# 🧱 CM1 : Panorama des architectures logicielles modernes

🎓 BUT Informatique — Ressource R4.01 « Architecture logicielle »

👨‍🏫 Enseignant·e : *à compléter*

🛠️ Objectif du cours :

Comprendre **pourquoi** on parle d’architecture logicielle et découvrir les **principales architectures** rencontrées dans le monde professionnel.

# Plan du cours (1/2)

1. Pourquoi une architecture logicielle ?
2. Notions clés : cohésion, couplage, responsabilités
3. Panorama des architectures :
    - Monolithique, N-tiers, MVC/MVVM
    - SOA, Microservices, Event-driven
    - Architectures centrées domaine

# Plan du cours (2/2)

4. Patterns utiles encore aujourd'hui
5. Rôle des briques logicielles
6. Mini-exercice de synthèse

# 🚀 1. Pourquoi parler d’architecture ?

Sans vraie architecture, on obtient vite :

- Du **code spaghetti** 🍝
- Une application **difficile à comprendre**
- Des bugs qui reviennent en boucle
- Une application **impossible à tester**
- Une appli qui ne supporte pas bien les évolutions

👉 L’architecture sert à organiser le logiciel pour qu’il soit **vivable** sur le long terme.

# 🎯 Objectifs d’une bonne architecture

Une bonne architecture doit aider à :

- 🔧 **Maintenir** : corriger, faire évoluer
- 📦 **Modulariser** : pouvoir changer une partie sans tout casser
- 🧪 **Tester** : isoler le métier pour le tester sans tout l’environnement
- 🌱 **Faire évoluer** : ajouter des fonctionnalités sans tout réécrire
- 🙋‍♀️ **Comprendre** : nouveaux développeurs qui arrivent sur le projet

## 🤖 Et avec GitHub Copilot, ChatGPT & co ?

**« L'IA code à ma place, donc l'architecture, ouf, plus besoin… »**

❌ **FAUX.** C'est même l'inverse.

# 🤖 Pourquoi l'architecture devient PLUS importante (1/2)

1. **L'IA suit des instructions, pas de décisions stratégiques**
    - Elle peut respecter une architecture… *si vous lui expliquez laquelle*
    - Elle ne sait pas si votre contexte justifie une exception
2. **Plus on génère vite, plus on a besoin de vision**
    - Sans direction claire → accumulation rapide de dette technique
    - L'IA produit du code cohérent *localement*, mais pas toujours *globalement*

# 🤖 Pourquoi l'architecture devient PLUS importante (2/2)

3. **L'IA est un amplificateur**
    - Bonne architecture + IA → productivité décuplée ✅
    - Pas d'architecture + IA → chaos à grande vitesse ❌
4. **Votre valeur = les décisions que l'IA ne peut pas prendre**
    - Où placer la frontière entre domaine et infrastructure ?
    - Ce couplage est-il acceptable *dans ce contexte* ?
    - Faut-il sacrifier la pureté pour la simplicité ici ?

# 🎯 A retenir !

> **L'IA est semblable à un développeur expérimenté et ultra-rapide…**
> **…qui débarque sur votre projet sans en connaître l'histoire ni la vision.**

Elle code (en général) très bien. Mais elle a besoin que **vous** lui donniez :

- 🧭 La direction (quelle architecture ?)
- 🚧 Les contraintes (quelles règles respecter ?)
- ⚖️ Les arbitrages (quand faire une exception ?)

# 💡 Codeur vs Ingénieur

**L'IA est une bonne codeuse, pas (encore) une ingénieure logicielle.**

Un **codeur** maîtrise un langage et produit du code qui fonctionne.
Un **ingénieur logiciel** conçoit des systèmes cohérents, maintenables, évolutifs, fait preuve de jugement sans appliquer aveuglément des règles.

> *Ce cours vise à faire de vous des ingénieurs,*
> *pas juste des codeurs assistés par IA.*

# 🧠 2. Trois notions clés

| Notion | Idée intuitive | Exemple simple |
|---|---|---|
| **Cohésion** | Ce qui va ensemble reste ensemble | Une classe « Panier » gère le panier, pas l'envoi d'e-mails |
| **Couplage faible** | Peu de dépendances directes | Un service utilise une interface, pas une implémentation concrète |
| **Responsabilités** | Une chose = un rôle | Un contrôleur web ne contient pas les règles métier |

👉 Ces notions se retrouvent dans **toutes** les architectures.

## 🔗 2.1. Les dépendances

Une dépendance = *quelque chose dont mon code a besoin pour fonctionner* :

- Base de données 🗄️ (MySQL, PostgreSQL…)
- Framework web 🌐 (Symfony, Spring, Django…)
- API externe 🔌 (service de paiement, météo…)
- Bibliothèque 📚 (PDF, Excel, logging…)

Plus le code **dépend directement** de ces éléments, plus il devient **fragile**.

# 🏛️ 3. Les grandes familles d’architectures

On va survoler :

1. Architecture **monolithique**
2. Architecture **en couches (N-tiers)**
3. **MVC / MVVM**
4. **SOA** (Service-Oriented Architecture)
5. **Microservices**
6. **Event-Driven** (pilotée par les événements)
7. Architectures **centrées domaine** (Clean, Hexagonale, Onion…)

🎯 Objectif : que vous sachiez les **reconnaître** et comprendre **leurs enjeux**.

# 3.1 Architecture monolithique


Tout est rassemblé dans **un seul bloc déployable**.

✅ Avantages :

- Simple à mettre en place
- Déploiement facile (un artefact à livrer)
- Très bien pour de **petites applications**

❌ Inconvénients :

- Devient vite **grosse et complexe**
- Difficile à faire évoluer par morceaux

## 3.2 Architecture en couches (Layered / N-tiers)

Idée : séparer par grandes responsabilités.


✅ Avantages :

- Modèle très répandu, compris par tous
- Sépare (un peu) l’interface, le métier et les données

❌ Inconvénients :

## 3.3 MVC (Model – View – Controller)

Spécialement pour les **interfaces utilisateur**.

```
View  <-->  Controller  <-->  Model
```

- **Model** : données et logique métier locale
- **View** : ce que voit l'utilisateur
- **Controller** : reçoit les actions de l'utilisateur, appelle le modèle, choisit la vue

Exemples : frameworks web (Laravel, Symfony), front web (certains patterns React), mobile (Android MVVM).

# MVC : intérêts et limites

✅ Intérêts :

- Sépare l’affichage de la logique
- Facilite la réutilisation du Model avec plusieurs Views (web, mobile…)
- Structure bien la partie **interface**

❌ Limites :

- Ne suffit pas à **structurer tout le back-end**
- Le Model finit parfois par contenir **trop de responsabilités**

👉 On utilise souvent MVC **à l’intérieur** d’une architecture plus globale (en couches, hexagonale, etc.).

# 3.4 SOA — Service-Oriented Architecture

Chaque « gros » besoin métier = un **service**.


✅ Avantages :

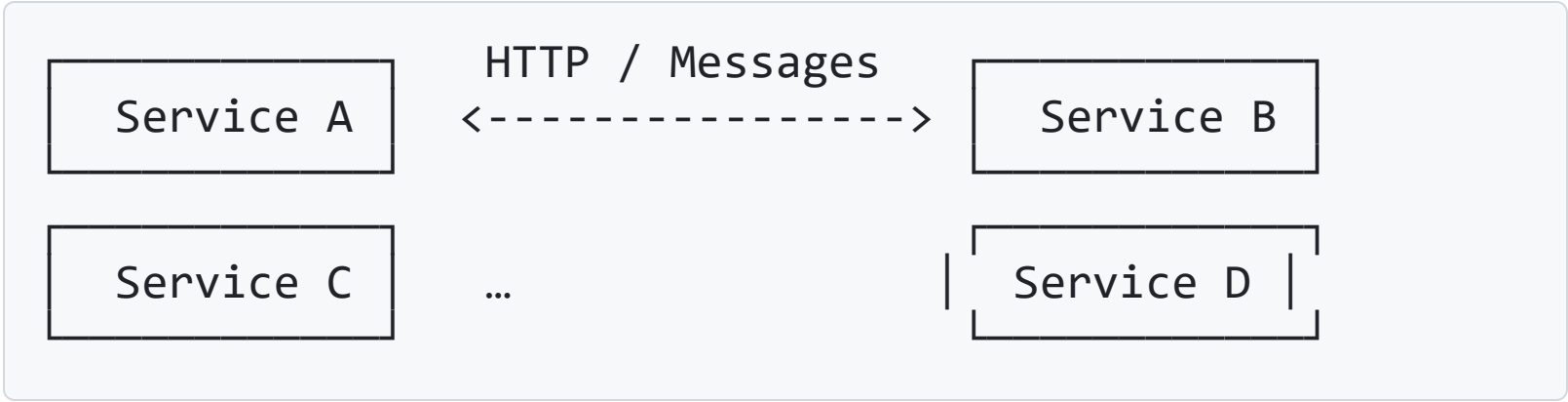
- Mutualisation de services dans une grande entreprise
- Standardisation via contrats (WSDL, SOAP, etc.)

❌ Inconvénients :

- Lourd à mettre en place
- Souvent associé à des outils complexes (ESB)
- Très présent dans les **SI historiques** (grands comptes)

# 3.5 Microservices 🧩

Chaque **microservice** est une petite application autonome, souvent centrée sur un sous-domaine.




✅ Avantages :

- Déploiement indépendant de chaque service
- Scalabilité fine (on peut scaler seulement le service critique)
- Permet des choix techniques différents par service

# 3.6 Quand (ne pas) utiliser les microservices ?

✅ Adapté quand :

- L’application est très grande
- Les équipes sont nombreuses (plusieurs dizaines de devs)
- Les besoins de scalabilité sont forts

❌ Surdimensionné quand :

- Projet de petite équipe (projet étudiant, PME…)
- Domaine métier encore flou
- Pas de moyens pour gérer la complexité infra



💡 Pour beaucoup de cas, un **monolithe bien structuré** est

# 3.7 Event-Driven Architecture (pilotée par les événements)

L’application réagit à des **événements**.

```
[Service A] -- produit un événement -->  [Bus de messages]  --> [Service B, C...]
```

Exemples d’événements :

- « Commande payée »
- « Utilisateur inscrit »
- « Stock mis à jour »

✅ Avantages :

- Asynchrone, scalable
- Permet de découpler les services

# 3.8 Architectures centrées domaine (Clean, Onion, Hexagonale…)

Idée clé :

> **Le code métier (domaine) ne doit pas dépendre de la technique.**
> La technique (framework, DB, IHM) dépend du domaine.

On parle parfois de :

- **Domain-Centric**
- **Ports & Adapters (architecture hexagonale)**
- **Clean Architecture**

👉 On y reviendra **en détail dans le CM2**.

## 🧱 4. Patterns utiles (1/2)

Quelques « briques » que l'on retrouve souvent :

- **Repository** : encapsule l'accès aux données (ex : `UserRepository`)
- **Service** métier : porte les règles métier (ex : `OrderService`)
- **Factory / Builder** : crée des objets complexes
- **DTO (Data Transfer Object)** : transporte les données entre couches

## 🧱 4. Patterns utiles (2/2)

- **Mapper** : convertit entités ↔️ DTO
- **Observer / Pub-Sub** : réagir à des événements
- **Dependency Injection (DI)** : délègue la création des dépendances

📌 Objectif : les connaître de nom, savoir **reconnaître** quand on les voit.

# 🔧 5. Briques logicielles & architecture

Les *savoirs de référence* de la ressource R4.01 incluent :

- Patrons d’architecture (ex : MVC, MVVM, etc.)
- Utilisation de **briques logicielles**, d’APIs, de **bibliothèques tierces**
- Développement de **services web**
- Organisation de **l’accès aux données** (BD, annuaires, services web…)

💡 L’architecture donne le **cadre** pour organiser ces briques intelligemment.

## Exemples de briques dans un projet web

- Framework : Symfony / Spring / Django
- ORM : Doctrine / Hibernate / Entity Framework
- Base de données : PostgreSQL / MySQL
- API externes : service de paiement, envoi de mails, SMS
- Bibliothèques : génération PDF, logs, sécurité…

👉 L’architecture choisie indique **où** et **comment** les utiliser.

## 📝 6. Mini-exercice (discussion)

Contexte : application de **gestion d’emprunts de livres** pour une médiathèque.
Fonctionnalités principales :

- Gérer les utilisateurs
- Gérer les livres
- Gérer les emprunts & retours
- Générer quelques statistiques simples

**Question :**
Parmi ces architectures, laquelle vous semble adaptée, et pourquoi ?

- Monolithe simple

# 🎯 Récapitulatif du CM1

Vous devez maintenant :

- Connaître les **motivations** d’une architecture logicielle
- Avoir une idée des principales **grandes familles d’architectures**
- Savoir que les choix d’architecture ont un impact sur :
  - La maintenabilité
  - La testabilité
  - L’évolutivité
  - L’impact environnemental

➡️ Prochain cours (CM2) : focus sur **l’architecture hexagonale (Ports & Adapters)**.

# 🏁 Fin du CM1

📂 Les slides seront disponibles sur le dépôt GitHub du cours.

❓ Questions ?