All • in • IT

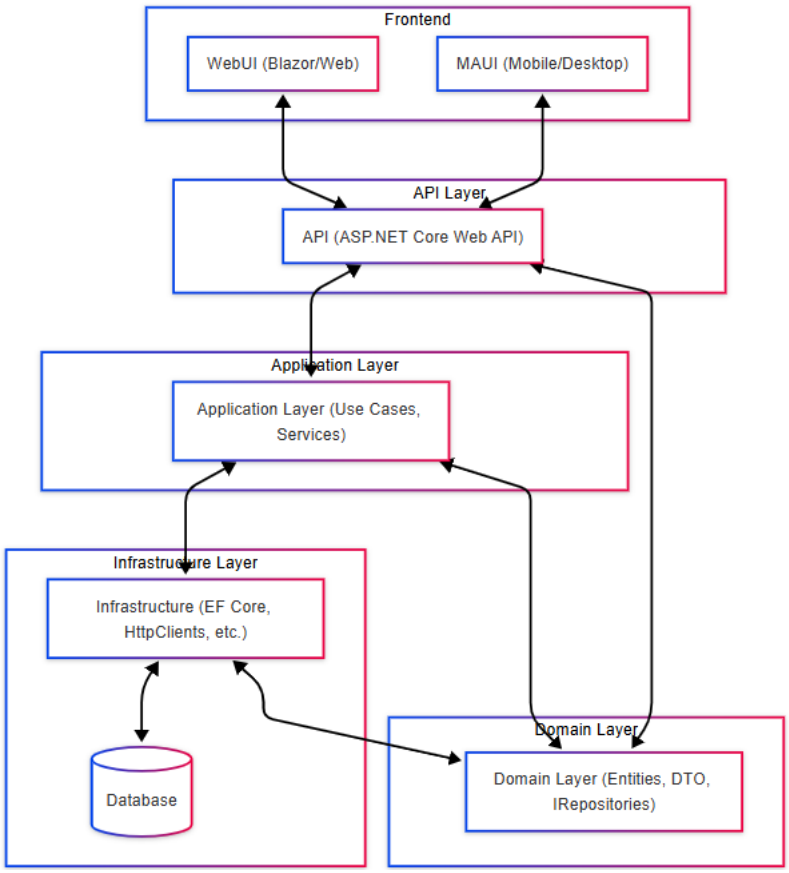
Dossier d'Architecture Logicielle

1. Introduction

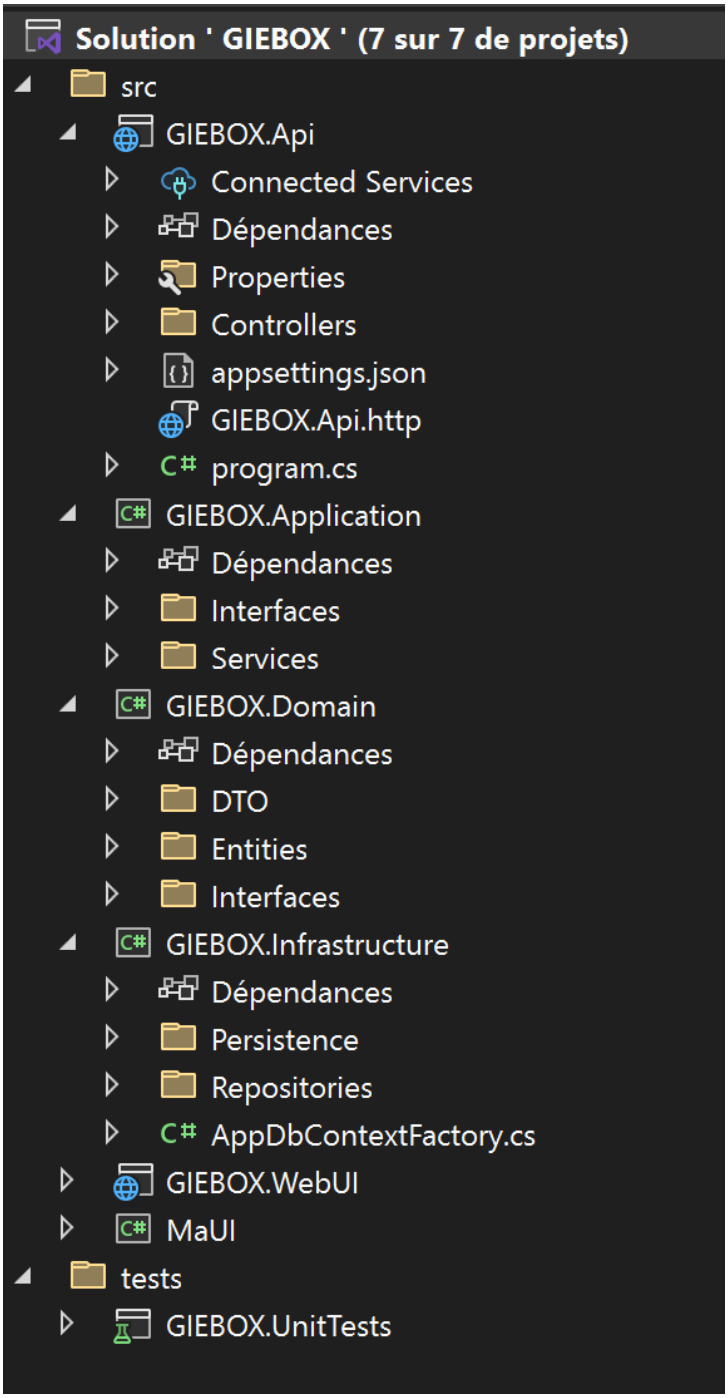
Ce document présente l'architecture technique de la solution logicielle nommée « GIEBOX ». Il décrit l'organisation des projets, les responsabilités de chaque couche, et les bonnes pratiques de conception suivies.

2. Organisation de la Solution

La solution Visual Studio « GIEBOX » est structurée selon une architecture en couches respectant les principes de séparation des responsabilités (Separation of Concerns) et de Clean Architecture.



3. Description des Projets



- **GIEBOX.Api** : Projet API REST qui expose les points d'entrée HTTP. Il contient les contrôleurs, le fichier de configuration `appsettings.json` et le point d'entrée `Program.cs`.

- **GIEBOX.Application** : Contient la *logique métier* de l'application. Cette couche définit les interfaces et implémente les services métiers.
- **GIEBOX.Domain** : Contient les entités, les objets de transfert de données (DTO) et les interfaces à l'intention de la couche Infrastructure.
- **GIEBOX.Infrastructure** : Gère la persistance des données. Contient les implémentations des interfaces du domaine, les accès à la base de données, et la classe `AppDbContextFactory`.
- **GIEBOX.WebUI** : Couche de présentation pour une interface web côté client : Blazor WebAssembly.
- **MaUI** : Couche de présentation pour une interface mobile / Androïd côté client : Maui.
- **tests** : Regroupe les projets de tests automatisés pour valider le comportement de l'application.

4. Bonnes Pratiques

- Respect du principe SOLID dans les couches Application et Domain.
- Usage d'interfaces pour l'injection de dépendances.
- Organisation claire en DTO / Services / Repositories.
- Séparation stricte des responsabilités entre les couches pour améliorer la maintenabilité et la testabilité.
- Présence d'un projet dédié aux tests pour garantir la qualité logicielle.

5. Rappel des Principes SOLID :

- S - Single Responsibility Principle (Responsabilité unique) :

Chaque classe ou module ne doit avoir qu'une seule responsabilité, c'est-à-dire une seule raison de changer. Cela permet une meilleure lisibilité, testabilité et maintenance du code.

- O - Open/Closed Principle (Ouvert/Fermé) :

Les entités logicielles doivent être ouvertes à l'extension, mais fermées à la modification. Cela signifie qu'on peut ajouter de nouveaux comportements sans modifier le code existant.

- L - Liskov Substitution Principle (Substitution de Liskov) :

Les objets d'une classe dérivée doivent pouvoir remplacer ceux de la classe de base sans altérer le comportement du programme.

- I - Interface Segregation Principle (Ségrégation des interfaces) :

Il vaut mieux plusieurs interfaces spécifiques qu'une interface générale. Une classe ne devrait pas être obligée d'implémenter des méthodes qu'elle n'utilise pas.

- D - Dependency Inversion Principle (Inversion des dépendances) :

Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais tous deux doivent dépendre d'abstractions. On privilégie les interfaces et l'injection de dépendances.

6. Mode de fonctionnement

- La version du framework DOT NET retenu pour ce projet est la version 8. Tout changement de version doit être validé par le chef de projet.
- Tout ajout de nugget ou autre framework doit être validé par le chef de projet.
- Pour autant que cela soit possible, les modifications sont à faire dans les répertoires "custom" quand ils existent.
- La logique métier est dans la couche Application et non dans la couche Infrastructure.
- Toute modification de structure la base de données (ajout de table, de champ ou suppression de table ou de champ) doit être validée par le chef de projet.

7. Stockage du code source

Pour la synchronisation des développements entre les intervenants et pour la sauvegarde du code source, ceux-ci seront stockés dans un environnement à distance, hébergé par ALL-IN-IT : git.services.

L'adresse de ce service : https://git.services.allin-it.fr/AIT/GIEBOX (accessible grâce à un login / password)

Il est donc primordial d'utiliser ce service pour sauvegarder et partager son travail.

Première étape (1 fois par projet) : clôner le dépôt distant sur son poste (via Visual-Studio par exemple). C'est à cette occasion que vous sera demandé vos identifiant et mot de passe. Vous aurez ainsi l'occasion d'avoir le code existant à jour sur votre machine de dev.

Pour tout ajout de nouvelle fonctionnalité (ou de correction d'un bug) : créer une branche (généralement depuis Master) et c'est sur cette branche que vos développements se feront.

Au minima 2 fois par jour, penser à **pousser** votre développement sur le dépôt. MAIS AVANT de pousser votre code modifié, ayez le réflexe de récupérer (**tirer**) l'existant (présent sur le dépôt distant) pour éviter toute synchronisation malheureuse au cas où

plusieurs travailleraient sur la même branche et sur les mêmes fichiers. Donc : **tirer** puis **pousser**.

Il est également bien d'avoir le réflexe de tirer avant de commencer sa matinée et avant de commencer son après-midi, histoire de voir ce qui a été ajouté par les collègues, histoire de voir si personne n'a déjà ajouté quelque chose qui vous serait utile (une nouvelle fonction manquante par exemple)

Une fois le développement de votre nouvelle fonctionnalité terminé (→ tous les tests unitaires passés avec succès), fusionner cette branche avec sa branche mère (généralement Master).

8. Conclusion

Cette architecture modulaire permet une maintenabilité optimale, facilite les tests unitaires et l'évolutivité du système. Elle favorise également une séparation nette entre les responsabilités métiers, techniques et de présentation.