# Architectures Back-end & Front-end

Web, Mobile et IA

# Sommaire

## Backend

- **Patterns d'Architecture**
  - Monolithe, Microservices
  - MVC, MVVM
  - CQRS, Event-Driven
  - Hexagonal, DI
  - Repository, Strategy
- **Clean Code, Clean Architecture**
- **Ecosystèmes technologiques**
  - Java, .Net, Python...

## Frontend

- **Patterns d'Interface**
  - Components
  - State Management
  - Routing
- **Technologies Web**
  - React, Vue, Angular
  - Frameworks modernes

## IA

- **MCP, A2A**

# Introduction

*Fondamentaux de l'architecture logicielle*

# Pourquoi l’architecture logicielle est cruciale ?

> "When you model using only the semantics that the business expert cares about, you get a model that the business expert understands." — **Eric Evans**, Domain-Driven Design

## IMPACT DIRECT SUR:

- Maintenabilité et évolutivité du code
- Performance et sécurité des applications
- Adaptation aux besoins métiers (ex: assurance, santé)
- Réduction des coûts de développement à long terme

## Évolution des architectures


| Architecture | Avantages | Inconvénients |
|---|---|---|
| Monolithe | Simple, facile à déployer | Diffcile à scaler, couplage fort |
| Microservices | Scalable, indépendant | Complexité opérationnelle |
| Serverless | Pas de gestion infra | Coûts imprévisibles, latence |

# Définitions clés

## Back-end

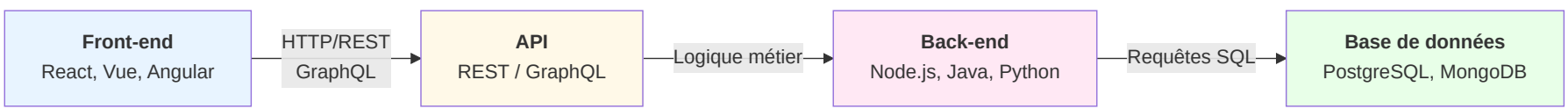
Logique métier, bases de données, APIs, serveurs. Invisible à l'utilisateur final.

## Front-end

Interface utilisateur, expérience client, interactions. Ce que l'utilisateur voit et utilise.

## API (Application Programming Interface)

Pont de communication entre back-end et front-end. Contrats et protocoles d'échange de données.



# Panorama des technologies

### 🔧 Back-end

- Java: Spring Boot, Spring Cloud
- Node.js: Express, NestJS
- Python: Django, FastAPI
- Go: Gin, Echo

### 🎨 Front-end

- React: Composants, Hooks
- Vue.js: Réactif, simple
- Angular: Complet, TypeScript
- Next.js: SSR, SSG, SSG

### 📱 Mobile

- React Native: Code réutilisable
- Flutter: Widgets natifs
- Swift/Kotlin: Natif

### ☁️ Infrastructure & BD

- Docker: Conteneurisation
- Kubernetes: Orchestration
- PostgreSQL, MongoDB

# Principes d’architecture applicative

## SÉPARATION DES PRÉOCCUPATIONS

Chaque couche a une responsabilité unique et bien définie.

```
Présentation (UI)
      ↓
Logique métier (Règles de gestion)
      ↓
Accès aux données (Persistance)
      ↓
Infrastructure (Serveurs, BD)
```

## Principes SOLID

- Single Responsibility Principle: Une classe = une responsabilité
- Open/Closed Principle: Ouvert à l’extension, fermé à la modification
- Liskov Substitution: Les sous-types peuvent remplacer le type parent
- Interface Segregation: Plusieurs interfaces spécifiques > une grosse interface
- Dependency Inversion: Dépendre des abstractions, pas des implémentations

# 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

## Maintenabilité

- Documentation
- Tests automatisés
- CI/CD pipeline

# 🏗 Patterns d’Architecture

*Solutions éprouvées pour structurer vos applications*

# Pourquoi utiliser des patterns ?

## LES PATTERNS RÉSOLVENTS DES PROBLÈMES RÉCURRENTS

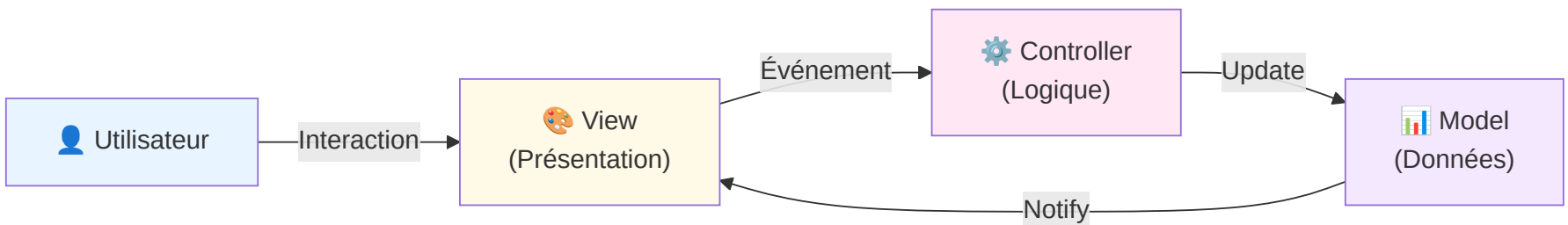
> "The purpose of design patterns is to give a name and a context to design problems and their solutions." — **Gang of Four**, Design Patterns

- Réutilisabilité: Solutions éprouvées et documentées
- Standardisation: Équipes alignées sur une même approche
- Collaboration: Facilite la communication entre développeurs
- Réduction des risques: Évite les pièges courants
- Maintenabilité: Code plus prévisible et compréhensible

# Pattern MVC (Model-View-Controller)

## SÉPARATION DES RESPONSABILITÉS:

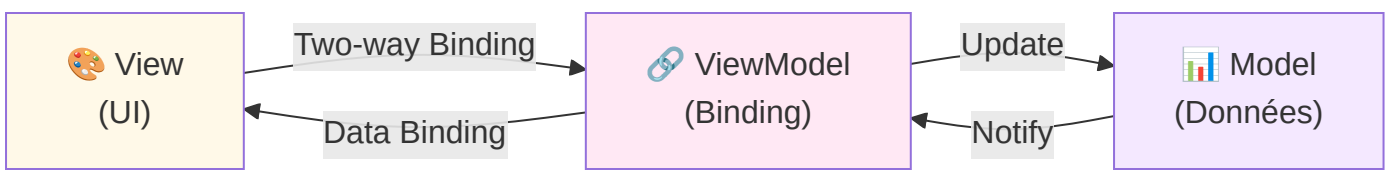
- Model: Données et logique métier
- View: Présentation et interface utilisateur
- Controller: Coordination et gestion des événements



# Pattern MVVM (Model-View-ViewModel)

## CARACTÉRISTIQUES:

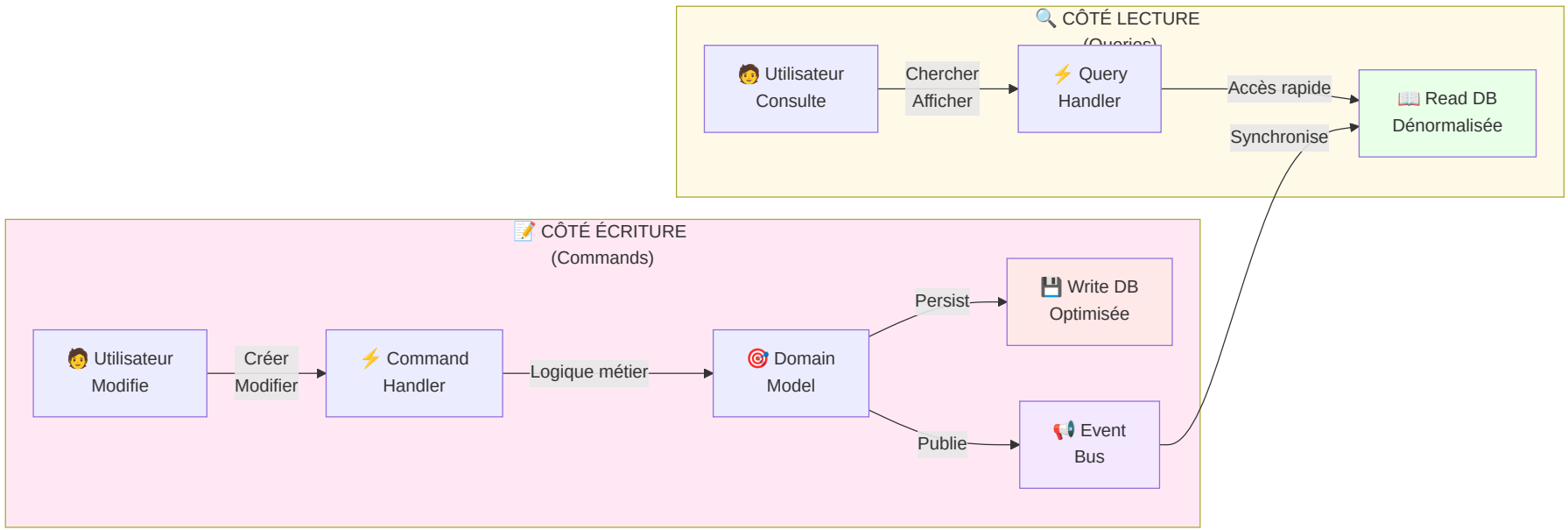
- Binding bidirectionnel: Sync automatique View ↔ ViewModel
- Testabilité: ViewModel indépendant de la Vue
- Réactivité: Mises à jour temps réel



# Pattern CQRS (Command Query Responsibility Segregation)

## CONCEPT CLÉ

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




## AVANTAGES

- ✅ **Optimisation indépendante**: Chaque modèle optimisé pour son usage
- ✅ **Scalabilité**: Lectures et écritures peuvent être déployées séparément

# 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 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/Guice injecte la dépendance
    public ContractService(DatabaseService db) {
        this.db = db;
    }
}
```

# 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 |

# 🔧 Écosystèmes Backend

*Découvrez les principaux frameworks et technologies*

## Vue d’ensemble

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

# Spring Boot (Java)

## CARACTÉRISTIQUES

- **Framework**: Spring Framework avec Spring Boot pour démarrage rapide
- **TypeScript/Langages**: Java (JVM ecosystem)
- **Popularité**: ⭐⭐⭐⭐⭐ Très populaire en entreprise
- **Apprentissage**: Moyen - courbe importante

## POINTS FORTS

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

## USE CASES

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

# NestJS (Node.js/TypeScript)

## CARACTÉRISTIQUES

- **Framework**: Node.js moderne avec TypeScript par défaut
- **Langages**: TypeScript/JavaScript
- **Popularité**: ⭐⭐⭐⭐ En croissance rapide
- **Apprentissage**: Facile - syntaxe proche de Angular

## POINTS FORTS

- Très rapide à développer
- Partage du code Frontend/Backend (JavaScript/TS)
- Développement agile
- Excellent pour API REST et GraphQL
- Écosystème npm très riche

## USE CASES

- 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**: Très rapide à développer, auto-documentation API
- **Use cases**: APIs légères, Machine Learning, Data Science

## DJANGO

- **Caractéristiques**: Framework complet "batteries included"
- **Avantages**: ORM puissant, admin panel généré, sécurité native
- **Use cases**: Applications web complètes, startups, prototypage rapide

## POINTS COMMUNS

- Popularité: ⭐⭐⭐⭐ Excellente pour l'IA/ML
- Apprentissage: Facile - syntaxe simple et claire
- Productivité: Très haute

# .NET / ASP.NET Core (C#)

## CARACTÉRISTIQUES

- **Framework**: ASP.NET Core (cross-platform)
- **Langages**: C# (langage moderne et puissant)
- **Popularité**: ⭐⭐⭐⭐ Très utilisé en entreprise
- **Apprentissage**: Moyen - C# plus complexe que Python

## POINTS FORTS

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

## USE CASES

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

# Ruby on Rails (Ruby)

## CARACTÉRISTIQUES

- **Framework**: Rails (Convention over Configuration)
- **Langages**: Ruby (syntaxe élégante)
- **Popularité**: ⭐⭐⭐ Moins nouveau, mais très efficace
- **Apprentissage**: Très facile - excellent pour débutants

## POINTS FORTS

- Rapidité de développement exceptionnelle
- Convention plutôt que configuration
- Excellente pour prototypes et MVPs
- Communauté très active et bienveillante
- Gestion de bases de données élégante

## USE CASES

- Startups et MVPs
- Applications web complètes
- Prototypage rapide
- Content management systems

## Comparaison Synthétique

| Critère | Spring Boot | NestJS | Python | .NET | Rails |
|---|---|---|---|---|---|
| **Vitesse dev** | Moyen | Rapide | Très rapide | Moyen | Très rapide |
| **Performance** | ⭐⭐⭐⭐⭐ | ⭐⭐⭐⭐ | ⭐⭐⭐ | ⭐⭐⭐⭐⭐ | ⭐⭐⭐ |
| **Scalabilité** | ⭐⭐⭐⭐⭐ | ⭐⭐⭐⭐ | ⭐⭐⭐ | ⭐⭐⭐⭐⭐ | ⭐⭐⭐ |
| **Courbe apprentissage** | Moyenne | Facile | Facile | Moyenne | Très facile |
| **Écosystème** | ⭐⭐⭐⭐⭐ | ⭐⭐⭐⭐ | ⭐⭐⭐⭐ | ⭐⭐⭐⭐⭐ | ⭐⭐⭐ |
| **Entreprise** | ⭐⭐⭐⭐⭐ | ⭐⭐⭐⭐ | ⭐⭐⭐⭐ | ⭐⭐⭐⭐⭐ | ⭐⭐⭐ |
| **Startup/Agile** | ⭐⭐⭐ | ⭐⭐⭐⭐⭐ | ⭐⭐⭐⭐⭐ | ⭐⭐ | ⭐⭐⭐⭐⭐ |

# Quelle pile choisir?

## SPRING BOOT 👉

- Vous avez une équipe Java expérimentée
- Vous développez un système critique d’entreprise
- Vous 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

# 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 corrects toujours |
| **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)

# Niveaux d'Isolation

## LECTURE AVEC PROBLÈMES POTENTIELS

| Niveau | Lecture Dirty | Non-Répétable | Fantôme |
|---|---|---|---|
| **READ UNCOMMITTED** | ❌ Oui | ❌ Oui | ❌ Oui |
| **READ COMMITTED** | ✅ Non | ❌ Oui | ❌ Oui |
| **REPEATABLE READ** | ✅ Non | ✅ Non | ❌ Oui |
| **SERIALIZABLE** | ✅ Non | ✅ Non | ✅ Non |

## DÉFINITIONS

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

# Implémentation dans les frameworks

## SPRING BOOT (JAVA)

```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
    }
}

// Gestion d'erreur
@Transactional
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
```

# ✨ Clean Code & Architecture

*Écrire du code maintenable et évolutif*

# 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

- Réduction des bugs: Code clair = moins d'erreurs
- Maintenabilité: Facile à modifier et à déboguer
- Collaboration: Équipes comprennent rapidement le code
- Évolutivité: Ajout de fonctionnalités sans refonte
- Productivité: Développeurs plus rapides et efficaces

# 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)

## FONCTION TROP GROSSE (MAUVAIS):

## FONCTIONS COURTES ET FOCALISÉES (BON):

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

```
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());
}
```

```
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**

✅ **Exception**

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

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

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

# Règle 4: DRY (Don’t Repeat Yourself)

## ❌ Code répété

## ✅ Extraction en méthode

Éliminer les répétitions de code.

```
// 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;
```

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

# Règle 5: Commentaires

## ❌ Commentaires inutiles

## ✅ Commentaires utiles

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

```
// 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();
    }
}
```

```
// 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
```

## Règle 6: Formatage et style

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: Les 7 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

Structure logicielle indépendante des frameworks, testable et maintenable.

Principe: Les couches intérieures ne dépendent jamais des couches extérieures

# Les 4 couches de Clean Architecture

## 🟪 Entities (Cœur métier)

## 🟪 Use Cases (Logique applicative)

## 🟪 Interface Adapters

## 🟪 Frameworks & Drivers

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; } } 2️⃣ 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©; } } 3️⃣ Interface Adapters Controllers, Gateways, Presenters

@RestController public class ContractController { @PostMapping("/contracts") public void create( @RequestBody Request req) { useCase.execute(req); } } 4️⃣ Frameworks & Drivers Spring, Hibernate, PostgreSQL, etc.

Détails techniques, facilement remplaçables

```
public class Contract {
    private String id;
    private Customer customer;
    private double premium;
```

# 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
│
```

# 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(
```

# 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
```

# Avantages de Clean Architecture

## ■ Pour le développement

## ■ Pour la maintenance

## ■ Pour le business

## ■ Pour l'architecture

- Logique métier isolée
- Tests unitaires simples
- Code découplé
- Facile à naviguer
- Changements localisés
- Moins de bugs
- Évolution facilitée
- Refactoring sûr
- Réduction des coûts

# Pièges à éviter

❌ **Over-engineering**
❌ **Entities contaminées**
❌ **DTOs oubliés**
❌ **Tests négligés**

Conseil: Adapter la complexité aux besoins

Conseil: Entities = POJO purs

Conseil: Toujours utiliser des DTOs

Conseil: 70%+ du code couvert

- Trop de couches
- Abstractions inutiles
- Code complexe pour du simple
- Annotations JPA/Spring
- Logique métier dispersée

## 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 | Prototypes, POC | Projets long terme |

# Récapitulatif: Clean Code & Architecture

### 🧠 Clean Code

### 🏛️ Clean Architecture

## IMPACT COMBINÉ:

7 règles:

4 couches:

- ✅ Code facile à lire et comprendre
- ✅ Logique métier isolée et testable
- ✅ Dépendances contrôlées
- ✅ Évolution sans refonte majeure
- ✅ Équipes productives et heureuses

# REST vs GraphQL

## COMPARAISON DES APPROCHES

| Aspect | REST | GraphQL |
| --- | --- | --- |
| Requête | Fixed endpoints (/users/1) | Flexible query (demander exactement ce qu’on veut) |
| Over-fetching | Oui (données superflues) | Non (données exactes) |
| Under-fetching | Oui (appels multiples) | Non (1 requête) |
| Caching | Facile (HTTP standard) | Plus difficile |
| Versioning | Nécessaire (/v1/, /v2/) | Pas nécessaire |
| Courbe d’apprentissage | Facile | Modérée |

# REST: Principes fondamentaux

## PRINCIPES CLÉS:

### Exemple d'endpoints REST:

REST: Representational State Transfer

- Client-Server: Séparation des préoccupations
- Stateless: Chaque requête contient toutes les infos
- Cacheable: Réponses peuvent être mises en cache
- Uniform Interface: Ressources identifiables par URI
- Méthodes HTTP standards: GET, POST, PUT, DELETE, PATCH

```
GET    /api/v1/contracts              # Récupérer tous les contrats
POST   /api/v1/contracts              # Créer un nouveau contrat
GET    /api/v1/contracts/123          # Récupérer un contrat spécifique
PUT    /api/v1/contracts/123          # Mettre à jour complètement
PATCH  /api/v1/contracts/123          # Mise à jour partielle
DELETE /api/v1/contracts/123          # Supprimer

GET    /api/v1/contracts/123/claims   # Sous-ressources
```

# REST: Bonnes pratiques

## BEST PRACTICES POUR UNE API REST ROBUSTE:

**Sécurité**

**Versioning**

**Documentation**

**Erreurs**

- OAuth2: Authentification
- JWT: Token sans état
- HTTPS: Chiffrement
- Rate limiting: Protection DOS
- CORS: Contrôle d’accès
- URL versioning: /v1/, /v2/
- Header versioning: X-API-Version
- Semantic versioning: 1.2.3

# Codes HTTP et gestion d’erreurs

## Réponse d’erreur standardisée:

```
{
  "error": {
    "code": "INVALID_CONTRACT",
    "message": "Le contrat ne peut pas être créé",
    "details": {
      "field": "customer_id",
      "reason": "Customer not found"
    },
    "timestamp": "2026-01-17T10:30:00Z",
    "requestId": "req-12345"
  }
}
```

| Code | Signification | Exemple |
|---|---|---|
| 200 | OK - Succès | Requête GET réussie |
| 201 | Created - Ressource créée | POST réussi |
| 400 | Bad Request - Erreur client | JSON invalide |

# GraphQL: Introduction

## CONCEPT CLÉ: DEMANDER EXACTEMENT CE QU'ON VEUT

❌ **REST (over-fetching)**

**GraphQL (seulement ce qu'il faut)**

GraphQL: Query language pour APIs

```
GET /api/v1/contracts/123

{
  "id": "123",
  "customer": { ... },
  "premium": 1200,
  "type": "AUTO",
  "status": "ACTIVE",
  "createdAt": "...",
  "updatedAt": "...",
  // Plein de données non nécessaires
}

Données non utilisées =
bande passante gaspillée
```

```
query {
  contract(id: "123") {
    id
```

# Schéma GraphQL

## Exemple de schéma pour assurance:

Structure typée des données et opérations disponibles

```
type Contract {
  id: ID!                    # ! = obligatoire
  customer: Customer!
  premium: Float!
  type: ContractType!
  status: Status!
  claims: [Claim!]!          # Liste obligatoire
  createdAt: DateTime!
}

type Customer {
  id: ID!
  name: String!
  email: String!
  age: Int!
  contracts: [Contract!]!
}

enum ContractType {
  AUTO
  HOME
  HEALTH
```

# GraphQL Queries (Lecture)

## QUERY SIMPLE:

## QUERY AVEC FILTRAGE ET PAGINATION:

## QUERY AVEC RELATIONS IMBRIQUÉES:

```
query GetContract {
  contract(id: "123") {
    id
    premium
    type
    customer {
      name
      email
    }
  }
}
```

```
query GetContracts {
  contracts(limit: 10, offset: 0) {
    id
    premium
    type
    status
    customer {
      id
      name
```

# GraphQL Mutations (Écriture)

## Mutation: Créer un contrat

Opérations de création, mise à jour, suppression

```
mutation CreateNewContract {
  createContract(input: {
    customerId: "cust-1"
    type: AUTO
    coverage: [COLLISION, THEFT]
    deductible: 500
  }) {
    id
    premium
    status
    customer {
      name
    }
  }
}

Réponse:
{
  "createContract": {
    "id": "contract-789",
    "premium": 1200.50,
    "status": "ACTIVE",
```

# GraphQL: Avantages et limitations

## ■ Avantages

## ✕ Limitations

- Pas de over-fetching
- Pas de under-fetching
- Requête unique
- Pas de versioning
- Typage fort
- Documentation auto
- Introspection
- Caching difficile (POST)
- Courbe apprentissage
- Complexité du serveur

# Implémentation GraphQL: Apollo Server

## Installation et setup:

## Code serveur GraphQL (Node.js):

```
npm install apollo-server-express
npm install graphql
```

```
const { ApolloServer, gql } = require('apollo-server-express');
const express = require('express');

// Schéma
const typeDefs = gql`
  type Contract {
    id: ID!
    premium: Float!
    type: String!
  }

  type Query {
    contract(id: ID!): Contract
  }

  type Mutation {
    createContract(premium: Float!, type: String!): Contract!
  }
`;
```

# Quand utiliser REST vs GraphQL?

| Scénario | REST | GraphQL | Recommandation |
|---|---|---|---|
| Ressources simples | ✅ Idéal | ⚠️ Overkill | REST |
| Relations complexes | ❌ Appels multiples | ✅ Requête unique | GraphQL |
| Clients variés | ❌ Over-fetching | ✅ Données précises | GraphQL |
| Mobile (bande passante) | ❌ Données superflues | ✅ Minimal | GraphQL |
| Caching HTTP | ✅ Facile | ❌ Complexe | REST |
| Adoption rapide | ✅ Facile à apprendre | ❌ Courbe apprentissage | REST |
| Real-time (WebSocket) | ❌ Non natif | ✅ Subscriptions | GraphQL |
| File uploads | ✅ Natif | ⚠️ Complexe | REST |

# Sécurité dans les APIs

## 🎟️ OAuth2

## 🔑 JWT (JSON Web Tokens)

### Flux d’authentification JWT:

Protocole d’authentification/autorisation

Token stateless, auto-contenu

- Authorization Code: Apps web
- Client Credentials: Services
- Implicit: Apps single-page
- Refresh Token: Session longue
- Header: Type et algorithme
- Payload: Données (user_id)
- Signature: Vérification intégrité
- Expiration: Courte durée

# Documentation API: Swagger/OpenAPI

## Exemple de specification OpenAPI (YAML):

Documenter et tester les APIs interactivement

```
openapi: 3.0.0
info:
  title: Insurance API
  version: 1.0.0
paths:
  /contracts:
    get:
      summary: List all contracts
      parameters:
        - name: limit
          in: query
          type: integer
          default: 10
      responses:
        '200':
          description: List of contracts
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Contract'
```

# Versioning d’API

## 🟪 URL Versioning

## 📋 Header Versioning

**Bonnes pratiques:**

Maintenir la compatibilité avec les clients existants

- Semantic Versioning: MAJOR.MINOR.PATCH (1.2.3)
- Backward compatibility: Supporter les anciennes versions (minimum 2 ans)
- Deprecation warnings: Notifier les clients
- Changelog: Documenter les changements

```
GET /api/v1/contracts
GET /api/v2/contracts

Avantages:
✅ Clair et explicite
✅ Caching facile
✅ Fournisseurs multiples

Inconvénients:
❌ URLs dupliquées
❌ Maintenance double
```

# Récapitulatif: API et GraphQL

## 🌐 REST API

## ◼ GraphQL

POINTS CLÉS:

- Standard HTTP (GET, POST, PUT, DELETE)
- Endpoints fixes par ressource
- Facile à cacher
- Versioning standard
- Idéal pour ressources simples
- Courbe apprentissage faible
- Query language typé
- Requêtes flexibles
- Pas over/under-fetching

# Microservices: Introduction

Architectures distribuées basées sur des services indépendants.

# Caractéristiques des Microservices

## PROPRIÉTÉS CLÉS:

**Autonomie**
**Communication**
**Résilience**
**Observabilité**

- Services indépendants
- Déploiement indépendant
- BD dédiée
- Équipes autonomes
- API REST / gRPC
- Message brokers (Kafka)
- Events asynchrones
- Découverte de services

# 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)

### ⬤ Asynchrone (Events)

```
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
```

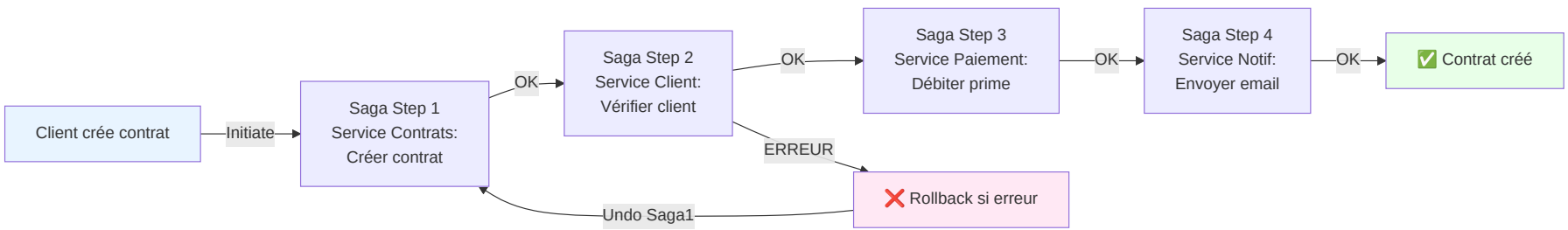
```
Service A
   ↓ (Publie event)
Kafka/RabbitMQ
   ↓ (Message broker)
Service B (reçoit)
```

# 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)



## SSR: Server-Side Rendering

### AVANTAGES:

Le serveur génère le HTML complet avant envoi au navigateur

- ✅ SEO: HTML complet pour crawlers
- ✅ Performance initiale: Page affichée rapidement
- ✅ Social media: Open Graph meta tags
- ✅ Contenu dynamique: Basé sur la requête

```
Error: Parse error on line 13:
...e page complète<br/>(SEO-friendly)| Brow
-----------------------^
Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND', 'SUBROUTINEEND', 'PIPE',
'CYLINDEREND', 'DIAMOND_STOP', 'TAGEND', 'TRAPEND', 'INVTRAPEND', 'UNICODE_TEXT', 'TEXT',
'TAGSTART', got 'PS'
```

## SSG: Static Site Generation

### AVANTAGES:

Générer les pages HTML à la compilation (build time)

- ✅ Performance: Fichiers pré-générés (très rapide)
- ✅ Sécurité: Pas de serveur Node.js exposé
- ✅ Coûts: Hosting CDN bon marché
- ✅ SEO: HTML complet et optimisé

```
Error: Parse error on line 16:
...->|Affiche page<br/>(très rapide)| Brows
-----------------------^
Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND', 'SUBROUTINEEND', 'PIPE',
'CYLINDEREND', 'DIAMOND_STOP', 'TAGEND', 'TRAPEND', 'INVTRAPEND', 'UNICODE_TEXT', 'TEXT',
'TAGSTART', got 'PS'
```

# SPA: Single Page Application

## AVANTAGES VS INCONVÉNIENTS:

**✅ Avantages**

**❌ Inconvénients**

Application côté client, tout le rendu en JavaScript

- UX fluide (pas de reload)
- Interaction rapide
- Offline possible
- Webapp native-like
- SEO difficile
- JS volumineux
- Première charge lente
- Pas de meta tags

## Comparaison: SSR vs SSG vs SPA

| Aspect | SSR | SSG | SPA |
|---|---|---|---|
| Quand générer | Runtime (chaque requête) | Build time (compilation) | Browser (JavaScript) |
| Performance | Bonne (HTML pré-rendu) | Excellente (fichiers statiques) | Mauvaise initiale |
| SEO | Parfait (HTML complet) | Parfait (HTML complet) | Difficile (JS exécuté) |
| Contenu dynamique | ✅ Oui (par requête) | ⚠️ Revalidation | ✅ Oui (en temps réel) |
| Coûts serveur | Modérés (besoin serveur) | Bas (CDN seulement) | Bas (BaaS) |
| Scalabilité | Limitée (load serveur) | Illimitée (CDN) | Illimitée (API) |
| Idéal pour | News, blogs (contenu dyn) | Docs, portfolio, blogs | Apps (Gmail, Trello) |

# Next.js: Hybrid Rendering

## Structure d’un projet Next.js:

## Exemples de stratégies:

Framework React supportant SSR, SSG et SPA dans le même projet

```
my-app/
├── pages/
│   ├── index.js           # Page d'accueil
│   ├── contracts/
│   │   ├── index.js       # Liste contrats
│   │   └── [id].js        # Détail contrat
│   └── api/
│       └── contracts.js   # API route
├── public/
├── styles/
└── package.json
```

```
// Page SSG (statique, très rapide)
export async function getStaticProps() {
  const contracts = await db.contracts.findAll();
  return { props: { contracts }, revalidate: 3600 };
}

// Page SSR (dynamique à chaque requête)
export async function getServerSideProps(context) {
  const id = context.params.id;
```

# Hydration: HTML + Interactivité

## PROCESSUS D'HYDRATION:

**⚠ Hydration Mismatch:**

Transformer du HTML statique en application interactive

Le HTML côté serveur et côté client DOIT être identique, sinon:

- ❌ React réécrit le DOM (performance)
- ❌ Perte du contenu serveur
- ❌ Erreurs de validation

```
Error: Parse error on line 7:
...r &lt;Component /&gt;
----------------------^
Expecting 'SOLID_OPEN_ARROW', 'DOTTED_OPEN_ARROW', 'SOLID_ARROW',
'BIDIRECTIONAL_SOLID_ARROW', 'DOTTED_ARROW', 'BIDIRECTIONAL_DOTTED_ARROW', 'SOLID_CROSS',
'DOTTED_CROSS', 'SOLID_POINT', 'DOTTED_POINT', got 'NEWLINE'
```

# Web Vitals: Métriques de performance

- **LCP**
- **FID**
- **CLS**

**Techniques d’optimisation:**

Mesurer l’expérience utilisateur réelle

Largest Contentful Paint

Temps d’affichage du contenu principal

Idéal: < 2.5 secondes

First Input Delay

Délai avant réaction aux interactions

Idéal: < 100ms

Cumulative Layout Shift

Stabilité visuelle (pas de changement de layout)

Idéal: < 0.1

# Récapitulatif: SSR vs SSG vs SPA

## DÉCISION TREE (ARBRE DE DÉCISION):

# Spring Boot: Introduction

## CARACTÉRISTIQUES CLÉS:

### Créer un projet Spring Boot:

Framework Java pour construire des microservices robustes

- Auto-configuration: Configuration intelligente par défaut
- Starters: Dépendances pré-configurées (spring-boot-starter-web)
- Embedded server: Pas besoin de Tomcat externe
- Production-ready: Monitoring, logging, health checks
- Actuator: Endpoints de monitoring (/health, /metrics)

```
# Via Spring Boot CLI
spring boot new my-api --from=web

# Via Maven
mvn archetype:generate \
  -DgroupId=com.myapp \
  -DartifactId=my-api \
  -DarchetypeArtifactId=maven-archetype-quickstart

# Via Spring Initializr
# https://start.spring.io
```

# Architecture Spring Boot

## STRUCTURE STANDARD:

## APPLICATION.PROPERTIES:

```
src/main/java/com/myapp/
├── Application.java           # Entry point @SpringBootApplication
├── controller/
│   └── ContractController.java # REST endpoints
├── service/
│   └── ContractService.java    # Logique métier
├── repository/
│   └── ContractRepository.java # Accès données
├── entity/
│   └── Contract.java           # JPA entity
└── config/
    └── SecurityConfig.java     # Configuration

src/main/resources/
├── application.properties      # Configuration (port, BD, etc)
└── schema.sql                  # DDL
```

```
# Serveur
server.port=8080
server.servlet.context-path=/api

# Base de données
```

# Contrôleurs REST Spring Boot

## Exemple complet de contrôleur:

```
@RestController
@RequestMapping("/api/contracts")
@Slf4j // Lombok logging
public class ContractController {

    private final ContractService service;

    @Autowired
    public ContractController(ContractService service) {
        this.service = service;
    }

    // GET /api/contracts?limit=10&offset=0
    @GetMapping
    public ResponseEntity&lt;List&lt;ContractDTO&gt;&gt; listContracts(
            @RequestParam(defaultValue = "10") int limit,
            @RequestParam(defaultValue = "0") int offset) {
        List&lt;ContractDTO&gt; contracts = service.list(limit, offset);
        return ResponseEntity.ok(contracts);
    }

    // GET /api/contracts/{id}
    @GetMapping("/{id}")
    public ResponseEntity&lt;ContractDTO&gt; getById(
```

# Services et Repository Pattern

## Service (Logique métier):

```
@Service
@Slf4j
public class ContractService {

    private final ContractRepository repo;
    private final PremiumCalculator calculator;
    private final EmailService emailService;

    @Autowired
    public ContractService(ContractRepository repo,
                           PremiumCalculator calculator,
                           EmailService emailService) {
        this.repo = repo;
        this.calculator = calculator;
        this.emailService = emailService;
    }

    @Transactional
    public ContractDTO create(CreateContractRequest req) {
        // Validation
        if (req.getCustomerId() == null)
            throw new InvalidContractException("Customer required");

        // Créer entité
```

# Spring Data JPA: Accès aux données

## Repository interface:

```
@Repository
public interface ContractRepository
    extends JpaRepository&lt;Contract, String&gt; {

    // Méthodes générées automatiquement:
    // save(T), delete(T), findById(ID), findAll(), etc.

    // Requêtes personnalisées (query methods)
    List&lt;Contract&gt; findByCustomerId(String customerId);

    List&lt;Contract&gt; findByType(String type);

    List&lt;Contract&gt; findByStatusAndCustomerId(
        String status, String customerId);

    // Requêtes JPQL/SQL natives
    @Query("SELECT c FROM Contract c WHERE c.premium > ?1")
    List&lt;Contract&gt; findHighPremium(double amount);

    @Query(value = "SELECT * FROM contracts WHERE active = true",
           nativeQuery = true)
    List&lt;Contract&gt; findAllActive();

    // Pagination et tri
```

# Spring Cloud: Microservices distribuées

## COMPOSANTS CLÉS:

**🔍 Service Discovery**
**▌ API Gateway**
**⚡ Circuit Breaker**
**▉ Distributed Tracing**

Framework pour construire des systèmes distribués

- Eureka: Service registry
- Consul: Service mesh
- Auto-registration et detection
- Spring Cloud Gateway
- Routing intelligent
- Load balancing
- Resilience4j, Hystrix

# Spring Cloud Config: Configuration centralisée

### Fichiers de config (application.yml):

Gérer la configuration des microservices depuis un endroit central

```
# config-repo/application.yml
server:
  port: 8080

spring:
  datasource:
    url: jdbc:postgresql://localhost/myapp
    username: admin
  jpa:
    hibernate:
      ddl-auto: update

# config-repo/application-prod.yml
server:
  port: 8080

spring:
  datasource:
    url: jdbc:postgresql://prod-db:5432/myapp
    username: prod-admin
    password: ${DB_PASSWORD} # Variable d'environnement
```

# Testing Spring Boot Applications

## Tests unitaires:

## Tests d’intégration:

```
@ExtendWith(MockitoExtension.class)
public class ContractServiceTest {

    @Mock
    private ContractRepository mockRepo;

    @InjectMocks
    private ContractService service;

    @Test
    public void shouldCreateContract() {
        // Given
        CreateContractRequest req = new CreateContractRequest(...);
        when(mockRepo.save(any()))
            .thenReturn(new Contract("123", 1200));

        // When
        ContractDTO result = service.create(req);

        // Then
        assertNotNull(result);
        assertEquals("123", result.getId());
        verify(mockRepo).save(any());
```

# Spring Security: Authentification & Autorisation

## Configuration Spring Security avec JWT:

Framework pour sécuriser les applications Spring

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
                .antMatchers("/api/login", "/api/register")
                    .permitAll()
                .antMatchers("/api/admin/**")
                    .hasRole("ADMIN")
                .anyRequest()
                    .authenticated()
            .and()
            .addFilterBefore(
                new JwtAuthenticationFilter(),
                UsernamePasswordAuthenticationFilter.class
            );
        return http.build();
```

# Monitoring: Spring Boot Actuator

**application.properties:**

## ENDPOINTS DISPONIBLES:

Endpoints pour monitorer la santé et les performances

```
# Activer Actuator
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always

# Ou limiter à certains endpoints
management.endpoints.web.exposure.include=health,metrics,info
```

| Endpoint | Description |
| --- | --- |
| /actuator/health | Santé générale (UP/DOWN) |
| /actuator/metrics | Métriques (CPU, mémoire, requêtes) |
| /actuator/prometheus | Format Prometheus (pour Grafana) |
| /actuator/loggers | Niveau de logging (modifiable) |

# Déploiement Spring Boot

## COMPILATION ET PACKAGING:

## AVEC DOCKER:

```
# Compiler et créer JAR
mvn clean package -DskipTests

# JAR créé: target/my-api-1.0.0.jar
# Exécutable standalone (embarque Tomcat)

# Lancer l'application
java -jar target/my-api-1.0.0.jar

# Avec variables d'environnement
java -Dspring.profiles.active=prod \
     -Dserver.port=8080 \
     -jar target/my-api-1.0.0.jar
```

```
FROM openjdk:11-jre-slim

WORKDIR /app

COPY target/my-api-1.0.0.jar app.jar

EXPOSE 8080
```

# Récapitulatif: Écosystème Spring Boot

## STACK COMPLET:

**Core**

**Operations**

## AVANTAGES:

- Spring Boot (app)
- Spring Data JPA (BD)
- Spring Security (auth)
- Spring Cloud (microservices)
- Actuator (monitoring)
- Logging (SLF4J)
- Docker (containerization)
- Kubernetes (orchestration)
- ✅ Configuration intelligente (convention over configuration)

# Node.js: Introduction

## CARACTÉRISTIQUES CLÉS:

### Installer Node.js:

JavaScript côté serveur - Runtime built on Chrome's V8 engine

- Event-driven: Basé sur les événements asynchrones
- Non-blocking I/O: N'attend pas les opérations disque/réseau
- Single-threaded: Un seul thread principal (avec worker threads)
- npm: Package manager avec millions de modules
- Cross-platform: Linux, macOS, Windows

```
# Via package manager (Linux)
sudo apt install nodejs npm

# Via Homebrew (macOS)
brew install node

# Via nvm (Node Version Manager - recommandé)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
nvm install 18.17.0
nvm use 18.17.0

# Vérifier
```

# npm: Gestion des dépendances

## Commandes essentielles:

## package.json structure:

Node Package Manager - Gérer les modules et dépendances

```
# Initialiser un projet
npm init -y

# Installer une dépendance
npm install express
npm i express            # Alias court

# Installer en développement (dev)
npm install --save-dev typescript
npm i -D typescript

# Installer une version spécifique
npm install express@4.18.2

# Installer toutes les dépendances (package.json)
npm install

# Mettre à jour les dépendances
npm update
npm outdated              # Voir les versions disponibles
```

# Express.js: Framework Web minimaliste

## Application Express basique:

Framework léger pour construire des APIs et applications web

```
const express = require('express');
const app = express();

// Middleware
app.use(express.json());
app.use(express.static('public'));

// Routes
app.get('/', (req, res) => {
  res.json({ message: 'Hello World!' });
});

// CRUD routes
app.get('/api/contracts', (req, res) => {
  res.json({ contracts: [] });
});

app.post('/api/contracts', (req, res) => {
  const { customerId, type } = req.body;
  res.status(201).json({ id: '123', customerId, type });
});
```

# NestJS: Framework pour Microservices

## CARACTÉRISTIQUES:

### Créer un projet NestJS:

Framework TypeScript avec architecture modulaire (inspiré par Angular)

- ✅ TypeScript natif (type-safe)
- ✅ Architecture modulaire (modules, controllers, services)
- ✅ Dependency Injection (intégré)
- ✅ Décorateurs (@Controller, @Get, @Post)
- ✅ Middleware et Guards (authentification)
- ✅ Testing intégré (Jest)

```
# Installer CLI
npm install -g @nestjs/cli

# Créer nouveau projet
nest new my-api

# Générer ressources
nest generate controller contracts
nest generate service contracts
nest generate module contracts
```

# Architecture NestJS

## STRUCTURE STANDARD:

### Module NestJS complet:

```
src/
├── contracts/
│   ├── contracts.module.ts       # Module (groupement)
│   ├── contracts.controller.ts   # Routes REST
│   ├── contracts.service.ts      # Logique métier
│   ├── contracts.entity.ts       # Entity (TypeORM)
│   └── dto/
│       ├── create-contract.dto.ts
│       └── update-contract.dto.ts
├── app.module.ts                 # Root module
├── main.ts                        # Entry point
└── common/
    ├── guards/
    ├── interceptors/
    └── decorators/
```

```
// contracts.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ContractsController } from './contracts.controller';
import { ContractsService } from './contracts.service';
import { Contract } from './contracts.entity';
```

# NestJS: Middleware, Guards & Interceptors

## Guard: Authentification avec JWT

Pipeline de traitement des requêtes

```
@Injectable()
export class JwtAuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
    const authHeader = request.headers.authorization;

    if (!authHeader) return false;

    const token = authHeader.replace('Bearer ', '');
    try {
      const payload = this.jwtService.verify(token);
      request.user = payload;
      return true;
    } catch (e) {
      return false;
    }
  }
}

// Utilisation
```

# TypeORM: ORM pour Node.js

## Entity TypeORM:

Object-Relational Mapping pour TypeScript

```
import { Entity, Column, PrimaryGeneratedColumn,
         OneToMany, CreateDateColumn } from 'typeorm';

@Entity('contracts')
export class Contract {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  customerId: string;

  @Column({ type: 'varchar', length: 50 })
  type: string;

  @Column({ type: 'decimal', precision: 10, scale: 2 })
  premium: number;

  @Column({ default: 'ACTIVE' })
  status: string;

  @OneToMany(() => Claim, claim => claim.contract)
  claims: Claim[];
```

# Testing avec Jest

## Test unitaire NestJS:

Framework de test pour Node.js et NestJS

```
import { Test, TestingModule } from '@nestjs/testing';
import { ContractsService } from './contracts.service';
import { ContractsController } from './contracts.controller';
import { getRepositoryToken } from '@nestjs/typeorm';
import { Contract } from './contracts.entity';

describe('ContractsService', () => {
  let service: ContractsService;
  let mockRepo: any;

  beforeEach(async () => {
    // Mock repository
    mockRepo = {
      find: jest.fn().mockResolvedValue([
        { id: '1', premium: 1200 }
      ]),
      findOne: jest.fn().mockResolvedValue(
        { id: '1', premium: 1200 }
      ),
      save: jest.fn().mockResolvedValue(
        { id: '1', premium: 1200 }
      ),
```

# Déploiement Node.js

## PM2 (PROCESS MANAGER):

## DOCKER:

## HEROKU:

```
# Installer PM2
npm install -g pm2

# Lancer application
pm2 start app.js

# Lancer en cluster mode (utiliser tous les cores)
pm2 start app.js -i max

# Monitorage
pm2 monit

# Logs
pm2 logs

# Persister après reboot
pm2 startup
pm2 save
```

```
FROM node:18-alpine
```

# Récapitulatif: Écosystème Node.js

## STACK NODE.JS COMPLET:

**Frameworks**

**Écosystème**

## AVANTAGES:

- Express: Minimaliste, flexible
- NestJS: Modulaire, TypeScript
- Fastify: Haute performance
- Koa: Middleware elegance
- npm: Package manager
- TypeORM: ORM
- Jest: Testing
- PM2: Process management
- ✅ JavaScript partout (frontend + backend)

# Frameworks Front-end: Panorama

## LES 3 GRANDS:

Frameworks JavaScript pour construire des interfaces utilisateur modernes

| Framework | Philosophie | Courbe d’apprentissage | Popularité |
|---|---|---|---|
| React | Component-based, JSX | Modérée | ⭐⭐⭐⭐⭐ (Très populaire) |
| Vue | Progressive, HTML-first | Facile | ⭐⭐⭐⭐ (Croissant) |
| Angular | Full-featured, TypeScript | Élevée | ⭐⭐⭐ (Entreprises) |

# React: Composants et JSX

## Composant fonctionnel React:

Library JavaScript pour construire des UIs avec des composants réutilisables

```
import React, { useState } from 'react';

// Composant fonctionnel avec Hooks
export function ContractForm({ onSubmit }) {
  const [formData, setFormData] = useState({
    customerId: '',
    type: 'AUTO'
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData(prev => ({
      ...prev,
      [name]: value
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    onSubmit(formData);
    setFormData({ customerId: '', type: 'AUTO' });
  };
```

# React Hooks: État et effets

## Hooks courants:

Réutiliser la logique avec des fonctions au lieu de classes

```
import { useState, useEffect, useContext, useReducer } from 'react';

// useState - Gérer l'état local
function ContractList() {
  const [contracts, setContracts] = useState([]);
  const [loading, setLoading] = useState(true);

  return &lt;&gt;...&lt;/&gt;;
}

// useEffect - Effets secondaires (API calls, timers)
useEffect(() => {
  const fetchContracts = async () => {
    const response = await fetch('/api/contracts');
    const data = await response.json();
    setContracts(data);
    setLoading(false);
  };
  fetchContracts();
}, []); // Dépendance vide = une seule exécution au mount

// useContext - Accéder au contexte global
```

# Gestion d’état en React

### Context API (intégré)

### Redux (library)

Gérer l’état global de l’application

// Créer contexte const UserContext = createContext(); // Provider function UserProvider({ children }) { const [user, setUser] = useState(null); return ( <UserContext.Provider value=> {children} </UserContext.Provider> ); } // Utiliser function MyComponent() { const { user } = useContext(UserContext); return {user.name} ; } Quand l’utiliser: État simple/petit

// Store + Reducer const store = createStore((state = {}, action) => { switch (action.type) { case 'ADD_CONTRACT': return { ...state, contracts: [...state.contracts, action.payload] }; default: return state; } }); // Actions const addContract = (contract) => ({ type: 'ADD_CONTRACT', payload: contract }); // Component function MyComponent() { const dispatch = useDispatch(); const contracts = useSelector( state => state.contracts ); return ( <button onClick={() => dispatch(addContract({...}))} >Add</button> ); } Quand l’utiliser: État complexe/gros apps

```
// Créer contexte
const UserContext = createContext();

// Provider
function UserProvider({ children }) {
  const [user, setUser] = useState(null);
```

# Vue.js: Progressive Framework

## Composant Vue:

Framework approachable et performant

```
&lt;template&gt;
  &lt;div class="contract-form"&gt;
    &lt;form @submit.prevent="submitForm"&gt;
      &lt;input
        v-model="form.customerId"
        placeholder="Customer ID"
      /&gt;
      &lt;select v-model="form.type"&gt;
        &lt;option&gt;AUTO&lt;/option&gt;
        &lt;option&gt;HOME&lt;/option&gt;
      &lt;/select&gt;
      &lt;button type="submit"&gt;Create&lt;/button&gt;
    &lt;/form&gt;

    &lt;div v-if="loading"&gt;Loading...&lt;/div&gt;
    &lt;div v-else-if="contracts.length"&gt;
      &lt;ul&gt;
        &lt;li v-for="c in contracts" :key="c.id"&gt;
          {{ c.type }} - {{ c.premium }}
        &lt;/li&gt;
      &lt;/ul&gt;
    &lt;/div&gt;
```

# Angular: Full-Featured Framework

## CARACTÉRISTIQUES:

### Structure Angular:

Framework complet TypeScript pour applications enterprise

- TypeScript natif (type-safe)
- Dependency Injection (intégré)
- RxJS & Observables (programmation réactive)
- Services et Components (architecture claire)
- Form handling (réactif et template-driven)
- Routing (navigation avancée)
- Testing (Jasmine + Karma intégrés)

```
src/app/
├── contracts/
│   ├── contract.model.ts
│   ├── contract.service.ts
│   ├── contract-list/
│   │   ├── contract-list.component.ts
│   │   └── contract-list.component.html
│   └── contract-form/
│       ├── contract-form.component.ts
```

## Comparaison: React vs Vue vs Angular

| Aspect | React | Vue | Angular |
|---|---|---|---|
| Courbe d’apprentissage | Modérée | Facile | Élevée |
| TypeScript | Optionnel | Optionnel | Natif |
| Taille bundle | ~50KB (minifié) | ~30KB (minifié) | ~500KB (minifié) |
| Performance | Excellente | Excellente | Bonne |
| Écosystème | Énorme | Croissant | Complet (intégré) |
| Gestion d’état | Redux, Zustand | Pinia, Vuex | NgRx (intégré) |
| Idéal pour | Apps modernes, startups | Projets rapides, prototype | Grandes entreprises |
| Emploi/marché | ⭐⭐⭐⭐⭐ Très demandé | ⭐⭐⭐⭐ Croissant | ⭐⭐⭐ Enterprise |

# Build Tools: Webpack, Vite, Parcel

### ⬢ Webpack

### ⚡ Vite

### 📄 Parcel

**Créer projet React avec Vite:**

Outils pour bundler et optimiser le code front-end

- Standard (CRA, Vue CLI)
- Très configurable
- Lent à démarrer
- Build complet
- Très rapide (ES modules)
- Dev server instant
- Configuration simple
- Recommandé 2026

# Styling Front-end: Options modernes

## APPROCHES DE STYLING:

- **CSS Modules**
- **CSS-in-JS (Styled Components)**
- **Tailwind CSS**
- **BEM (CSS traditionnel)**

```
/* styles.module.css */
.container {
  padding: 20px;
  background: #f0f0f0;
}
```

```
import styles from './styles.module.css';
&lt;div className={styles.container}&gt;...&lt;/div&gt;
```

```
import styled from 'styled-components';

const Container = styled.div`
  padding: 20px;
  background: #f0f0f0;
  &:hover {
    background: #e0e0e0;
  }
`;
```

# Récapitulatif: Écosystème Front-end

## STACK FRONT-END MODERNE (2026):

**Base**

**Tooling**

## CONCEPTS CLÉS:

- Framework: React (+ Vite)
- Styling: Tailwind ou Styled-Components
- State: Zustand ou Redux
- Routing: React Router v6
- Build: Vite
- Testing: Vitest + React Testing Library
- Linting: ESLint
- Formatting: Prettier
- ✅ Composants réutilisables

# Développement Mobile: Panorama

## 3 APPROCHES PRINCIPALES:

Approches pour développer des applications mobiles

| Approche | Langages | Performance | Temps de dev |
|---|---|---|---|
| Native | Swift (iOS) / Kotlin (Android) | ⭐⭐⭐⭐⭐ Excellente | ⏱️ Lent (deux bases de code) |
| Cross-platform | React Native / Flutter | ⭐⭐⭐⭐ Très bonne | ⏱️⏱️ Modéré (une base de code) |
| Web App | HTML/CSS/JS (Responsive) | ⭐⭐⭐ Correcte | ⏱️⏱️⏱️ Rapide |

# React Native: JavaScript sur iOS/Android

## Application React Native:

Framework pour construire des apps mobiles avec React

```
import React, { useState, useEffect } from 'react';
import { View, Text, TextInput, TouchableOpacity,
         FlatList, StyleSheet, SafeAreaView } from 'react-native';

export default function ContractApp() {
  const [contracts, setContracts] = useState([]);
  const [customerId, setCustomerId] = useState('');

  useEffect(() => {
    fetchContracts();
  }, []);

  const fetchContracts = async () => {
    const response = await fetch('https://api.example.com/contracts');
    const data = await response.json();
    setContracts(data);
  };

  const createContract = async () => {
    const response = await fetch('https://api.example.com/contracts', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
```

# Architecture React Native

## Setup Expo (recommandé pour débutants):

Comment React Native communique avec iOS/Android

```
# Installer Expo CLI
npm install -g expo-cli

# Créer nouveau projet
expo init my-app
cd my-app

# Lancer sur téléphone/simulateur
expo start

# Scanner QR code ou appuyer 'i' (iOS) ou 'a' (Android)
```

# Flutter: Framework cross-platform Google

## AVANTAGES FLUTTER:

### Application Flutter simple:

Développer des apps iOS/Android/Web avec Dart

- ✅ Performance native: Compilé vers ARM (pas de bridge)
- ✅ UI attractive: Material Design et Cupertino (iOS)
- ✅ Hot reload: Modifier le code et voir en temps réel
- ✅ Une base de code: iOS, Android, Web, Desktop
- ✅ Dart: Langage moderne, typé
- ✅ Moins de code: que React Native (généralement)

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp();

  @override
  Widget build(BuildContext context) {
```

# Native Development: Swift & Kotlin

## Swift (iOS)

## Kotlin (Android)

Développement natif pour apps haute performance

```
import Foundation

struct Contract {
  let id: String
  let type: String
  let premium: Double
}

class ContractService {
  private let baseURL =
    "https://api.example.com"

  func fetchContracts()
    async throws -> [Contract] {
    let url = URL(string:
      "\(baseURL)/contracts")!
    let (data, _) = try await
      URLSession.shared
        .data(from: url)
    return try JSONDecoder()
      .decode([Contract].self,
```

## Comparaison: React Native vs Flutter vs Native

| Critère | React Native | Flutter | Native (Swift/Kotlin) |
|---|---|---|---|
| Langage | JavaScript | Dart | Swift / Kotlin |
| Performance | ⭐⭐⭐⭐ (Bridge lent) | ⭐⭐⭐⭐⭐ (Compilé) | ⭐⭐⭐⭐⭐ (Optimisé) |
| Courbe d’apprentissage | Facile (React known) | Modérée | Élevée |
| Écosystème | ⭐⭐⭐⭐⭐ (npm) | ⭐⭐⭐⭐ (pub.dev) | ⭐⭐⭐⭐ (natif) |
| Base de code | 1 (iOS + Android) | 1 (iOS + Android + Web) | 2 (ios/ android/) |
| Temps de dev | ⏱️⏱️⏱️ Rapide | ⏱️⏱️⏱️ Rapide | ⏱️⏱️ Lent |
| Idéal pour | Apps classiques, startups | Apps complètes (multi-plateforme) | Haute performance, jeux |

# Architecture Mobile: Offline-first

## Implémentation offline-first:

Gérer la connectivité instable des appareils mobiles

```
// React Native avec AsyncStorage + WatermelonDB

import { Database } from '@nozbe/watermelondb';
import SQLiteAdapter from '@nozbe/watermelondb/adapters/sqlite';
import NetInfo from '@react-native-community/netinfo';

class SyncService {
  async syncData() {
    const isConnected = await NetInfo.fetch();

    if (!isConnected.isConnected) {
      // Offline: sauvegarder localement
      await this.saveLocalCache(data);
      await this.queueForSync(data);
      return;
    }

    // Online: synchroniser avec le serveur
    const queuedActions = await this.getQueuedActions();
    for (const action of queuedActions) {
      await this.sendToServer(action);
    }
```

# Gestion des données mobiles

**SQLite (Native)**

**Realm (Mobile)**

**Hive (Flutter)**

**AsyncStorage (React Native)**

Stocker et synchroniser les données localement

- ✅ SQL queries
- ✅ Relations complexes
- ✅ Très stable
- ❌ Syntaxe verbose
- ✅ Synchronisation Cloud
- ✅ API simple (objets)
- ✅ Transactions ACID
- ❌ Licence propriétaire (gratuit)

# Performance Mobile: Optimisations

## OPTIMISATIONS CRITIQUES:

**Rendu optimisé**

**Gestion mémoire**

**Bundle size**

**Network**

- ✅ Code splitting
- ✅ Tree shaking (remove dead code)
- ✅ Compression (gzip)
- ✅ Lazy loading images
- ❌ Énormes assets
- ✅ GraphQL (moins de données)
- ✅ Pagination et pagination
- ✅ Cache HTTP (ETags)

# Distribution: App Stores

## 🍎 Apple App Store

## 🤖 Google Play Store

### Commandes pour Expo (React Native):

Déployer l’application sur les stores

- 1. Rejoindre Apple Developer Program (99$/an)
- 2. Créer App ID dans Developer Portal
- 3. Générer certificats et provisioning profiles
- 4. Archive l’app avec Xcode
- 5. Soumettre via TestFlight
- 6. Approbation (1-48h)
- 1. Rejoindre Google Play Developer Program (25$ une fois)
- 2. Créer signingKey
- 3. Build APK/AAB avec Android Studio
- 4. Remplir le formulaire dans Play Console
- 5. Soumettre pour review
- 6. Approbation (~2h)

# Récapitulatif: Développement Mobile

## STACK MOBILE RECOMMANDÉ (2026):

**Cross-platform**

**Native (si needed)**

**Tooling**

## CHECKLIST DÉPLOIEMENT:

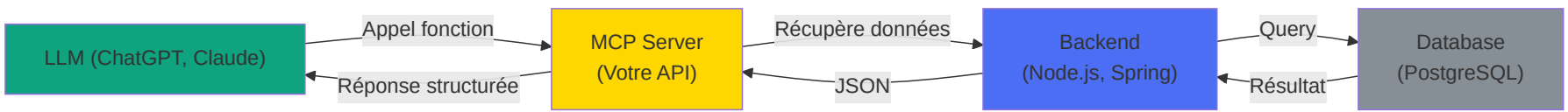
- Framework: React Native ou Flutter
- State: Redux ou Provider
- API: REST ou GraphQL
- Storage: SQLite/Realm/Hive
- iOS: Swift + SwiftUI
- Android: Kotlin + Jetpack
- IDE: Xcode, Android Studio
- Testing: XCTest, JUnit

# 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

```
// Node.js/Express avec MCP SDK
const mcp = require('@anthropic-sdk/mcp');
const express = require('express');

const server = new mcp.MCPServer({
  name: 'insurance-api',
  version: '1.0.0'
});

// Enregistrer des ressources/outils
server.resource('contract', async (id) => {
  const contract = await db.contracts.findOne(id);
  return {
    type: 'contract',
    id,
    data: contract
  };
});

server.tool('create_claim', {
  description: 'Créer un sinistre',
```

# Exposer les APIs pour l'IA

BEST PRACTICES:

**■ Schémas clairs**

**🔑 Authentification**

**▲ Limitations & Guardrails:**

Préparer votre backend pour l'intégration IA

- Rate limiting: Max 100 requêtes/min pour IA
- Scopes: L'IA ne peut accéder qu'aux données appropriées
- Validation: Valider tous les inputs
- Logs: Auditer toutes les actions IA

```
{
  "contract": {
    "id": "string",
    "customerId": "string",
    "type": "enum(AUTO|HOME|HEALTH)",
    "premium": {
      "type": "number",
      "minimum": 0,
      "unit": "EUR"
    },
    "status": "enum(ACTIVE|EXPIRED)",
```

# Use Cases: IA dans Assurance/Santé

## CAS D'USAGE ASSURANCE:

**🟪 Analyse automatique de sinistres**

**🟪 Recommandations personnalisées**

**🟪 Génération de documents**

## CAS D'USAGE SANTÉ:

Flux: Client décrit sinistre → Claude analyse → Extraction automatique données → Création claim dans BDD → Notation de risque

// Prompt exemple const prompt = `Tu es un expert en assurance automobile. Analyse ce sinistre: "${claimDescription}" Extrais les informations dans ce format JSON: { "type": "collision|theft|damage", "severity": "low|medium|high", "estimatedAmount": number, "actionRequired": string[] }`; 2️⃣ Recommandations personnalisées Flux: Historique client → Claude analyse → Produits recommandés → Propositions adaptées

Flux: Données contrat → Claude génère → Email/PDF avec détails clause personnalisées

- 📋 Diagnostic assistance: Analyse symptômes + historique → suggestions
- 💊 Gestion médicaments: Détection interactions, contrindications
- 📊 Rapports médicaux: Génération automatique résumés

# Sécurité & Gouvernance: IA en production

POINTS CRITIQUES:

**🔒 Sécurité données**

**⚖️ Conformité légale**

**Architecture sécurisée:**

Protéger les données et respecter la réglementation

- ✅ Chiffrer données avant LLM
- ✅ Pas d’infos sensibles en prompt
- ✅ PII masking/tokenization
- ✅ Utiliser des modèles privés
- ✅ RGPD (droit à l’oubli)
- ✅ HIPAA (santé US)
- ✅ Explainabilité IA
- ✅ Audit trail complet

# Monitoring: IA en production

## Métriques à tracker:

Surveiller la qualité et la performance des réponses IA

```
// Instrumenter les appels IA
const aiMetrics = {
  // Performance
  latency: new Histogram('ai_latency_ms'),
  tokenUsage: new Counter('ai_tokens_used'),
  costs: new Gauge('ai_monthly_cost'),

  // Qualité
  hallucinations: new Counter('ai_hallucinations'),
  userRejections: new Counter('ai_responses_rejected'),
  accuracy: new Gauge('ai_accuracy_score'),

  // Erreurs
  rateLimitExceeded: new Counter('ai_rate_limit'),
  timeouts: new Counter('ai_timeouts'),
  authErrors: new Counter('ai_auth_errors')
};

// Instrumenter
const startTime = Date.now();
try {
  const response = await llm.analyze(data);
```

# Futur: Agents IA autonomes

## Exemple: Traitement sinistre automatique

La prochaine génération: agents capables de décisions autonomes

```
// Agent autonome
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 autonomement:
// 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!
```

# Récapitulatif: MCP & IA en Production

## ARCHITECTURE COMPLÈTE:

**Backend side**

**LLM side**

## ROADMAP 2026:

- API: REST/GraphQL
- MCP Server: Expose ressources
- Auth: OAuth2/tokens
- Audit: Logs détaillés
- Model: Claude, GPT-4
- Tools: API calls structurées
- Agents: Loop autonome
- Monitoring: Métriques qualité
- ✅ Phase 1: Exposer APIs simples (GET)

# Conclusion: Architectures Modern (2026)

## CE QUE VOUS AVEZ APPRIS:

**Backend**

**Frontend/Mobile**

**Advanced**

**Tendances 2026+**

- ✅ Spring Boot (enterprise)
- ✅ Node.js/NestJS (moderne)
- ✅ Clean Architecture
- ✅ API design (REST/GraphQL)
- ✅ Microservices patterns
- ✅ React (standard)
- ✅ State management
- ✅ React Native/Flutter

# Architecture Decision Matrix

Choisir la bonne stack selon vos contraintes

| Projet | Backend | Frontend | Mobile | Database |
|---|---|---|---|---|
| Startup SaaS(Rapide) | Node.js + NestJS | React + Vite | React Native (Expo) | PostgreSQL |
| Assurance (Web)(Scalabilité) | Spring Boot | React + Redux | Native Swift/Kotlin | Oracle/PostgreSQL |
| E-commerce(Performance) | Node.js | Next.js (SSR) | Flutter | PostgreSQL + Redis |
| Santé(Sécurité) | Spring Boot (Java) | Angular | Native Swift/Kotlin | HIPAA PostgreSQL |
| MVP/Prototype(Temps minimum) | Firebase/Supabase | Vue.js | Flutter | Firebase |

# Critères de sélection technologique

## ■ Critères positifs

## ▲ Signes d'alerte

## ● Framework de décision (2-3 min par techno):

Évaluer les technologies avant de les choisir

- Maturité: Utilisé en production
- Communauté: Actif, nombreux ressources
- Documentation: Complète et claire
- Performance: Benchmarks publics
- Scalabilité: Peut grandir
- Maintenabilité: Code lisible, patterns clairs
- Emploi: Demande marché
- Hype: Nouvelle techno = risque
- Dépendances: Trop de libs externes

# DevOps: Déploiement en production

## PIPELINE CI/CD STANDARD:

### Terraform (Infrastructure as Code):

Passer du développement à la production

```
# AWS ECS + RDS
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "eu-west-1"
}

# RDS Database
resource "aws_db_instance" "contracts" {
  identifier     = "contracts-db"
  engine         = "postgres"
  engine_version = "14.0"
  instance_class = "db.t3.micro"
  allocated_storage = "20"
```

# Sécurité: Bonnes pratiques

## 🔒 Application Security

## 🔐 Data Security

## 🛡️ Infrastructure Security

## 🔍 Monitoring & Audit

Protéger l’application et les données

- ✅ HTTPS (TLS 1.3+)
- ✅ OWASP Top 10
- ✅ Input validation
- ✅ SQL Injection prevention
- ✅ XSS protection (CSP)
- ✅ CSRF tokens
- ✅ Rate limiting
- ✅ CORS configuration

# Performance: Monitoring & Optimization

## ■ Métriques clés (APM)

## 🚀 Optimisations backend

## ⚡ Optimisations frontend

## 🛠️ Tools

Mesurer et optimiser les performances

- Latency: p50/p95/p99 (ms)
- Throughput: req/sec
- Error rate: % requêtes
- Availability: uptime %
- CPU/Memory: utilisation
- Database: query time
- Cache hit: %
- Cold starts: serverless

# Organisation: Structure d’équipe

## PETIT PROJET (3-5 PERSONNES):

## PROJET MOYEN (15-20 PERSONNES):

**Responsabilités clés:**

Organiser les équipes pour la scalabilité

- Backend: API design, database, scalability
- Frontend: UX, performance, responsive
- Mobile: App store releases, native features
- DevOps: Deployment, monitoring, security
- QA: Testing, quality, regression
- Tech Lead: Architecture, decisions, mentoring

```
Tech Lead (1)
├── Full-stack Developer (2)
├── DevOps/Infrastructure (1)
└── QA/Testing (1)

Rôles croisés: tout le monde touche à tout
```

```
Engineering Manager (1)
├── Backend Chapter Lead (1)
```

# Carrière: Progression dans le tech

**Individual Contributor**

**Leadership**

**Spécialisation**

Évolution de carrière en 2026

💰 Salaire: €30k → €120k+

💰 Salaire: €45k → €200k+

💰 Potentiel: €100k → ∞

- Junior Dev (0-2 ans)
- Mid Dev (2-5 ans)
- Senior Dev (5-8 ans)
- Staff/Principal (8+ ans)
- Tech Lead (IC → Lead)
- Manager (People mgmt)

# Apprentissage continu: Rester à jour

**🎓 Ressources gratuites**

**💰 Ressources payantes**

**🧠 Best practices learning**

**⚡ À ne pas négliger**

L’industrie change tous les 2-3 ans

- YouTube: Tech talks, tutorials
- Dev.to: Articles techniques
- GitHub: Code open-source
- Podcasts: Tech interviews
- Docs officielles: React, Node, Spring
- Twitter/X: Tech community
- Open Source: Contribuer
- Udemy: Cours structurés ($)

# Ressources & Références

## 📚 Ouvrages de Référence

**Clean Code** - Robert C. Martin

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

**Clean Architecture** - Robert C. Martin

> "A software architect is a programmer who has stopped programming and has started thinking about programs."

**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

### 🔧 Documentation officielle

- Spring Boot: https://spring.io/projects/spring-boot
- NestJS: https://nestjs.com
- React: https://react.dev
- Vue.js: https://vuejs.org
- Angular: https://angular.io
- Python/FastAPI: https://fastapi.tiangolo.com
- .NET: https://dotnet.microsoft.com

### 🎓 Plateformes d'apprentissage

- Coursera (Spécialisations complètes)
- Frontend Masters (Spécialistes reconnus)
- Udemy (Cours pratiques)
- LeetCode (Algorithmes & entretiens)
- Educative (Tutoriels interactifs)
- YouTube (Chaînes techniques)

### 🌐 Communautés & Forums

- Stack Overflow
- Dev.to

- Dev.to
- Hacker News
- GitHub (Open source)
- Discord (Communities)
- Meetup (Local)
- Conferences: JSConf, React Conf, etc.

# Insights clés de cette formation

- **Pas de technologie universelle**
- **Fondamentaux > frameworks**
- **DevOps c’est la production**
- **Sécurité dès le départ**
- **La scalabilité humaine**
- **L’IA change le jeu**

Les leçons importantes

Chaque choix tech est un trade-off: performance vs facilité, coût vs flexibilité. Évaluer vos priorités.

Comprendre l’architecture, les patterns, les principes est plus important que maîtriser React ou Spring Boot.

Le code qui fonctionne en dev = 0. Seul le code en prod compte. Investir dans CI/CD et monitoring.

Ajouter la sécurité après est 10x plus cher. Planifier dès l’architecture initiale.

C’est pas juste du code - c’est comment l’équipe grandit. Clean code et documentation sauvent des vies.

2026 onwards: l’IA n’est pas optionnel. Apprendre à utiliser les LLMs et MCP dès maintenant.

# Prochaines étapes: De la théorie à la pratique

## ⬛ Semaine 1

## ⬛ Semaines 2-4

## ⬛ Semaines 5-8

## ⬤ Projets recommandés pour pratiquer:

Appliquer ce que vous avez appris

- Choisir un projet
- Évaluer technologies
- Setup dev env
- Premiere requête HTTP
- Implémenter API
- Build frontend
- Connecter frontend/backend
- Tests unitaires

# Questions & Discussion

## QU’AVEZ-VOUS ENVIE DE DISCUTER?

✋ Levez la main pour poser vos questions 💬 Débat sur technologies, architecture, ou carrière 🤔 Cas d’usage spécifiques à votre contexte

Temps estimé: 30-45 minutes

Pas de question bête - cette partie est pour VOUS

# Contact & Ressources Finales

## RESTER EN CONTACT

**Email**

**LinkedIn**

**GitHub**

**Certification**

**Ressources partagées de cette formation:**

contact@example.com

Pour questions/retours

linkedin.com/company/example

Suivre nos updates

github.com/example

Code examples & projets

Attestation formation

Disponible après complétion

# Merci! 🙏

## MERCI POUR CES 6 HEURES!

**Vos prochains défis:**

Vous avez maintenant les fondamentaux pour construire des architectures back-end & front-end modernes en 2026+ ✨

- 🚀 Construire un projet personnel
- 📚 Approfondir une technologie
- 🤝 Contribuer à l’open source
- 💼 Partager vos connaissances
- 🌱 Continuer à apprendre

# Bonus: Glossaire Technique

## A-M

## N-Z

- API: Application Programming Interface
- ACID: Atomicity, Consistency, Isolation, Durability
- CI/CD: Continuous Integration/Deployment
- CORS: Cross-Origin Resource Sharing
- CRUD: Create, Read, Update, Delete
- DevOps: Development + Operations
- Docker: Container orchestration
- DPI: Dependency Injection
- GraphQL: Query language for APIs
- HTTPS: HTTP Secure (TLS/SSL)