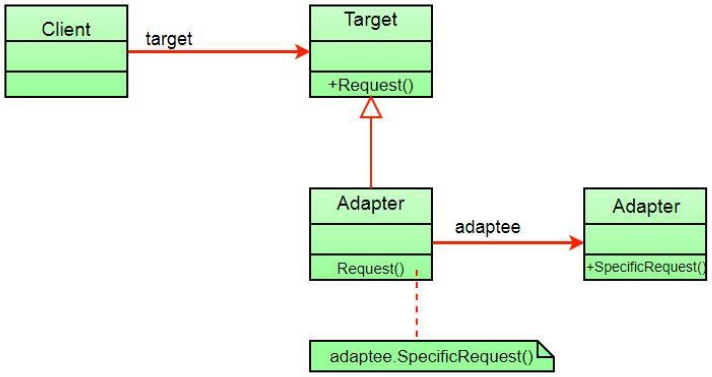
Rapport SF

Architecture Logicielle



Formateur : Duc Alain

Date : 10 novembre 2023, 13H00 à 16H00

Durée : Une demi-journée

Présents : David Guillaume, Dasek Joiakim, Cardoso Rafael, Laurent Térence, Uka Zotrim

Rapporteur : David Guillaume

Table des matières

Table des matières

Introduction

L'architecture logicielle revêt une importance essentielle dans l'organisation des projets de développement. Elle amène des outils afin de garantir son objectif principal qui est l'organisation et la structuration du code.

En effet, l'évolution des cas d'utilisation, l'évolution des technologies tels que le changement ou l'ajout de prise en charge de bases de données, ou tout autre évolution sur le code ... amènent ce-dernier à se complexifier. Il est donc essentiel de réaliser et d'intégrer une architecture afin de répondre à ces défis.

Architecture logicielle

- L'architecture logicielle doit être étudier en amont de la phase de développement.
- Elle structure et organise le code.
- Elle permet un langage commun entre l'équipe de développement par sa structuration.

La documentation

Une documentation d'architecture logicielle doit atteindre un juste milieu, évitant d'être trop simpliste, ce qui pourrait entraver la compréhension, tout en évitant d'être excessivement complexe, car cela la rendrait difficile à utiliser, non maintenable, voire inutile. Elle devrait être suffisamment détaillée pour fournir une compréhension approfondie sans perdre en accessibilité, favorisant ainsi une utilisation efficace et une maintenance facilitée.

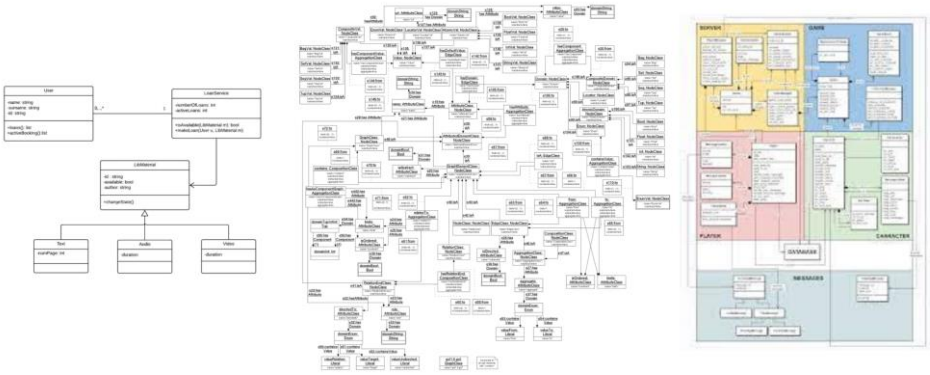


Figure 1 : Exemple de documentation d'architecture logicielle

Évolution des coûts

Il est très essentiel de se rappeler que plus les modifications sont près de la mise en production, plus cela est couteux.

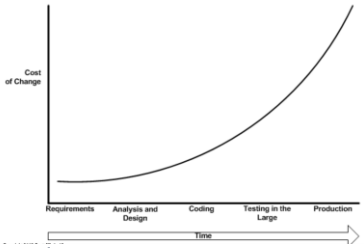


Figure 2 : Évolution des coûts / avancée du projet

→ L'architecture préliminaire évite ce genre de situation.

Principes

Une architecture logicielle bien conçue repose sur des principes à respecter.

Principes d'architecture

- **Couplage lâche (Loose coupling)**
  Ce principe vise à réduire les dépendances entre les composants d'un système. En minimisant les liens forts entre les modules, on favorise une plus grande flexibilité, permettant à chaque composant d'évoluer de manière indépendante.

- **Séparation des préoccupations (Separation of concern)**
  Ce principe consiste à organiser le système de manière à ce que chaque composant se concentre sur une tâche spécifique. Cela facilite la compréhension, la maintenance et l'évolution du code, en évitant les mélanges de fonctionnalités.

  → Par exemple, on retrouve dans ce principe la séparation entre la « vue », le « contrôleur » et le « modèle » du concept MVC.

- **Dissimulation de l'information (Information hiding)**
  Il s'agit de limiter l'accès direct aux détails internes d'un composant, ne rendant disponibles que les informations nécessaires à l'extérieur. Cela favorise l'encapsulation et réduit la complexité en permettant aux composants de fonctionner de manière autonome.

- **Structuration hiérarchique (Hierarchical structuring)**
  Ce principe implique une organisation en niveaux ou en couches, où chaque niveau a des responsabilités spécifiques. Cela facilite la compréhension globale du système et permet une gestion plus claire des différentes parties de l'architecture logicielle.

Principes SOLID

Les principes SOLID sont un ensemble de cinq directives de conception de logiciels orientées objet, formulées visant à créer des systèmes plus compréhensibles, flexibles et maintenables.

- **S (Single Responsibility Principle - Principe de Responsabilité Unique)**
  Un objet ou une classe ne devrait avoir qu'une seule raison de changer, c'est-à-dire qu'il devrait avoir une seule responsabilité dans le système.
- **O (Open/Closed Principle - Principe Ouvert/Fermé)**

Les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension mais fermées à la modification. On devrait pouvoir ajouter de nouvelles fonctionnalités sans modifier le code existant.

- **L (Liskov Substitution Principle - Principe de Substitution de Liskov)**
  Les objets d'une classe de base doivent pouvoir être remplacés par des objets de ses classes dérivées sans altérer la cohérence du programme. Cela garantit une utilisation cohérente des sous-classes.
- **I (Interface Segregation Principle - Principe de Ségrégation des Interfaces)**
  Il vaut mieux avoir plusieurs interfaces spécifiques qu'une seule interface générale. Cela évite d'imposer aux classes des méthodes qu'elles n'utiliseront pas.
- **D (Dependency Inversion Principle - Principe d'Inversion de Dépendance)**
  Les modules de haut niveau ne devraient pas dépendre des modules de bas niveau, mais plutôt des abstractions. Les détails concrets ne doivent pas être dépendants des détails abstraits, mais plutôt l'inverse. Cela favorise une conception plus souple et extensible.

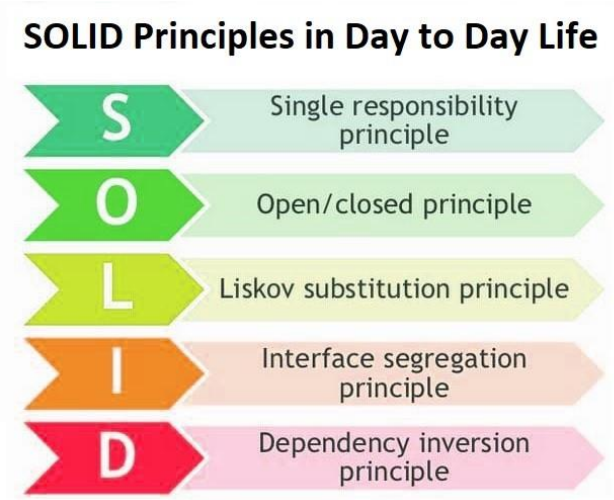


Figure 3 : S.O.L.I.D

Design Pattern

Les design patterns, ou modèle de conception, sont des solutions réutilisables à des problèmes courants rencontrés lors de la conception de logiciels. Ils offrent des modèles éprouvés pour résoudre efficacement des types spécifiques de problèmes dans le développement logiciel.

- Ils sont flexibles et adaptables
- L'objectif n'est pas d'implémenter tous les design patterns, ce qui pourrait être contreproductif, mais plutôt de les intégrer de manière à simplifier et structurer le code.

La modularisation

La modularisation vise à rendre le programme **compréhensible** pour tous les développeurs en facilitant la navigation. Elle consiste à organiser le code en modules regroupés dans des packages.

La mise en œuvre du principe de couplage lâche (loose coupling) garantit que les modifications n'affectent qu'un module spécifique, simplifiant ainsi la maintenance et l'évolution du logiciel.

En résumé, la modularisation améliore la lisibilité, la maintenance et la flexibilité du code en le structurant de manière logique et en favorisant l'indépendance des modules.

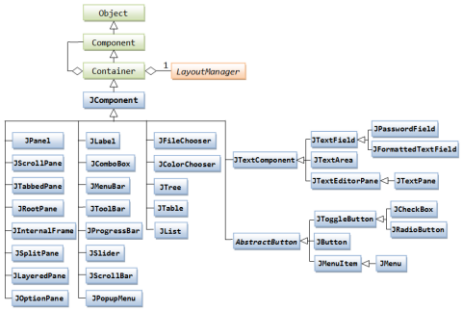


Figure 4 : Organisation des classes

Les types d'architecture

Il y a différents niveaux d'architecture tels que :

- Niveau matériel
- Niveau logiciel
- Niveau réseau (de plus en plus avec le Cloud)

Architecture logiciel

Les types d'architectures de développement comprennent :

1. **3 Tiers :**
    - **Présentation :** Gère l'interface utilisateur, par exemple, les boutons.
    - **Application :** Gère la logique métier, par exemple, la création de patients.
    - **Données :** Gère l'accès et la manipulation des données, par exemple, les opérations d'insertion.
2. **MVC (Modèle-Vue-Contrôleur) :**
    - **Modèle :** Gère la logique métier et les données.
    - **Contrôleur :** Gère les interactions utilisateur et orchestre les actions.
    - **Vue :** Gère l'interface utilisateur et l'affichage.
3. **N-Tiers :**
    - Architecture distribuée où les fonctionnalités sont réparties sur plusieurs niveaux (tiers).

- Les couches peuvent inclure la présentation, la logique métier, la gestion des données, etc.

Ces architectures offrent des approches différentes pour organiser et structurer les composants d'une application, en fonction des besoins spécifiques du projet.

L'utilisation d'interfaces est pratique car elle simplifie les tests unitaires.

Dans une implémentation stricte, les couches du système n'ont généralement accès qu'aux couches adjacentes, limitant ainsi la visibilité. Cependant, il peut y avoir des situations où un accès direct aux couches 1 à 3 est nécessaire.

L'architecture organisée tel que [Server / Logique] [Client] ou [Server] [Logique / Client] propose deux configurations possibles. Dans la première, la logique applicative est répartie entre le serveur et le client, tandis que dans la seconde, elle est centralisée soit du côté serveur, soit du côté client. Ce choix architectural influence la répartition des tâches et des responsabilités entre le serveur et le client dans le contexte d'une application. Il convient d'étudier la configuration en amont.

Architecture d'infrastructure

Les types d'architectures d'infrastructure comprennent :

1. **Client-Server :**
    - Les clients demandent des services et les serveurs fournissent ces services.
    - La communication se fait généralement via des requêtes et des réponses.
2. **Master-Slave :**
    - Un nœud principal (master) coordonne les activités des nœuds subordonnés (slaves).
    - Les slaves effectuent des tâches et rapportent au maître.
3. **Pipe-Filter (BizTalk) :**
    - Les composants (filtres) traitent les données à travers des canaux de communication (pipes).
    - Chaque filtre se concentre sur une tâche spécifique.
4. ...

Il est crucial de choisir la bonne architecture en fonction des besoins spécifiques de chaque projet. Chacune de ces architectures offre des avantages et des inconvénients, et la sélection dépend des exigences particulières de l'application et de l'infrastructure.

Les Web Services

Les services web sont essentiels de nos jours, et il est crucial de créer les interfaces de programmation (API). Deux approches courantes sont REST, qui utilise des objets JSON avec HTTP, et SOAP, qui repose sur un descripteur WSDL pour générer un client (nécessitant l'envoi du fichier de description).

1. **REST (Representational State Transfer) :**
    - Utilise des objets JSON pour représenter les données.
    - S'appuie sur le protocole HTTP pour les opérations (GET, POST, PUT, DELETE).
    - Favorise une architecture simple, légère et flexible.
    - Souvent utilisé pour des services web plus simples et orientés ressources.
2. **SOAP (Simple Object Access Protocol) :**
    - Repose sur un descripteur WSDL (XML) (Web Services Description Language) pour définir les opérations et les types de données.
    - Crée un client en utilisant ce descripteur, nécessitant l'envoi du fichier de description (WSDL).
    - Supporte divers protocoles sous-jacents, tels que HTTP, SMTP, et plus.
    - Utilisé dans des environnements où la formalité de la communication est primordiale, par exemple, dans les applications d'entreprise complexes.

Les services REST, utilisant JSON sur HTTP, sont appréciés pour leur simplicité, légèreté et flexibilité, adaptés aux services orientés ressources. En revanche, SOAP, basé sur WSDL, assure une formalité et une complexité nécessaires dans des environnements d'entreprise complexes. Cependant, il exige l'envoi du descripteur WSDL, ce qui peut ajouter une couche de complexité. Le choix entre REST et SOAP dépend ainsi des exigences spécifiques du projet, cherchant un équilibre entre simplicité et formalité en fonction du contexte d'utilisation.

- L'ancêtre du Webservice peut être compris comme étant RMI, RPC.
- La part de JSON augmente alors que celle d'XML diminue.

Les Micro-services

Les micro-services constituent une approche de développement où les services web sont subdivisés en entités autonomes, chacune étant généralement dotée de **sa propre base de données**. Cette décomposition favorise l'indépendance entre les services, car chaque micro-service est responsable d'une tâche spécifique, ce qui simplifie le développement, le déploiement et la maintenance.

Un aspect essentiel des micro-services est la séparation des données, chaque entité gérant sa propre base de données. Cette autonomie évite les conflits potentiels et facilite la gestion des données spécifiques à chaque service, contribuant ainsi à une architecture plus robuste.

Idéalement, les micro-services fonctionnent de manière autonome, sans communication directe entre eux hormis l'API. Cette autonomie vise à améliorer la scalabilité et les performances du système.

La conception précise d'une application basée sur des micro-services est cruciale. Il faut tenir compte de l'accès simultané à deux services pour garantir une interaction fluide et efficace. La gestion des erreurs, la sécurité et la cohérence des données sont des aspects critiques à considérer dans cette démarche.

Les micro-services offrent une solution agile et évolutive pour le développement logiciel, mettant l'accent sur l'indépendance, la scalabilité et la performance. Cependant, une conception réfléchie et précise est nécessaire pour assurer le succès de leur implémentation.

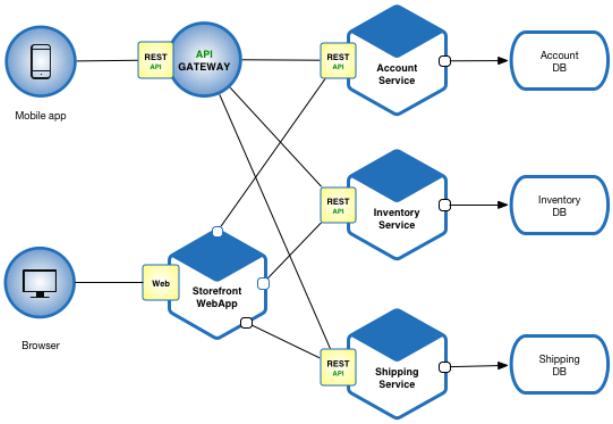


Figure 5 : Architecture en micro-services

Les micro-services et l'architecture orientée services (SOA) sont des approches architecturales similaires, mais présentent des nuances significatives. Les micro-services découpent une application en services autonomes, chacun ayant sa propre base de données, favorisant l'agilité et la scalabilité. En revanche, SOA organise les fonctionnalités en services, qui peuvent être moins autonomes et partager des ressources. Les micro-services, souvent considérés comme une évolution de la SOA, privilégient l'indépendance, la flexibilité et le déploiement indépendant, tandis que SOA peut avoir une portée plus large et une approche moins modulaire.

- Docker est très utilisé dans ce type d'architecture.
- L'intégration continue (CI) est une pratique de développement qui implique la fusion régulière du code développé par une équipe dans un référentiel partagé, accompagnée d'une automatisation des tests. Son objectif principal est de détecter rapidement les erreurs d'intégration et de résoudre les conflits, assurant ainsi une progression rapide du projet. Dans le contexte des architectures modernes, comme les micro-services, l'intégration continue est cruciale pour garantir la cohérence et la qualité du code à mesure que de petits modules indépendants sont fréquemment mis à jour. Elle accélère le déploiement, facilite la collaboration au sein de l'équipe de développement, et minimise les risques d'erreurs en identifiant précocement les problèmes potentiels.

Confrontation sur les projets

Le temps restant a été orienté sur la confrontation et discussion de l'architecture de projets.

- S’intéresser à InfluxDB, base de données optimisée pour valeurs temporelles en NoSQL.
- Architecture Flutter
    - Providers, repositories et services (la documentation est bien fournie)
    - L’utilisation d’un pattern Adapter est une bonne idée
    - .env pour la configuration MQTT
- Documentation de l’architecture
    - Documentation de haut niveau (Flutter, device, MQTT)
    - Zoomer en s’approchant

Bibliographie

Support de cours du formateur

https://media.geeksforgeeks.org/wp-content/uploads/classDiagram.jpg

https://media.licdn.com/dms/image/C5112AQFJTYZ05cR_Lw/article-cover_image-shrink_600_2000/0/1545715493806?e=2147483647&v=beta&t=SW9hQCcZVMdMNLw0fkIh5KsnwxIRtmISAPmE9Uu9NVA

https://www3.ntu.edu.sg/home/ehchua/programming/java/images/Swing_JComponentClassDiagram.png

https://microservices.io/i/Microservice_Architecture.png