




Annexe : Comparaison des architectures

Pourquoi l'hexagonale pour ce module ?

 **Objectif** : Comprendre pourquoi l'architecture hexagonale a été choisie plutôt que d'autres approches courantes.

Slides :

1. Vue d'ensemble des alternatives
2. Tableau comparatif détaillé
3. Exemples concrets de chaque architecture
4. Justification pédagogique approfondie

Panorama des architectures

Question légitime : *Pourquoi pas une architecture qu'on connaît déjà ?*

Les alternatives courantes

| Architecture | Popularité | Complexité | Usage typique |
|--------------------|------------|----------------|---|
| Layered (3-tier) | ★★★★★ | Faible | Applications d'entreprise classiques |
| MVC | ★★★★★ | Faible | Applications web backend (Django, Rails) |
| MVVM | ★★★★ | Faible-Moyenne | Applications frontend (Angular, React, WPF) |
| Microservices | ★★★★ | Très élevée | Systèmes distribués, grande échelle |
| Hexagonale | ★★★ | Moyenne | Modernisation, DDD, qualité logicielle |
| Clean Architecture | ★★★ | Moyenne | Variante de l'hexagonale |
| Event-Driven | ★★★ | Élevée | Systèmes asynchrones, haute disponibilité |

Tableau comparatif (1/2)

| Critère | Layered | MVC | MVVM | Microservices | Hexagonale |
|-----------------------|----------------|-------------------|----------------------|----------------|------------|
| Complexité | Faible | Faible | Faible-Moyenne | Très élevée | Moyenne |
| Testabilité métier | ⚠ Permissive | ⚠ Model couplé DB | ⚠ Model couplé infra | ✅ Si bien fait | ✅ Forcée |
| Inversion dépendances | ❌ Optionnelle | ❌ Rare | ❌ UI/ViewModel | ✅ Nécessaire | ✅ Au cœur |
| Adapté 20h TD | ⚠ Trop simple | ⚠ Focus UI | ⚠ Frontend | ❌ Hors scope | ✅ Parfait |
| Enseigne SOLID | ⚠ Contournable | ❌ Non | ⚠ Partiel | ⚠ Acquis | ✅ Obligé |

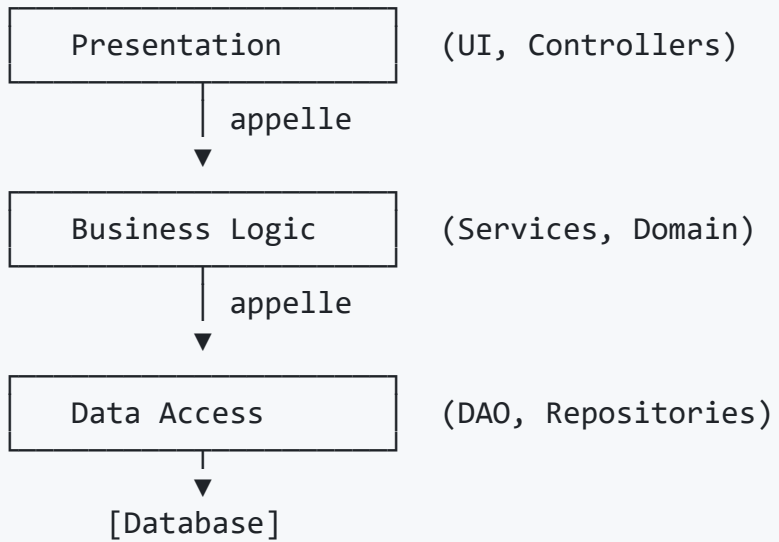
Tableau comparatif (2/2)

| Critère | Layered | MVC | MVVM | Microservices | Hexagonale |
|------------------|-----------------|----------------|--------------|---------------|-----------------|
| Changement infra | ✗ Impact métier | ✗ Réécriture | ⚠ Adapter VM | ✓ Par design | ✓ 1 adapter |
| Production | ✓ Legacy | ✓ Web standard | ✓ Frontend | ✓ Netflix | ✓ Modernisation |

Tableau comparatif (3/3)

| Critère | Layered | MVC | MVVM | Microservices | Hexagonale |
|-----------------|-----------|-----------|-----------|---------------|-------------|
| Autres patterns | ⚠ Limitée | ⚠ Limitée | ⚠ Limitée | ✅ Oui | ✅ DDD, CQRS |

🔍 Architecture Layered (3-tier)



Architecture Layered (suite)

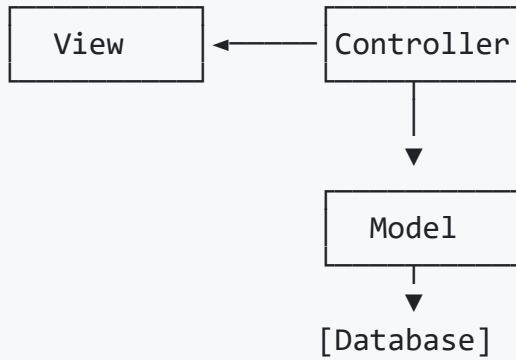
Avantages :

- Simple à comprendre
- Structure familière
- Bonne séparation visuelle

Problèmes :

- Dépendances top-down (métier → DB)
- Difficile à tester le métier seul
- Changement de DB = impact métier

🎨 Architecture MVC (Model-View-Controller)



✅ Avantages :

- Séparation UI/logique
- Pattern bien connu
- Frameworks matures (Django, Rails)

Architecture MVC (suite)

✗ Problèmes :

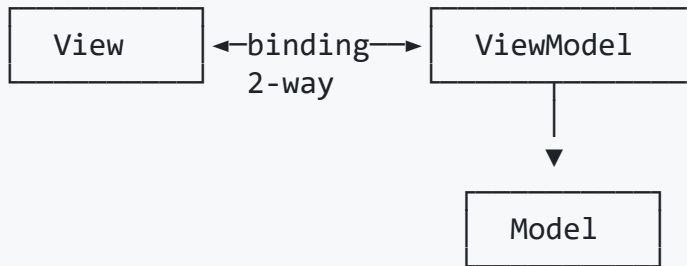
- Scope = organisation UI
- Model souvent couplé à l'ORM
- Pas d'inversion de dépendances
- Logique métier éparpillée

Exemple typique :

```
class TicketModel(db.Model): # ✗ Héritage de l'ORM
    title = db.Column(db.String)
    # Le métier dépend de la DB
```

Architecture MVVM (Model-View-ViewModel)

Évolution de MVC pour interfaces riches (WPF, Angular, React)



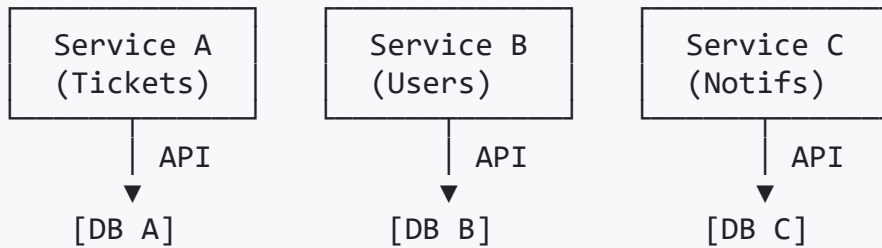
Différence clé avec MVC

| Aspect | MVC | MVVM |
|----------------|-----------------|--------------------------|
| Point d'entrée | Controller | View |
| Flux données | Unidirectionnel | Bidirectionnel (binding) |
| Tests UI | Difficile | ViewModel testable |

Architecture MVVM (suite)

- ✓ **Avantages** : Data binding auto, ViewModel testable
- ✗ **Limites** : Scope UI, Model couplé infra, pas d'inversion métier
- 🎯 **Usage** : Angular, React, Vue.js, WPF

Architecture Microservices




✓ Avantages :

- Scaling indépendant
- Technologies hétérogènes
- Équipes autonomes
- Déploiement indépendant

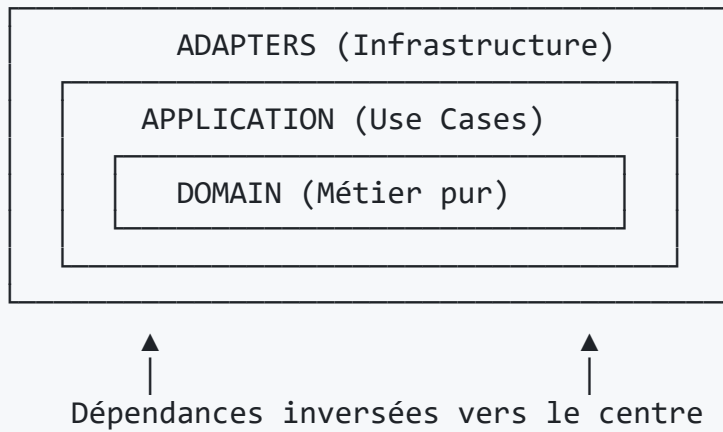
Architecture Microservices (suite)

Problèmes pour ce module :

- Complexité infrastructure (orchestration, découverte services)
- Gestion transactions distribuées
- Monitoring, logging distribué
- Hors scope d'un cours de 20h
- Suppose maîtrise du monolithe d'abord

 **Verdict** : Pertinent en M2, pas en BUT2

Architecture Hexagonale (rappel)



Architecture Hexagonale (suite)

Avantages :

- Métier totalement indépendant
- Testabilité maximale
- Évolutivité infrastructure
- Force SOLID

Trade-off :

- Verbosité (plus de fichiers)
- Courbe d'apprentissage
- Peut sembler over-engineering pour petits projets

🎓 Pourquoi l'hexagonale pour CE module ?

1. Objectif pédagogique : enseigner les fondamentaux

Ce qu'on veut que vous maîtrisiez :

- ☒ Inversion de dépendances (pas juste en théorie)
- ☒ Séparation domaine/infrastructure (visible structurellement)
- ☒ Testabilité (forcée par l'architecture)
- ☒ Évolutivité (changement d'adapter sans toucher au métier)

Pourquoi l'hexagonale ? (suite)

Avec Layered/MVC : On PEUT mal faire sans que ça bloque

Avec Hexagonale : Impossible de progresser sans respecter les principes

2. Format : 20h (ni trop, ni pas assez)

| Architecture | Complexité | Verdict pour 20h |
|---------------|---------------------|-------------------------------------|
| Layered | Trop simple | ✗ Vous connaissez déjà (S1-S2) |
| MVC | Trop simple | ✗ Focus UI, pas architecture métier |
| Microservices | Trop complexe | ✗ Infrastructure > architecture |
| Hexagonale | Juste ce qu'il faut | ✓ Challenge sans être inaccessible |

2. Format : 20h (suite)

Progression réaliste :

- TD1 : Domain (2h) → faisable
- TD2 : Use cases + Ports (4h) → on comprend l'inversion
- TD3 : Repository SQLite (4h) → on voit la puissance des adapters
- TD4 : API REST (4h) → on assemble tout

3. Testabilité : apprendre à BIEN tester

Mauvaise pratique (Layered/MVC) :


```
def test_create_ticket():  
    db.create_all() # Setup DB  
    client = TestClient(app) # Setup API  
    response = client.post("/tickets", json={"title": "Bug"})  
    assert response.status_code == 200
```

→ Toute l'infra pour tester 3 lignes de métier 🤖

Testabilité (suite)

Bonne pratique (Hexagonale) :

```
def test_create_ticket():  
    ticket = Ticket(title="Bug", status=Status.OPEN)  
    assert ticket.status == Status.OPEN
```

→ Test unitaire pur, rapide 

4. Transférabilité : base pour toutes les architectures modernes

Une fois l'hexagonale maîtrisée, vous comprenez :

| Architecture | Lien avec Hexagonale |
|-----------------------|--------------------------------------|
| Clean Architecture | Même principe (dépendances → centre) |
| Onion Architecture | Variante (couches concentriques) |
| DDD | Domaine au centre, bounded contexts |
| CQRS / Event Sourcing | Séparation via ports |
| Microservices | Chaque service = hexagone |

💡 Hexagonale = clé pour l'architecture moderne

Exemples concrets en production

Cas d'usage Layered (legacy courant)

- Applications d'entreprise années 2000-2010
- Maintenabilité difficile
- Dette technique élevée

Cas d'usage MVC


- Sites web traditionnels (Django, Rails)
- CMS (WordPress, Drupal)
- Admin panels

Exemples concrets (suite)

Cas d'usage Microservices

- Netflix (800+ services)
- Amazon, Uber
- Systèmes haute disponibilité

Cas d'usage Hexagonale

- **Modernisation de monolithes** (refactoring progressif)
- **Startups tech** (qualité dès le départ)
- **Projets DDD** (e-commerce, finance)
- **Votre projet BUT** 

Évolution : de Layered à Hexagonale

Scénario réaliste en entreprise :

```
Année 1 : Layered (quick & dirty)
  ↓ (dette technique s'accumule)
Année 3 : Tests impossible, bugs récurrents
  ↓ (décision de refactorer)
Année 4 : Migration vers Hexagonale
  ↓
Année 5 : Testable, maintenable, évolutif ✓
```

Votre avantage : Apprendre l'hexagonale dès le départ

- Vous évitez les erreurs courantes
- Vous êtes opérationnels pour la modernisation de legacy

Récapitulatif : Pourquoi Hexagonale ?

| Critère | Justification |
|---------------|--|
| Pédagogique | Force à appliquer SOLID (pas juste à les connaître) |
| Format 20h | Ni trop simple (acquis), ni trop complexe (hors scope) |
| Testabilité | Apprendre à BIEN tester (pas juste faire des tests) |
| Production | Cas d'usage réel (modernisation monolithes) |
| Transférable | Base pour Clean, DDD, microservices |
| Différenciant | Peu enseigné en formation, recherché en entreprise |

L'hexagonale n'est pas "meilleure" dans l'absolu.

Elle est la plus formatrice pour apprendre les fondamentaux d'architecture.

? Questions fréquentes

Q : Et si mon projet est petit, l'hexagonale n'est pas overkill ?

R : Oui, pour un script de 100 lignes. Non pour une application évolutive. Seuil \approx 500+ lignes.

Q : Microservices, c'est l'avenir, pourquoi pas les apprendre ?

R : Oui, APRÈS avoir maîtrisé le monolithe. Microservices = multiplier les problèmes par N.

? Questions fréquentes (suite)


Q : MVC suffit pour le web, non ?

R : MVC organise l'UI. Il faut une architecture métier EN PLUS (hexagonale marche bien avec MVC).

Q : C'est quoi la différence avec Clean Architecture ?

R : Quasi identique. Clean = généralisation de l'hexagonale par Uncle Bob.

 **Fin de l'annexe**

 [Retour au cours principal](#)