

Architectures des Applications

Objectifs du cours

Ce cours a pour objectif de fournir une **compréhension approfondie des architectures applicatives**, en équilibrant **théorie et pratique** pour les profils **développement (DLW)** et **administration système (IRC)**.

À la fin du cours, les étudiants seront capables de :

- Identifier et comparer les **architectures monolithique, n-tiers et microservices**.
 - Comprendre les principes des **architectures avancées** (DDD, Clean, Hexagonale).
 - Expliquer les **implications en hébergement et déploiement** de chaque modèle.
 - Utiliser **UML pour représenter et justifier des choix d'architecture**.
 - Appliquer ces concepts dans un **cas pratique** avec une **évaluation finale**.
-

Organisation du cours

Le cours est divisé en **trois jours** :

Jour 1 : Introduction & architectures classiques

- Pourquoi l'architecture est essentielle ?
- Modèle monolithique : principes, avantages, limites.
- Architecture n-tiers : découpage en couches et hébergement.
- Microservices : modularité et complexité.
- Démonstrations et exemples en PHP et JavaScript.
- Diagrammes UML : cas d'utilisation, composants, déploiement.

Jour 2 : Architectures avancées & déploiement

- Domain-Driven Design (DDD) : structurer le domaine métier.
- Clean Architecture : séparation des responsabilités.
- Hexagonale : découplage et ports/adapters.
- Hébergement et déploiement : serveurs, conteneurs, cloud.
- Démonstrations et exemples en PHP et JavaScript.
- Diagrammes UML : cas d'utilisation, composants, déploiement.

Jour 3 : Études de cas & évaluation

- Études de cas et choix d'architectures.
- Préparation à l'évaluation.
- Examen final : analyse et justification d'une architecture.

Introduction aux architectures applicatives

Objectifs de cette section

Dans cette introduction, nous allons poser les bases de l'architecture applicative et comprendre son importance.
À la fin de cette section, vous serez capable de :

- Définir ce qu'est une **architecture applicative** et pourquoi elle est essentielle.
- Identifier les critères qui influencent **le choix d'une architecture**.
- Comprendre comment **UML** va nous aider à représenter et analyser les architectures.
- Mettre en place un **projet backend simple (gestion de tâches)** qui servira d'exemple tout au long du cours.

1. Qu'est-ce qu'une architecture applicative ?

L'architecture d'une application définit **comment ses composants sont organisés et interagissent entre eux**. Elle structure le code et impacte directement :

- **La maintenabilité** : facilité à faire évoluer l'application.
- **La scalabilité** : capacité à supporter une charge croissante.
- **La performance** : rapidité et efficacité du traitement des données.
- **La sécurité** : protection des données et des interactions entre les services.

Exemple concret :

Une simple **application de gestion de tâches** peut être architecturée de plusieurs façons :

- **En monolithique**, tout le code est regroupé dans une seule application.
- **En n-tiers**, le frontend, le backend et la base de données sont séparés.
- **En microservices**, chaque fonctionnalité (gestion des tâches, des utilisateurs, etc.) est un service indépendant.

2. Comment choisir une architecture ?

Il n'existe pas d'**architecture parfaite**, mais un ensemble de **compromis** adaptés aux besoins.

Les critères de choix :

Critère	Impact sur l'architecture
Complexité du projet	Un petit projet fonctionne bien en monolithique , un projet plus vaste peut nécessiter dun-tiers ou du microservices .
Fréquence des mises à jour	Une mise à jour fréquente de certaines fonctionnalités favorise une approche modulaire (microservices, hexagonale).
Performance et scalabilité	Une application avec beaucoup d'utilisateurs simultanés doit être scalable (microservices, API séparées).
Coût et temps de développement	Un monolithe est rapide à développer , alors qu'une architecture microservices est plus longue et coûteuse .

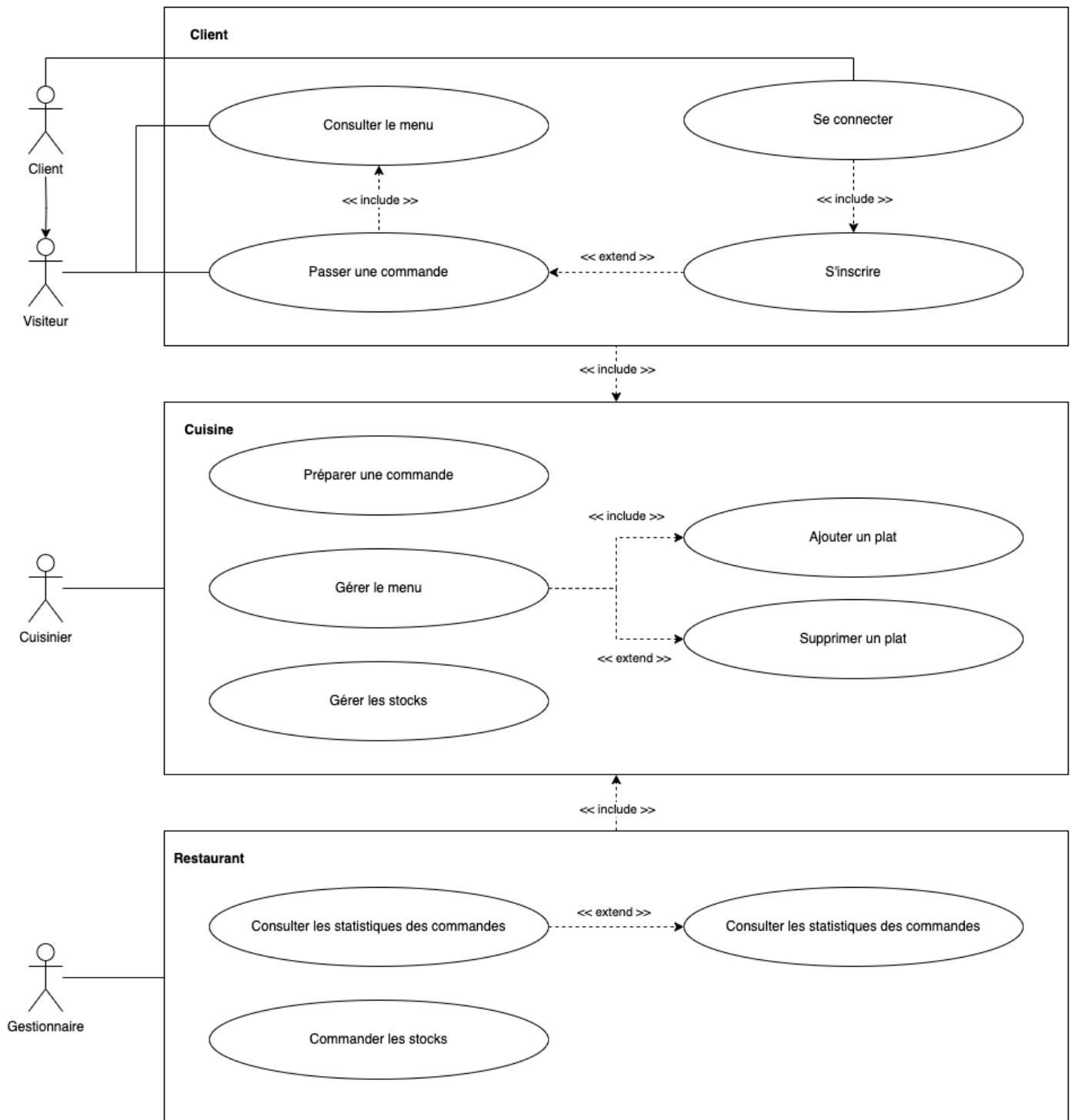
3. UML : un outil essentiel pour comprendre les architectures

Nous utiliserons **UML (Unified Modeling Language)** pour représenter **les architectures et leurs composants**.
Pas besoin d'être expert, l'idée est d'utiliser **quelques diagrammes simples** :

Diagramme de cas d'utilisation (exemple)

Ce diagramme présente simplement les interactions entre l'utilisateur et le système.

Cas d'Utilisation : Système de commande de repas en ligne

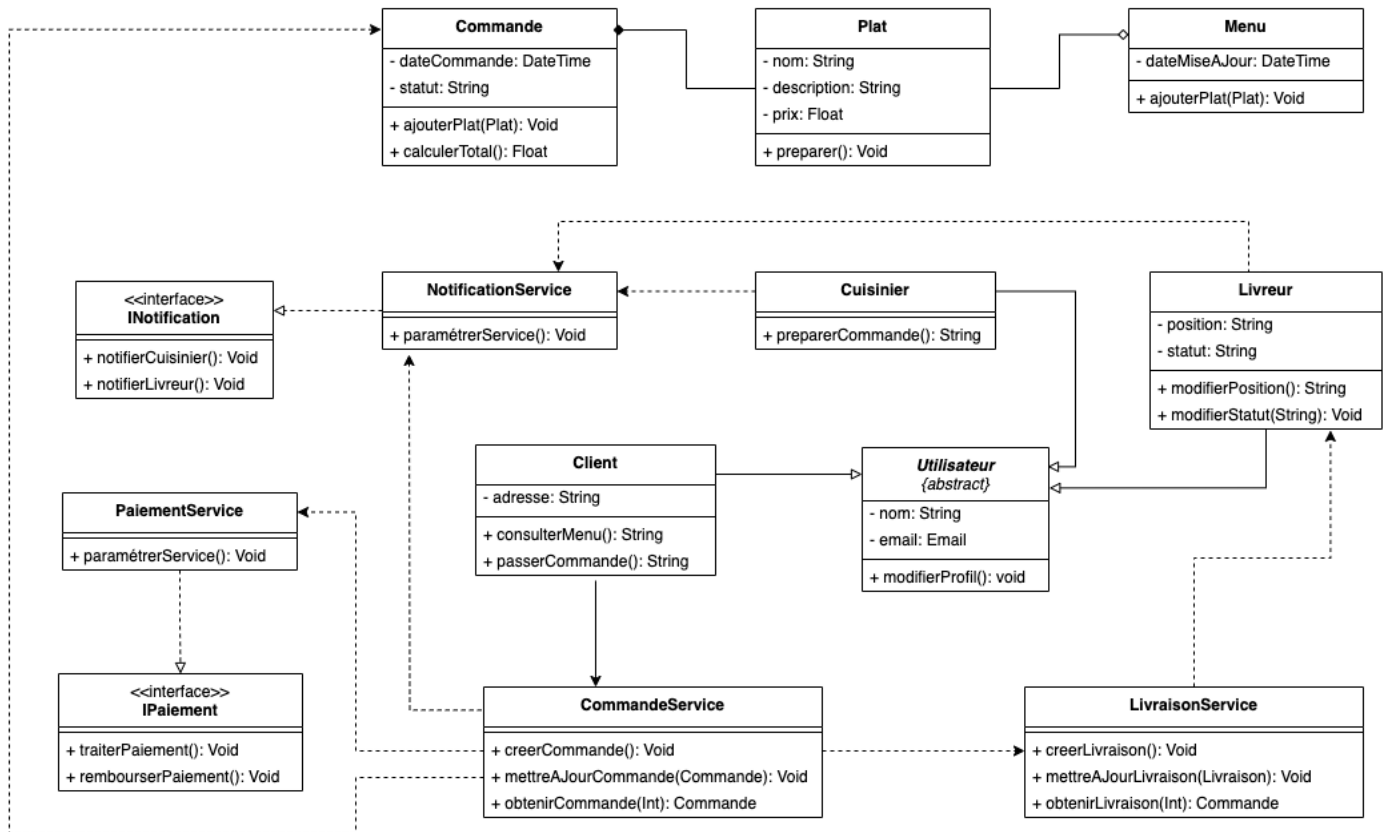


[Support de cours UML - Diagramme de Cas d'Utilisation](#)

Diagramme de classes (exemple)

C'est la modélisation des entités principales.

Classes : Gestion des commandes de repas en ligne



Support de cours UML - Diagramme de Classes

Diagramme de composants (exemple)

Ce diagramme montre l'organisation des blocs applicatifs.

Composants : Gestion des commandes de repas en ligne

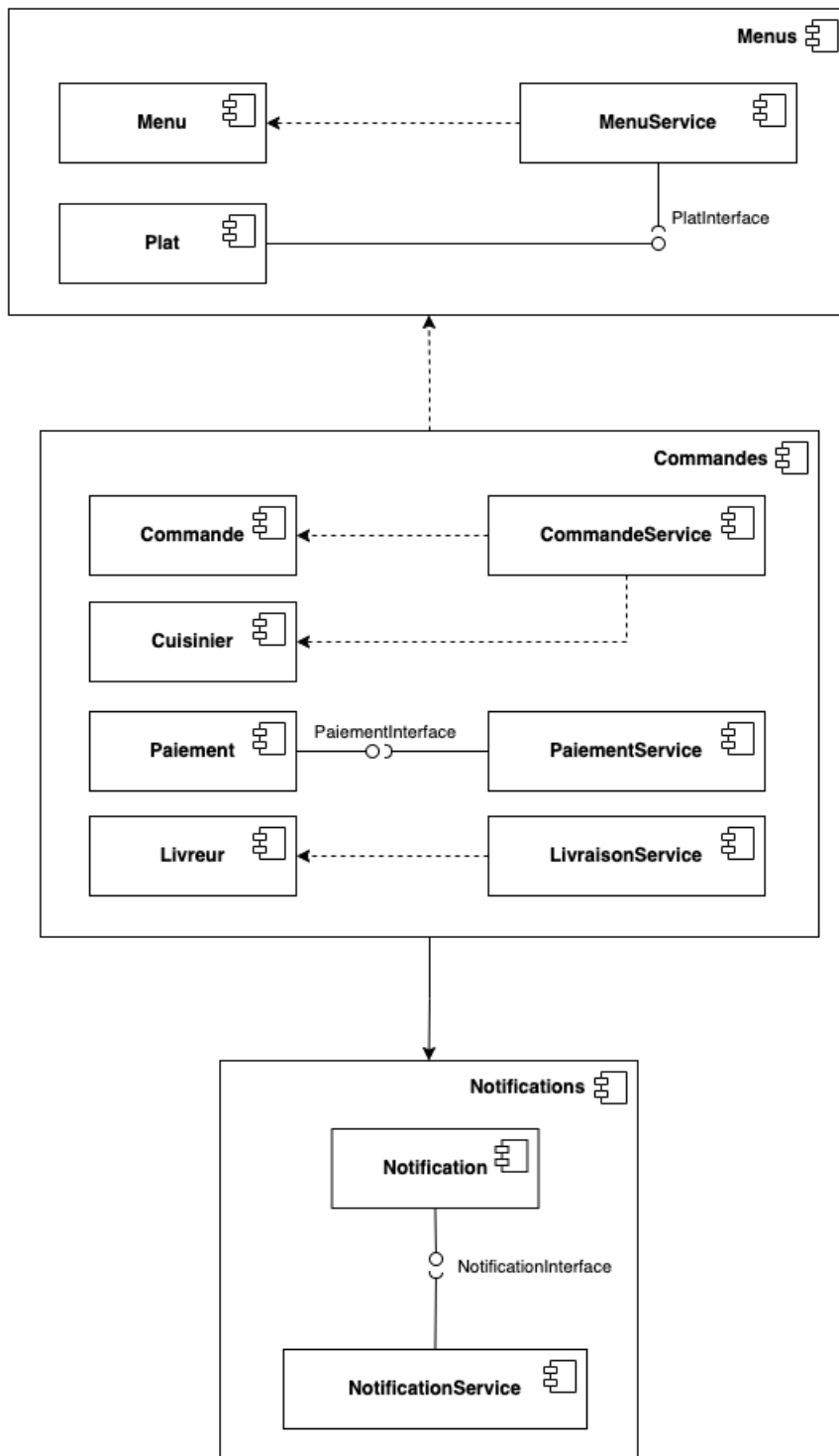
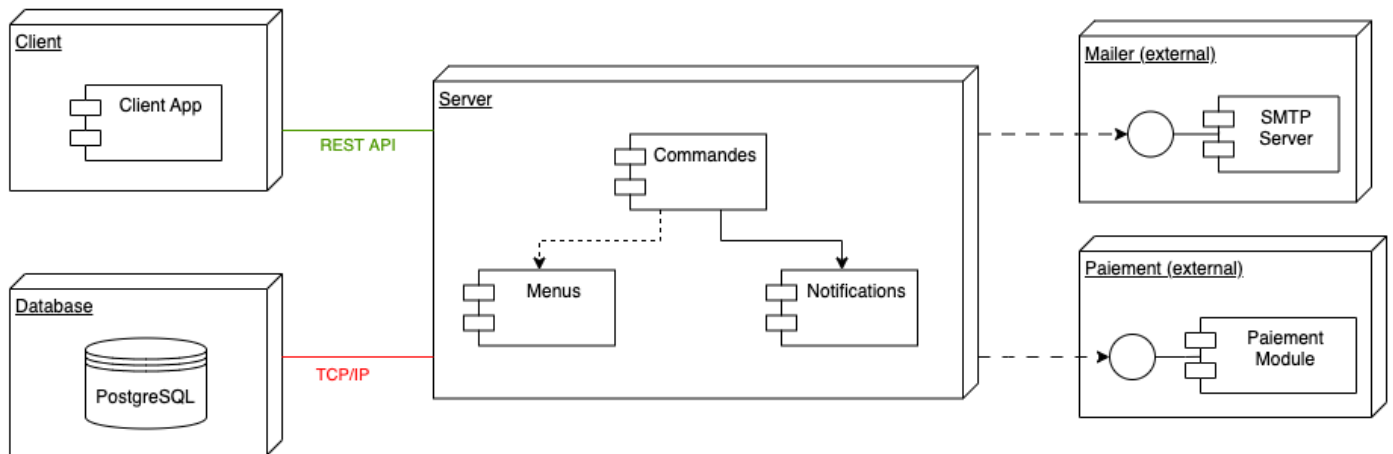


Diagramme de déploiement (exemple)

Ce diagramme montre où et comment sont hébergés les composants.



4. Mise en place du projet backend : les développeurs à l'oeuvre

Nous allons développer **une application de gestion de tâches** qui nous servira d'exemple tout au long du cours. Elle sera proposée **en PHP et en JavaScript (Node.js)** pour que chacun puisse travailler avec son langage préféré.

Arborescence du projet

```

/gestion-taches
├── /php                # Version en PHP
│   ├── index.php      # Point d'entrée principal
│   ├── db.php         # Connexion à la base de données
│   ├── tasks.php      # Gestion des tâches
│   ├── .env           # Configuration (base de données, etc.)
│   └── composer.json  # Dépendances PHP
├── /nodejs            # Version en Node.js
│   ├── index.js       # Point d'entrée principal
│   ├── db.js          # Connexion à la base de données
│   ├── tasks.js       # Gestion des tâches
│   ├── .env           # Configuration (base de données, etc.)
│   └── package.json   # Dépendances Node.js
└── README.md          # Documentation du projet
    
```

5. Mise en pratique pour les administrateurs infrastructure

L'objectif est de **comprendre les bases de l'hébergement d'une application backend** et d'**anticiper son déploiement**.

Cette mise en pratique vous permettra d'identifier **les éléments techniques**, de **préparer un environnement simple**, et d'**établir les premiers réflexes en matière de sécurité et de maintenance**.

Découverte des composants techniques

Objectif :

Se familiariser avec les **différents éléments techniques** qui composent l'application.

- Quelle est la **différence** entre un serveur web, une base de données et une API ?
- L'application est prévue pour fonctionner avec **deux technologies backend** (PHP et Node.js).
 - Que faut-il installer sur un serveur pour exécuter **une application PHP** ?
 - Que faut-il installer sur un serveur pour exécuter **une application Node.js** ?
- À quoi sert une **base de données** et comment l'application s'y connecte-t-elle ?
- Où sont stockés les **fichiers de configuration** et pourquoi sont-ils importants ?

Livrable attendu : Une brève **explication écrite** des composants et de leur rôle dans l'application.

Préparation d'un environnement de test

Objectif :

Installer un environnement dédié aux tests **end-to-end** permettant de valider le bon fonctionnement de l'application avant son déploiement en production.

- Quelle est la différence entre un **serveur de production** et un **serveur d'intégration** ?
- Pourquoi un **environnement de test** doit-il être le plus proche possible de l'environnement de production ?
- Quelle différence y a-t-il entre un **serveur local** et une **machine virtuelle ou un conteneur** pour exécuter des tests ?
- Comment peut-on exécuter un **serveur PHP** ou **Node.js** en local sans serveur distant ?
- À quoi sert **SQLite**, et pourquoi est-ce un bon choix pour tester une base de données en local ?
- Quels outils peuvent être utilisés pour **simuler des requêtes et tester les réponses de l'API** ?
- Comment s'assurer que les modifications effectuées **n'impactent pas négativement** l'application existante ?

Astuce :

- Des outils comme **Postman** ou **cURL** permettent d'envoyer des requêtes HTTP et de tester le bon fonctionnement de l'API.
- L'utilisation de **Docker** permet de créer un environnement isolé et reproductible pour les tests.

Livable attendu :

Une **courte procédure** décrivant comment tester l'application en local et vérifier le bon fonctionnement des requêtes API.

Organisation des fichiers et sauvegardes

Objectif :

Apprendre à identifier **les fichiers importants** et à organiser les données pour **éviter les pertes**.

- Quels fichiers sont **essentiels** au bon fonctionnement de l'application ?
- Pourquoi est-il important de **sauvegarder régulièrement la base de données** ?
- Quelle différence entre une **sauvegarde complète** et une **sauvegarde incrémentale** ?
- Où et comment stocker les fichiers de sauvegarde (sur son PC, sur un serveur distant, dans le cloud...) ?

Astuce : Dans un projet réel, il existe des outils pour automatiser les sauvegardes, comme `cron` sous Linux ou des services comme **Amazon S3**.

Livable attendu : Une **liste des fichiers essentiels** à sauvegarder et une **proposition de stratégie simple** de sauvegarde.

Premiers pas en sécurité

Objectif :

Comprendre les **premières étapes** pour sécuriser un serveur qui hébergerait une application backend.

- Pourquoi **ne jamais stocker les mots de passe** directement dans le code source ?
- Qu'est-ce qu'un **fichier .env** et comment protège-t-il les informations sensibles ?
- Pourquoi faut-il **limiter les accès au serveur** (ex : interdiction de se connecter en "root") ?
- Quels sont les **risques liés à une API publique**, et comment peut-on limiter l'accès à certaines fonctionnalités ?

Astuce : Même en local, prendre **de bonnes habitudes** dès le début permet d'éviter de graves erreurs en production.

Livable attendu : Une **liste des bonnes pratiques** pour sécuriser un backend, même en environnement de test.

Architecture Monolithique

Objectifs de cette section

Dans cette section, nous allons explorer le modèle **monolithique**, qui est la forme la plus simple d'architecture applicative. À la fin de cette section, vous serez capable de :

- Définir ce qu'est une **application monolithique**.
 - Identifier ses **avantages et inconvénients**.
 - Comprendre son **fonctionnement et ses limites**.
 - Mettre en place une **application backend monolithique en PHP et Node.js**.
 - Déterminer les **implications en infrastructure** et préparer son **hébergement et sa maintenance**.
-

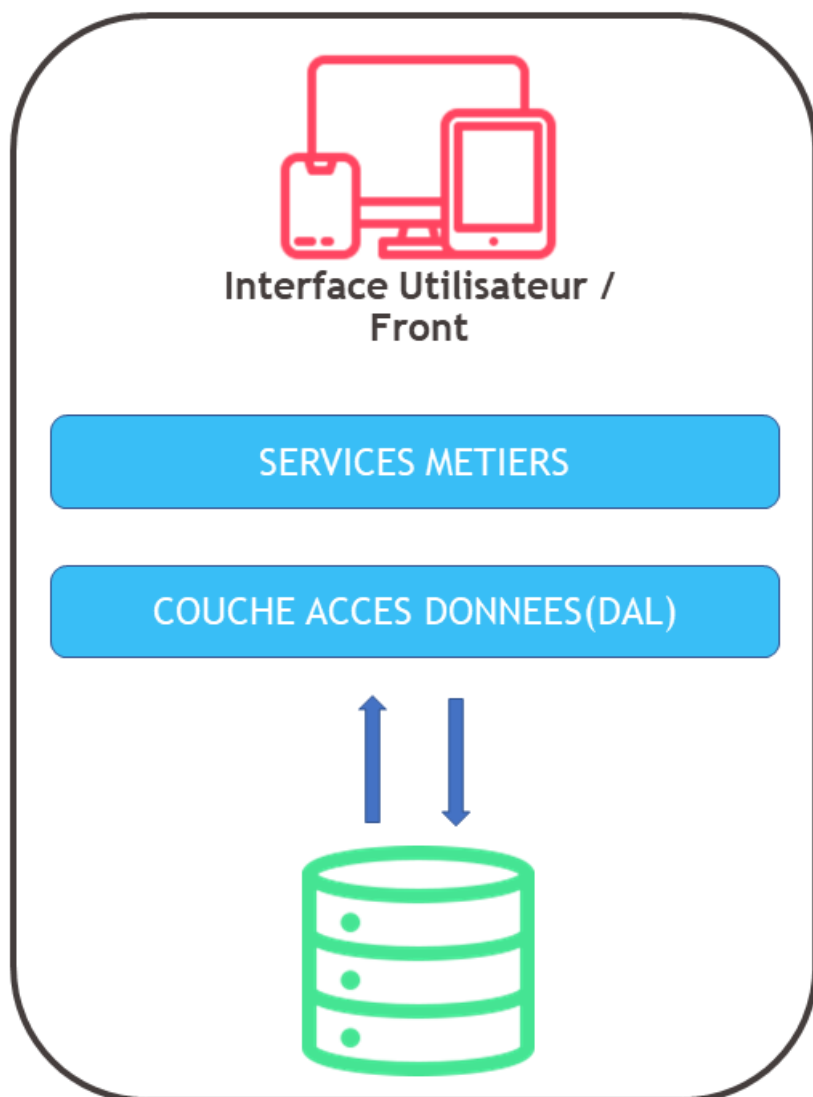
1. Qu'est-ce qu'une architecture monolithique ?

Une **application monolithique** est un **bloc unique**, où toutes les fonctionnalités (interface utilisateur, logique métier, accès aux données) sont **regroupées dans un seul codebase et exécutées sur un même serveur**.

Caractéristiques principales

- Une seule application contenant **toutes les fonctionnalités**.
- Une **base de données centralisée**.
- Une **exécution sur un seul serveur ou une seule machine virtuelle**.

Schéma d'une architecture monolithique :



2. Avantages et Inconvénients du monolithique

Aspect	Avantages	Inconvénients
Développement	Facile à mettre en place et à comprendre	Peut devenir difficile à maintenir sur le long terme
Performance	Pas de communication réseau entre les composants	Difficulté à monter en charge
Déploiement	Une seule unité à gérer	Impossible de déployer certaines parties séparément
Évolutivité	Simple pour les petits projets	Peu adapté aux grandes applications
Sécurité	Moins de points d'entrée = moins de vulnérabilités	Un bug critique peut affecter tout le système

3. Mise en place d’un backend monolithique

Nous allons maintenant créer une **application de gestion de tâches** en **architecture monolithique**. Cette application comprendra une API REST permettant d'ajouter, modifier et supprimer des tâches dans une **base de données locale**.

Les technologies utilisées :

- **PHP + SQLite** pour la version PHP.
- **Node.js + SQLite** pour la version JavaScript.

Arborescence du projet à modifier par les développeurs

```
/gestion-taches
├── /php                                # Implémentation en PHP
│   ├── index.php                      # Point d'entrée
│   ├── db.php                        # Connexion à la base de données
│   ├── Task.php                      # Modèle Task
│   ├── taskController.php            # Contrôleur des tâches
│   ├── api.php                      # API REST pour gérer les tâches
│   ├── .env                          # Configuration de la base de données
│   └── composer.json                 # Dépendances PHP
├── /nodejs                            # Implémentation en Node.js
│   ├── index.js                     # Point d'entrée
│   ├── db.js                       # Connexion à la base de données
│   ├── task.js                     # Modèle Task
│   ├── taskController.js           # Contrôleur des tâches
│   ├── api.js                     # API REST pour gérer les tâches
│   ├── .env                        # Configuration de la base de données
│   └── package.json                 # Dépendances Node.js
└── README.md                       # Documentation
```

4. Mise en pratique pour les administrateurs infrastructure

En plus du développement du backend, les administrateurs système devront analyser **l'hébergement et la gestion d'un monolithe**.

Identification des composants techniques

Objectif :
Analyser l'architecture et les services nécessaires.

- Quels sont les **composants techniques** du projet ?
- Quels services doivent être installés et configurés sur le serveur ?
- Quelle est la **configuration minimale requise** pour héberger l'application ?

Livrable attendu : Un schéma des composants avec une description de leur rôle.

Hébergement et installation

Objectif :

Déterminer comment et où héberger l'application.

- Faut-il utiliser un **serveur physique, un VPS, un hébergement mutualisé, ou un hébergement cloud** ?
- Quels outils doivent être installés sur le serveur pour PHP et pour Node.js ?
- Faut-il prévoir un **serveur unique ou plusieurs instances** ?

Livable attendu : Un plan d'installation détaillé pour PHP et Node.js.

Sauvegardes et maintenance

Objectif :

Prévoir une stratégie de sauvegarde et de mise à jour.

- Quels fichiers et bases de données doivent être sauvegardés ?
- Quelle fréquence de sauvegarde recommander ?
- Comment gérer les **mises à jour du code et des dépendances** ?

Livable attendu : Une proposition de stratégie de sauvegarde.

Sécurité et monitoring

Objectif :

Sécuriser l'application et surveiller ses performances.

- Quels ports doivent être ouverts et quels services doivent être protégés ?
- Comment configurer **un pare-feu et un accès SSH sécurisé** ?
- Quels outils utiliser pour surveiller **l'état du serveur et la consommation des ressources** ?

Livable attendu : Une check-list des mesures de sécurité à appliquer.