MODULE 5

Architectures Logicielles

Génie Logiciel et Qualité — M1 MIAGE | 3h CM | 2h TD | 4h TP

Monolithes | Domain-Centric | Distribué

Objectifs du module

1. **Identifier** les principaux patterns d'architecture et leurs caractéristiques
2. **Comparer** les architectures selon des critères objectifs
3. **Justifier** le choix d'une architecture en fonction du contexte
4. **Appliquer** l'architecture hexagonale et documenter avec des ADR

> *"Il n'existe pas d'architecture parfaite. Chaque choix est un compromis adapté à un contexte."*

1

Fondamentaux

Qu'est-ce que l'architecture logicielle ?

Qu'est-ce que l'architecture logicielle ?

"L'architecture logicielle d'un système est l'ensemble des structures nécessaires pour raisonner sur le système."

— Bass, Clements & Kazman

Ce que l'architecture définit

Structure — Organisation des composants

Communication — Interactions (API, events)

Déploiement — Distribution du système

Qualités — Attributs non-fonctionnels

ARCHITECTURE

Décisions structurelles majeures

Choix technologiques fondamentaux

Difficile à changer

DESIGN

Décisions niveau classes

Patterns de conception (GoF)

Plus facile à refactorer

Triangle des compromis : Rapidité ↔ Scalabilité ↔ Simplicité

Attributs de qualité

L'architecture est guidée par les exigences non-fonctionnelles

⚡ Performance

Latence et débit acceptables

📈 Scalabilité

Gestion de la croissance

🔒 Disponibilité

Uptime requis (99.9%)

🔧 Maintenabilité

Facilité de modification

🧪 Testabilité

Facilité de test

🔄 Évolutivité

Ajout de nouvelles fonctionnalités

Règle d'or : Chaque attribut impacte les autres → Prioriser selon le contexte

2

Architectures Classiques

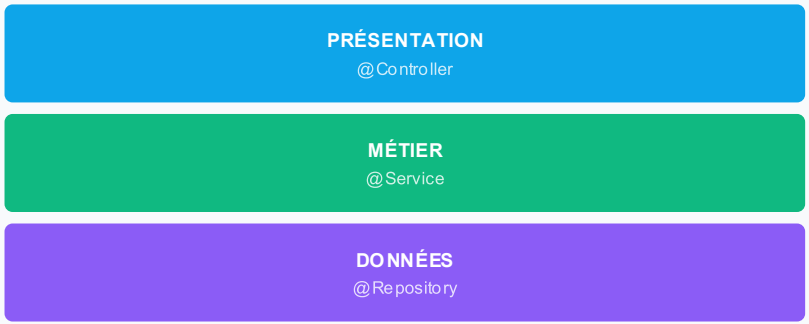
N-Tiers, Monolithe et MVC

Architecture N-Tiers

Layer (Couche) = Séparation logique

Tier = Séparation physique

Architecture 3-Tiers



✅ **Avantages**

Séparation des responsabilités

Équipes spécialisées

Scaling par tier

❌ **Inconvénients**

Couplage vertical fort

Modifications en cascade

Modèle anémique

```
// Règle de dépendance
Présentation → Métier → Données
⚠ Jamais dans l'autre sens !
```

Architecture Monolithique

> *"Presque toutes les success stories de microservices ont commencé par un monolithe devenu trop gros."*
>
> — Martin Fowler

Caractéristiques

Unité de déploiement unique
Une seule base de code
Une seule base de données
Même processus runtime

✅ **Avantages**

Simplicité, Performance (in-memory), Transactions ACID, Refactoring facile

❌ **Inconvénients**

Scalabilité limitée, Déploiement risqué, Stack unique, Build lent

Quand choisir le monolithe ?

✓ Équipe < 10 développeurs
✓ Domaine mal compris / exploratoire
✓ Time-to-market critique
✓ Cohérence forte requise (ACID)

```
// Exemples de succès
StackOverflow, Shopify (1.27M req/s)
```

Monolithe Modulaire (vs Big Ball of Mud)

⚠️ Big Ball of Mud (Anti-pattern)

Modifications → impacts inattendus

Personne ne comprend le système

"Peur de toucher au code"

✅ Solution : Monolithe Modulaire

Modules bien définis

Frontières explicites

Communication via API publiques

Référence par ID, pas par objet

```
// Structure modulaire BiblioTech

com.bibliotech/
    ├── catalogue/
    │   ├── api/ // Public
    │   └── internal/ // Privé
    ├── prets/
    │   ├── api/
    │   └── internal/
    ├── membres/
    └── shared/ // Minimal !
```

MVC et ses variantes

MVC (Model-View-Controller)

Trygve Reenskaug, Xerox PARC, 1979

View Controller Model

MVP
Presenter testable

MVVM
Data binding

MVI
État immutable

Pattern	Flux	Testabilité	Usage
MVC	△	Modérée	Spring MVC
MVP	↔	Élevée	Desktop
MVVM	⇄	Élevée	WPF, Jetpack
MVI	→	Très élevée	Apps réactives

3

Architectures Centrées Domaine

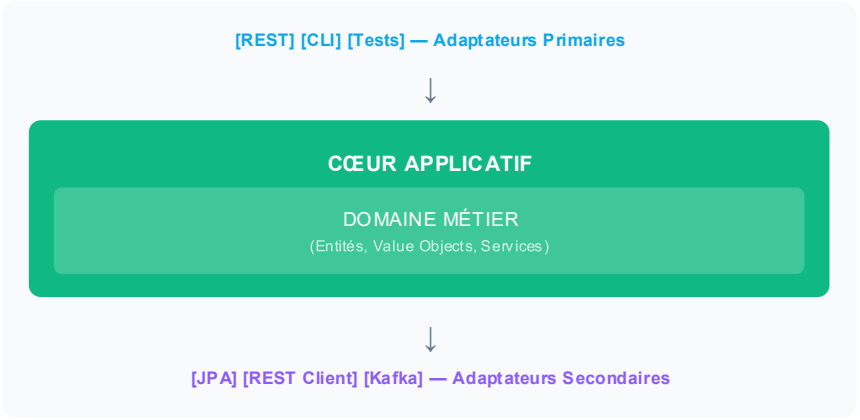
Hexagonale, Clean Architecture, DDD

Architecture Hexagonale (Ports & Adapters)

> *"Créez votre application pour qu'elle fonctionne sans UI ni base de données."*
> — Alistair Cockburn, 2005

Terminologie

- **Port Primaire** — Interface d'entrée (Use Case)
- **Port Secondaire** — Interface de sortie (SPI)
- **Adaptateur** — Implémentation concrète



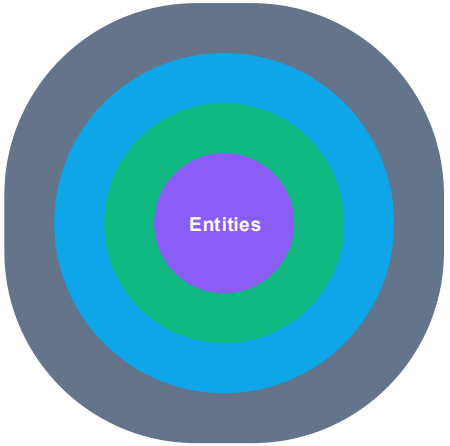
Clean Architecture (Uncle Bob, 2012)

RÈGLE FONDAMENTALE

"Les dépendances du code source doivent pointer uniquement vers l'intérieur."

Les 4 cercles

- **Entities** — Règles métier entreprise
- **Use Cases** — Règles métier application
- **Interface Adapters** — Controllers, Gateways
- **Frameworks** — Web, DB, UI



← Dépendances vers l'intérieur ←

Introduction au DDD (Domain-Driven Design)

Patterns tactiques

Entity (Entité)
Objet avec identité persistante

Value Object
Objet immutable (ISBN, Email)

Aggregate
Frontière de cohérence avec racine

```
// Value Object (immutable)
public record ISBN(String val) {
    public ISBN {
        if (!estValide(val))
            throw new ISBNInvalide();
    }
}
```

Règles de Vaughn Vernon

1. Modéliser les vrais invariants
2. Petits agrégats
3. Référencer par identité uniquement
4. Cohérence éventuelle hors frontières

Comparaison des architectures centrées domaine

Aspect	Hexagonale (2005)	Onion (2008)	Clean (2012)
Créateur	Alistair Cockburn	Jeffrey Palermo	Robert C. Martin
Focus	Isolation externe	Couches en oignon	Règle de dépendance
Terminologie	Ports, Adapters	Domain Model, Services	Entities, Use Cases

4

Architectures Distribuées

Microservices, Event-Driven, Serverless

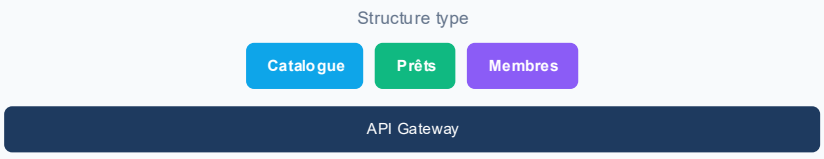
Architecture Microservices

"Un ensemble de petits services, chacun s'exécutant dans son propre processus."

— Martin Fowler & James Lewis

9 Caractéristiques clés

Services composants | Par capacité métier | Produits vs projets | Smart endpoints | Gouvernance décentralisée | Données décentralisées | Infra automatisée | Design for failure | Design évolutif



✅ Avantages

Scaling indépendant, Déploiement indépendant, Polyglottisme

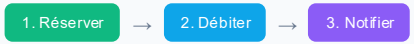
❌ Défis

Complexité distribuée, Transactions, Debugging

Patterns Microservices avec Spring Cloud

Pattern	Solution Spring
API Gateway	Spring Cloud Gateway
Service Discovery	Eureka, Consul
Circuit Breaker	Resilience4j
Config centralisée	Spring Cloud Config
Distributed Tracing	Micrometer + Zipkin

Pattern Saga (transactions distribuées)



En cas d'échec : Transactions compensatoires

Anti-patterns à éviter

Monolithe distribué | Base partagée | Appels sync en chaîne | Nano-services

Architecture Event-Driven

Types d'événements (Martin Fowler)

Event Notification
Signal minimal, le consommateur rappelle

Event-Carried State Transfer
Données complètes dans l'événement

Event Sourcing
État = séquence d'événements

CQRS
Séparation lecture / écriture

Communication Kafka


```
// Spring Kafka
@KafkaListener
(topics = "pret-events")
void handle(PretEvent e) {...}
```

Architecture Serverless

FaaS
AWS Lambda, Azure Functions, GCP Functions

BaaS
Firebase, Supabase, Auth0

Caractéristiques

Sans serveur à gérer | Scaling automatique | Paiement à l'usage | Event-driven | Stateless

```
// Spring Cloud Function
@Bean
Function<SearchReq, List<Livre>> search() {
    return req -> catalogueService.search(req);
}
```

⚠ **Cold Start en Java**

Configuration	Cold Start
Java 17 JVM standard	~10-12s
+ AWS SnapStart	~1s
GraalVM Native Image	~300ms

Monolithe vs Microservices : Comment choisir ?

Facteur	→ Monolithe	→ Microservices
Taille équipe	< 10 développeurs	Multiples équipes autonomes
Connaissance domaine	Frontières floues	Bounded contexts clairs
Maturité DevOps	Limitée	CI/CD mature, conteneurs
Besoins scaling	Uniformes	Différenciés par service
Cohérence données	ACID requis	Eventual consistency OK

5

Prise de Décision

ADR, Matrices et Évolution

Architecture Decision Records (ADR)

"Les décisions architecturales sont des choix de conception difficiles à faire et/ou coûteux à changer."

— Michael Nygard

```
# ADR [N]: [Titre]
## Statut
Proposé | Accepté | Déprécié
## Contexte
[Situation, problème, forces]
## Décision
[Choix et justification]
## Conséquences
[Impacts positifs/négatifs]
```

Quand écrire un ADR ?

Choix de technologie (BDD, framework) | Pattern architectural | Approche d'intégration | Compromis non-fonctionnels

✅ **Bonnes pratiques**

Écrire au moment de la décision | Inclure alternatives rejetées | Versionner avec le code (Git) | Ne jamais supprimer, déprécier

Outils : adr-tools, Log4brains, MADR

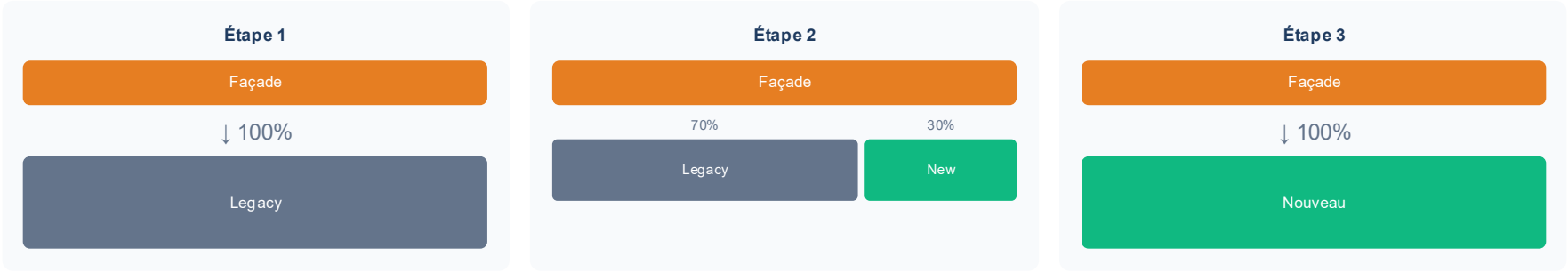
Matrice de décision architecturale

Exemple : Choix d'architecture pour BiblioTech

Critère	Poids	Monolithe	Modulaire	Microservices
Vitesse dev initiale	**4**	5 (20)	4 (16)	2 (8)
Scalabilité long terme	**3**	2 (6)	4 (12)	5 (15)
Simplicité opérationnelle	**5**	5 (25)	4 (20)	1 (5)
Cohérence données	**5**	5 (25)	5 (25)	2 (10)
Évolutivité architecture	**3**	2 (6)	4 (12)	5 (15)
TOTAL	**20**	**82**	**85**	**53**

Pattern Strangler Fig (Évolution progressive)

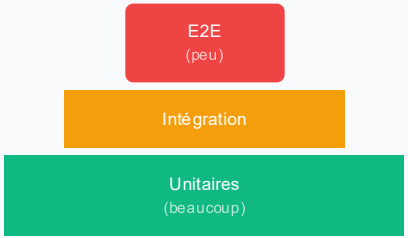
Remplacer un système legacy progressivement (Martin Fowler)



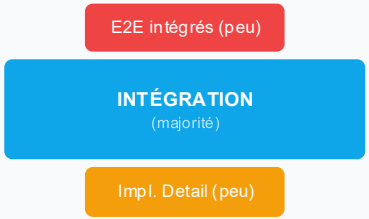
Autres stratégies : Branch by Abstraction | Parallel Run | Anti-Corruption Layer | Feature Toggles

Stratégies de test par architecture

Pyramide des tests — Monolithe



Nid d'abeille — Microservices



Contract Testing

Essentiel pour MS

Pact, Spring Cloud Contract

Clé : Tester les contrats entre services

6

Tendances Modernes

2024-2025

Tendances 2024-2025

Platform Engineering

Plateformes internes (IDP) avec "golden paths" standardisés

Spring Modulith

Framework officiel pour monolithes modulaires bien structurés

Virtual Threads (Java 21+)

Project Loom — millions de threads légers, concurrence simplifiée

IA et Architecture

RAG, MCP, Spring AI, Vector DBs, LLM integration patterns

```yaml
// Virtual Threads Spring Boot 3.2+
spring:
  threads:
    virtual:
      enabled: true
```

Gartner : 80% des orgs auront des platform teams d'ici 2026

Récapitulatif : 7 principes à retenir

1. Commencer par le monolithe sauf si frontières claires
2. Modularité > distribution — appliquer DDD partout
3. Hexagonale/Clean à toute échelle — isoler le domaine
4. Documenter les décisions avec des ADR
5. Évolution > révolution — Strangler Fig
6. Adapter la stratégie de test à l'architecture
7. La technologie sert l'architecture, pas l'inverse

Règle d'or

La meilleure architecture est celle que votre équipe peut **construire, opérer et faire évoluer** efficacement.

Ressources et prochaines étapes

📚 Livres recommandés

Clean Architecture — Robert C. Martin

Building Microservices — Sam Newman

Domain-Driven Design — Eric Evans

Software Architecture: The Hard Parts

🌐 Ressources en ligne

martinfowler.com/architecture | spring.io/projects/spring-modulith | adr.github.io

🎯 Pour BiblioTech (TP)

1. Analyser l'architecture actuelle
2. Refactorer en monolithe modulaire
3. Appliquer l'hexagonale au module Prêts
4. Documenter avec des ADR

Suite du parcours GLQ

Module 6
Tests avancés

Module 7
CI/CD DevOps