉS

- **Exchange**: Reçoit messages, les route selon type (direct, topic, fanout)
- **Queue**: Stocke messages
- **Binding**: Règle qui lie exchange ↔ queue

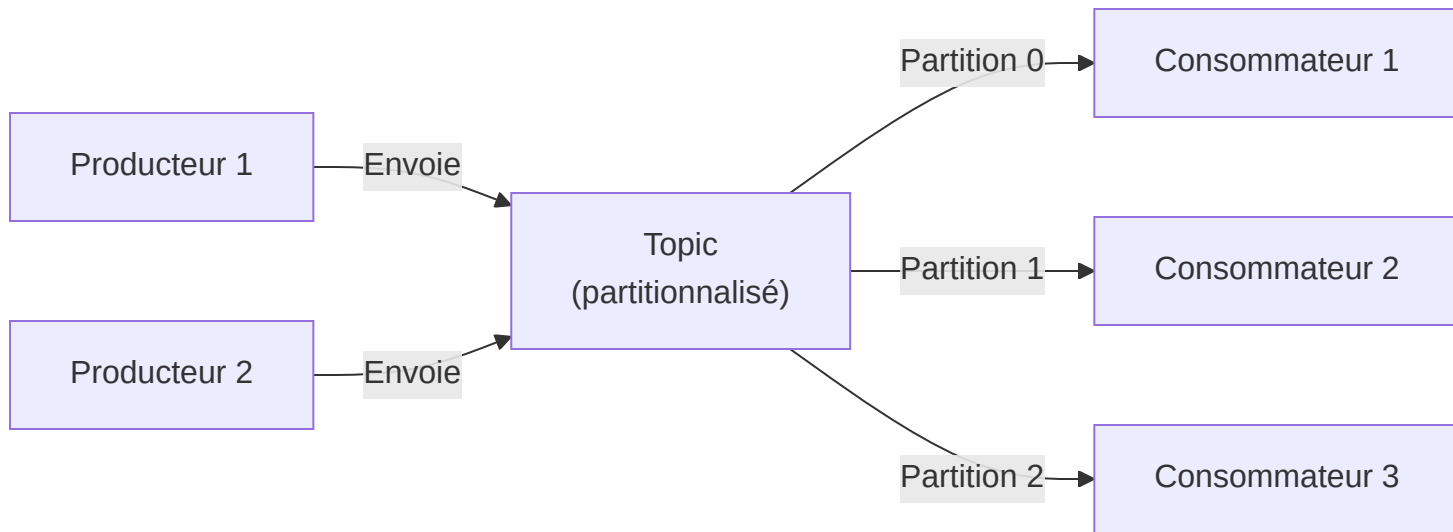
EXEMPLE SPRING: RABBITMQ

```
@Configuration
public class RabbitConfig {

    // Déclarer queue, exchange, binding
    @Bean
    public Queue orderQueue() {
        return new Queue("order.queue", true); // durable
    }
}
```

Apache Kafka

ARCHITECTURE



CONCEPTS CLÉS

- **Topic**: Flux d'événements partitionné
- **Partition**: Garantit ordre dans partition; scalabilité horizontale
- **Consumer Group**: Groupe de consommateurs, parallélisation
- **Offset**: Position dans le stream; replay possible







Comparaison: JMS vs AMQP vs Kafka

Critère	JMS	AMQP (RabbitMQ)	Kafka
Latence (p50)	5-50 ms	1-10 ms	5-20 ms
Throughput	10k-100k msg/s	100k-1M msg/s	100k-1M+ msg/s
Persistence	Configurable	✅ Queue durable	✅ Durée configurable
Replay	❌ Non	❌ Non	✅ Oui (offset)
Ordering	Par queue	Par queue	Par partition
Cas d'usage	Legacy, transactions	RPC, RPC-like	Event streaming, logs

Recommandations

Scénario	Choix	Raison
API synchrone, transactionnel	✗ Ne pas utiliser	Async != sync
Microservices, RPC-like	RabbitMQ (AMQP)	Routing flexible, latence basse
Event sourcing, audit log	Kafka	Replay, durabilité, scalabilité
Legacy J2EE, ACID garanti	JMS + ActiveMQ	Écosystème établi
Pics de charge, buffering	Kafka ou RabbitMQ	Absorber surcharge

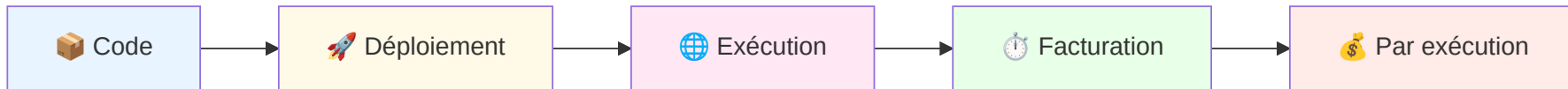
Bonnes pratiques

-  **Idempotence**: Consommateur doit gérer duplicates
-  **Dead Letter Queue**: Gérer messages non-traités
-  **Monitoring**: Lag de queues, consumer lag (Kafka)
-  **Sérialisation**: JSON ou Avro (schéma évolutif)
-  **Timeouts**: Configurer selon latence réseau
-  **Partitionnement** (Kafka): Clé = stable pour ordre



Principes du Serverless

CARACTÉRISTIQUES CLÉS



AVANTAGES

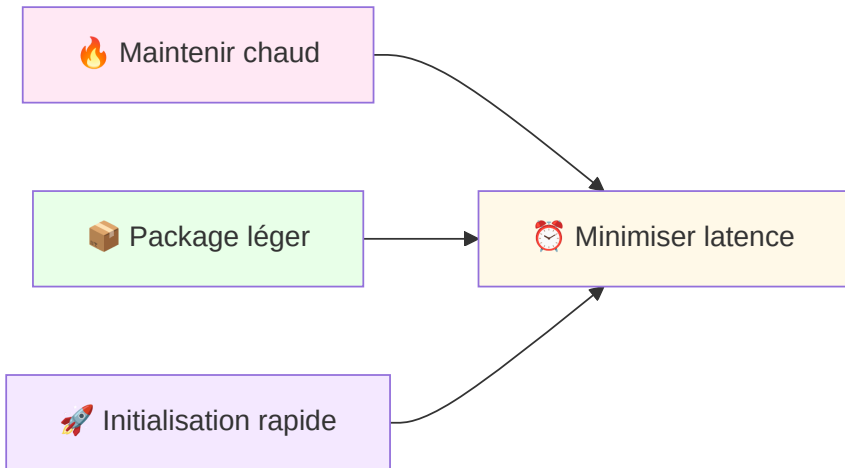
- **Pas de gestion serveur**: Focus sur le code métier
- **Scalabilité automatique**: Gestion transparente de la charge
- **Facturation précise**: Pay-as-you-go
- **Déploiement rapide**: Mise en production instantanée

DÉFIS

- **Cold starts**: Latence initiale
- **Timeouts**: Limites d'exécution
- **Vendor lock-in**: Dépendance au fournisseur cloud

Patterns Serverless Avancés

1. COLD START OPTIMIZATION



2. COMPOSITION DE FONCTIONS

Comparaison des Fournisseurs Cloud

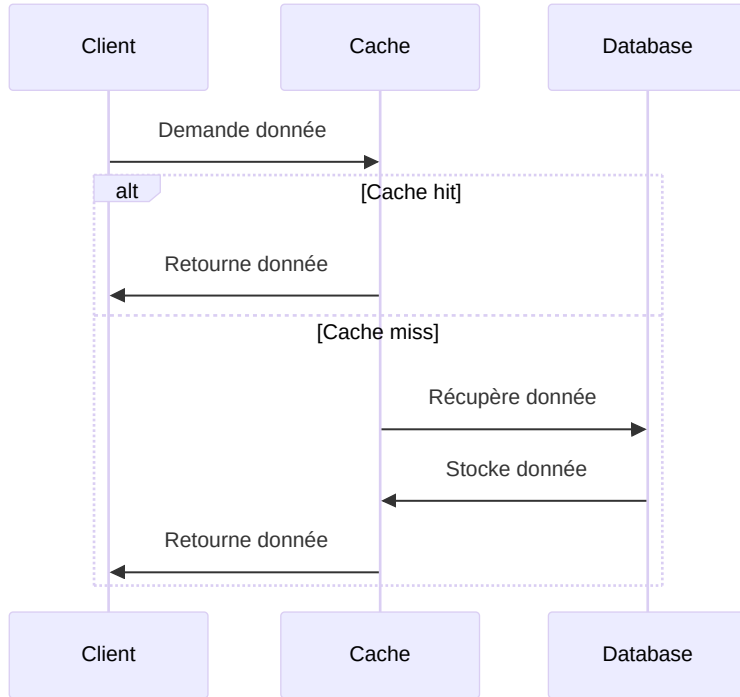
Fournisseur	Service	Langages	Timeout Max	Points forts
AWS	Lambda	Node, Python, Java, Go	15 min	Écosystème complet
Azure	Functions	C#, JavaScript, Python	10 min	Intégration Microsoft
Google	Cloud Functions	Node, Python, Go	9 min	Scalabilité rapide
Cloudflare	Workers	JavaScript	30 sec	Edge computing



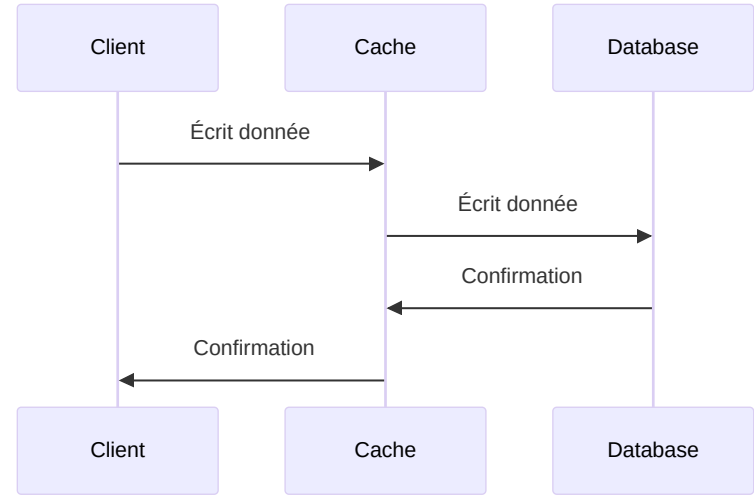
Caching Avancé

Patterns de Cache

CACHE-ASIDE (LAZY LOADING)



WRITE-THROUGH



-> le cache est toujours à jour

Stratégies d'Invalidation

1. TIME-BASED (TTL)



2. EVENT-BASED



Comparaison Redis vs Memcached

Critère	Redis	Memcached
Persistence	✓ Oui	✗ Non
Structures	✓ Riches	✗ Clé-valeur
Réplication	✓ Master-Slave	✗ Basique
Performance	⚠ Très élevée	✓ Extrême
Utilisation	Cache + BD	Cache pur

CAS D'USAGE

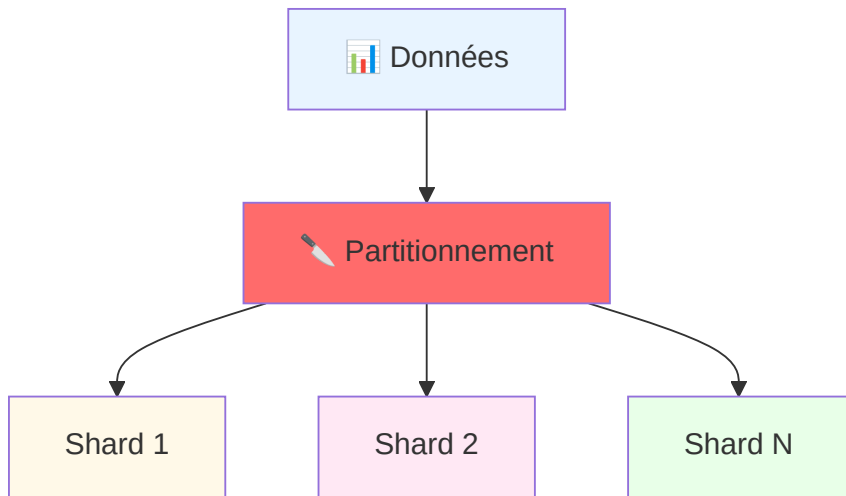
- **Redis**: Sessions, leaderboards, pub/sub
- **Memcached**: Cache simple, performances pures



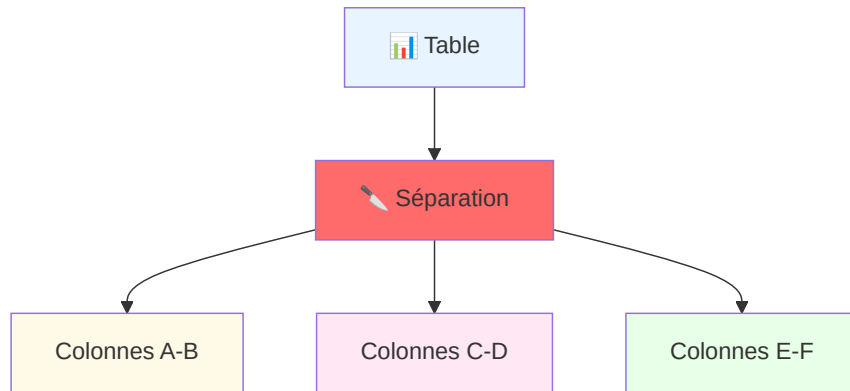
Database Sharding et Partitioning

Définitions

SHARDING HORIZONTAL

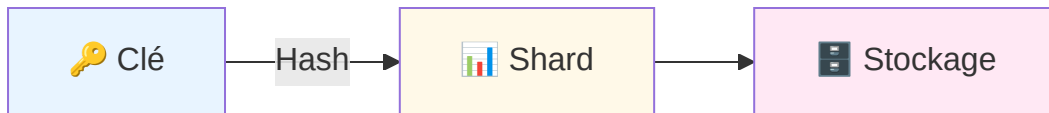


PARTITIONING VERTICAL

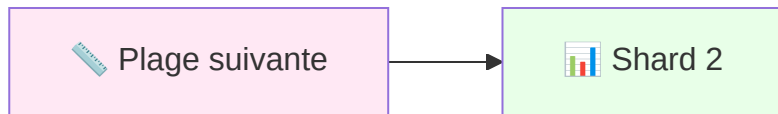


Stratégies de Sharding

1. KEY-BASED SHARDING

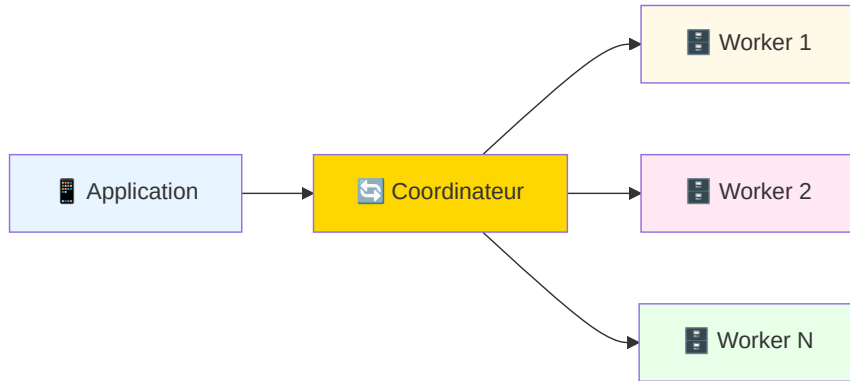


2. RANGE-BASED SHARDING

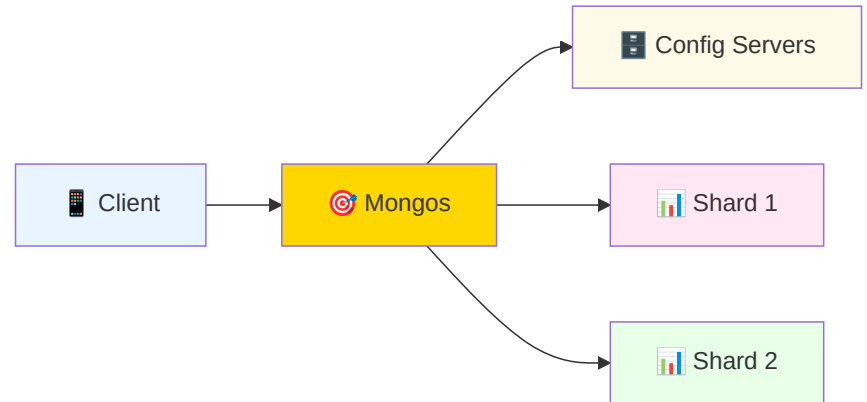


Implémentation Pratique

POSTGRESQL AVEC CITUS



MONGODB SHARDING





Domain-Driven Design

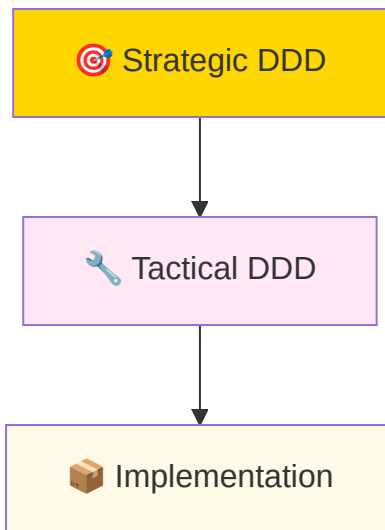
Pourquoi le DDD ?

Au début des années 2000, les projets logiciels complexes souffraient souvent d'un fossé entre besoins métier et implémentations techniques, malgré l'essor de l'orienté objet.

Eric Evans synthétise ses expériences pour proposer un cadre structuré centré sur le domaine métier, langage ubiquitaire et modélisation collaborative, influencé par l'agilité naissante (XP) et visant à rendre les gros systèmes maintenables.

Strategic vs Tactical DDD

NIVEAUX DE DDD



STRATEGIC DDD

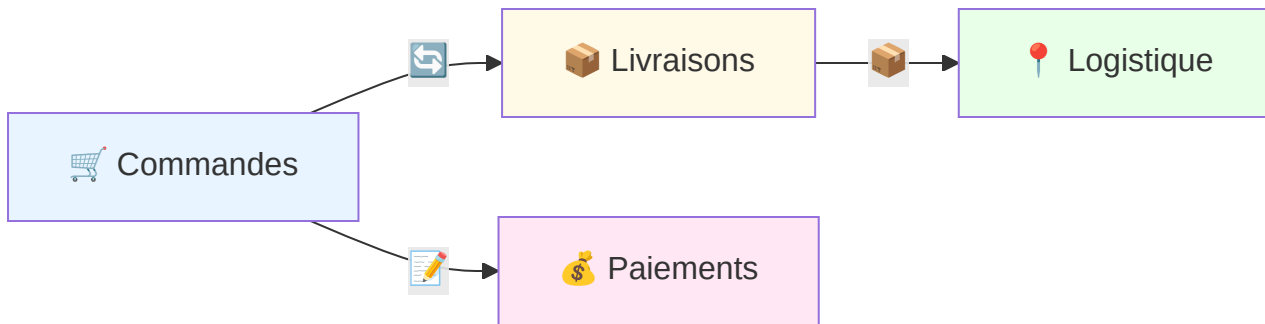
- **Bounded Contexts:** Frontières claires
- **Context Mapping:** Relations entre contextes
- **Ubiquitous Language:** Langage commun

TACTICAL DDD

- **Aggregates:** Cohérence transactionnelle
- **Domain Events:** Communication asynchrone
- **Entities vs Value Objects:** Modélisation fine

Bounded Contexts et Context Mapping

EXEMPLE D'ARCHITECTURE



TYPES DE RELATIONS

- Shared Kernel : Deux équipes partagent un petit morceau de code métier commun (ex. : définition d'un "Client").
- Customer/Supplier : Un module utilise l'API d'un autre (le "fournisseur"), en acceptant ses règles.
- Partnership : Collaboration étroite bidirectionnelle entre deux modules égaux.
- Conformist : Un module s'adapte complètement aux règles d'un dominant.
- Anticorruption Layer : Une "barrière" traduit un modèle externe vers le vôtre, sans pollution.

Event Storming

PROCESSUS COLLABORATIF

ÉTAPES CLÉS

1. **Événements métiers**: "CommandePayée",
"LivraisonPlanifiée"
2. **Commandes**: Actions déclenchantes
3. **Aggregates**: Groupes cohérents
4. **Bounded Contexts**: Frontières logiques

Exemple (source : <https://draft.io/fr/example/eventstorming>)

Récapitulatif DDD

AVANTAGES

- **Alignement métier / tech**: Langage commun
- **Modularité**: Contextes indépendants
- **Maintenabilité**: Modèle clair
- **Évolutivité**: Adaptation facile



✨ **Clean Code &
Architecture**

Clean Code: Introduction

POURQUOI LE CLEAN CODE ?

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." — **Robert C. Martin**, Clean Code

Code sale vs Code propre

❌ Code sale (mauvais)

```
function calc(c) {  
    let p = 0;  
    if (c.age < 25)  
        p = c.sal * 0.15;  
    else if (c.age < 65)  
        p = c.sal * 0.1;  
    else  
        p = c.sal * 0.2;  
  
    // TODO: ajouter taxes  
    // FIXME: bug ici  
    return p;  
}
```

✅ Code propre (bon)

```
double calculateInsurancePremium(  
    Customer customer) {  
    int age = customer.getAge();  
    double salary = customer.getSalary();  
  
    PremiumRate rate =  
        determinePremiumRate(age);  
  
    return salary * rate.getPercentage();  
}
```

Règle 1: Nommage clair

NOMS RÉVÉLATEURS D'INTENTION

✗ Mauvais

✓ Bon

Raison

d

elapsedTimeInDays

Spécifique et clair

calcP()

calculatePremium()

Verbe + nom explicite

list1, list2

activeContracts, expiredContracts

Contexte et utilité clairs

Manager

ContractManager

Plus précis et domaine-spécifique

Règle 2: Fonctions courtes (SRP)

Single Responsibility Principle: Une fonction = une seule raison de changer

FONCTION TROP GROSSE (MAUVAIS):

```
public void processContract(Contract c) {  
    // Validation  
    if (c.getSalary() < 0) throw new Exception(...);  
  
    // Calcul de prime  
    double premium = c.getSalary() * 0.1;  
  
    // Enregistrement  
    database.save(c);  
  
    // Envoi email  
    emailService.send(c.getEmail(), premium);  
  
    // Logging  
    logger.info("Contrat traité: " + c.getId());  
}
```

Fonctions courtes et focalisées (bon):

```
public void processContract(Contract c) {  
    validateContract(c);  
    double premium = calculatePremium(c);  
}
```

Règle 3: Gestion des erreurs

PRÉFÉRER LES EXCEPTIONS AUX CODES DE RETOUR:

✗ Code de retour

```
int status =
    contractService.save(c);

if (status == 0) {
    System.err.println("Erreur!");
} else if (status == 1) {
    System.out.println("Saved");
}
```

✓ Exception

```
try {
    contractService.save(c);
    logger.info("Contrat sauvé");
} catch (
    InvalidContractException e) {
    logger.error(
        "Contrat invalide: "
        + e.getMessage()
    );
}
```

Règle 4: DRY (Don't Repeat Yourself)

Éliminer les répétitions de code.

✗ Code répété

```
// ContractService
double premium = salary * 0.1;
if (premium < 100) premium = 100;
return premium;

// CustomerService
double amount = salary * 0.1;
if (amount < 100) amount = 100;
return amount;

// BenefitService
double benefit = salary * 0.1;
if (benefit < 100) benefit = 100;
return benefit;
```

✓ Extraction en méthode

```
// PricingCalculator
private double calculateAmount(
    double salary) {
    double amount = salary * 0.1;
    return Math.max(amount, 100);
}
```

Règle 5: Commentaires

Le code doit se commenter lui-même. Les commentaires ne doivent expliquer que le POURQUOI, pas le QUOI.

❌ Commentaires inutiles

```
// Incrémenter i
i++;

// Vérifier si la liste
// n'est pas vide
if (list.size() > 0) {
    // Boucler sur les éléments
    for (Item item : list) {
        // Ajouter à total
        total += item.getValue();
    }
}
```

✅ Commentaires utiles

```
// Limite minimale définie par
// la régulation assurance (2024)
final double MINIMUM_PREMIUM = 100;

// Algorithme de pricing Bayésien
// basé sur historique client
// Source: ACME-2023 Paper
private double
    calculateAdaptivePremium(
        Customer c) {
    ...
}
```

Règle 6: Formatage et style

LA COHÉRENCE EST CLÉ






- Indentation: 2 ou 4 espaces (pas de tabs)
- Longueur de ligne: Max 100-120 caractères
- Noms de classes: PascalCase (ContractService)
- Noms de variables: camelCase (myVariable)
- Noms de constantes: UPPER_SNAKE_CASE (MAX_SIZE)
- Espaces: Autour des opérateurs ($x = y + z$)

Règle 7: Testabilité

PROPRIÉTÉS D'UN CODE TESTABLE:

Exemple: Test unitaire simple

Code testable = code découplé

-  Dépendances injectées (pas "new Database()")
-  Logique métier indépendante du framework
-  Pas de singletons globaux
-  Pas d'appels à des APIs externes en dur
-  Méthodes courtes et déterministes

```
@Test
public void testCalculatePremiumForYoungDriver() {
    Customer young = new Customer(20, 30000);
    double premium = service.calculatePremium(young);
    assertEquals(4500, premium, 0.01);
}
```

Récapitulatif: quelques règles du Clean Code

#	Règle	Bénéfice
1	Nommage clair	Comprendre rapidement l'intention
2	Fonctions courtes (SRP)	Facile à tester et maintenir
3	Gestion des erreurs	Code plus lisible et robuste
4	DRY (pas de répétition)	Modifications en un seul endroit
5	Commentaires utiles	Comprendre le POURQUOI
6	Formatage cohérent	Équipe sur la même longueur d'onde
7	Testabilité	Confiance dans le code

Clean Architecture: Introduction

"A software architect is a programmer who has stopped programming and has started thinking about programs." — **Robert C. Martin**, Clean Architecture

Les 4 couches de Clean Architecture 1/2

1 Entities (Cœur métier)

Objets métiers purs, pas de frameworks

```
public class Contract {  
    private String id;  
    private Customer customer;  
    private double premium;  
    public boolean isValid() {  
        return premium > 0  
            && customer != null;  
    }  
}
```

2 Use Cases (Logique applicative)

Règles métier spécifiques à l'app

```
public class CreateContractUseCase { private ContractRepository repo; public void execute( CreateContractRequest req)  
{ Contract c = new Contract(...); validateContract©; repo.save©; } }
```

Les 4 couches de Clean Architecture 2/2

3 Interface Adapters

Controllers, Gateways, Presenters

```
@RestController
public class ContractController {
    @PostMapping("/contracts")
    public void create(
        @RequestBody Request req) {
        useCase.execute(req);
    }
}
```

4 Frameworks & Drivers

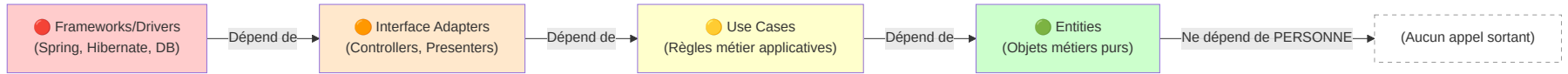
Spring, Hibernate, PostgreSQL, etc.

Détails techniques, facilement remplaçables

```
@RestController
public class ContractController {
    @PostMapping("/contracts")
    public void create(
        @RequestBody Request req) {
        useCase.execute(req);
    }
}
```

Direction des dépendances

Règle d'or: Les dépendances pointent toujours vers l'intérieur



Structure de projet Clean Architecture

```
src/
├── main/java/com/myapp/
│   ├── domain/                # ● Entities
│   │   ├── Contract.java
│   │   ├── Customer.java
│   │   └── ContractRepository.java (interface)
│   ├── application/           # ● Use Cases
│   │   ├── CreateContractUseCase.java
│   │   ├── UpdateContractUseCase.java
│   │   └── dto/
│   │       └── CreateContractRequest.java
│   ├── infrastructure/        # ● Adapters & Drivers
│   │   ├── controller/
│   │   │   └── ContractController.java
│   │   ├── persistence/
│   │   │   ├── PostgresContractRepository.java
│   │   │   └── ContractEntity.java (JPA)
│   │   └── external/
│   │       └── EmailServiceAdapter.java
│   └── config/
│       └── DependencyInjectionConfig.java
```

```
|
└── test/
    ├── java/com/myapp/
    │   ├── domain/
    │   ├── application/
    │   └── infrastructure/
```

Cas d'usage: CreateContractUseCase

ÉTAPES DU PROCESSUS:

Code complet:

```
@Service
public class CreateContractUseCase {
    private final ContractRepository repo;
    private final EmailService emailService;
    private final PremiumCalculator calculator;

    @Inject // Dependency Injection
    public CreateContractUseCase(
        ContractRepository repo,
        EmailService emailService,
        PremiumCalculator calculator) {
        this.repo = repo;
        this.emailService = emailService;
        this.calculator = calculator;
    }
}
```

```
public ContractResponse execute(
    CreateContractRequest request) {
    // 1. Validation
    validateRequest(request);

    // 2. Création entité
    Contract contract = new Contract(
        request.getCustomerId(),
        request.getType()
    );

    // 3. Calcul de prime
    double premium = calculator
        .calculate(contract);
    contract.setPremium(premium);

    // 4. Persistance
    Contract saved = repo.save(contract);

    // 5. Notification
    emailService.sendConfirmation(
        saved.getCustomer().getEmail(),
        saved
```

Tests unitaires faciles

Test du CreateContractUseCase

Un avantage clé de Clean Architecture: testabilité.

```
public class CreateContractUseCaseTest {
    private CreateContractUseCase useCase;
    private ContractRepository mockRepo;
    private EmailService mockEmail;
    private PremiumCalculator mockCalc;

    @Before
    public void setup() {
        // Créer des mocks (faux objets)
        mockRepo = mock(ContractRepository.class);
        mockEmail = mock(EmailService.class);
        mockCalc = mock(PremiumCalculator.class);

        // Injector les dépendances
        useCase = new CreateContractUseCase(
            mockRepo, mockEmail, mockCalc
        );
    }
}
```

```
@Test
public void shouldCreateContractWithValidData() {
    // Given
    CreateContractRequest req =
        new CreateContractRequest("cust-1", "AUTO");
    when(mockCalc.calculate(any()))
        .thenReturn(1200.0);
    when(mockRepo.save(any()))
        .thenReturn(new Contract(...));

    // When
    ContractResponse response = useCase.execute(req);

    // Then
    assertNotNull(response);
    verify(mockEmail).sendConfirmation(...);
    verify(mockRepo).save(...);
}
```

Avantages de Clean Architecture

✓ Pour le développement

- Logique métier isolée
- Tests unitaires simples
- Code découplé
- Facile à naviguer

✓ Pour la maintenance

- Changements localisés
- Moins de bugs
- Évolution facilitée
- Refactoring sûr

✓ Pour le business

- Réduction des coûts
- Time-to-market amélioré
- Moins de bugs en prod
- Équipes plus productives

✓ Pour l'architecture

- Framework agnostique
- Technologie replaceable
- Scalabilité intégrée
- Future-proof

Pièges à éviter

✗ Over-engineering

- Trop de couches
- Abstractions inutiles
- Code complexe pour du simple

Conseil: Adapter la complexité aux besoins

✗ Entités contaminées

- Annotations JPA/Spring
- Logique métier dispersée
- Dépendances externes

Conseil: Entités = POJO purs

✗ DTOs oubliés

- Entités retournées au client
- Leaks d'implémentation
- Couplage fort

Conseil: Toujours utiliser des DTOs

✗ Tests négligés

- Tests intégration lents
- Pas de tests unitaires
- Couverture faible

Conseil: 70%+ du code couvert

Comparaison: Approches d'architecture

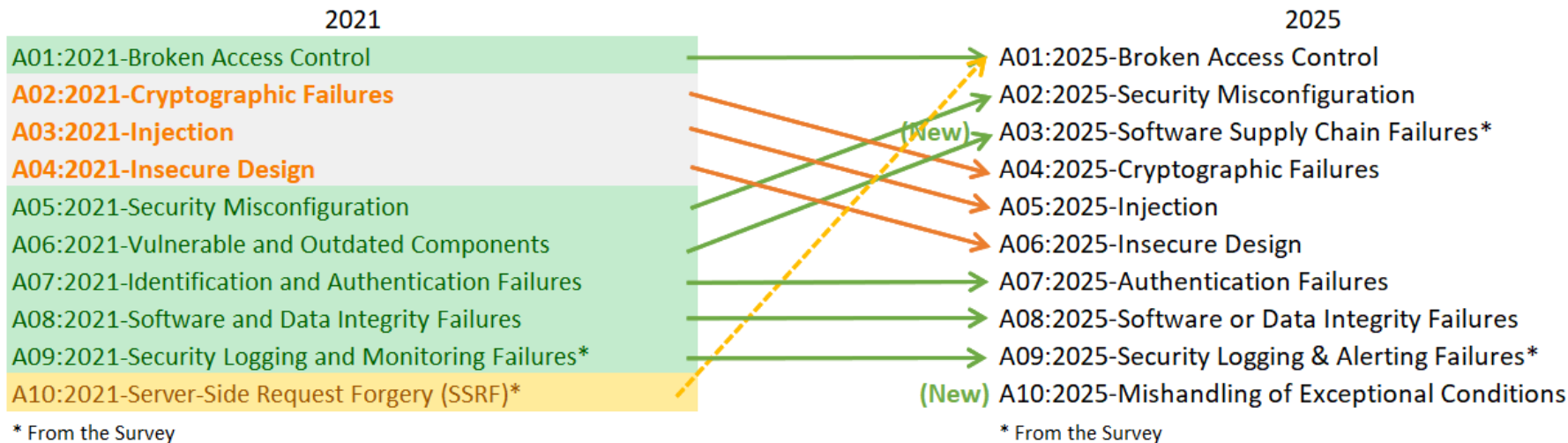
Aspect	Architecture simple	Clean Architecture
Testabilité	Difficile (couplage fort)	Facile (découplage)
Complexité initiale	Faible	Modérée à élevée
Maintenance long terme	Difficile (dette tech)	Facile (structure claire)
Scalabilité	Limitée	Excellente
Changement technologie	Coûteux (réécriture)	Simple (adaptateurs)
Productivité équipe	Diminue avec la taille	Stable et prévisible
Idéal pour	Projet simples ou courts termes	Projets stratégiques long terme



Sécurité Avancée

Protéger vos applications contre les menaces
courantes

OWASP Top 10 - Les 10 risques critiques des applications Web



Source : https://owasp.org/Top10/2025/0x00_2025-Introduction/

Injection SQL & Validation d'entrées

✗ CODE VULNÉRABLE

```
@GetMapping("/users")
public List<User> searchUsers(@RequestParam String name) {
    // DANGEREUX: Concaténation directe
    String query = "SELECT * FROM users WHERE name = '" + name + "'";
    return jdbcTemplate.query(query, new UserRowMapper());
}
```

Attaque: ``name = '' OR '1'='1`` → Récupère tous les utilisateurs

✓ Code sécurisé avec Spring

APPROCHE 1: PREPARED STATEMENTS

```
@GetMapping("/users")
public List<User> searchUsers(@RequestParam String name) {
    String query = "SELECT * FROM users WHERE name = ?";
    return jdbcTemplate.query(query, new Object[]{name}, new UserRowMapper());
}
```

APPROCHE 2: VALIDATION D'ENTRÉES

```
@GetMapping("/users")
public List<User> searchUsers(
    @RequestParam
    @Pattern(regexp = "^[a-zA-Z0-9\\s]*$")
    String name) {
    return userRepository.findByName(name);
}
```

APPROCHE 3: ORM (JPA/HIBERNATE)

```
@GetMapping("/users")
public List<User> searchUsers(@RequestParam String name) {
    // ✓ JPA paramétrise automatiquement
    return userRepository.findByName(name);
}
```

XSS (Cross-Site Scripting) & CSRF

XSS: INJECTION DE SCRIPT CÔTÉ CLIENT

```
<!-- ✖ Vulnérable -->
<h1>${userInput}</h1>

<!-- ✔ Sécurisé (échappement) -->
<h1 th:text="${userInput}"></h1>
```

Attaque: ``userInput = "<script>alert('Piratage')``

`</script>"``

CSRF: FALSIFICATION DE REQUÊTE CROSS-SITE



✓ Protection CSRF avec Spring

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf() // ✓ CSRF activé par défaut
                .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .and()
            .authorizeHttpRequests()
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated();

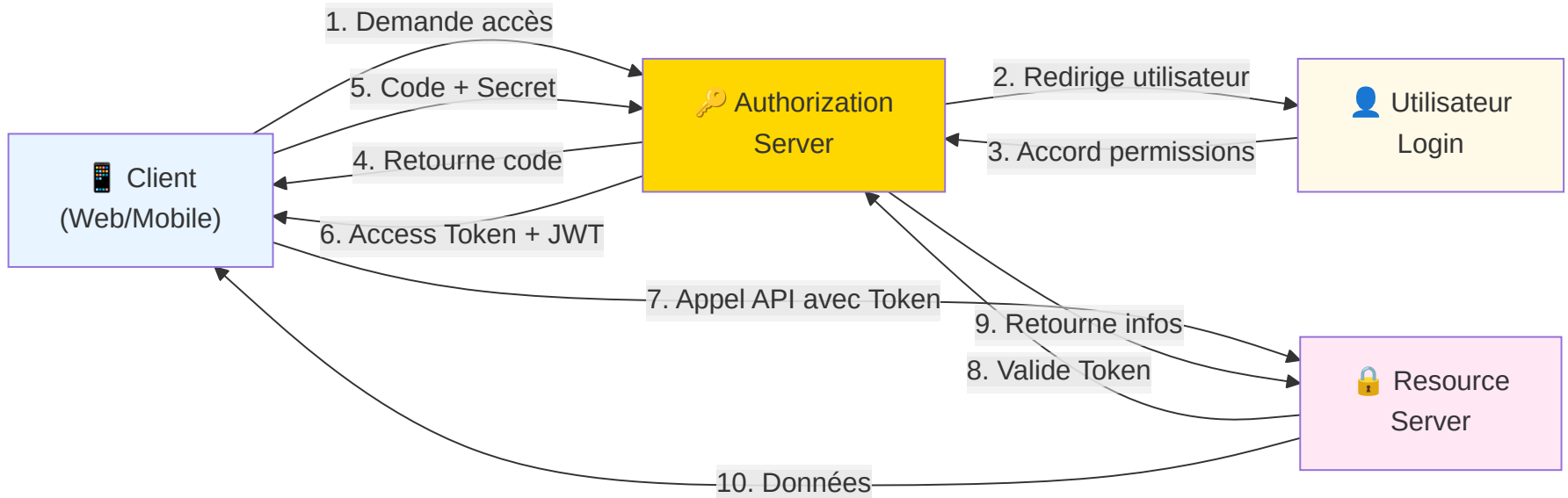
        return http.build();
    }
}
```

DANS LE FORMULAIRE HTML

```
<form method="POST" action="/transfer">
    <input type="hidden" name="_csrf" th:value="${_csrf.token}" />
    <input type="text" name="amount" />
    <button type="submit">Valider</button>
</form>
```

OAuth2 & JWT - Flux d'authentification

ARCHITECTURE OAUTH2



✓ Configuration OAuth2 avec Spring

AUTHORIZATION SERVER

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients
            .inMemory()
                .withClient("mobile-app")
                .secret("{bcrypt}$2a$10$...") // ✓ Password encodé
                .authorizedGrantTypes("password", "refresh_token")
                .authorities("ROLE_CLIENT")
                .scopes("read", "write")
                .accessTokenValiditySeconds(3600) // 1 heure
                .refreshTokenValiditySeconds(86400); // 24 heures
    }
}
```

RESOURCE SERVER (API PROTÉGÉE)

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {
```

✓ Contrôle d'accès avec annotations

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {

    @GetMapping("/{id}")
    @PreAuthorize("hasRole('USER')")
    public Order getOrder(@PathVariable Long id) {
        return orderService.findById(id);
    }

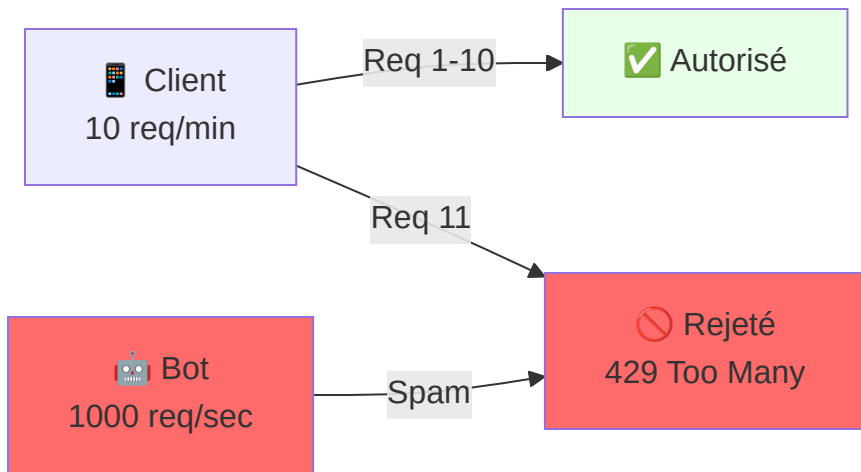
    @PostMapping
    @PreAuthorize("hasRole('USER') and @orderService.canCreateOrder(#request)")
    public Order createOrder(@RequestBody CreateOrderRequest request) {
        return orderService.create(request);
    }

    @DeleteMapping("/{id}")
    @PreAuthorize("hasRole('ADMIN') or @orderService.isOwner(#id, authentication.principal.id)")
    public void deleteOrder(@PathVariable Long id, Authentication authentication) {
        orderService.delete(id);
    }

    @GetMapping("/admin/stats")
    @PreAuthorize("hasAnyRole('ADMIN', 'ANALYST')")
    public OrderStats getStats() {
        return orderService.getStatistics();
    }
}
```

Rate Limiting & Protection DDoS

CONCEPT: LIMITER LES REQUÊTES PAR CLIENT



✓ Rate Limiting avec Resilience4j

```
@RestController
@RequestMapping("/api/payments")
public class PaymentController {

    @PostMapping("/process")
    @RateLimiter(
        name = "paymentLimiter",
        fallbackMethod = "paymentFallback"
    )
    public PaymentResponse processPayment(@RequestBody PaymentRequest request) {
        return paymentService.process(request);
    }

    // Fallback: appelé si limite dépassée
    public PaymentResponse paymentFallback(PaymentRequest request,
                                           RequestNotPermitted ex) {

        return PaymentResponse.builder()
            .status("RATE_LIMIT_EXCEEDED")
            .message("Trop de requêtes. Réessayez dans 1 minute.")
            .build();
    }
}

// Configuration application.yml
/*
resilience4j:
```

Secret Management & Configuration sensible

❌ MAUVAISE PRATIQUE

```
@Configuration
public class DatabaseConfig {

    // ❌ DANGEREUX: mot de passe en dur
    @Bean
    public DataSource dataSource() {
        return DriverManager.getConnection(
            "jdbc:postgresql://localhost:5432/db",
            "admin",
            "password123" // 🚨 Exposé dans le code!
        );
    }
}
```

✓ Injection de secrets avec @Value

```
@Configuration
public class DatabaseConfig {

    @Value("${database.url}")
    private String dbUrl;

    @Value("${database.username}")
    private String dbUsername;








    @Value("${database.password}")
    private String dbPassword;

    @Bean
    public DataSource dataSource() {
        // ✓ Secrets injectés à l'exécution
        return DriverManager.getConnection(
            dbUrl,
            dbUsername,
            dbPassword
        );
    }
}
```








FICHER `APPLICATION-PROD.YML` (SÉCURISÉ)

Bonnes pratiques en résumé

SÉCURITÉ AU DÉVELOPPEMENT

-  Valider TOUTES les entrées
-  Utiliser des Prepared Statements
-  Encoder les sorties HTML/URL
-  HTTPS/TLS obligatoire
-  Secrets en variables d'env
-  Logs sans données sensibles
-  CORS configuré strictement

DÉTECTION & RÉACTION

-  Logging sécurité (logs centralisés)
-  Alertes sur anomalies
-  Rate limiting par IP/API key
-  Circuit breaker en cas d'attaque
-  WAF (Web Application Firewall)
-  Monitoring de dépendances vulnérables
-  Rotation de secrets régulière



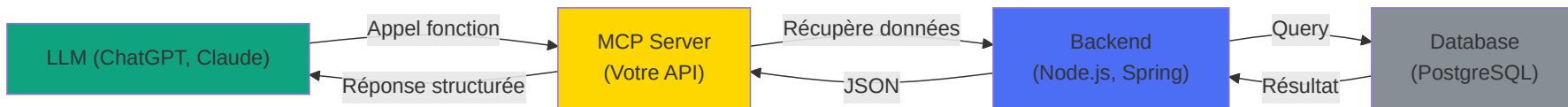
MCP & Intégration IA

MCP & Intégration IA: Nouvelle ère

Cas d'usage:

Connecter les backends avec les modèles d'IA

- 📋 Assurance: Analyse automatique des sinistres avec Claude
- 🏥 Santé: Diagnostic assistance basé sur données patients
- ✍️ Génération contenu: Documents, email, rapports automatisés
- 🔍 Recherche: Sémantique sur base de données



MCP: Model Context Protocol

ARCHITECTURE MCP:

MCP Server (côté backend):

Standard ouvert pour connecter LLMs aux tools/APIs

```
# server.py
from fastmcp import FastMCP
from pydantic import BaseModel
from typing import Dict, Any, List
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Modèle pour l'utilisateur
class UserProfile(BaseModel):
    name: str
    role: str
    department: str = "Engineering"
    years_experience: int = 0

mcp = FastMCP(
    name="user-db-server",
    version="1.0.0",
```

```
# Ressource : liste statique d'utilisateurs
@mcp.resource("user_profiles")
def fetch_users() -> List[Dict[str, Any]]:
    """Retourne la liste des profils utilisateurs."""
    return [
        {"name": "Alice", "role": "Dev", "department": "Engineering"},
        {"name": "Bob", "role": "PM", "department": "Product"},
    ]

# Outil : créer un utilisateur
@mcp.tool()
def create_user(name: str, role: str, department: str = "Engineering"):
    """Crée un nouveau profil utilisateur."""
    user = UserProfile(name=name, role=role, department=department)
    logger.info(f"User créé : {user}")
    return f"User {user.name} créé avec succès !"

if __name__ == "__main__":
    logger.info("Démarrage serveur MCP...")
    mcp.run(transport="stdio") # Ou "http://localhost:8000"
```

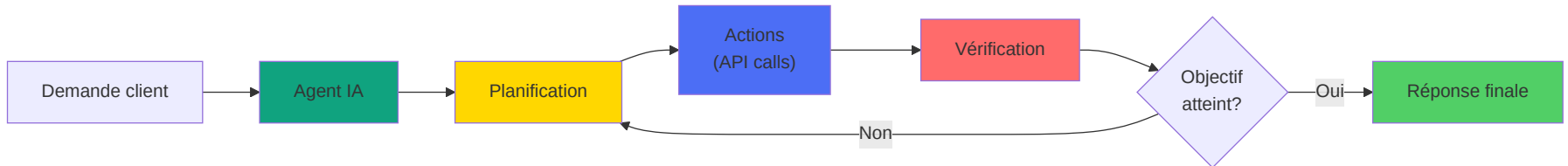
Futur: Agents IA autonomes

Exemple: Traitement sinistre automatique

La prochaine génération: agents capables de décisions autonomes

```
const claimAgent = new Agent({  
  tools: ['get_contract', 'create_claim', 'estimate_damage', 'notify_client', 'schedule_inspection']  
});  
  
const result = await claimAgent.run(  
  `Traiter ce sinistre: Description du sinistre...`  
);
```

```
// Résultat: Agent a de façon autonome:  
// 1. ✅ Cherché le contrat  
// 2. ✅ Créé le dossier sinistre  
// 3. ✅ Estimé les dégâts  
// 4. ✅ Notifié le client  
// 5. ✅ Programmé l'inspection  
// Tout dans une seule chaîne de pensée!
```





**Conclusion : ressources et
références**

Ressources & Références

Ouvrages de Référence

Design Patterns - Gang of Four (Gamma, Helm, Johnson, Vlissides)

"The purpose of design patterns is to give a name and a context to design problems and their solutions."

Building Microservices - Sam Newman

"Microservices are small, autonomous services that work together. The microservice architectural style is an approach to developing a single application as a suite of small services."

Domain-Driven Design - Eric Evans

"When you model using only the semantics that the business expert cares about, you get a model that the business expert understands."


Refactoring: Improving the Design of Existing Code - Martin Fowler


"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

Clean Code - Robert C. Martin

Questions & Discussion

QU'AVEZ-VOUS ENVIE DE DISCUTER?

 Levez la main pour poser vos questions

 Débat sur technologies, architecture...

 Cas d'usage spécifiques à votre contexte

Pas de question bête - cette partie est pour VOUS

Merci! 🙏

