UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Architecture Logicielle

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction à l'Architecture Logicielle

Introduction à l'Architecture Logicielle

Définition

- L'architecture logicielle implique la définition des composants ou modules d'un système logiciel, ainsi que des interfaces par lesquelles ces modules interagissent.
- Elle inclut également les directives qui régissent leur interaction, leur intégration, et la stratégie de leur déploiement et maintenance.
- Les architectes logiciels utilisent divers outils et modèles, comme les diagrammes UML (Unified Modeling Language), pour décrire la structure, le comportement et les interactions des composants au sein du système.

Introduction à l'Architecture Logicielle

Importance de l'Architecture Logicielle

1. **Facilite la communication:** L'architecture fournit une vue de haut niveau du système qui est compréhensible pour les parties prenantes à tous les niveaux. Cela aide les développeurs, les gestionnaires de projet, et les parties prenantes non techniques à comprendre le système et ses interactions.
2. **Prise de décision éclairée:** L'architecture sert de support à la prise de décision en offrant une vue globale sur les contraintes techniques et les trade-offs. Cela permet aux décideurs de choisir les meilleures stratégies en fonction des objectifs du projet et des contraintes opérationnelles.

Introduction à l'Architecture Logicielle

Importance de l'Architecture Logicielle

3. **Amélioration de la qualité:** Une bonne architecture permet d'assurer la qualité du système en termes de performance, de fiabilité, de réutilisabilité et de sécurité. Elle aide à détecter les problèmes potentiels tôt dans le cycle de développement et à élaborer des stratégies pour les résoudre.
4. **Maintenabilité et évolutivité:** Une architecture bien conçue est flexible et peut évoluer pour répondre à des exigences changeantes sans nécessiter une refonte complète. Elle facilite également la maintenance en organisant le système de manière logique et compréhensible.

Introduction à l'Architecture Logicielle

Importance de l'Architecture Logicielle

5. **Réduction des coûts et des risques:** En anticipant les défis techniques et en planifiant à l'avance, une architecture solide réduit les risques associés à la non-conformité aux exigences, les dépassements de coûts, et les échecs dans la mise en œuvre.
6. **Réutilisation des composants:** Une architecture qui encourage la modularité permet la réutilisation de composants dans d'autres systèmes ou projets, ce qui peut accélérer le développement et réduire les coûts.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Rôles de l'Architecte Logiciel

1. **Concepteur de la structure du système:**

- Définit l'architecture globale du système en choisissant les technologies appropriées et en structurant les composants logiciels de manière à répondre aux exigences du projet tout en maximisant la performance et la maintenabilité.
- S'assure que l'architecture choisie respecte à la fois les exigences techniques et les contraintes budgétaires ou de délai.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Rôles de l'Architecte Logiciel

2. **Responsable de la qualité technique:**

- Veille à ce que le système soit robuste, sécurisé, et facile à maintenir.
- Intègre des pratiques et des standards de qualité dans la conception du système, comme les tests, les revues de code, et les métriques de qualité.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Rôles de l'Architecte Logiciel

3. **Coordinateur entre les équipes de développement et les parties prenantes:**

- Assure la communication entre les développeurs, les gestionnaires de projet, les utilisateurs finaux, et les autres parties prenantes pour garantir que l'architecture répond à toutes les exigences et attentes.
- Traduit les exigences techniques pour les parties prenantes non techniques et vice versa.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Rôles de l'Architecte Logiciel

4. **Guide et mentor pour l'équipe de développement:**

- Fournit un leadership technique et des conseils aux développeurs.
- Aide l'équipe à comprendre l'architecture et les principes sous-jacents.
- Organise des formations et des séminaires pour maintenir l'équipe à jour avec les nouvelles technologies et méthodes.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Responsabilités de l'Architecte Logiciel

1. **Définition de l'architecture:**

- Élabore les modèles architecturaux, les plans et les prototypes.
- Définit les normes de codage, les outils et les plateformes à utiliser.
- Assure la cohérence de l'architecture à travers toutes les phases du projet.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Responsabilités de l'Architecte Logiciel

2. **Évaluation et gestion des risques:**

- Identifie les risques techniques potentiels et propose des solutions pour les atténuer.
- Évalue régulièrement l'efficacité de l'architecture actuelle et propose des modifications si nécessaire.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Responsabilités de l'Architecte Logiciel

3. **Optimisation des performances et de l'évolutivité:**

- Conçoit des systèmes capables de s'adapter à l'augmentation des charges de travail ou à l'évolution des exigences sans performances dégradées.
- Analyse les performances du système et identifie les goulets d'étranglement.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Responsabilités de l'Architecte Logiciel

4. **Documentation de l'architecture:**

- Rédige des documents détaillés sur l'architecture pour assurer une référence claire et une maintenance aisée.
- Maintient une documentation à jour à mesure que le système évolue.

Introduction à l'Architecture Logicielle

Rôles et responsabilités d'un architecte logiciel

Responsabilités de l'Architecte Logiciel

5. **Innovation et recherche:**

- Se tient informé des dernières tendances et technologies en architecture logicielle.
- Explore de nouvelles approches et technologies pour améliorer continuellement la conception du système.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Styles et patrons d'architecture

Styles et patrons d'architecture

Définition

- Les **styles et patrons d'architecture logicielle** sont des approches structurées utilisées pour résoudre des problèmes de conception communs lors de la création de systèmes logiciels.
- Ces **styles et patrons** aident à structurer le système de manière à faciliter la gestion de la complexité et à améliorer la maintenabilité et la scalabilité.

Styles et patrons d'architecture

Styles d'Architecture

1. **Architecture Monolithique:**

- ***Description:*** Dans ce style, l'application est développée comme un bloc unique qui fonctionne de manière autonome. Toutes les fonctions de l'application sont gérées en un seul et même endroit.
- ***Utilisation typique:*** Applications simples ou de petite taille où la simplicité de déploiement est prioritaire.

Styles et patrons d'architecture

Styles d'Architecture

1. **Architecture Monolithique:**

L'architecture monolithique est un style d'architecture logicielle où une application est conçue comme une unité unique et indivisible. Ce type d'architecture est souvent utilisé pour des applications plus petites ou des systèmes où la complexité n'est pas extrêmement élevée

Styles et patrons d'architecture

Styles d'Architecture

Les caractéritiques de l'architecture Monolithique:

- **Unité unique de déploiement :** Toutes les fonctionnalités de l'application sont regroupées en un seul paquet logiciel ou en une seule base de code, et déployées ensemble. Cela signifie que chaque fois qu'un changement est apporté, même mineur, toute l'application doit être recompilée et redéployée.
- **Simplicité de développement :** Étant donné que tout est contenu dans une seule base de code, le développement, le déploiement et le test de l'application peuvent être plus simples, en particulier pour les petites équipes ou les projets moins complexes.

Styles et patrons d'architecture

Styles d'Architecture

Les caractéritiques de l'architecture Monolithique:

- **Performance :** Les applications monolithiques peuvent offrir de bonnes performances pour des charges de travail spécifiques car elles évitent la latence associée aux appels de service ou aux communications réseau internes qu'on trouve dans les architectures distribuées.
- **Limitations à la scalabilité :** La scalabilité horizontale (ajout de plus de machines) d'une application monolithique peut être plus complexe par rapport à une architecture basée sur des microservices. Pour scaler, il faut souvent déployer l'entier de l'application sur chaque serveur, ce qui peut être coûteux en ressources.

Styles et patrons d'architecture

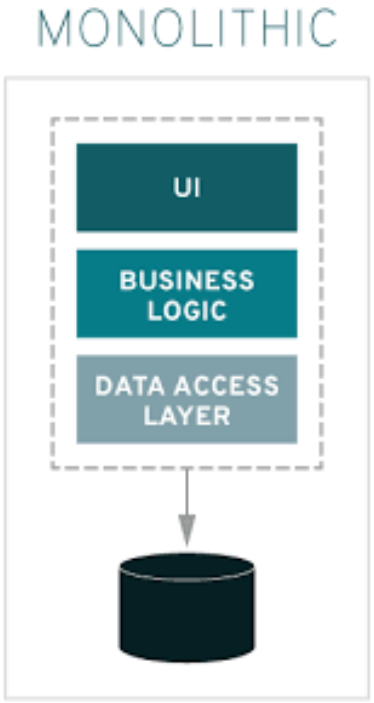
Styles d'Architecture

Les caractéritiques de l'architecture Monolithique:

- **Dépendances et couplage :** Les composants dans une application monolithique sont souvent étroitement couplés, ce qui peut rendre difficile la mise à jour de certaines parties de l'application sans affecter d'autres parties. Cela peut également compliquer la maintenance et l'évolution de l'application.
- **Résilience :** Dans un monolithe, un bug ou une défaillance dans une partie de l'application peut potentiellement affecter l'ensemble du système, réduisant la résilience globale.

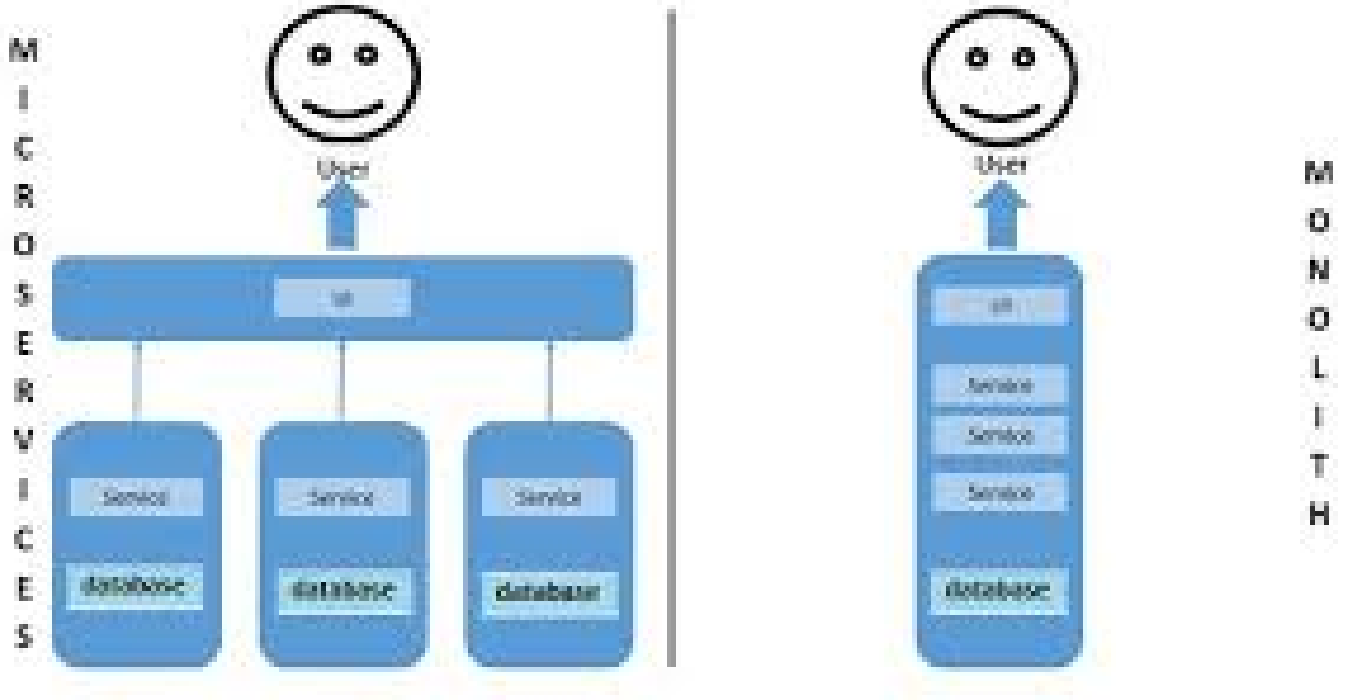
Introduction à la sécurité informatique

1. Architecture Monolithique:



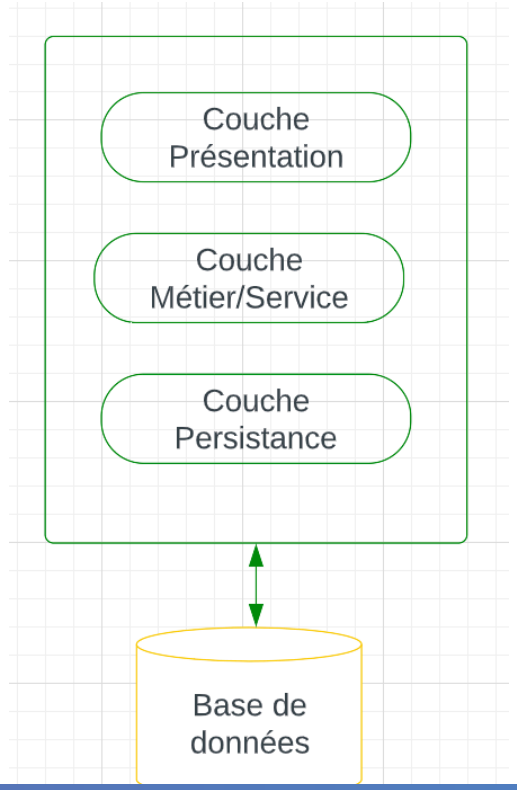
Introduction à la sécurité informatique

1. Architecture Monolithique:



Introduction à la sécurité informatique

1. Architecture Monolithique:



Styles et patrons d'architecture

Styles d'Architecture

2. **Architecture Orientée Service (SOA):**

- **Description:** Consiste en services faiblement couplés qui peuvent être réutilisés et combinés pour créer des applications. Chaque service encapsule une fonctionnalité métier complète.
- **Utilisation typique:** Intégration d'applications d'entreprise et systèmes où la flexibilité et la réutilisation des services sont essentielles.

Styles et patrons d'architecture

Styles d'Architecture

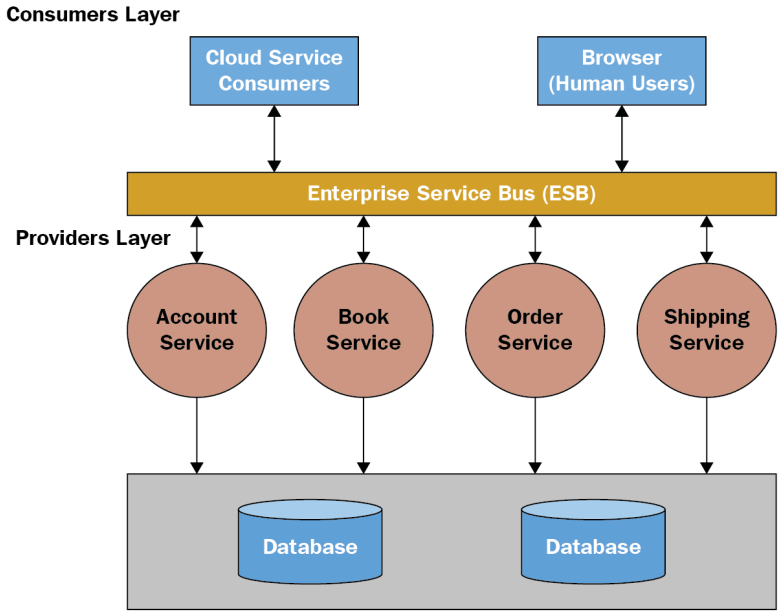
2. **Architecture Orientée Services (SOA):**

- L'architecture orientée services (SOA) est un style architectural qui permet la conception d'applications en décomposant les fonctionnalités en services distincts, généralement sur un réseau.
- Ceux-ci communiquent entre eux via des interfaces bien définies et des protocoles standards, souvent basés sur des messages.
- SOA vise à améliorer la flexibilité, la réutilisabilité, l'évolutivité et l'interopérabilité des applications d'entreprise.

Styles et patrons d'architecture

Styles d'Architecture

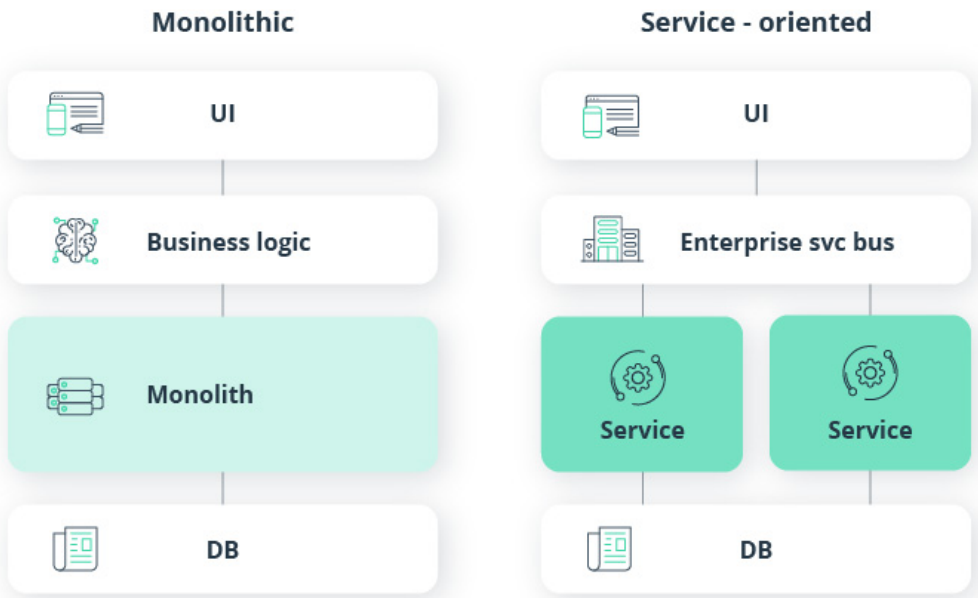
2. **Architecture Orientée Services (SOA):**



Styles et patrons d'architecture

Styles d'Architecture

2. **Architecture Orientée Services (SOA):**



Styles et patrons d'architecture

Styles d'Architecture

Les caractéritiques de l'architecture SOA:

- **Modularité :** SOA organise les fonctionnalités en services indépendants, ce qui permet de gérer, de développer et de déployer les services de manière indépendante les uns des autres.
- **Réutilisation des services :** Les services sont conçus pour être réutilisables dans différents contextes et applications. Cela permet une économie de coûts et de temps en réduisant la duplication des efforts de développement.

Styles et patrons d'architecture

Styles d'Architecture

Les caractéritiques de l'architecture SOA:

- **Interopérabilité :** SOA favorise l'utilisation de normes ouvertes et de protocoles communs (comme HTTP, REST, SOAP) pour faciliter l'interaction entre différents systèmes et technologies. Cela est crucial pour intégrer divers systèmes hétérogènes au sein d'une entreprise.
- **Loose Coupling (Faible Couplage) :** Les services sont faiblement couplés, ce qui signifie que les services interagissent les uns avec les autres sans avoir besoin de connaître les détails des implémentations internes. Cela simplifie les modifications et les mises à jour sans affecter les consommateurs du service.

Styles et patrons d'architecture

Styles d'Architecture

Les caractéritiques de l'architecture SOA:

- **Abstraction :** Les détails de l'implémentation des services sont cachés derrière une interface. Les consommateurs de services interagissent avec ces interfaces et non avec l'implémentation même du service, ce qui permet de modifier l'implémentation sans perturber les utilisateurs.
- **Orchestration :** SOA permet l'orchestration de services pour créer des processus métier complexes en combinant plusieurs services plus simples. Cela est souvent géré par des moteurs d'orchestration qui coordonnent l'ordre d'exécution des services et gèrent les transactions et les états du processus.

Styles et patrons d'architecture

Styles d'Architecture

Les caractéritiques de l'architecture SOA:

- **Gouvernance :** SOA nécessite une gestion et une gouvernance solides pour s'assurer que les services sont développés selon des normes cohérentes, bien documentés, et que leur utilisation est en accord avec les politiques de l'entreprise.

Styles et patrons d'architecture

Styles d'Architecture

Avantages de SOA

Flexibilité : Facilité d'intégration et d'adaptation aux changements des exigences d'affaires.
Scalabilité : Les services peuvent être déployés sur plusieurs serveurs ou dans le cloud pour gérer des charges de travail accrues.
Maintenabilité : Mises à jour et améliorations plus simples grâce à l'indépendance des services.

Styles et patrons d'architecture

Styles d'Architecture

3. **Architecture Microservices :**

- **Description:** Approche où l'application est divisée en petits services autonomes qui communiquent via des API. Chaque microservice est responsable d'une fonctionnalité spécifique et peut être déployé indépendamment.
- **Utilisation typique:** Applications complexes et évolutives où la déployabilité, la maintenabilité et la scalabilité sont critiques.

Styles et patrons d'architecture

Styles d'Architecture

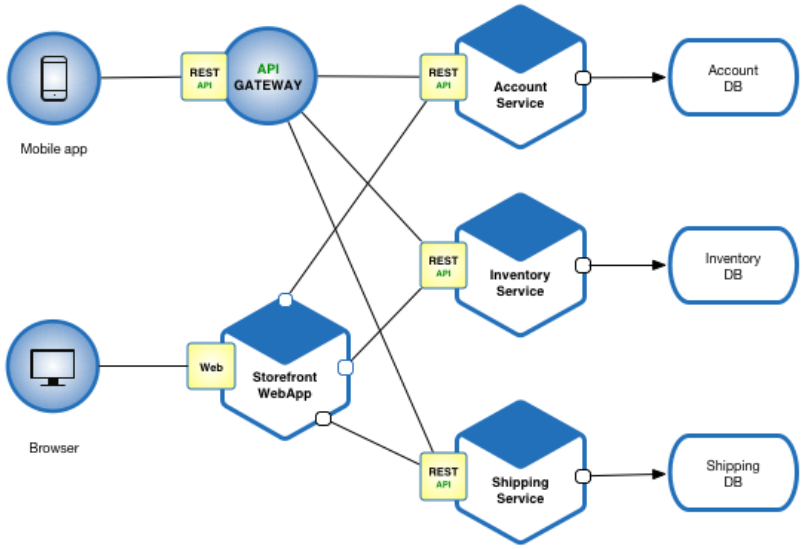
3. **Architecture Microservices :**

- L'architecture de microservices est un style architectural où une application est structurée en tant qu'ensemble de services indépendants, chacun exécutant une fonctionnalité spécifique.
- Ce style de conception a gagné en popularité en raison de sa flexibilité, sa scalabilité et son efficacité pour les applications complexes et à grande échelle.

Styles et patrons d'architecture

Styles d'Architecture

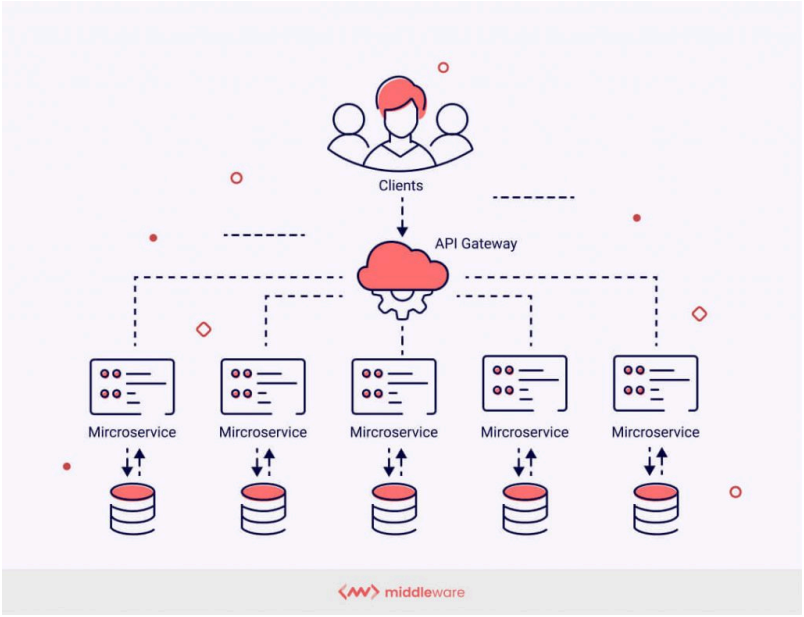
3. **Architecture Microservices :**



Styles et patrons d'architecture

Styles d'Architecture

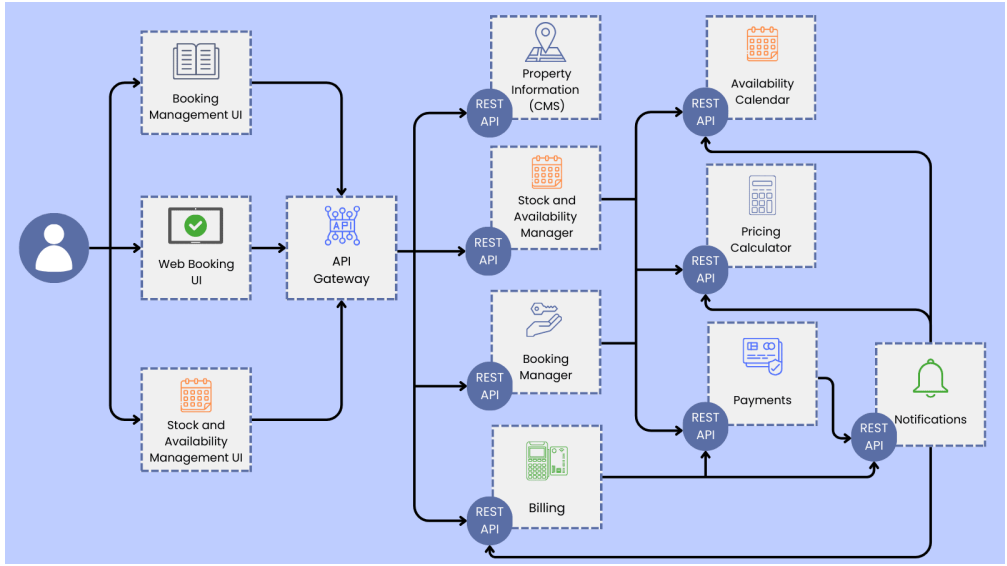
3. **Architecture Microservices :**



Styles et patrons d'architecture

Styles d'Architecture

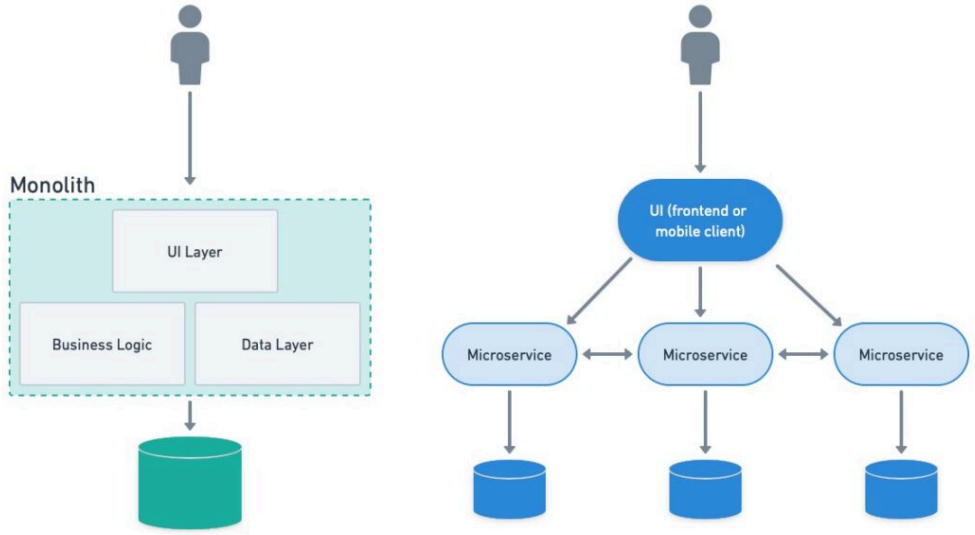
3. **Architecture Microservices :**



Styles et patrons d'architecture

Styles d'Architecture

3. **Architecture Microservices :**



Styles et patrons d'architecture

Styles d'Architecture

Caractéristiques principales des microservices

- **Décomposition fonctionnelle :** Les applications sont divisées en petits services autonomes, chacun responsable d'une fonctionnalité précise ou d'un domaine métier spécifique.
- **Indépendance :** Chaque microservice est développé, déployé, et géré de manière indépendante des autres. Cela permet des mises à jour et des maintenances sans perturber le reste de l'application.

Styles et patrons d'architecture

Styles d'Architecture

Caractéristiques principales des microservices

- **Polyglotte :** Les microservices peuvent être écrits dans différents langages de programmation, utilisant les technologies les plus adaptées à leurs besoins spécifiques.
- **Communication légère :** Les services communiquent entre eux en utilisant des protocoles légers, souvent à base d'API HTTP/REST, mais également par des messages asynchrones.

Styles et patrons d'architecture

Styles d'Architecture

Caractéristiques principales des microservices

- **Décentralisation de la gestion des données :** Chaque service gère sa propre base de données, ce qui évite les dépendances croisées entre services et optimise les performances.
- **Automatisation :** Le déploiement et la gestion des microservices sont souvent automatisés avec des outils de CI/CD (Continuous Integration / Continuous Deployment) et des plateformes d'orchestration comme Kubernetes.

Styles et patrons d'architecture

Styles d'Architecture

Avantages des microservices

- **Agilité :** La petite taille et l'autonomie des services facilitent les modifications rapides et l'expérimentation, accélérant le développement de nouvelles fonctionnalités.
- **Scalabilité :** Il est possible de scaler les services indépendamment en fonction de leurs besoins de ressources, ce qui est plus efficace que de scaler une application monolithique entière.

Styles et patrons d'architecture

Styles d'Architecture

Avantages des microservices

- **Résilience :** Les problèmes dans un microservice spécifique affectent moins probablement l'ensemble de l'application, permettant une meilleure gestion des pannes et une disponibilité accrue.
- **Réutilisation :** Les services peuvent être réutilisés dans différents contextes et applications, facilitant le partage de fonctionnalités entre équipes et projets.

Styles et patrons d'architecture

Styles d'Architecture

Défis des microservices

1. **Complexité de gestion :** La multiplication des services peut entraîner une complexité accrue en termes de gestion, de monitoring, et de sécurisation des communications entre services.
2. **Consistance des données :** Maintenir la cohérence des données à travers les différents services peut être complexe, surtout avec des transactions s'étendant sur plusieurs services.
3. **Latence de communication :** La communication entre services peut introduire des délais supplémentaires, surtout lorsque les services sont physiquement distants ou mal optimisés.
4. **Overhead :** Le déploiement d'une architecture à microservices peut nécessiter des ressources supplémentaires en termes de développement, de test, et d'opérations, nécessitant des compétences et des outils spécialisés.

Styles et patrons d'architecture

Styles d'Architecture

4. Architecture Serverless

- **Description:** Les développeurs écrivent principalement des fonctions gérées et exécutées par un fournisseur cloud, qui gère dynamiquement l'allocation des ressources.
- **Utilisation typique:** Applications avec des motifs de charge variable, idéal pour réduire les coûts d'infrastructure en payant uniquement pour les ressources utilisées lors de l'exécution des fonctions.

Styles et patrons d'architecture

Styles d'Architecture

4. Architecture Serverless

- L'architecture serverless, également connue sous le nom de "FaaS" (Function as a Service), est un modèle de développement où les développeurs peuvent écrire et déployer du code sans se soucier de la gestion de l'infrastructure sous-jacente.
- Les fournisseurs de cloud, comme AWS, Microsoft Azure, et Google Cloud Platform, gèrent l'exécution du code, les serveurs, la gestion des ressources, la maintenance, et la scalabilité.
- Le code dans une architecture serverless est généralement exécuté en réponse à des événements et des triggers, et le fournisseur facture uniquement pour le temps de calcul utilisé lors de l'exécution du code

Styles et patrons d'architecture

Styles d'Architecture

Caractéristiques principales de l'architecture Serverless

- **Évolutivité automatique :** Le fournisseur de cloud gère l'allocation de ressources et la scalabilité, permettant au système de s'adapter automatiquement à la charge sans intervention manuelle.
- **Facturation basée sur l'utilisation :** Les coûts sont basés uniquement sur le temps de calcul et les ressources consommées lors de l'exécution du code, sans frais pour les serveurs inactifs.

Styles et patrons d'architecture

Styles d'Architecture

Caractéristiques principales de l'architecture Serverless

- **Gestion de l'infrastructure par le fournisseur :** La gestion des serveurs, la mise à jour des systèmes d'exploitation, et les patchs de sécurité sont tous gérés par le fournisseur, réduisant la charge de travail des développeurs.
- **Temps de réponse aux événements :** Le code est souvent conçu pour exécuter des fonctions en réponse à des événements spécifiques (comme des requêtes HTTP, des modifications dans une base de données, ou des messages de file d'attente).

Styles et patrons d'architecture

Styles d'Architecture

Avantages de l'architecture Serverlesss

- **Déploiement rapide :** Les développeurs peuvent se concentrer sur la logique métier et le code, plutôt que sur la configuration et la gestion de l'infrastructure.
- **Réduction des coûts opérationnels :** Élimine le besoin d'achat, de gestion, et de maintenance de serveurs physiques ou virtuels, ce qui peut réduire considérablement les coûts.
- **Agilité :** Permet une innovation rapide en réduisant le temps entre le développement d'une fonctionnalité et sa mise en production.
- **Optimisation des ressources :** Comme le système est élastique, il n'y a pas de surprovisionnement de ressources; on utilise et on paie seulement ce qui est nécessaire.

Styles et patrons d'architecture

Styles d'Architecture

Défis de l'architecture Serverless

- **Dépendance au fournisseur :** Le code peut devenir dépendant des API spécifiques du fournisseur de cloud, ce qui peut limiter la portabilité et augmenter le risque de verrouillage avec un fournisseur.
- **Complexité de test :** Tester les applications serverless peut être complexe, en particulier pour simuler les environnements et les interactions des services gérés par le fournisseur.
- **Latences à froid :** Les fonctions qui n'ont pas été utilisées récemment peuvent subir une "latence à froid" lorsqu'elles sont réactivées, ce qui entraîne un délai supplémentaire lors du premier appel après une période d'inactivité.
- **Gestion de l'état :** Étant donné que les fonctions sont stateless par défaut, la gestion des états entre les exécutions de fonctions peut nécessiter des services supplémentaires ou des stratégies complexes.

Styles et patrons d'architecture

Patrons d'Architecture

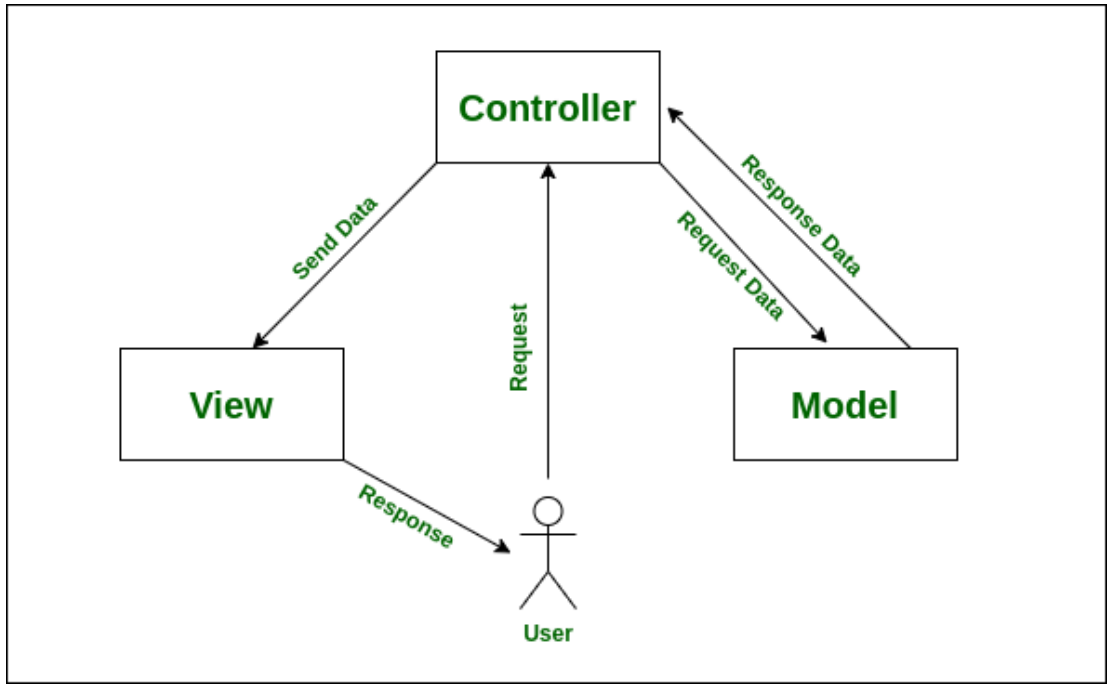
MVC (Model-View-Controller)

- **Description:** Sépare l'application en trois composants principaux : le modèle (données), la vue (interface utilisateur), et le contrôleur (logique de traitement). Permet une séparation claire entre la présentation et la logique métier.
- **Utilisation typique:** Applications web où une séparation claire entre l'interface utilisateur et la logique métier est nécessaire.

Styles et patrons d'architecture

Patrons d'Architecture

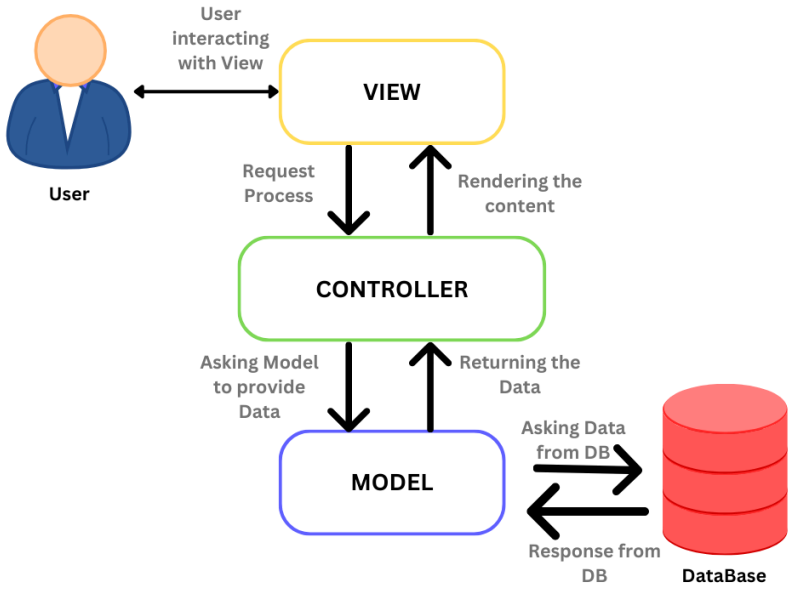
MVC (Model-View-Controller)



Styles et patrons d'architecture

Patrons d'Architecture

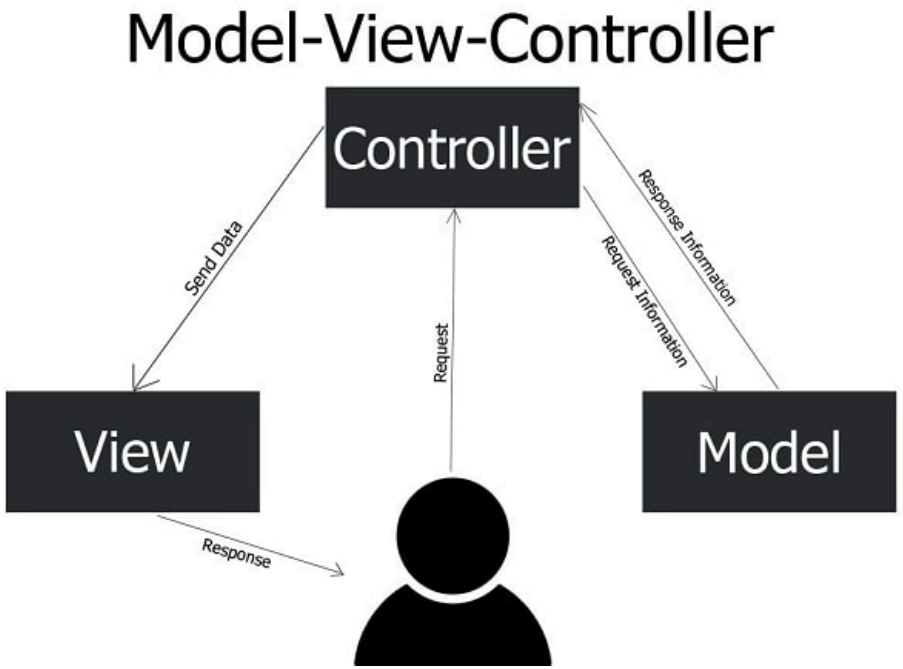
MVC (Model-View-Controller)



Styles et patrons d'architecture

Patrons d'Architecture

MVC (Model-View-Controller)



Styles et patrons d'architecture

Patrons d'Architecture

MVC (Model-View-Controller)

- Le **modèle Model-View-Controller (MVC)** est un motif d'architecture logicielle qui sépare une application en trois composants principaux : le modèle, la vue, et le contrôleur.
- Ce motif est très utilisé pour développer des interfaces utilisateur qui séparent la logique métier de l'interface utilisateur, ce qui améliore la modularité et facilite la gestion et la maintenance du code.

Styles et patrons d'architecture

Patrons d'Architecture

MVC (Model-View-Controller)

1. **Modèle (Model)**

- Le modèle représente la logique métier et les données de l'application.
- Il gère les règles de l'entreprise, les données, les algorithmes et les fonctions de base.
- Le modèle communique avec la base de données, encapsule l'état de l'application et déclenche des événements que le contrôleur peut utiliser pour mettre à jour la vue.
- Il est complètement indépendant de la vue et du contrôleur, ce qui signifie que le modèle n'a pas de connaissance directe de ce qui se passe sur l'interface utilisateur.

Styles et patrons d'architecture

Patrons d'Architecture

MVC (Model-View-Controller)

2. **Vue (View)**

- La vue est la présentation de l'interface utilisateur
- Elle affiche les données reçues du modèle à l'utilisateur et envoie les interactions de l'utilisateur (comme les clics de souris, les entrées clavier) au contrôleur.
- La vue est généralement composée de tous les éléments nécessaires pour l'interaction utilisateur comme les boutons, les menus, et les champs de texte.
- Elle écoute les modifications du modèle, qui peuvent parfois entraîner une mise à jour de l'affichage présenté à l'utilisateur.

Styles et patrons d'architecture

Patrons d'Architecture

MVC (Model-View-Controller)

3. **Contrôleur (Controller)**

- Le contrôleur agit comme un intermédiaire entre le modèle et la vue.
- Il écoute les événements de l'interface utilisateur envoyés par la vue et exécute les actions appropriées sur le modèle.
- Quand le modèle est mis à jour, le contrôleur informe la vue pour qu'elle se rafraîchisse ou modifie son affichage en fonction des données modifiées.
- Le contrôleur décode les entrées reçues de la vue, les transforme en requêtes pour le modèle ou les vues appropriées.

Styles et patrons d'architecture

Patrons d'Architecture

Avantages de MVC

- **Séparation des préoccupations :** Chaque composant a une responsabilité claire, facilitant ainsi la maintenance et l'évolution de l'application.
- **Développement parallèle :** Comme les composants sont indépendants, plusieurs développeurs peuvent travailler simultanément sur le modèle, la vue, et le contrôleur sans interférence.
- **Réutilisation du code :** Le modèle, une fois créé, peut être utilisé par différentes vues pour différentes interfaces utilisateur.
- **Adaptabilité :** Changer l'interface utilisateur d'une application peut être aussi simple que de créer une nouvelle vue sans changer le modèle sous-jacent.

Styles et patrons d'architecture

Patrons d'Architecture

Défis de MVC

- **Complexité :** Le motif MVC peut introduire une complexité supplémentaire dans la structure de l'application, ce qui peut être un surcoût pour des projets simples.
- **Synchronisation :** Maintenir la synchronisation entre la vue et le modèle peut être difficile, surtout quand l'interface utilisateur doit réagir en temps réel aux changements.
- **Gestion des mises à jour :** Les dépendances entre les vues et les modèles peuvent parfois rendre les mises à jour de l'interface utilisateur plus complexes.

Styles et patrons d'architecture

Patrons d'Architecture

MVVM (Model-View-ViewModel)

- **Description:** Variante de MVC, MVVM relie la Vue à un ViewModel qui agit comme un médiateur qui réfléchit les données du Modèle et les actions de l'utilisateur.
- **Utilisation typique:** Applications avec des interfaces riches, souvent utilisées dans des applications de bureau ou mobiles.

Styles et patrons d'architecture

Patrons d'Architecture

MVVM (Model-View-ViewModel)

- **MVVM (Model-View-ViewModel)** est un modèle architectural qui facilite la séparation des préoccupations dans les applications, notamment celles développées sur des plateformes comme WPF (Windows Presentation Foundation), Silverlight, natively in .NET, et plus récemment utilisé dans des frameworks de développement web comme Angular.

Styles et patrons d'architecture

Patrons d'Architecture

Composants de MVVM:

1. **Model**

- **Rôle:** Représente les données et la logique métier de l'application.
- **Caractéristiques:** Peut inclure la validation, les interfaces avec les bases de données, et d'autres opérations de données.

Styles et patrons d'architecture

Patrons d'Architecture

Composants de MVVM:

2. **View**

- **Rôle:** Présente les données à l'utilisateur; c'est la couche d'interface utilisateur.
- **Caractéristiques:** Composée de tous les éléments visuels comme les boutons, les zones de texte, les graphiques, etc. Elle est passive, ce qui signifie qu'elle ne contient aucune logique métier ni logique de manipulation des données.

Styles et patrons d'architecture

Patrons d'Architecture

Composants de MVVM:

3. **ViewModel**

- **Rôle:** Fait le lien entre le Model et la View. Il expose des méthodes, des commandes et d'autres propriétés qui maintiennent l'état de la View.
- **Caractéristiques:** Agit comme un intermédiaire en fournissant des données du Model de manière que la View puisse les utiliser plus facilement.

Styles et patrons d'architecture

Patrons d'Architecture

Fonctionnement MVVM:

- **Binding de données:** Le ViewModel fournit un lien de données entre le Model et la View, permettant ainsi des mises à jour interactives. Si les données dans le Model changent, ces changements sont automatiquement reflétés dans la View via le ViewModel.
- **Indépendance de la View:** La View ne connaît que le ViewModel et aucunement le Model direct. Cette séparation permet un développement et des tests plus faciles des composants.
- **Abstraction de commandes:** Le ViewModel traite les actions de l'utilisateur via des commandes plutôt que des événements, ce qui réduit le couplage entre la View et le ViewModel.

Styles et patrons d'architecture

Patrons d'Architecture

Avantages MVVM:

- **Testabilité:** La séparation claire entre la logique de présentation et la logique métier facilite les tests unitaires.
- **Maintenabilité:** Il est plus facile de modifier ou de maintenir l'application sans que cela affecte d'autres couches.
- **Réutilisabilité:** Les composants du ViewModel peuvent être réutilisés dans différentes Views si nécessaire.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Principes de conception

Principes de conception

Definition

Les principes de conception sont des lignes directrices qui aident à structurer et organiser le code de manière à rendre les logiciels plus compréhensibles, maintenables, évolutifs et robustes. Ils fournissent un cadre pour prendre des décisions éclairées en matière de conception logicielle et permettent d'éviter les pièges courants qui peuvent rendre le code difficile à modifier ou à étendre.

Principes de conception

Objectifs des principes de conception

- **Modularité :** Séparer les fonctionnalités pour qu'elles soient indépendantes et réutilisables.
- **Cohérence :** Assurer que le code soit facile à comprendre pour les autres développeurs.
- **Maintenance :** Faciliter les modifications futures du code sans risques de casser des fonctionnalités existantes.
- **Extensibilité :** Permettre d’ajouter de nouvelles fonctionnalités sans devoir modifier de manière significative le code existant.

Principes de conception

Exemples de principes de conception courants

- **SOLID :** Ensemble de cinq principes pour la programmation orientée objet (SRP, OCP, LSP, ISP, DIP).
- **DRY (Don't Repeat Yourself) :** Chaque morceau de connaissance doit avoir une seule représentation au sein du système.
- **YAGNI (You Aren't Gonna Need It) :** Ne pas implémenter de fonctionnalités qui ne sont pas immédiatement nécessaires.
- **KISS (Keep It Simple, Stupid) :** Garder les choses simples et éviter la complexité inutile.
- **Separation of Concerns (SoC) :** Séparer les différents aspects d'un programme, chaque aspect étant responsable d'un comportement spécifique.
- **Encapsulation :** Regrouper les données et le comportement d'un objet ensemble tout en cachant les détails internes.
- **Cohésion :** Les éléments d'un module doivent être étroitement liés en termes de fonctionnalité.
- **Couplage faible :** Minimiser les dépendances entre différents modules pour éviter que des changements dans un module nécessitent des modifications dans d'autres.existant.

Principes de conception

SOLID

S - Single Responsibility Principle (SRP) : Un module ou une classe doit avoir une seule responsabilité ou une seule raison de changer. Cela signifie que chaque classe doit se concentrer sur une seule tâche ou fonction.
O - Open/Closed Principle (OCP) : Une entité (classe, module, fonction) doit être ouverte à l'extension mais fermée à la modification. Autrement dit, il doit être possible d'ajouter de nouvelles fonctionnalités sans modifier le code existant.
L - Liskov Substitution Principle (LSP) : Les objets d'une classe dérivée doivent pouvoir remplacer les objets de la classe mère sans altérer le bon fonctionnement du programme. Cela implique que les classes filles doivent conserver les comportements et les caractéristiques de la classe mère.
I - Interface Segregation Principle (ISP) : Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas. Il vaut mieux avoir plusieurs interfaces spécifiques plutôt qu'une seule interface générale.
D - Dependency Inversion Principle (DIP) : Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Tous deux doivent dépendre d'abstractions. En d'autres termes, il faut dépendre d'abstractions (interfaces ou classes abstraites) plutôt que de classes concrètes.

Principes de conception

DRY

- Le principe **DRY (Don't Repeat Yourself)** est un concept clé en développement logiciel et architecture qui consiste à éviter la duplication inutile de code, de logique métier ou de processus.
- En d'autres termes, l'idée est de centraliser et de réutiliser les composants, les fonctionnalités ou les concepts afin de réduire la redondance.

Principes de conception

DRY

1. *Modularité*

- **Définition :** DRY encourage la division d'une application en modules réutilisables, où chaque module encapsule une fonctionnalité unique.
- **Exemple :** Dans une architecture de microservices, chaque service est conçu pour être autonome et responsable d'une seule fonctionnalité ou d'un ensemble de fonctionnalités connexes. Cela permet de réutiliser les services sans répéter la même logique dans plusieurs services.

2. *Réutilisation du Code*

- **Définition :** DRY promeut la réutilisation du code pour éviter les répétitions. Cela peut se faire sous la forme de fonctions, bibliothèques ou services communs.
- **Exemple :** Dans une architecture orientée services (SOA) ou microservices, certains services, comme les services d'authentification, de journalisation ou de facturation, peuvent être utilisés par plusieurs autres services, plutôt que de réimplémenter ces fonctionnalités dans chaque service.

Principes de conception

DRY

3. *Centralisation de la Logique Métier*

- **Définition :** La logique métier ou les règles spécifiques à un domaine ne doivent pas être dupliquées dans plusieurs services ou composants. Elle doit être centralisée pour garantir que tout changement ou mise à jour soit reflété dans une seule source de vérité.
- **Exemple :** Utiliser un modèle Domain-Driven Design (DDD) pour isoler la logique métier dans des Bounded Contexts permet de respecter le principe DRY en regroupant la logique d'un domaine dans une seule unité fonctionnelle.

4. *Factorisation des Composants Techniques*

- **Définition :** DRY en architecture concerne aussi la factorisation des composants techniques communs à travers différentes parties d'une application.
- **Exemple :** Dans les architectures n-tiers ou microservices, l'intégration de composants communs comme des services de gestion de la configuration (par exemple, Spring Cloud Config) ou des services de sécurité (comme Keycloak) permet d'éviter de dupliquer ces éléments dans chaque service.

Principes de conception

DRY

5. *Outils de Réutilisation*

- **Définition :** DRY peut être renforcé par l'utilisation d'outils ou de frameworks qui encouragent la réutilisation de composants partagés.
- **Exemple :** Des bibliothèques partagées dans des environnements comme Maven ou Gradle dans des projets Java permettent de centraliser des fonctionnalités transversales (validation de données, gestion des erreurs, etc.) et de les réutiliser à travers plusieurs services ou modules.

6. *Centralisation de la Configuration*

- **Définition :** La configuration d'une application ne doit pas être dupliquée dans plusieurs endroits. DRY encourage l'utilisation de mécanismes pour centraliser la gestion de la configuration.
- **Exemple :** Dans une architecture distribuée, l'utilisation d'un service de configuration centralisé permet de centraliser les paramètres de configuration, plutôt que de les définir manuellement dans chaque service. Cela réduit la répétition et le risque d'erreurs.

Principes de conception

DRY

7. *Eviter la Duplication des Tests*

- **Définition :** DRY peut également s'appliquer aux tests. Plutôt que de dupliquer des scénarios de tests similaires, les tests doivent être factorisés et centralisés autant que possible.
- **Exemple :** En automatisant des tests unitaires et d'intégration réutilisables dans des suites de tests centralisées, vous garantissez que les tests couvrent toutes les parties pertinentes de l'application sans avoir à les répéter dans chaque module ou service.

8. *Gestion des APIs*

- **Définition :** Dans une architecture API, DRY incite à créer des API réutilisables plutôt que d'avoir des points de terminaison qui dupliquent la logique ou les données.
- **Exemple :** Utiliser une API Gateway pour exposer des fonctionnalités communes à plusieurs services et éviter de dupliquer la logique d'accès dans chaque service consommateur.

Principes de conception

DRY

7. *Eviter la Duplication des Tests*

- **Définition :** DRY peut également s'appliquer aux tests. Plutôt que de dupliquer des scénarios de tests similaires, les tests doivent être factorisés et centralisés autant que possible.
- **Exemple :** En automatisant des tests unitaires et d'intégration réutilisables dans des suites de tests centralisées, vous garantissez que les tests couvrent toutes les parties pertinentes de l'application sans avoir à les répéter dans chaque module ou service.

8. *Gestion des APIs*

- **Définition :** Dans une architecture API, DRY incite à créer des API réutilisables plutôt que d'avoir des points de terminaison qui dupliquent la logique ou les données.
- **Exemple :** Utiliser une API Gateway pour exposer des fonctionnalités communes à plusieurs services et éviter de dupliquer la logique d'accès dans chaque service consommateur.

Principes de conception

Avantages de DRY en architecture

- **Maintenance simplifiée :** En centralisant la logique, les modifications futures sont plus faciles à mettre en œuvre puisqu'il n'est pas nécessaire de modifier plusieurs endroits.
- **Cohérence :** En ayant une seule source de vérité, on évite des comportements incohérents ou des divergences entre différentes parties du système.
- **Réduction du coût :** Moins de redondance signifie moins de code à maintenir, tester et déployer, ce qui réduit les coûts globaux de développement et d’exploitation.

Principes de conception

KISS

- Le principe **KISS (Keep It Simple, Stupid)** est une philosophie de conception qui promeut la simplicité et l'efficacité. Il vise à éviter la complexité inutile et à privilégier des solutions simples, claires et faciles à maintenir.
- L'idée est de résoudre les problèmes avec des solutions directes et éprouvées, en évitant la sur-ingénierie.

Principes de conception

KISS

1. *Simplicité dans la conception des systèmes*

- **Définition :** KISS encourage à garder la conception d'un système aussi simple que possible, en évitant d'ajouter des fonctionnalités ou des éléments complexes qui ne sont pas strictement nécessaires.
- **Exemple :** Dans une architecture microservices, éviter de surcharger les services avec des responsabilités multiples. Chaque service doit se concentrer sur une seule tâche et être simple à comprendre et à maintenir.

2. *Modularité*

- **Définition :** La modularité est essentielle pour la simplicité. En divisant un système en modules bien définis et indépendants, il est plus facile de le gérer et de le faire évoluer.
- **Exemple :** Dans une architecture n-tiers, séparer les couches (présentation, application, données) en gardant des responsabilités claires et distinctes permet de simplifier la structure globale du système.

Principes de conception

KISS

3. *Eviter la sur-ingénierie*

- **Définition :** KISS recommande de ne pas créer des solutions trop complexes pour des problèmes simples. La simplicité prime sur des solutions sur-optimisées ou futuristes.
- **Exemple :** Si une base de données relationnelle répond au besoin d'une application, inutile d'introduire une base NoSQL plus complexe sans raisons justifiées.

4. *Utilisation de solutions éprouvées*

- **Définition :** Utiliser des outils, des bibliothèques et des frameworks connus et bien établis aide à maintenir la simplicité et la stabilité.
- **Exemple :** Choisir des frameworks comme Spring Boot ou Laravel, qui sont bien documentés et populaires, permet de maintenir la simplicité et de bénéficier de l'expérience collective de la communauté.

Principes de conception

KISS

5. *Simplicité dans le code*

- **Définition :** Le code doit être clair et facile à lire. KISS encourage l'écriture de code simple, lisible et facilement compréhensible par les autres développeurs.
- **Exemple :** Utiliser des conventions de nommage cohérentes, des méthodes courtes et éviter les optimisations complexes inutiles permettent d'améliorer la lisibilité et la maintenabilité du code.

6. *Facilité de maintenance*

- **Définition :** Un système simple est plus facile à maintenir, car il est plus compréhensible et moins sujet à des erreurs.
- **Exemple :** Une architecture avec des composants bien séparés et une documentation concise est plus facile à maintenir que des systèmes complexes avec des couches multiples et interdépendantes.

Principes de conception

KISS

7. *Réduction des dépendances*

- **Définition :** Limiter les dépendances entre composants permet de réduire la complexité et de maintenir un système simple et flexible.
- **Exemple :** Dans un projet de microservices, éviter de créer des dépendances croisées entre les services permet de garantir que chaque service reste autonome et simple à déployer.

8. *Refus des fonctionnalités inutiles*

- **Définition :** Ne pas ajouter des fonctionnalités complexes ou inutiles au système. Il faut résoudre le problème présent, sans anticiper des cas futurs qui pourraient ne jamais arriver.
- **Exemple :** Dans la conception d'une API, il est préférable de n'exposer que les endpoints nécessaires. Ajouter des fonctionnalités "au cas où" complique le développement et la maintenance.

Principes de conception

Avantages de KISS en architecture

- **Facilité de développement et d'évolution :** Des systèmes simples sont plus faciles à développer, à tester et à étendre au fur et à mesure que les besoins évoluent.
- **Moins d'erreurs :** Les architectures complexes sont souvent plus sujettes à des erreurs et à des bugs. KISS réduit ce risque en minimisant la complexité.
- **Économie de ressources :** Un système simple utilise souvent moins de ressources (temps de développement, matériel, etc.) et peut être mis en œuvre plus rapidement.

Principes de conception

YAGNI

- Le principe **YAGNI (You Aren't Gonna Need It)** est une pratique de développement issue de l'extrême programming (XP) qui consiste à ne pas implémenter des fonctionnalités dont vous n'avez pas immédiatement besoin.
- L'idée est de ne pas développer des fonctionnalités futures "au cas où", mais de se concentrer uniquement sur ce qui est nécessaire à l'instant.

Principes de conception

YAGNI

1. *Eviter l'anticipation de fonctionnalités*

- **Définition :** YAGNI recommande de ne pas coder des fonctionnalités en avance ou par prévision de besoins futurs.
- **Exemple :** Lors de la conception d'une application, ne pas ajouter des options ou des configurations si elles ne sont pas nécessaires dans la version actuelle du projet.

2. *Focalisation sur les besoins actuels*

- **Définition :** Ce principe encourage à se concentrer sur les besoins immédiats du projet, ce qui permet de livrer plus rapidement des fonctionnalités tangibles.
- **Exemple :** Plutôt que de prévoir une architecture qui gère des millions d'utilisateurs futurs dès le début, créer une architecture simple qui répond aux besoins actuels et l'améliorer au fil du temps en fonction de la croissance.

Principes de conception

YAGNI

3. *Réduction de la complexité*

- **Définition :** YAGNI aide à éviter la complexité inutile dans le code, ce qui conduit à des systèmes plus simples et plus maintenables.
- **Exemple :** Ne pas écrire des algorithmes complexes pour gérer des cas limites improbables ou anticipés. Se concentrer sur des scénarios simples et ajouter des fonctionnalités au fur et à mesure qu'elles deviennent nécessaires.

4. *Agilité dans le développement*

- **Définition :** Le principe YAGNI s'aligne avec la méthodologie agile, qui prône des cycles de développement courts, en se concentrant sur la livraison rapide des fonctionnalités les plus prioritaires.
- **Exemple :** Dans un développement agile, les équipes se concentrent sur des incréments de valeur immédiate plutôt que de construire des fonctionnalités complètes qui ne seront utilisées que bien plus tard.

Principes de conception

YAGNI

5. *Eviter la surcharge de fonctionnalités*

- **Définition :** Ne pas ajouter des fonctionnalités supplémentaires qui compliquent inutilement l'application.
- **Exemple :** Dans une API, se limiter aux endpoints nécessaires sans anticiper de futures intégrations. Si des extensions sont nécessaires plus tard, elles pourront être ajoutées sans surcharger l'architecture dès le départ.

6. *Limiter la dette technique*

- **Définition :** En appliquant YAGNI, on réduit la dette technique en n'introduisant pas de code ou de fonctionnalités inutiles qui devront être maintenues ou supprimées plus tard.
- **Exemple :** En ne codant que ce qui est nécessaire à un moment donné, on réduit les risques de devoir refactorer des parties non utilisées ou mal adaptées au contexte actuel.

Principes de conception

YAGNI

7. *Simplification du code*

- **Définition :** En évitant d'anticiper des fonctionnalités ou des cas d'usage futurs, le code reste simple et facile à comprendre.
- **Exemple :** Si une fonctionnalité de recherche avancée n'est pas nécessaire dans un MVP (Minimum Viable Product), il est préférable de créer une simple recherche textuelle et d'ajouter les fonctionnalités avancées ultérieurement si nécessaire.

8. *Amélioration continue*

- **Définition :** YAGNI encourage un développement itératif où l'on ajoute des fonctionnalités au fur et à mesure, basées sur les besoins réels plutôt que supposés.
- **Exemple :** Ajouter des fonctionnalités supplémentaires au produit à la demande, en fonction des retours des utilisateurs, plutôt que d'implémenter des fonctionnalités qui pourraient ne jamais être utilisées.

Principes de conception

Avantages de YAGNI en architecture

- **Réduction de la complexité :** En évitant les fonctionnalités superflues, l'architecture et le code restent simples et plus faciles à maintenir.
- **Meilleure gestion du temps :** En ne développant que ce qui est nécessaire, le temps de développement est optimisé, permettant de livrer plus rapidement les fonctionnalités critiques.
- **Agilité accrue :** YAGNI permet aux équipes de réagir plus rapidement aux changements de priorités ou aux nouvelles demandes, en minimisant les efforts gaspillés sur des fonctionnalités inutiles.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Patrons de conception

Patrons de conception

Definition

- Les patterns de conception (ou **design patterns**) sont des solutions récurrentes et éprouvées à des problèmes de conception courants dans le développement logiciel.
- Ce ne sont pas des morceaux de code prédéfinis, mais plutôt des modèles abstraits qui peuvent être appliqués pour résoudre des problèmes spécifiques dans la conception de logiciels, tout en suivant de bonnes pratiques.
- Ils offrent des approches structurées pour rendre les systèmes plus maintenables, réutilisables et évolutifs.

Patrons de conception

Caractéristiques des patterns de conception

- **Réutilisabilité :** Les patterns de conception sont génériques et peuvent être réutilisés dans différents contextes et projets.
- **Lisibilité :** Ils permettent de créer du code plus lisible et compréhensible, car ils répondent à des problèmes bien connus.
- **Faible couplage :** Les patterns favorisent des structures logicielles où les composants sont faiblement couplés, facilitant ainsi la maintenance et l'extension du code.

Patrons de conception

Classification des patterns de conception

1. **Patterns de création :** Ils concernent la manière dont les objets sont créés. Ils cherchent à abstraire ou à contrôler le processus d'instanciation.

- Exemples : Singleton, Factory, Builder, Prototype.

2. **Patterns structurels :** Ils définissent la manière dont les classes et objets sont structurés pour former des ensembles plus complexes. Ils se concentrent sur la composition des objets et des classes.

- Exemples : Adapter, Facade, Composite, Proxy.

3. **Patterns comportementaux :** Ils concernent la communication entre les objets et les classes. Ils facilitent les interactions et la gestion du flux de travail.

- Exemples : Observer, Command, Strategy, State.

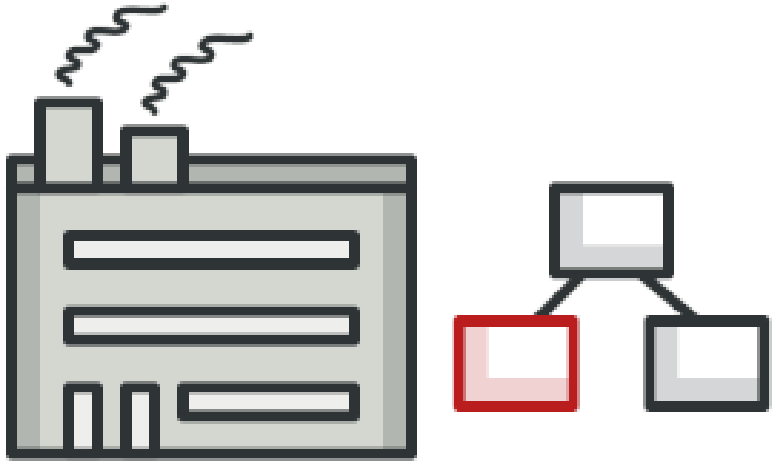
Patrons de conception

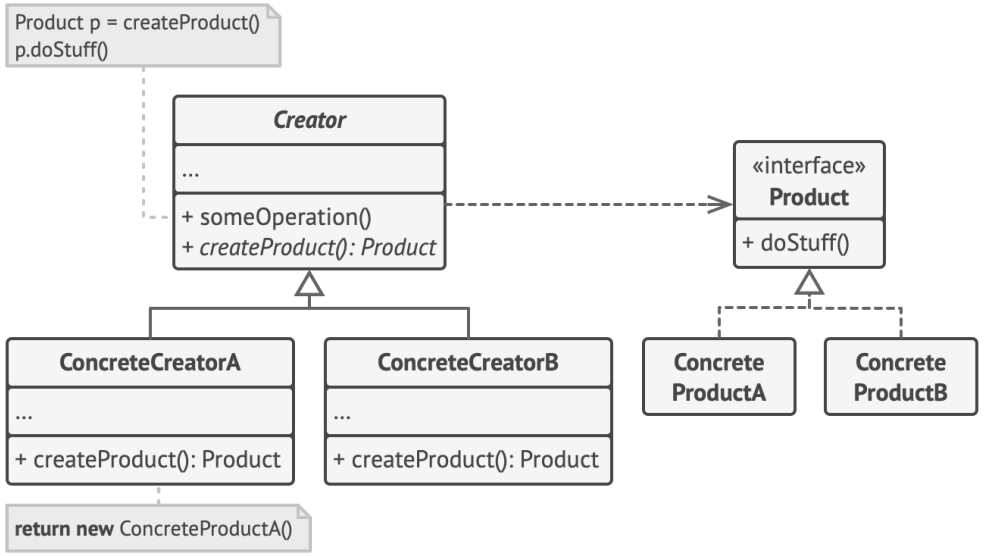
Pourquoi utiliser les patterns de conception ?

- **Réduction de la complexité :** Les patterns apportent des solutions simplifiées à des problèmes complexes.
- **Maintenabilité :** En adoptant des patterns, il devient plus facile de modifier et d'étendre le système à l'avenir.
- **Meilleure communication :** Les patterns sont largement connus et reconnus, ce qui facilite la compréhension et la collaboration entre les développeurs.
- **Flexibilité et évolutivité :** Ils permettent de concevoir des systèmes qui sont plus flexibles face aux changements de spécifications.

Patrons de conception

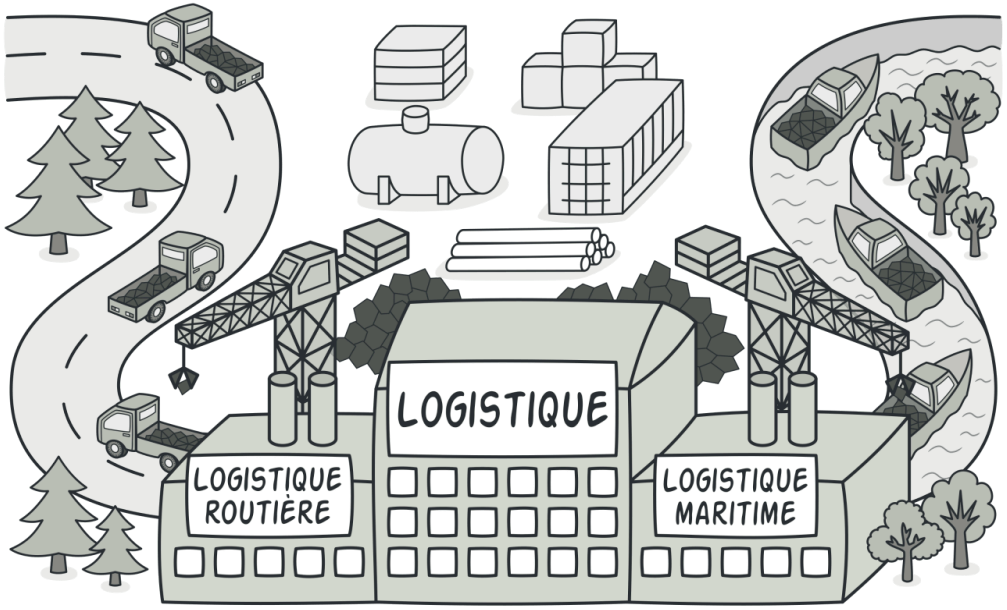
Patterns de création : Factory

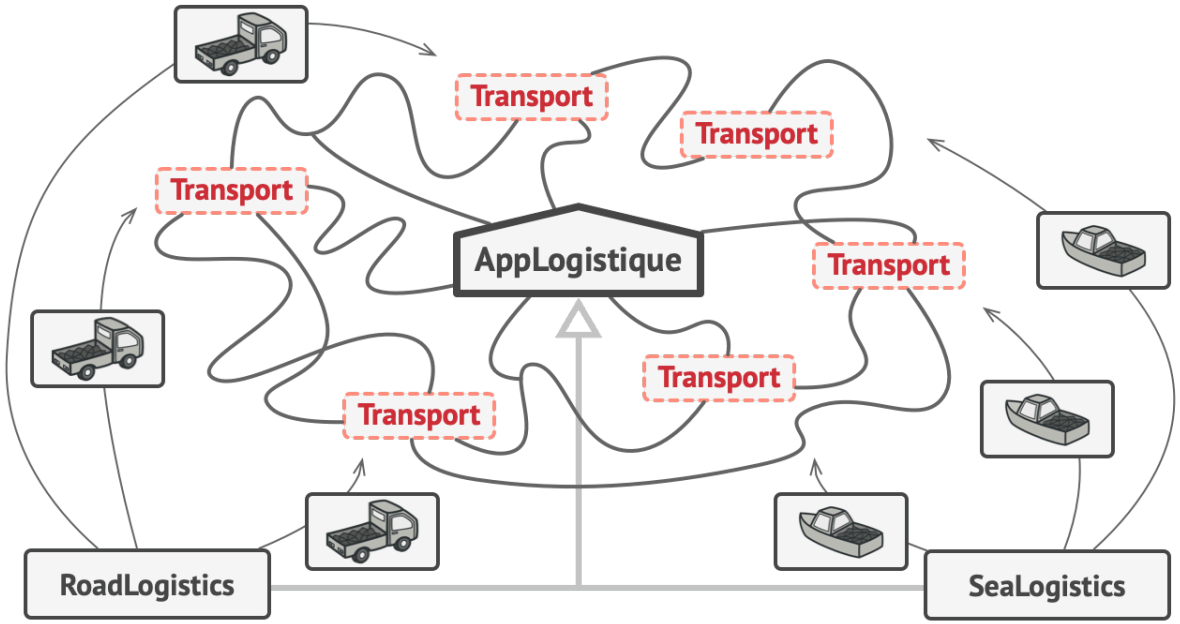




Patrons de conception

Patterns de création : Factory





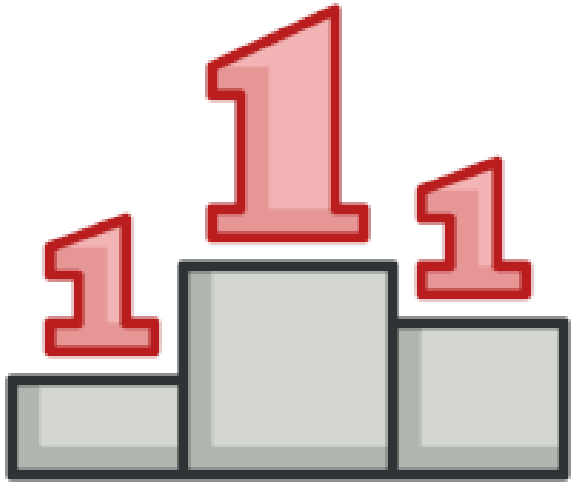
Patrons de conception

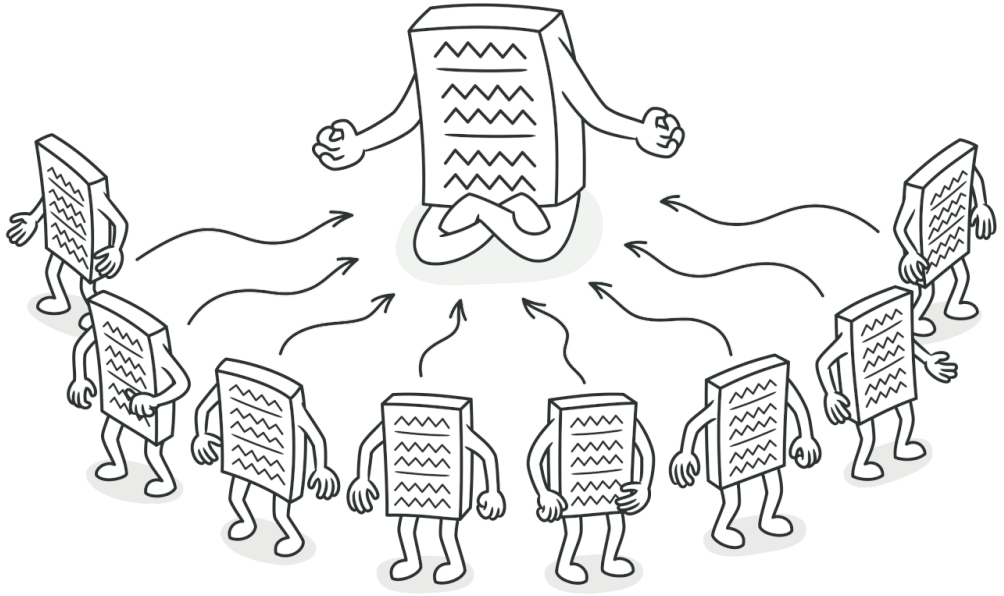
Patterns de création : Factory

- **Définition :** Le pattern Factory est utilisé pour créer des objets sans exposer la logique de création au client et en renvoyant des instances de classes appropriées.
- **Exemple :** Factory est souvent utilisé dans des frameworks où des objets de différentes sous-classes doivent être créés en fonction de certaines conditions (par exemple, création d'objets en fonction du type de produit).
- **Avantages :** Encapsule la logique de création, permet de respecter le principe d’ouverture/fermeture (OCP).
- **Inconvénients :** Complexité accrue si la hiérarchie de classes devient large.

Patrons de conception

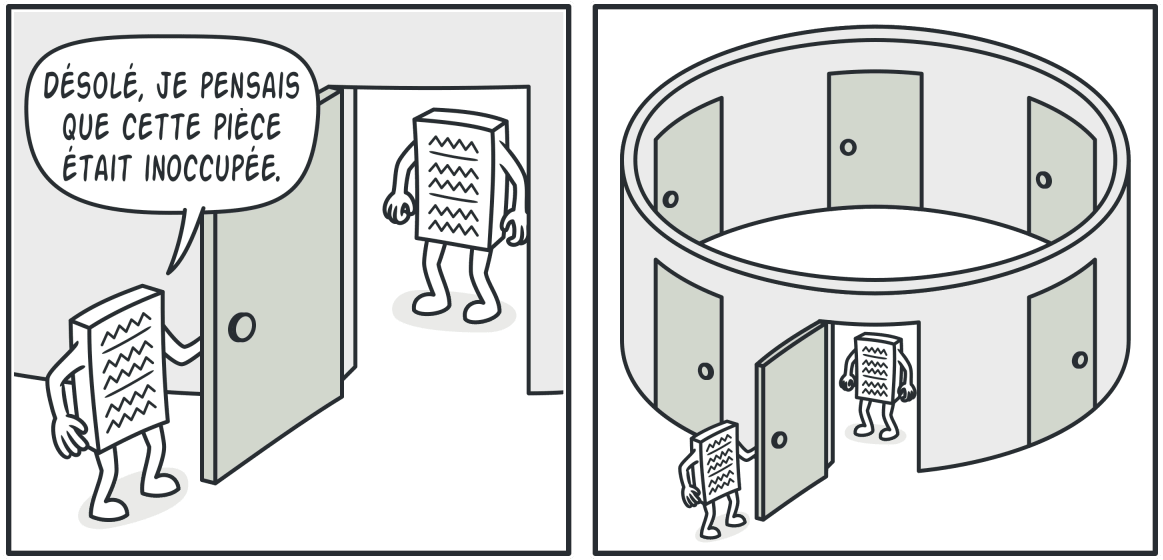
Patterns de création : Singleton

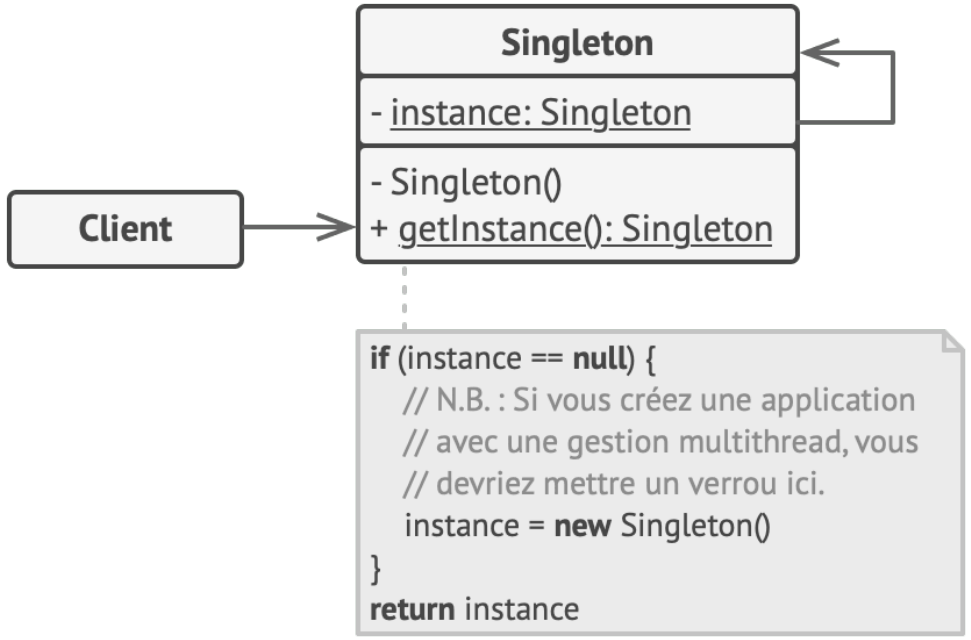




Patrons de conception

Patterns de création : Singleton





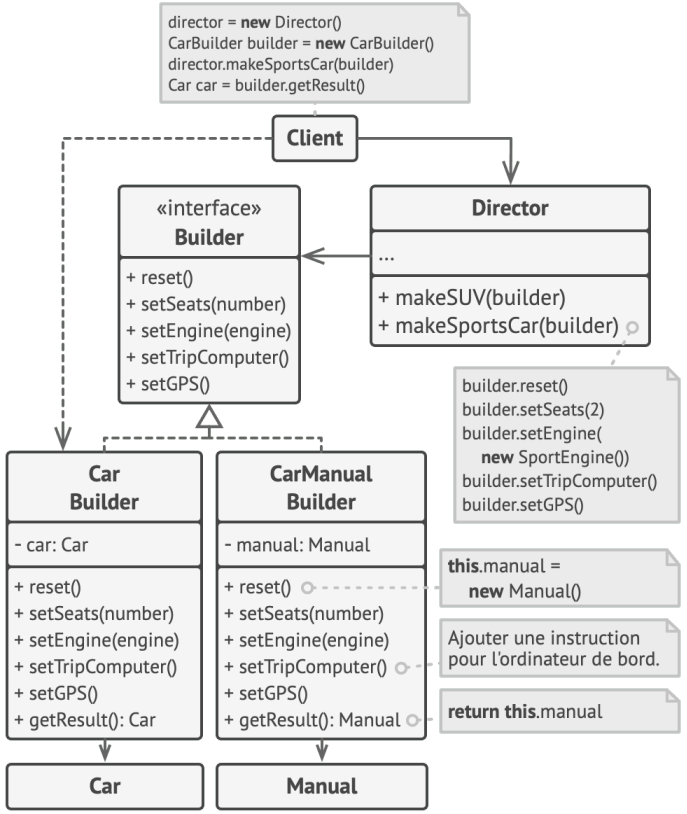
Patrons de conception

Patterns de création : Singleton

- **Définition :** Le pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.
- **Exemple :** Utilisé pour la gestion des ressources partagées telles que les connexions aux bases de données, les fichiers de configuration ou les instances de services.
- **Avantages :** Centralisation de l'accès à une ressource partagée, réduction de la consommation de ressources.
- **Inconvénients :** Peut entraîner un couplage fort et rendre les tests plus complexes.

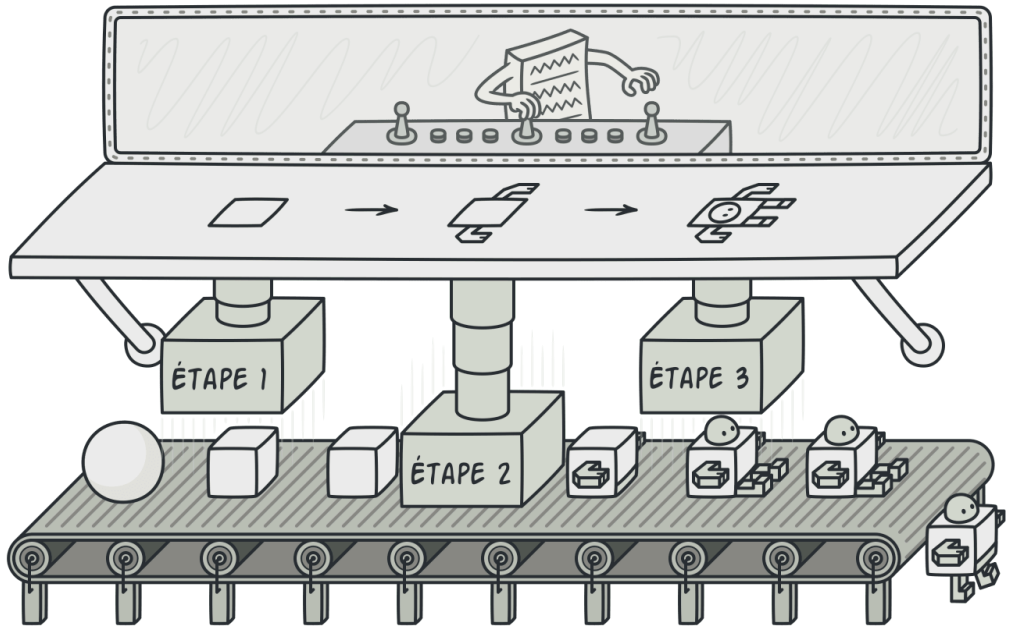
Patrons de conception

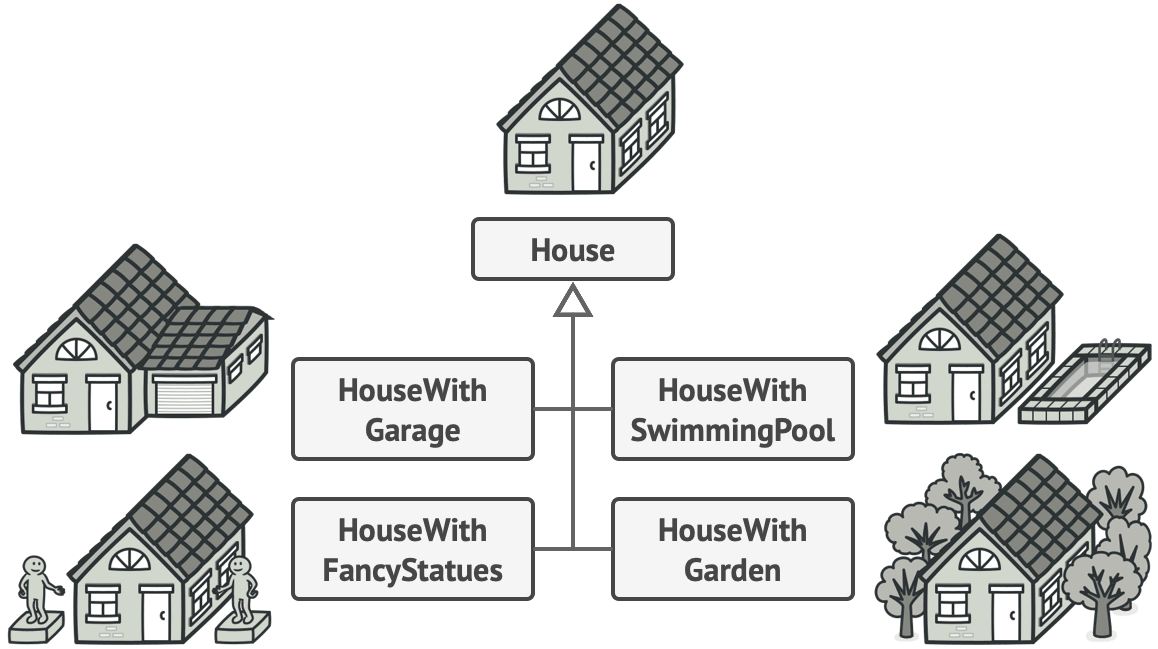
Patterns de création : Builder



Patrons de conception

Patterns de création : Builder





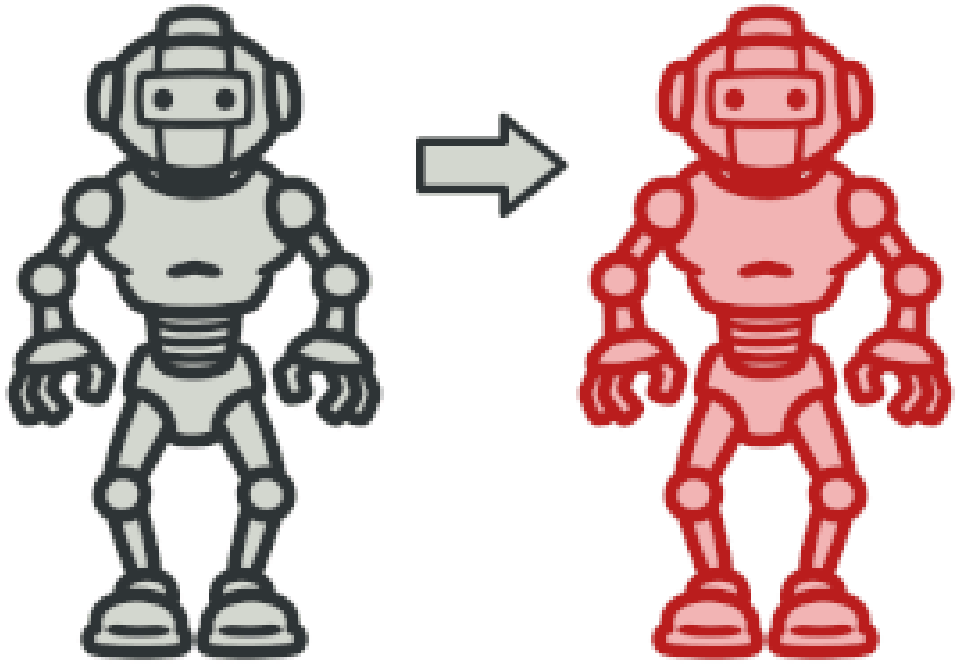
Patrons de conception

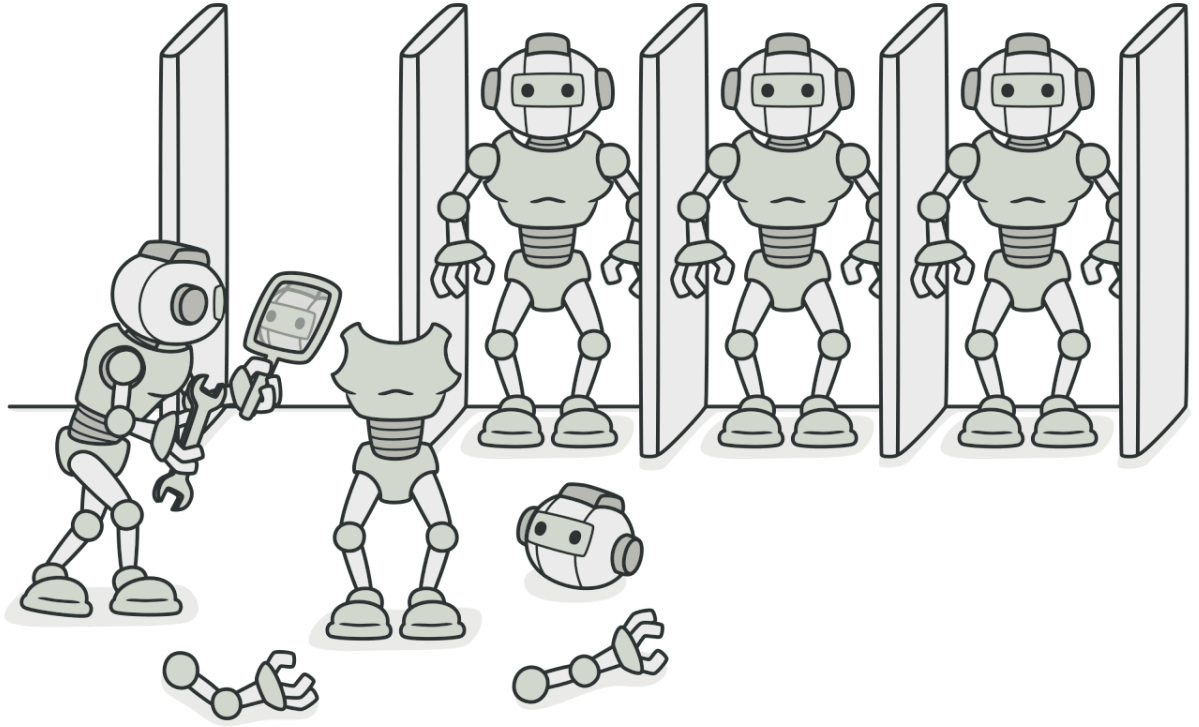
Patterns de création : Builder

- **Définition :** Le pattern **Builder** sépare la construction d'un objet complexe de sa représentation, permettant ainsi de créer différents types d'objets avec la même construction. Il est particulièrement utile pour les objets avec beaucoup de paramètres ou des processus de construction complexes.
- **Exemple :** Utilisé pour créer des objets avec plusieurs options ou étapes, comme la construction d'un objet `House` où vous pouvez spécifier le nombre de chambres, la couleur des murs, le type de toiture, etc.
- **Avantages :** Améliore la lisibilité du code lorsque l'objet a beaucoup d'options, permet une création flexible et une meilleure gestion des objets immuables.
- **Inconvénients :** Peut ajouter de la complexité et nécessite la création d'une classe Builder supplémentaire.

Patrons de conception

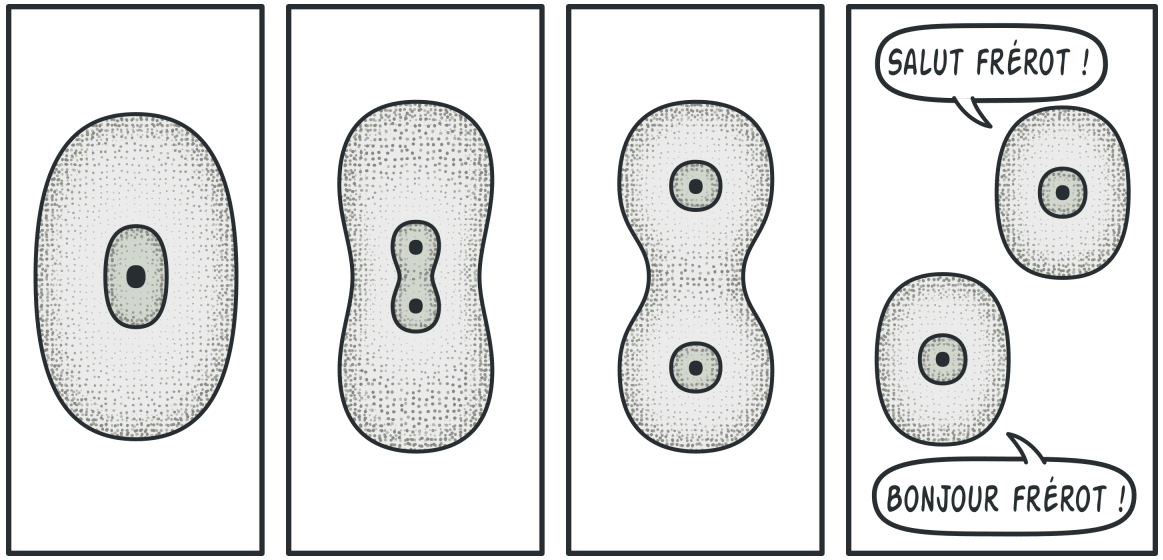
Patterns de création : Prototype

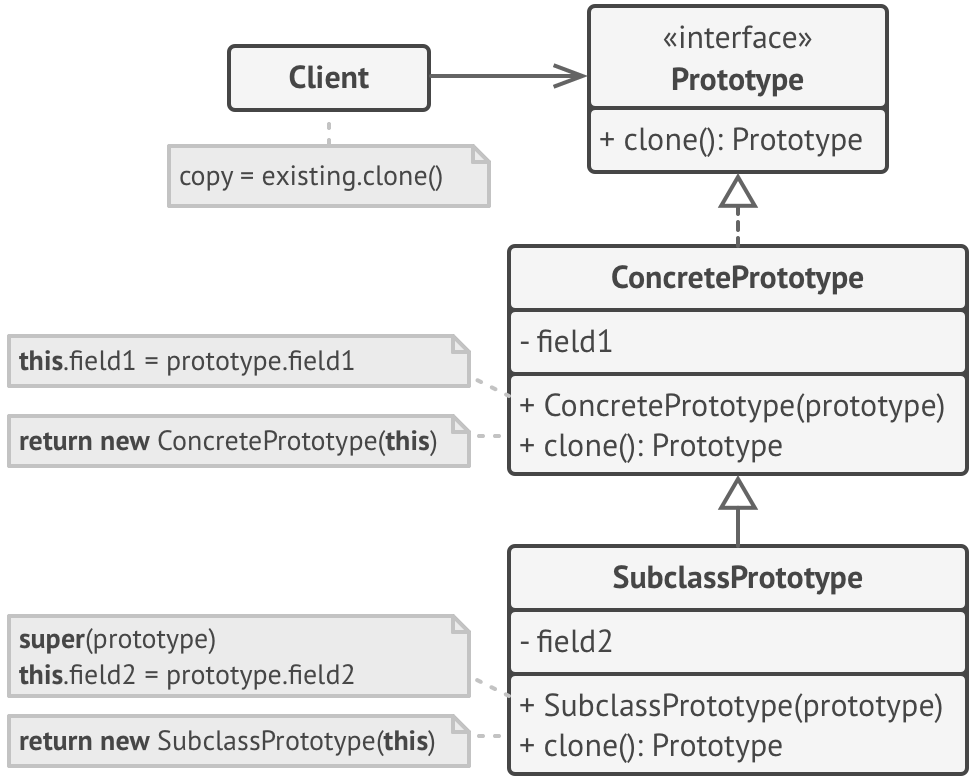




Patrons de conception

Patterns de création : Prototype





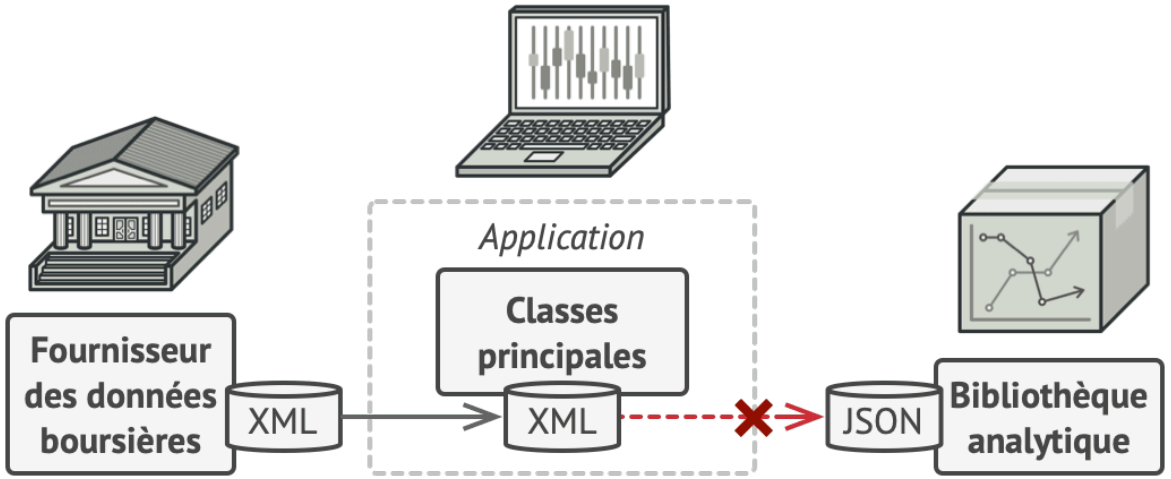
Patrons de conception

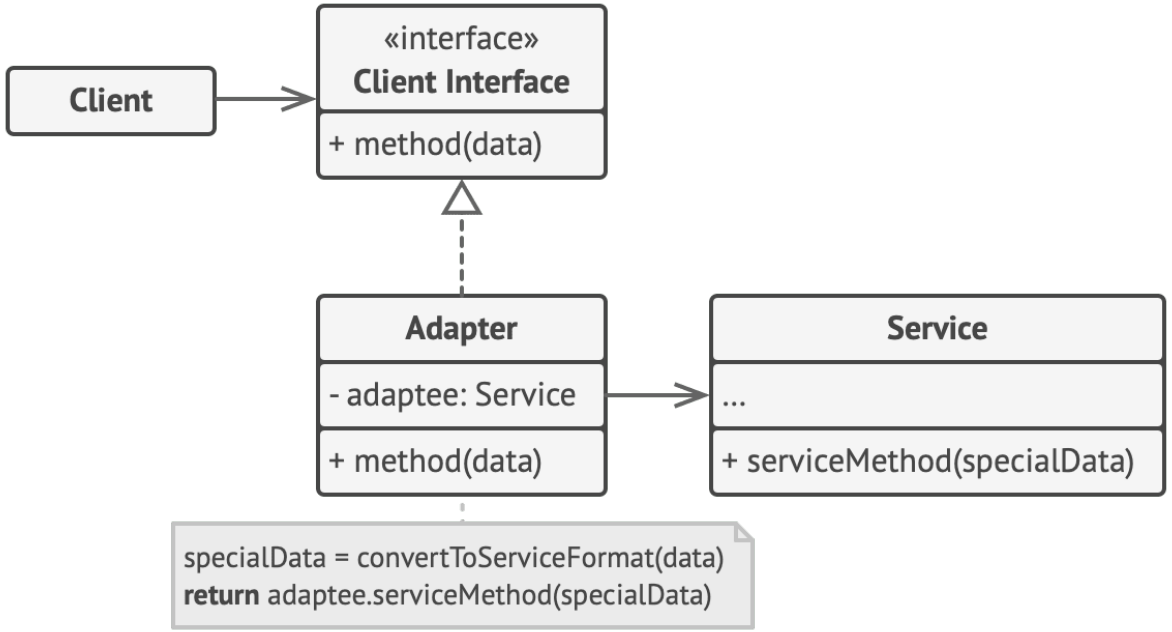
Prototype

- **Définition :** Le pattern **Prototype** est utilisé pour créer de nouveaux objets en copiant (clonant) un prototype existant, plutôt qu'en instanciant une nouvelle classe. Cela permet d'éviter les coûts de création d'un objet "à partir de zéro" et peut être utile lorsque la création d'objets est coûteuse.
- **Exemple :** Utilisé lorsqu'il est nécessaire de créer plusieurs objets similaires mais dont l'initialisation complète est coûteuse, comme dans des systèmes graphiques où des objets complexes doivent être dupliqués (par exemple, des formes ou des graphismes).
- **Avantages :** Permet de créer rapidement des copies d'objets sans devoir passer par des constructeurs lourds. Réduit le couplage entre les classes créatrices et les classes instanciées.

Patrons de conception

Patterns structurels : Adapter





Patrons de conception

Patterns structurels : Bridge

- **Définition :** Le pattern Bridge est utilisé pour séparer l'abstraction de son implémentation afin qu'elles puissent évoluer indépendamment. Il crée une interface qui agit comme un pont pour relier les deux.
- **Exemple :** Dans une application graphique, vous pourriez avoir des formes (abstraction) comme des cercles et des rectangles, et différentes plateformes graphiques (implémentations) comme OpenGL ou DirectX.
- **Avantages :** Découplage entre abstraction et implémentation, permettant des évolutions indépendantes. Meilleure extensibilité grâce à une hiérarchie claire. Réduction de la duplication de code lorsque plusieurs combinaisons abstraction/implémentation sont nécessaires.
- **Inconvénients :** Complexité accrue due à l'ajout de plusieurs couches d'abstraction. Peut être surdimensionné pour des systèmes simples avec des relations fixes entre abstraction et implémentation.

Patrons de conception

Patterns structurels : Facade

- **Définition :** Le pattern Facade fournit une interface simplifiée à un ensemble de classes ou à un sous-système complexe.
- **Exemple :** Utilisé dans des bibliothèques ou frameworks pour offrir une API simple tout en cachant la complexité sous-jacente.
- **Avantages :** Réduction de la complexité pour le client, améliore la lisibilité du code.
- **Inconvénients :** Peut cacher des détails importants, risque de sur-simplification.

Patrons de conception

Patterns structurels : Composite

- **Définition :** Le pattern Composite permet de structurer les objets sous forme d'arbres, où chaque nœud peut être une feuille ou un autre composite.
- **Exemple :** Utilisé pour représenter des hiérarchies comme les systèmes de fichiers (dossiers et fichiers) ou des arbres de composants graphiques.
- **Avantages :** Permet de traiter les objets individuels et composites de la même manière.
- **Inconvénients :** Peut rendre la conception plus complexe si mal utilisé.

Patrons de conception

Patterns comportementaux : Observer

- **Définition :** Le pattern Observer définit une relation de type un-à-plusieurs entre des objets, de sorte que lorsque l'état d'un objet change, tous ses observateurs en sont informés et mis à jour automatiquement.
- **Exemple :** Utilisé dans des systèmes où plusieurs objets dépendent d'un objet central, comme les interfaces graphiques, les événements UI ou les systèmes de notification.
- **Avantages :** Faible couplage entre les objets, extensibilité facile.
- **Inconvénients :** Peut entraîner une propagation excessive de notifications si mal implémenté.

Patrons de conception

Patterns comportementaux : Command

- **Définition :** Le pattern Command transforme une requête en un objet autonome, permettant de paramétrer les clients avec des requêtes, files d'attente ou journaux de requêtes.
- **Exemple :** Utilisé pour implémenter des actions réversibles comme dans un système de gestion d'historique (Undo/Redo), ou pour envoyer des requêtes à des services distants.
- **Avantages :** Séparation claire entre l'émetteur et l'exécuteur de la commande, flexibilité et réutilisabilité.
- **Inconvénients :** Peut introduire de la complexité supplémentaire avec la gestion des objets Command.

Patrons de conception

Patterns comportementaux : Strategy

- **Définition :** Le pattern Strategy permet de définir une famille d'algorithmes, de les encapsuler dans des classes séparées, et de les rendre interchangeables à l'exécution, selon le comportement souhaité.
- **Exemple :** Utilisé dans des systèmes où plusieurs approches sont possibles pour accomplir une tâche, comme les calculs de frais de livraison (par poids, par distance, ou forfaitaire) ou les algorithmes de tri (bulle, rapide, fusion).
- **Avantages :** Favorise l'extensibilité et respecte le principe "Ouvert/Fermé" (Open/Closed Principle). Permet de changer facilement d'algorithme sans modifier le contexte.
- **Inconvénients :** Peut complexifier le code en introduisant plusieurs classes pour chaque stratégie, rendant la gestion de ces classes plus difficile.

Patrons de conception

Patterns comportementaux : Médiateur

- **Définition :** Le pattern Médiateur (Mediator) permet de centraliser la communication entre plusieurs objets dans un système. Au lieu que les objets se communiquent directement entre eux, ils passent par un médiateur, ce qui réduit leurs dépendances et simplifie leur gestion.
- **Exemple :** Utilisé dans des systèmes comme une **chatroom** où plusieurs utilisateurs échangent des messages. Plutôt que chaque utilisateur communique avec tous les autres directement, ils utilisent un médiateur (chatroom) pour envoyer et recevoir des messages.
- **Avantages :**
  - **Réduction des dépendances directes** entre les objets participants.
  - Facilite la maintenance en centralisant la logique d'interaction dans un composant unique.
  - Améliore la lisibilité du système en évitant les relations enchevêtrées.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Architectures Distribuées et Microservices

Architectures Distribuées et Microservices

Introduction aux Microservices

- Les microservices sont une approche d'architecture logicielle qui consiste à décomposer une application en plusieurs services autonomes, chacun étant responsable d'une fonctionnalité spécifique.
- Ces services sont indépendants, peuvent être développés, déployés, et maintenus de manière autonome, ce qui permet une grande flexibilité et une évolutivité accrue.

Architectures Distribuées et Microservices

Introduction aux Microservices

Avantages des Microservices

1. **Indépendance des services**
   Chaque microservice est indépendant, ce qui permet aux équipes de développement de travailler sur différents services en parallèle sans interférer les uns avec les autres. Cela facilite le déploiement de nouvelles fonctionnalités et la correction de bugs sans perturber l'ensemble de l'application.
2. **Scalabilité**
   Les microservices peuvent être scalés individuellement, ce qui permet de mieux gérer la charge en fonction des besoins spécifiques de chaque service. Par exemple, si un service de gestion des utilisateurs reçoit plus de trafic, il peut être scalé indépendamment des autres services.

Architectures Distribuées et Microservices

Introduction aux Microservices

Avantages des Microservices

3. **Technologies hétérogènes**
   Dans une architecture microservices, chaque service peut être développé avec des technologies différentes. Par exemple, un service pourrait être écrit en Java, un autre en Node.js, et un autre en Python, selon les besoins spécifiques du service. Cela permet une plus grande flexibilité technologique.

4. **Déploiement rapide et continu**
   Les microservices facilitent les déploiements continus et rapides, car chaque service peut être mis à jour indépendamment des autres. Cela favorise l'adoption de pratiques comme l'intégration continue et le déploiement continu (CI/CD).

Architectures Distribuées et Microservices

Introduction aux Microservices

Avantages des Microservices

5. **Résilience et tolérance aux pannes**
   L'isolement des services permet d'éviter que la défaillance d'un service n'affecte l'ensemble de l'application. Par exemple, si un microservice tombe en panne, les autres services peuvent continuer à fonctionner normalement.
6. **Maintenance simplifiée**
   Les microservices étant plus petits et indépendants, il est plus facile de les comprendre et de les maintenir. Les équipes peuvent se concentrer sur des parties spécifiques du système sans être gênées par la complexité globale.

Architectures Distribuées et Microservices

Introduction aux Microservices

Inconvénients des Microservices

1. **Complexité de gestion**
   Bien que les microservices apportent des avantages en termes de découplage, ils introduisent une complexité accrue dans la gestion de l'architecture. Il est nécessaire de gérer un grand nombre de services indépendants, ce qui peut devenir difficile sans des outils d'orchestration et de gestion appropriés.
2. **Communication inter-services**
   Les microservices communiquent entre eux via des API, ce qui implique des appels réseau qui peuvent introduire de la latence et des problèmes de performance. De plus, la gestion de la communication entre plusieurs services nécessite l'utilisation de technologies comme les files d'attente de messages, les brokers, ou des API Gateway.

Architectures Distribuées et Microservices

Introduction aux Microservices

Inconvénients des Microservices

3. **Consistance des données**
   Chaque microservice ayant sa propre base de données, maintenir la consistance des données entre plusieurs services peut devenir un défi. Les transactions distribuées et les stratégies de gestion des données (comme la base de données eventuelle) sont souvent nécessaires pour gérer cela.

4. **Surveillance et dépannage**
   La gestion des logs, la surveillance des performances et le dépannage peuvent être plus difficiles dans une architecture microservices en raison de la dispersion des services à travers plusieurs serveurs ou containers. Un système de monitoring centralisé et des outils de traçage sont nécessaires pour assurer la visibilité sur l'ensemble des services.

Architectures Distribuées et Microservices

Introduction aux Microservices

Inconvénients des Microservices

5. **Overhead opérationnel**
   Le déploiement et la gestion de multiples services peuvent entraîner un overhead opérationnel significatif, notamment en matière de déploiement, de gestion des containers, de mise en réseau, et de mise à l'échelle.
6. **Surcharge de développement**
   Développer des microservices implique de gérer plus de complexité en termes de gestion de la configuration, de la sécurité, de l'intégration continue et des tests. Cela peut entraîner un coût de développement plus élevé pour les petites équipes ou les applications simples.

Architectures Distribuées et Microservices

Introduction aux Microservices

Comparaison avec les Architectures Monolithiques

1. **Architecture Monolithique**
   Une architecture monolithique est une approche où l'ensemble de l'application est développée et déployée en tant qu'unité unique. Tous les composants de l'application sont intégrés dans une seule base de code, et l'application entière doit être déployée et mise à jour en une seule fois.

   Avantages :

   - Simplicité de développement et de déploiement pour les petites applications.
   - Moins de complexité en termes de gestion des dépendances et de communication inter-composants.
   - Moins de besoins en infrastructure complexe.

   Inconvénients :

   - Difficile à faire évoluer et à maintenir à mesure que l'application devient plus grande.
   - Le déploiement de nouvelles fonctionnalités nécessite souvent de redéployer toute l'application.
   - La scalabilité est limitée à l'application dans son ensemble, ce qui peut poser des problèmes de performance pour les applications avec une forte charge.

Architectures Distribuées et Microservices

Caractéristique	Microservices	Monolithique
Modularité	Haute, services indépendants	Faible, tout est dans une seule unité
Scalabilité	Chaque service peut être scalé indépendamment	L'application entière doit être scalée
Complexité	Plus élevée (gestion des services, communication inter-services)	Moins élevée (tout est intégré)
Déploiement	Déploiement indépendant des services	Déploiement de l'application complète
Technologies	Possibilité d'utiliser plusieurs technologies	Une seule technologie pour l'ensemble
Performance	Peut être affectée par la latence réseau	Moins de latence, tout est sur le même serveur

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

Une **architecture microservices** repose sur plusieurs composants clés qui assurent la modularité, l'indépendance des services, et la communication entre eux.

1. Microservices

Rôle :

- Unités autonomes responsables d'une fonctionnalité métier spécifique.
- Chaque microservice peut être déployé, testé et mis à jour indépendamment.
- Possède souvent sa propre base de données (pattern "Database per service").
- Communique avec d'autres services via des API ou des messages.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

2. API Gateway

Rôle :

- Point d'entrée unique pour toutes les requêtes des clients vers les microservices.
- Route les requêtes vers les services appropriés.
- Offre des fonctionnalités transversales :
  - Authentification et autorisation.
  - Limitation de débit (Rate Limiting).
  - Agrégation des réponses provenant de plusieurs services.
  - Transformation des requêtes ou réponses.

Exemple d'outils :

- Kong, NGINX, Spring Cloud Gateway, AWS API Gateway.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

3. Service Discovery

Rôle :

- Permet aux microservices de localiser dynamiquement d'autres services sans dépendre de leurs adresses IP ou ports statiques.
- Assure l'enregistrement des services lorsqu'ils démarrent et leur désinscription lorsqu'ils s'arrêtent.

Exemple d'outils :

- Netflix Eureka, Consul, Zookeeper.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

4. Config Server

Rôle :

- Fournit une source centralisée pour la gestion des configurations des microservices.
- Permet de modifier les configurations dynamiquement sans redémarrer les services.

Exemple d'outils :

- Spring Cloud Config, HashiCorp Consul.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

5. Communication Interservices

Rôle :

- Assure la communication entre les microservices.
- Peut être synchrone (via HTTP ou gRPC) ou asynchrone (via des message brokers).

Exemple d'outils :

- Synchrone : REST, gRPC.
- Asynchrone : RabbitMQ, Kafka, AWS SQS.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

6. Message Broker

Rôle :

- Facilite la communication asynchrone en permettant l'échange de messages ou d'événements entre microservices.
- Découple les services et améliore la résilience du système.

Exemple d'outils :

- RabbitMQ, Apache Kafka, ActiveMQ.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

7. Load Balancer

Rôle :

- Répartit les requêtes entre plusieurs instances d'un microservice pour équilibrer la charge.
- Augmente la disponibilité et réduit les risques de surcharge.

Exemple d'outils :

- HAProxy, AWS Elastic Load Balancer, NGINX.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

8. Monitoring et Observabilité

Rôle :

- Surveille la santé et les performances des microservices.
- Fournit une vue centralisée des métriques et des logs.
- Permet le suivi des requêtes entre microservices (Distributed Tracing).

Exemple d'outils :

- Monitoring : Prometheus, Grafana.
- Tracing : Jaeger, Zipkin.
- Logging : ELK Stack (Elasticsearch, Logstash, Kibana), Fluentd.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

9. Circuit Breaker

Rôle :

- Prévient les effets de cascade en cas de panne d'un service.
- Permet de limiter les appels vers des services dégradés ou non disponibles.

Exemple d'outils :

- Resilience4j, Netflix Hystrix.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

10. Service Mesh

Rôle :

- Simplifie la communication, la sécurité et la gestion des microservices.
- Offre des fonctionnalités comme le routage, la sécurité mTLS (mutual TLS), et l'observabilité.

Exemple d'outils :

- Istio, Linkerd, Consul Service Mesh.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

11. Distributed Caching

Rôle :

- Stocke temporairement des données fréquemment utilisées pour améliorer les temps d'accès.
- Réduit la charge sur les bases de données et les microservices.

Exemple d'outils :

- Redis, Memcached.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

12. Authentication et Authorization

Rôle :

- Gère la sécurité des microservices.
- Implémente des protocoles comme OAuth2 et OpenID Connect pour l'authentification.
- Gère les rôles et permissions des utilisateurs.

Exemple d'outils :

- Keycloak, Auth0, Okta.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

13. CI/CD Pipeline

Rôle :

- Automatisation de la construction, des tests et du déploiement des microservices.
- Facilite les déploiements fréquents et sûrs.

Exemple d'outils :

- Jenkins, GitLab CI/CD, CircleCI.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

14. Bases de Données

Rôle :

- Chaque microservice possède sa propre base de données, souvent adaptée à ses besoins spécifiques.
- Peut être relationnelle ou NoSQL.

Exemple d'outils :

- SQL : PostgreSQL, MySQL.
- NoSQL : MongoDB, Cassandra.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

15. Fault Tolerance et Résilience

Rôle :

- Implémente des mécanismes pour rendre le système tolérant aux pannes :
  - Retry (réessais des requêtes échouées).
  - Timeout (limiter les temps d'attente).
  - Bulkhead (isoler les services pour éviter la propagation des pannes).

Exemple d'outils :

- Resilience4j.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

16. Centralized Logging

Rôle :

- Agrège les logs provenant de différents microservices pour faciliter le suivi des erreurs et le débogage.

Exemple d'outils :

- ELK Stack, Loki.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

17. Scheduler (Orchestration des Tâches)

Rôle :

- Planifie l'exécution des tâches ou des jobs asynchrones.

Exemple d'outils :

- Kubernetes CronJobs, Apache Airflow.

Architectures Distribuées et Microservices

Les composants d'une architecture microservice

17. Scheduler (Orchestration des Tâches)

Rôle :

- Planifie l'exécution des tâches ou des jobs asynchrones.

Exemple d'outils :

- Kubernetes CronJobs, Apache Airflow.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Communication entre Microservices

Communication entre Microservices

1. API REST

API REST (Representational State Transfer) est une approche standard pour les communications réseau dans une architecture microservices. Elle utilise les protocoles HTTP/HTTPS pour les opérations CRUD (Create, Read, Update, Delete) :

- **Légère** : Utilise le protocole HTTP standard pour la communication.
- **Sans état** : Chaque appel API est indépendant et contient toutes les informations nécessaires pour comprendre la requête.
- **Cacheable** : Les réponses peuvent être mises en cache pour améliorer la performance.
  Les microservices s'exposent en tant que ressources avec des URL uniques, où les actions sont contrôlées par des méthodes HTTP telles que GET, POST, PUT, et DELETE.

Communication entre Microservices

2. GraphQL

GraphQL est une alternative aux API REST qui permet aux clients de demander exactement les données dont ils ont besoin, rien de plus, rien de moins. Cela peut réduire la charge sur le réseau en évitant les réponses surchargées et en minimisant le nombre de requêtes :

- **Déclaration des besoins en données** : Le client spécifie précisément quelles données il souhaite récupérer.
- **Single Endpoint** : GraphQL utilise un seul endpoint, généralement `/graphql`, pour toutes les requêtes, ce qui simplifie la logistique de la communication entre services.

Communication entre Microservices

3. gRPC

gRPC est un framework de communication inter-services développé par Google, basé sur le protocole HTTP/2. Il est conçu pour être rapide et efficace, idéal pour les environnements de microservices où les performances et la faible latence sont cruciales :

- **Contract-first** : Utilise des fichiers de définition de service (protobuf) pour définir le schéma des données et des services.
- **Support des streaming** : gRPC prend en charge le streaming unidirectionnel et bidirectionnel, permettant une communication en temps réel plus efficace.

Communication entre Microservices

4. Event-Driven Architecture (EDA)

Dans une **architecture pilotée par les événements**, les services communiquent via des événements qui sont émis par une source et écoutés par un ou plusieurs services. Cela favorise une forte découplage entre les services :

- **Kafka** : Une plateforme de streaming distribuée qui permet la publication et la souscription à des streams de records. Très utile pour le traitement des événements en temps réel.
- **RabbitMQ** : Un broker de messages qui supporte plusieurs protocoles de messagerie, y compris AMQP. Il est utilisé pour faciliter la communication asynchrone dans un système distribué.