

# Architectures Back-end & Front-end

Web, Mobile et IA

**Durée:** 6 heures

**Public cible:** Développeurs, Architectes, Chefs de projet

**Approche:** Théorie + Études de cas (assurance/santé)

# POURQUOI L'ARCHITECTURE LOGICIELLE EST CRUCIALE ?

## 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
<b>Monolithe</b>	Simple, facile à déployer	Difficile à scaler, couplage fort
<b>Microservices</b>	Scalable, indépendant	Complexité opérationnelle
<b>Serverless</b>	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.

# 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

# PRINCIPES FONDAMENTAUX

## 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)
```

**Bénéfice:** Testabilité et maintenabilité améliorées

# SOLID & BONNES PRATIQUES

## 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

# DÉFIS DE L'ARCHITECTURE MODERNE



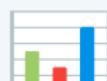
## PERFORMANCE

- Latence réduite
- Caching efficace
- Scalabilité



## SÉCURITÉ

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



## SCALABILITÉ



## MAINTENABILITÉ

# CAS D'USAGE: ASSURANCE

## PLATEFORME DE GESTION DE CONTRATS



Syntax error in text  
mermaid version 11.12.2

**Défi:** Gérer des milliers de contrats simultanément avec calculs complexes et conformité légale

# CAS D'USAGE: SANTÉ

## PLATEFORME DE TÉLÉCONSULTATION

**Défi:** Garantir la sécurité des données sensibles, la conformité RGPD et la disponibilité 24/7

# POURQUOI UTILISER DES PATTERNS ?

## LES PATTERNS RÉSOLVENTS DES PROBLÈMES RÉCURRENTS

- **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)



Syntax error in text  
mermaid version 11.12.2

## 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  $\leftrightarrow$  ViewModel
- **Testabilité:** ViewModel indépendant de la Vue
- **Réactivité:** Mises à jour temps réel

**Popularité:** Angular, WPF, Xamarin. Idéal pour interfaces

# PATTERN CQRS (COMMAND QUERY RESPONSIBILITY SEGREGATION)

Séparer les modèles de lecture et écriture.



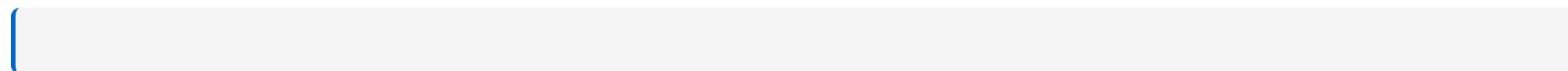
Syntax error in text  
mermaid version 11.12.2

## AVANTAGES:

- Optimisation indépendante des lectures et écritures

# PATTERN EVENT-DRIVEN ARCHITECTURE

Services réactifs aux événements métiers asynchrones.



## CAS D'USAGE ASSURANCE:

- **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)

Isoler le cœur métier des détails techniques.



Syntax error in text  
mermaid version 11.12.2

## BÉNÉFICES:

- 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(); //  
  
    public void createContract(Contract c) {  
        db.save(c);  
    }  
}
```

# PATTERN REPOSITORY

Abstraction de la couche d'accès aux données.



Syntax error in text  
mermaid version 11.12.2

## AVANTAGES:

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

# PATTERN STRATEGY

Encapsuler une famille d'algorithmes interchangeables.

## EXEMPLE: CALCUL DE PRIME D'ASSURANCE

### STRATEGY: BASEPRICINGSTRATEGY

```
interface PricingStrategy {  
    calculatePremium(contract);  
}
```

```
class Standard  
    implements PricingStrategy { .
```

### UTILISATION

```
ContractService {  
    strategy: PricingStrategy;  
  
    calculate() {  
        return this.strategy  
            .calculatePremium(contract);  
    }  
}
```

# 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	Découpage asynchrone	Microservices, systèmes réactifs

# CLEAN CODE: INTRODUCTION

## POURQUOI LE 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

**Statistique:** 80% du temps de développement est consacré à la

# CODE SALE VS CODE PROPRE

## CODE SALE (MAUVAIS)

```
function calc(c) {  
    let p = 0;  
    if (c.age < 25)  
        p = c.sal * 0.1;  
    else if (c.age < 65)  
        p = c.sal * 0.1;  
    else  
        p = c.sal * 0.2;  
  
    // TODO: ajuster ta
```

## 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

**Conseil:** Préférer long et clair plutôt que court et vague

# 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;
```

# 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 = 1;
return premium;
```

```
// CustomerService
double amount = salary * 0.1;
```



## EXTRACTION EN MÉTHODE

```
// PricingCalculator
private double calculateAmount
    double salary) {
    double amount = salary * 0.1;
    return Math.max(amount, 10);
```

# 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) {  
    // Boucle sur les éléments
```



## 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 calculatePremium(...)
```

# 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

Code testable = code découplé

## PROPRIÉTÉS D'UN CODE TESTABLE:

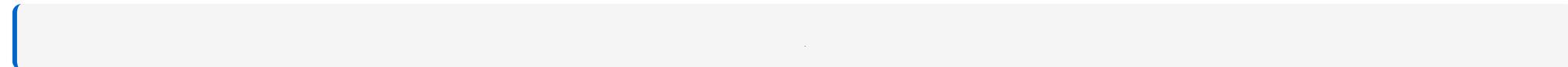
-  Dépendances injectées (pas "new Database()")
-  Logique métier indépendante du framework
-  Pas de singltons globaux
-  Pas d'appels à des APIs externes en dur
-  Méthodes courtes et déterministes

# 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

Structure logicielle indépendante des frameworks, testable et maintenable.



**Principe:** Les couches intérieures ne dépendent jamais des couches extérieures

# LES 4 COUCHES DE CLEAN ARCHITECTURE

## 1 ENTITIES (CŒUR MÉTIER)

Objets métiers purs, pas de frameworks

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

## 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(. . .);  
        repo.save(c);  
    }  
}
```

# DIRECTION DES DÉPENDANCES

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



Syntax error in text  
mermaid version 11.12.2



*Entities peuvent être testées sans aucune dépendance externe*

# STRUCTURE DE PROJET CLEAN ARCHITECTURE

```
src/
└── main/java/com/myapp/
    ├── domain/          # 🟢 Entities
    │   ├── Contract.java
    │   ├── Customer.java
    │   └── ContractRepository.java (interface)
    └── application/     # 💛 Use Cases
```

# CAS D'USAGE: CREATECONTRACTUSECASE

## ÉTAPES DU PROCESSUS:

- 1. Validation:** Vérifier les données d'entrée
- 2. Création entité:** Instancier l'objet Contract
- 3. Calcul de prime:** Appliquer les règles métier
- 4. Persistance:** Sauvegarder en BD via Repository
- 5. Notification:** Envoyer email de confirmation

# TESTS UNITAIRES FACILES

Un avantage clé de Clean Architecture: testabilité.

## TEST DU CREATECONTRACTUSECASE

```
public class CreateContractUseCaseTest {  
    private CreateContractUseCase useCase;  
    private ContractRepository mockRepo;  
    private EmailService mockEmail;  
    private PremiumCalculator mockCalc;  
  
    @Before  
    public void setup() {  
        // Créer des mocks (faux objets)
```

# AVANTAGES DE CLEAN ARCHITECTURE



## POUR LE DÉVELOPPEMENT

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



## POUR LA MAINTENANCE

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

# PIÈGES À ÉVITER



## OVER-ENGINEERING

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

**Conseil:** Adapter la complexité aux besoins



## ENTITIES CONTAMINÉES

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

**Conseil:** Entities = POJO purs



# 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

### 7 règles:

1. Nommage clair
2. Fonctions courtes
3. Gestion erreurs
4. DRY
5. Commentaires utiles



## CLEAN ARCHITECTURE

### 4 couches:

1. Entities (métier pur)
2. Use Cases (appli)
3. Adapters (interface)
4. Frameworks (détails)

# 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

# REST: PRINCIPES FONDAMENTAUX

**REST:** Representational State Transfer

**PRINCIPES CLÉS:**

- **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

# REST: BONNES PRATIQUES

## BEST PRACTICES POUR UNE API REST ROBUSTE:



### SÉCURITÉ

- **OAuth2:** Authentification
- **JWT:** Token sans état
- **HTTPS:** Chiffrement
- **Rate limiting:** Protection DOS
- **CORS:** Contrôle d'accès



### VERSIONING

- **URL versioning:** /v1/, /v2/
- **Header versioning:** X-API-Version
- **Semantic versioning:** 1.2.3
- **Backward compatibility**
- **Deprecation warning**

# CODES HTTP ET GESTION D'ERREURS

<b>Code</b>	<b>Signification</b>	<b>Exemple</b>
<b>200</b>	OK - Succès	Requête GET réussie
<b>201</b>	Created - Ressource créée	POST réussi
<b>400</b>	Bad Request - Erreur client	JSON invalide
<b>401</b>	Unauthorized - Auth requise	Token expiré
<b>403</b>	Forbidden - Pas d'accès	Permissions insuffisantes
<b>404</b>	Not Found - Ressource absente	Contrat inexistant
<b>500</b>	Server Error - Erreur serveur	Exception non gérée

# GRAPHQL: INTRODUCTION

GraphQL: Query language pour APIs

CONCEPT CLÉ: DEMANDER EXACTEMENT CE QU'ON VEUT



**REST (OVER-FETCHING)**

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

```
{  
  "id": "123",  
  "customer": { ... },  
  "premium": 1200,  
  "type": "AUTO",  
  "status": "ACTIVE"
```



**GRAPHQL  
(SEULEMENT CE QU'IL FAUT)**

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

# SCHÉMA GRAPHQL

Structure typée des données et opérations disponibles

## EXEMPLE DE SCHÉMA POUR ASSURANCE:

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

# GRAPHQL QUERIES (LECTURE)

## QUERY SIMPLE:

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

## QUERY AVEC FILTRAGE ET PAGINATION:

# GRAPHQL MUTATIONS (ÉCRITURE)

Opérations de création, mise à jour, suppression

## MUTATION: CRÉER UN CONTRAT

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

# GRAPHQL: AVANTAGES ET LIMITATIONS



## AVANTAGES

- Pas de over-fetching
- Pas de under-fetching
- Requête unique
- Pas de versioning
- Typage fort
- Documentation auto



## LIMITATIONS

- Caching difficile (POST)
- Courbe apprentissage
- Complexité du serveur
- N+1 queries problem
- File uploads complexe
- Real-time (WebSocket)

# IMPLÉMENTATION GRAPHQL: APOLLO SERVER

## INSTALLATION ET SETUP:

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

## CODE SERVEUR GRAPHQL (NODE.JS):

```
const { ApolloServer } = require('apollo-server-express');  
const typeDefs = require('./schemas/typeDefs');  
const resolvers = require('./schemas/resolvers');
```

# QUAND UTILISER REST VS GRAPHQL?

Scénario	REST	GraphQL	Recommandation
Ressources simples	Idéal	Overkill	<b>REST</b>
Relations complexes	Appels multiples	Requête unique	<b>GraphQL</b>
Clients variés	Over-fetching	Données précises	<b>GraphQL</b>
Mobile (bande passante)	Données superflues	Minimal	<b>GraphQL</b>

# SÉCURITÉ DANS LES APIs



## OAUTH2

Protocole  
d'authentification/autorisation

- **Authorization Code:** Apps web
- **Client Credentials:** Services
- **Implicit:** Apps single-page
- **Refresh Token:** Session longue



## JWT (JSON WEB TOKENS)

Token stateless, auto-contenu

- **Header:** Type et algorithme
- **Payload:** Données (user\_id)
- **Signature:** Vérification intégrité
- **Expiration:** Courte durée

# DOCUMENTATION API: SWAGGER/OPENAPI

Documenter et tester les APIs interactivement

## EXEMPLE DE SPECIFICATION OPENAPI (YAML):

```
openapi: 3.0.0
info:
  title: Insurance API
  version: 1.0.0
paths:
  /contracts:
    get:
      summary: List all contracts
```

# VERSIONING D'API

Maintenir la compatibilité avec les clients existants



## URL VERSIONING

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

Avantages:

- Clair et explicite
- Caching facile
- Fournisseurs multiples

- . . . .



## HEADER VERSIONING

```
GET /api/contracts  
X-API-Version: 2
```

Avantages:

- URL unique
- Moins de duplication

Inconvénients:

- . . . .

# RÉCAPITULATIF: API ET GRAPHQL



## REST API

- Standard HTTP (GET, POST, PUT, DELETE)
- Endpoints fixes par ressource
- Facile à cacher
- Versioning standard
- Idéal pour ressources simples



## GraphQL

- Query language typé
- Requêtes flexibles
- Pas over/under-fetching
- Pas de versioning
- Idéal pour relations complexes
- Courbe apprentissage

# MICROSERVICES: INTRODUCTION

Architectures distribuées basées sur des services indépendants.



Syntax error in text  
mermaid version 11.12.2

# 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

# 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



### ASYNCHRONE (EVENTS)

Service A

↓ (Publie event)

Kafka/RabbitMQ

↓ (Message broker)

Service B (reçoit)

Service C (reçoit)

# SAGA PATTERN: TRANSACTIONS DISTRIBUÉES

Maintenir la cohérence des données sur plusieurs services

## DEUX APPROCHES:

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

# SSR: SERVER-SIDE RENDERING

Le serveur génère le HTML complet avant envoi au navigateur



Syntax error in text  
mermaid version 11.12.2

## AVANTAGES:

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

# SSG: STATIC SITE GENERATION

Générer les pages HTML à la compilation (build time)



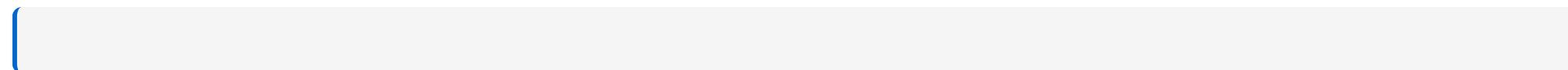
Syntax error in text  
mermaid version 11.12.2

## AVANTAGES:

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

# SPA: SINGLE PAGE APPLICATION

Application côté client, tout le rendu en JavaScript



## AVANTAGES VS INCONVÉNIENTS:



### AVANTAGES

- UX fluide (pas de reload)
- Interaction rapide



### INCONVÉNIENTS

- SEO difficile
- JS volumineux

# 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)
	Modérés (besoin d'un serveur)	Bas (CDN)	Bas (CDN)

# NEXT.JS: HYBRID RENDERING

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

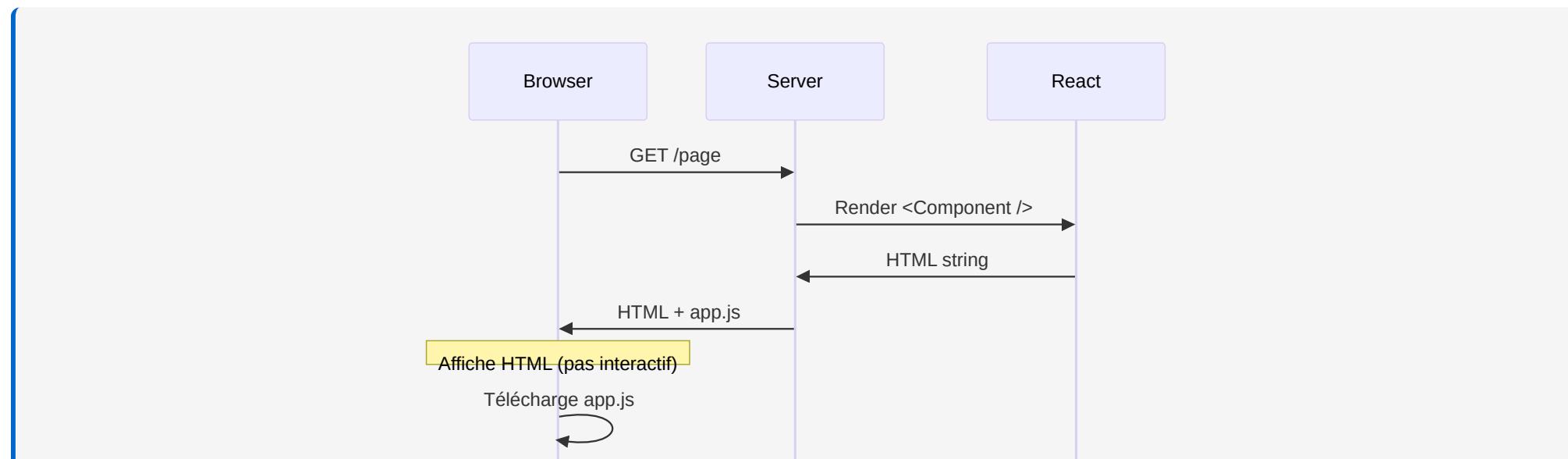
## STRUCTURE D'UN PROJET NEXT.JS:

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

# HYDRATION: HTML + INTERACTIVITÉ

Transformer du HTML statique en application interactive

## PROCESSUS D'HYDRATION:



# WEB VITALS: MÉTRIQUES DE PERFORMANCE

Mesurer l'expérience utilisateur réelle



**LCP**

**Largest Contentful Paint**

**Paint**

Temps d'affichage du contenu principal

**Idéal:** < 2.5 secondes



**FID**

**First Input Delay**

Délai avant réaction aux interactions

**Idéal:** < 100ms



**CLS**

**Cumulative Layout Shift**

Stabilité visuelle (pas de changement de layout)

# RÉCAPITULATIF: SSR VS SSG VS SPA

DÉCISION TREE (ARBRE DE DÉCISION):



Syntax error in text  
mermaid version 11.12.2

# SPRING BOOT: INTRODUCTION

Framework Java pour construire des microservices robustes

## CARACTÉRISTIQUES CLÉS:

- **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)

# ARCHITECTURE SPRING BOOT

## STRUCTURE STANDARD:

```
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/
```

## APPLICATION.PROPERTIES:

# 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) {
```

# 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
```

# SPRING DATA JPA: ACCÈS AUX DONNÉES

## REPOSITORY INTERFACE:

```
@Repository  
public interface ContractRepository  
    extends JpaRepository<Contract, String> {  
  
    // Méthodes générées automatiquement:  
    // save(T), delete(T), findById(ID), findAll(), etc.  
  
    // Requêtes personnalisées (query methods)  
    ...  
}
```

# SPRING CLOUD: MICROSERVICES DISTRIBUÉES

Framework pour construire des systèmes distribués

## COMPOSANTS CLÉS:



### SERVICE DISCOVERY

- **Eureka:** Service registry
- **Consul:** Service mesh
- Auto-registration et detection



### API GATEWAY

- **Spring Cloud Gateway**
- Routing intelligent
- Load balancing

# SPRING CLOUD CONFIG: CONFIGURATION CENTRALISÉE

Gérer la configuration des microservices depuis un endroit central

## FICHIERS DE CONFIG (APPLICATION.YML):

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

spring:
```

# TESTING SPRING BOOT APPLICATIONS

## TESTS UNITAIRES:

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

    @Mock
    private ContractRepository mockRepo;

    @InjectMocks
    private ContractService service;
```

# SPRING SECURITY: AUTHENTIFICATION & AUTORISATION

Framework pour sécuriser les applications Spring

## CONFIGURATION SPRING SECURITY AVEC JWT:

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
  
    @Bean
```

# MONITORING: SPRING BOOT ACTUATOR

Endpoints pour monitorer la santé et les performances

## APPLICATION.PROPERTIES:

*# 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
```

# DÉPLOIEMENT SPRING BOOT

## COMPILETION ET PACKAGING:

```
# 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 DOCKER:

# RÉCAPITULATIF: ÉCOSYSTÈME SPRING BOOT

## STACK COMPLET:

### CORE

- Spring Boot (app)
- Spring Data JPA (BD)
- Spring Security (auth)
- Spring Cloud (microservices)

### OPERATIONS

- Actuator (monitoring)
- Logging (SLF4J)
- Docker (containerization)
- Kubernetes (orchestration)

# NODE.JS: INTRODUCTION

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

## CARACTÉRISTIQUES CLÉS:

- **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

# NPM: GESTION DES DÉPENDANCES

Node Package Manager - Gérer les modules et dépendances

## COMMANDES ESSENTIELLES:

```
# Initialiser un projet  
npm init -y
```

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

# EXPRESS.JS: FRAMEWORK WEB MINIMALISTE

Framework léger pour construire des APIs et applications web

## APPLICATION EXPRESS BASIQUE:

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

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

// Routes
```

# NESTJS: FRAMEWORK POUR MICROSERVICES

Framework TypeScript avec architecture modulaire (inspiré par Angular)

## CARACTÉRISTIQUES:

-  **TypeScript natif** (type-safe)
-  **Architecture modulaire** (modules, controllers, services)
-  **Dependency Injection** (intégré)
-  **Décorateurs** (@Controller, @Get, @Post)

# ARCHITECTURE NESTJS

## STRUCTURE STANDARD:

```
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
```

## MODULE NESTJS COMPLET:

# NESTJS: MIDDLEWARE, GUARDS & INTERCEPTORS

Pipeline de traitement des requêtes

## GUARD: AUTHENTIFICATION AVEC JWT

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

  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
```

# TYPEORM: ORM POUR NODE.JS

Object-Relational Mapping pour TypeScript

## ENTITY TYPEORM:

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

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

    @Column()
```

# TESTING AVEC JEST

Framework de test pour Node.js et NestJS

## TEST UNITAIRE 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;
```

# DÉPLOIEMENT NODE.JS

## PM2 (PROCESS MANAGER):

```
# 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
```

## DOCKER:

# RÉCAPITULATIF: ÉCOSYSTÈME NODE.JS

## STACK NODE.JS COMPLET:

### FRAMEWORKS

- **Express**: Minimaliste, flexible
- **NestJS**: Modulaire, TypeScript
- **Fastify**: Haute performance
- **Koa**: Middleware elegance

### ÉCOSYSTÈME

- **npm**: Package manager
- **TypeORM**: ORM
- **Jest**: Testing
- **PM2**: Process management

# FRAMEWORKS FRONT-END: PANORAMA

Frameworks JavaScript pour construire des interfaces utilisateur modernes

## LES 3 GRANDS:

Framework	Philosophie	Courbe d'apprentissage	Popularité
<b>React</b>	Component-based, JSX	Modérée	 (Très populaire)
<b>Vue</b>	Progressive,	Facile	

# REACT: COMPOSANTS ET JSX

Library JavaScript pour construire des UIs avec des composants réutilisables

## COMPOSANT FONCTIONNEL REACT:

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

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

# REACT HOOKS: ÉTAT ET EFFETS

Réutiliser la logique avec des fonctions au lieu de classes

## HOOKS COURANTS:

```
import { useState, useEffect, useContext, useReducer } from

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

  return <>...</>;
}
```

# GESTION D'ÉTAT EN REACT

Gérer l'état global de l'application

## CONTEXT API (INTÉGRÉ)

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

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

  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}
```

## REDUX (LIBRARY)

```
// Store + Reducer
const store = createStore(reducer);
store.dispatch({ type: 'ADD_CONTRACT', payload: '...' });

switch (action.type) {
  case 'ADD_CONTRACT':
    return { ...state, contracts: [...state.contracts, action.payload] };
  default: return state;
}

// Reducer
const reducer = (state, action) => {
  switch (action.type) {
    case 'ADD_CONTRACT':
      return { ...state, contracts: [...state.contracts, action.payload] };
    default: return state;
  }
}
```

# VUE.JS: PROGRESSIVE FRAMEWORK

Framework approachable et performant

## COMPOSANT VUE:

```
<template>
  <div class="contract-form">
    <form @submit.prevent="submitForm">
      <input
        v-model="form.customerId"
        placeholder="Customer ID"
      />
      <select v-model="form.type">
```

# ANGULAR: FULL-FEATURED FRAMEWORK

Framework complet TypeScript pour applications enterprise

## CARACTÉRISTIQUES:

- **TypeScript natif** (type-safe)
- **Dependency Injection** (intégré)
- **RxJS & Observables** (programmation réactive)
- **Services et Components** (architecture claire)

# 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)

# BUILD TOOLS: WEBPACK, VITE, PARCEL

Outils pour bundler et optimiser le code front-end



## WEBPACK

- Standard (CRA, Vue CLI)
- Très configurable
- Lent à démarrer



## VITE

- Très rapide (ES modules)
- Dev server instant
- Configuration simple



## PARCEL

- Zero-config
- Facile pour petits projets
- Auto-build

# STYLING FRONT-END: OPTIONS MODERNES

## APPROCHES DE STYLING:



### CSS MODULES

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



### CSS-IN-JS (STYLED COMPONENTS)

```
import styled from 'styled-components'

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

# RÉCAPITULATIF: ÉCOSYSTÈME FRONT-END

STACK FRONT-END MODERNE (2026):

## BASE

- **Framework:** React (+ Vite)
- **Styling:** Tailwind ou Styled-Components
- **State:** Zustand ou Redux
- **Routing:** React Router v6

## TOOLING

- **Build:** Vite
- **Testing:** Vitest + React Testing Library
- **Linting:** ESLint
- **Formatting:** Prettier

# DÉVELOPPEMENT MOBILE: PANORAMA

Approches pour développer des applications mobiles

## 3 APPROCHES PRINCIPALES:

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)

# REACT NATIVE: JAVASCRIPT SUR IOS/ANDROID

Framework pour construire des apps mobiles avec React

## APPLICATION REACT NATIVE:

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

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

# ARCHITECTURE REACT NATIVE

Comment React Native communique avec iOS/Android



Syntax error in text  
mermaid version 11.12.2

## SETUP EXPO (RECOMMANDÉ POUR DÉBUTANTS):

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

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

# FLUTTER: FRAMEWORK CROSS-PLATFORM GOOGLE

Développer des apps iOS/Android/Web avec Dart

## AVANTAGES FLUTTER:

-  **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
-  **Portabilité:** facile à intégrer dans une application existante

# NATIVE DEVELOPMENT: SWIFT & KOTLIN

Développement natif pour apps haute performance

## SWIFT (IOS)

```
import Foundation

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

## KOTLIN (ANDROID)

```
import retrofit2.http.GET
import kotlinx.coroutines.flow.Flow

data class Contract(
    val id: String,
    val type: String,
    val premium: Double
)
```

# 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)
		1 (iOS + Android +	

# ARCHITECTURE MOBILE: OFFLINE-FIRST

Gérer la connectivité instable des appareils mobiles



Syntax error in text  
mermaid version 11.12.2

## IMPLÉMENTATION OFFLINE-FIRST:

```
// React Native avec AsyncStorage + WatermelonDB

import { Database } from '@nozbe/watermelondb';
import SQLiteAdapter from '@nozbe/watermelondb/adapters/sqlite';
```

# GESTION DES DONNÉES MOBILES

Stocker et synchroniser les données localement

## SQLITE (NATIVE)

- SQL queries
- Relations complexes
- Très stable
- Syntaxe verbose

## REALM (MOBILE)

- Synchronisation Cloud
- API simple (objets)
- Transactions ACID
- Licence propriétaire

# PERFORMANCE MOBILE: OPTIMISATIONS

## OPTIMISATIONS CRITIQUES:

### RENDU OPTIMISÉ

```
// React Native
const ContractItem = React.memo(({  
    contract, onPress  
}) => (  
    <TouchableOpacity onPress={onPress}>  
        <Text>{contract.type}</Text>  
    </TouchableOpacity>
```

### GESTION MÉMOIRE

```
// Éviter les fuites mémoire
useEffect(() => {  
    const subscription =  
        fetchContracts();  
  
    return () => {  
        // Cleanup
```

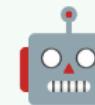
# DISTRIBUTION: APP STORES

Déployer l'application sur les stores



## APPLE APP STORE

- 1. Rejoindre Apple Developer Program (99\$/an)
- 2. Créer App ID dans Developer Portal
- 3. Générer certificats et provisioning profiles



## GOOGLE PLAY STORE

- 1. Rejoindre Google Play Developer Program (25\$ une fois)
- 2. Créer signingKey
- 3. Build APK/AAB avec Android Studio

# RÉCAPITULATIF: DÉVELOPPEMENT MOBILE

## STACK MOBILE RECOMMANDÉ (2026):

### CROSS- PLATFORM

- **Framework:** React Native ou Flutter
- **State:** Redux ou Provider

### NATIVE (SI NEEDED)

- **iOS:** Swift + SwiftUI
- **Android:** Kotlin + Jetpack

### TOOLING

- **Version Control:** Git
- **CI/CD:** GitHub Actions, EAS
- **Analytics:**

# MCP & INTÉGRATION IA: NOUVELLE ÈRE

Connecter les backends avec les modèles d'IA

## CAS D'USAGE:

-  **Assurance:** Analyse automatique des sinistres avec Claude
-  **Santé:** Diagnostic assistance basé sur données patients

# MCP: MODEL CONTEXT PROTOCOL

Standard ouvert pour connecter LLMs aux tools/APIs

ARCHITECTURE MCP:

## MCP SERVER (CÔTÉ BACKEND):

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

const server = new mcp.MCPServer({
```

# EXPOSER LES APIs POUR L'IA

Préparer votre backend pour l'intégration IA

## BEST PRACTICES:



### SCHÉMAS CLAIRS

```
{  
  "contract": {  
    "id": "string",  
    "customerId": "string",  
    "type": "enum(AUTO|HOME|HEALT  
  "premium": {  
    "type": "number",  
    "minimum": 0
```



### AUTHENTIFICATION

```
// Utiliser des tokens d'accès limités  
const aiToken = jwt.sign(  
  {  
    sub: 'ai-assistant',  
    scopes: ['read:contracts',  
             'create:claims']  
  },  
  SECRET
```

# USE CASES: IA DANS ASSURANCE/SANTÉ

CAS D'USAGE ASSURANCE:

1

## ANALYSE AUTOMATIQUE DE SINISTRES

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

```
// Prompt exemple  
const prompt =
```

# SÉCURITÉ & GOUVERNANCE: IA EN PRODUCTION

Protéger les données et respecter la réglementation

## POINTS CRITIQUES:



### SÉCURITÉ DONNÉES

- Chiffrer données avant LLM
- Pas d'infos sensibles en prompt



### CONFORMITÉ LÉGALE

- RGPD (droit à l'oubli)
- HIPAA (santé US)

# MONITORING: IA EN PRODUCTION

Surveiller la qualité et la performance des réponses IA

## MÉTRIQUES À TRACKER:

```
// 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
    errorRate: new Counter('ai_error_rate'),
    successRate: new Counter('ai_success_rate')
```

# FUTUR: AGENTS IA AUTONOMES

La prochaine génération: agents capables de décisions autonomes



Syntax error in text  
mermaid version 11.12.2

## EXEMPLE: TRAITEMENT SINISTRE AUTOMATIQUE

```
// Agent autonome
const claimAgent = new Agent({
  tools: [
    'get_contract'
```

# RÉCAPITULATIF: MCP & IA EN PRODUCTION

## ARCHITECTURE COMPLÈTE:

### BACKEND SIDE

- API: REST/GraphQL
- **MCP Server:** Expose ressources
- Auth: OAuth2/tokens

### LLM SIDE

- Model: Claude, GPT-4
- Tools: API calls structurées
- Agents: Loop autonome
- Monitoring: Métriques

# CONCLUSION: ARCHITECTURES MODERN (2026)

CE QUE VOUS AVEZ APPRIS:

## BACKEND

-  Spring Boot (enterprise)
-  Node.js/NestJS (moderne)
-  Clean Architecture
-  API design

## FRONTEND/MOBILE

-  React (standard)
-  State management
-  React Native/Flutter
-  Performance optimization

# ARCHITECTURE DECISION MATRIX

Choisir la bonne stack selon vos contraintes

Projet	Backend	Frontend	Mobile	Database
<b>Startup SaaS (Rapide)</b>	Node.js + NestJS	React + Vite	React Native (Expo)	PostgreSQL
<b>Assurance (Web) (Scalabilité)</b>	Spring Boot	React + Redux	Native Swift/Kotlin	Oracle/PostgreSQL

# CRITÈRES DE SÉLECTION TECHNOLOGIQUE

Évaluer les technologies avant de les choisir



## CRITÈRES POSITIFS

- **Maturité:** Utilisé en production
- **Communauté:** Actif, nombreux ressources
- **Documentation:** Complète et claire



## SIGNES D'ALERTE

- **Hype:** Nouvelle techno = risque
- **Dépendances:** Trop de libs externes
- **Breaking changes:** Mises à jour régulières et brusques

# DEVOPS: DÉPLOIEMENT EN PRODUCTION

Passer du développement à la production

**PIPELINE CI/CD STANDARD:**

**TERRAFORM (INFRASTRUCTURE AS CODE):**

```
# AWS ECS + RDS
terraform {
    required_providers {
```

# SÉCURITÉ: BONNES PRATIQUES

Protéger l'application et les données



## APPLICATION SECURITY

- HTTPS (TLS 1.3+)
- OWASP Top 10
- Input validation
- SQL Injection prevention
- XSS protection (CSP)



## DATA SECURITY

- Encryption at rest
- Encryption in transit
- Hashing passwords (bcrypt)
- JWT avec expiration
- Secrets management

# PERFORMANCE: MONITORING & OPTIMIZATION

Mesurer et optimiser les performances



## MÉTRIQUES CLÉS (APM)

- **Latency:**  
p50/p95/p99 (ms)
- **Throughput:**  
,



## OPTIMISATIONS BACKEND

```
// 1. Caching
app.use(cacheControl({
  'API-RESPONSE': '5 minutes',
  'STATIC': '1 year'
}));

// 2. Compression
```

# ORGANISATION: STRUCTURE D'ÉQUIPE

Organiser les équipes pour la scalabilité

## PETIT PROJET (3-5 PERSONNES):

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

Rôles croisés: tout le monde touche à tout

# CARRIÈRE: PROGRESSION DANS LE TECH

Évolution de carrière en 2026



## INDIVIDUAL CONTRIBUTOR

- Junior Dev (0-2 ans)
- Mid Dev (2-5 ans)



## LEADERSHIP

- Tech Lead (IC → Lead)
- Manager (People)



## SPÉCIALISATION

- Expert Domain (Ex: IA)
- Architect (Design)

# APPRENTISSAGE CONTINU: RESTER À JOUR

L'industrie change tous les 2-3 ans



## RESSOURCES GRATUITES

- **YouTube:** Tech talks, tutorials
- **Dev.to:** Articles techniques
- **GitHub:** Code open-source



## RESSOURCES PAYANTES

- **Udemy:** Cours structurés (\$)
- **Coursera:** Certifications (\$\$)
- **Frontend Masters:** Avancé

# RESSOURCES & RÉFÉRENCES

Liens et documentation pour approfondir



## DOCUMENTATION OFFICIELLE

- [Spring Boot](#)
- [NestJS](#)
- [React](#)
- [Vue.js](#)
- [Angular](#)



## LIVRES CLASSIQUES

- **Clean Code** - Robert Martin
- **Design Patterns** - Gang of Four
- **Microservices Patterns** - Chris Richardson
- **System Design Interview** -

# INSIGHTS CLÉS DE CETTE FORMATION

Les leçons importantes

1

## PAS DE TECHNOLOGIE UNIVERSELLE

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

2

## FONDAMENTAUX > FRAMEWORKS

Comprendre l'architecture, les patterns, les principes est plus

# PROCHAINES ÉTAPES: DE LA THÉORIE À LA PRATIQUE

Appliquer ce que vous avez appris



## SEMAINE 1

- Choisir un projet
- Évaluer technologies
- Setup dev env



## SEMAINES 2-4

- Implémenter API
- Build frontend
- Connecter



## SEMAINES 5-8

- Docker & CI/CD
- Monitoring
- Sécurité

# 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

# CONTACT & RESSOURCES FINALES

## RESTER EN CONTACT



[contact@example.com](mailto:contact@example.com)

Pour questions/retours



[linkedin.com/company/example](https://linkedin.com/company/example)

Suivre nos updates



[github.com/example](https://github.com/example)



CERTIFICATION

[Attestation formation](#)

MERCI!



**MERCI POUR CES 6 HEURES!**

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

2026+ ✨

# BONUS: GLOSSAIRE TECHNIQUE

## A-M

- **API:** Application Programming Interface
- **ACID:** Atomicity, Consistency, Isolation, Durability
- **CI/CD:** Continuous Integration/Deployment

## N-Z

- **ORM:** Object-Relational Mapping
- **OWASP:** Open Web App Security Project
- **PII:** Personally Identifiable Information