Conception, Projet

Cours 6 - Conception générale

Table des matières

Introduction aux modèles généraux

Maintenant, nous avons un cahier de charge exhaustif, des directives haut niveau claires et transferrables dans le domaine informatique ainsi qu'une méthodologie par rapport à l'assurance qualité vis-à-vis les besoins du client et de notre équipe.

Plus le niveau d'abstraction est bas, plus on s'approche du technique et pratique.

Ceci-dit, il n'est pas encore le temps de coder, il nous faut choisir une architecture logicielle, une conception des systèmes et un processus de tests qui s'agence bien avec notre projet, ressources et capacité d'innovation.

Introduction aux modèles généraux

Voici les modèles généraux que nous allons abordés, dans cet ordre de choix (sinon cela ne fonctionne pas) :

1. Architecture logicielle (Globalité du système)
2. Conception (Composantes et/ou séparations du système)
3. Tests (Comment le code sera validé, **la stratégie pour y arriver**, non la technologie/méthode spécifique)

Introduction aux modèles généraux - À noter

NB : La littérature peut avoir une vue différente de la théorie.

Par exemple, l'architecture logicielle et la conception sont tous ensemble dans l'architecture logicielle.

Dans le cadre du cours, nous allons utiliser la définition séparée de architecture logicielle plus haut niveau d'abstraction que la conception.

Architecture logicielle

L'architecture logicielle est un niveau d'abstraction assez haut, juste en dessous du cahier de charges, où l'on peut choisir comment le système sera bâtit fondamentalement.

Nous avons les choix suivants, plusieurs d'autres existes, mais nous allons garder l'étendue à ceux-ci dans notre cours :

1. Monolitique
2. Microservice
3. Monolitique Microservice (Containrized singleton service as monolithic app)

Architecture logicielle - Monolitique

En bref, l'architecture monolithique au niveau tehcnique veut dire tout le code dans le même endroit (une application) et le tout devient massif et difficile à dinstinguer.

Ceci provient du mot monolithe, étant un morceau de pierre massif.

Au fur et à mesure d'ajouter du code, les différents systèmes s'entremèle et deviennent un gros morceau de code.

Cette architecture n'a pas d'avantages nobles et beaucoups de désavantages si elle est utilisée en elle même sous sa forme primitive.

Architecture logicielle - Monolitique

Avantages (non nobles) :

- Vitesse de production d'applications très rapide, bon pour livrer des produits rapidement (i.e. freelance)
- Débogage simple puisque tout est dans une application

Désavantages :

- Code peu maintenable
- Application singleton, tout doit fonctionner ensemble
- Logique d'affaire mélangée avec la logique technique
- Difficile/impossible à mettre à jour graduellement (bout par bout)
- Difficile à déployer
- Difficile à adapter avec la clientèle (scaling)

Architecture logicielle - Microservice

Les microservices sont une approche architecturale qui consiste à décomposer le logiciel en petits composants ou services indépendants.

Chaque service joue un rôle unique et communique avec les autres services au moyen d'une interface bien définie.

Comme ils s'exécutent indépendamment, il est possible de mettre à jour, modifier, déployer ou mettre à l'échelle chaque service selon les besoins de service.

Architecture logicielle - Microservice

Avantages :

- Facile à maintenir
- Coût faible lorsque l'application est à moyenne/grande échelle
- Indépendant l'un à l'autre des microservices (i.e. blog : users, posts, etc.)

Désavantages :

- Un coût initial plus haut dans l'hébergement petite échelle
- Difficile de suivre le débogage
- Demande une expertise plus avancée

Monolithique vs. Microservices

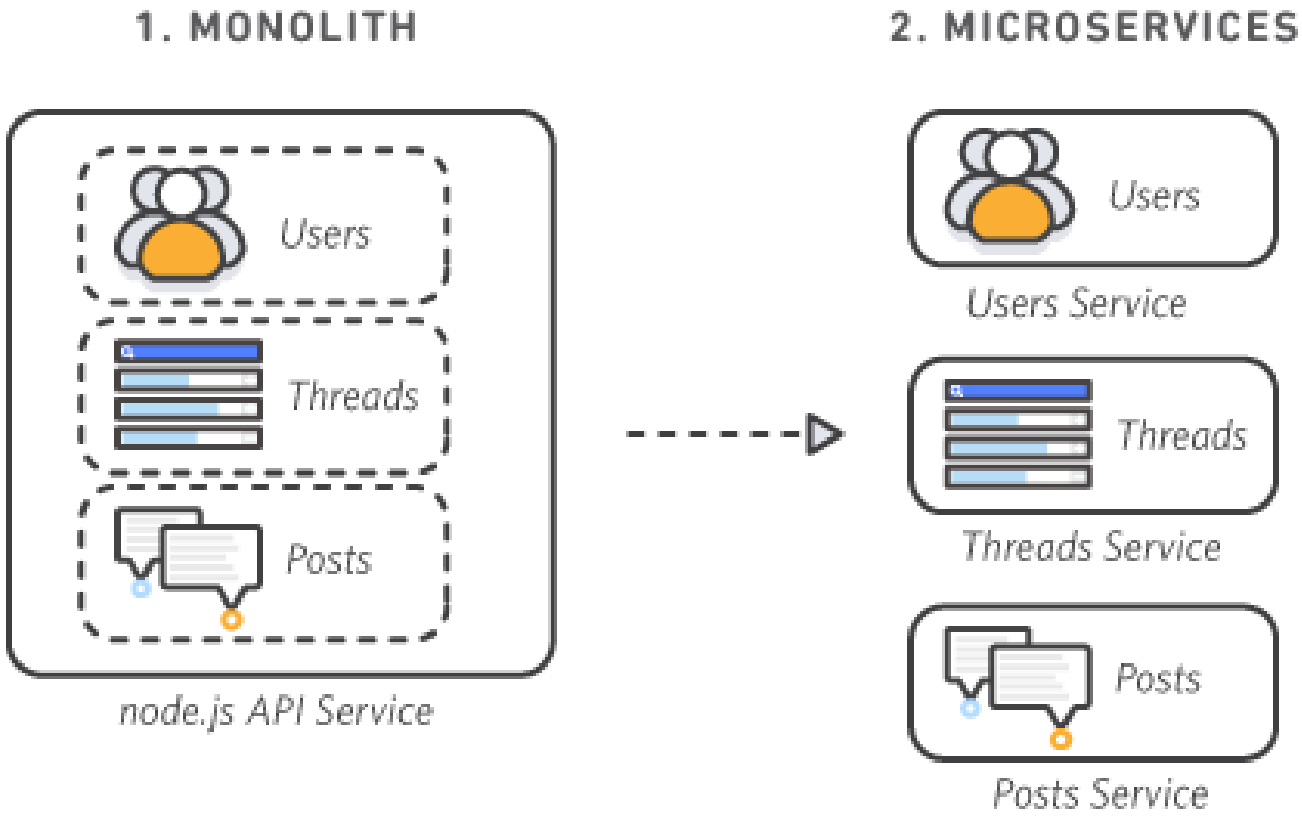


Figure: `https://aws.amazon.com/fr/compare/the-difference-between-monolithic-and-microservices-architecture/?nc1=h_ls`

Architecture Monolithique Microservices

Il est possible de **fusionner les deux types d'architectures** pour avoir le meilleur des deux mondes : des applications avec une architecture simple (API monolithique) et un système découplé pour une architecture globale facilement maintenable (microservice).

Architecture Monolithique Microservices

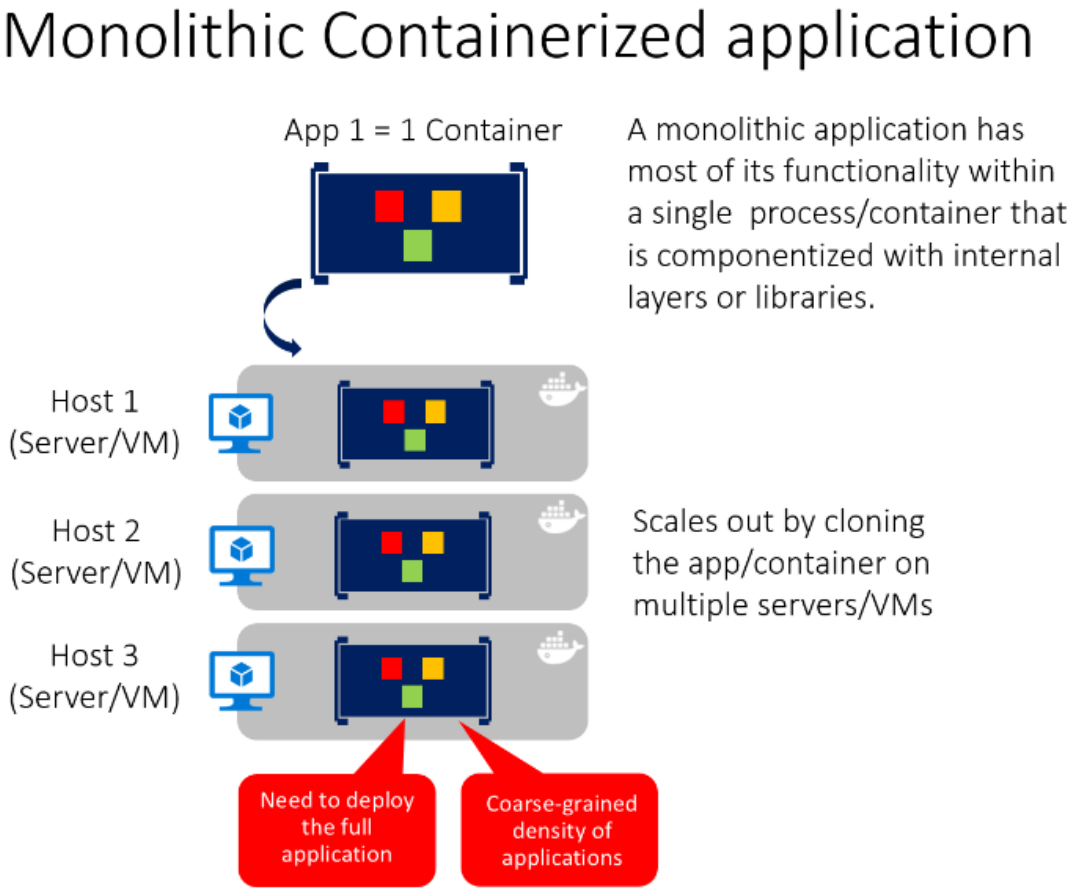


Figure: `https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures`

Architecture Monolithique Microservices

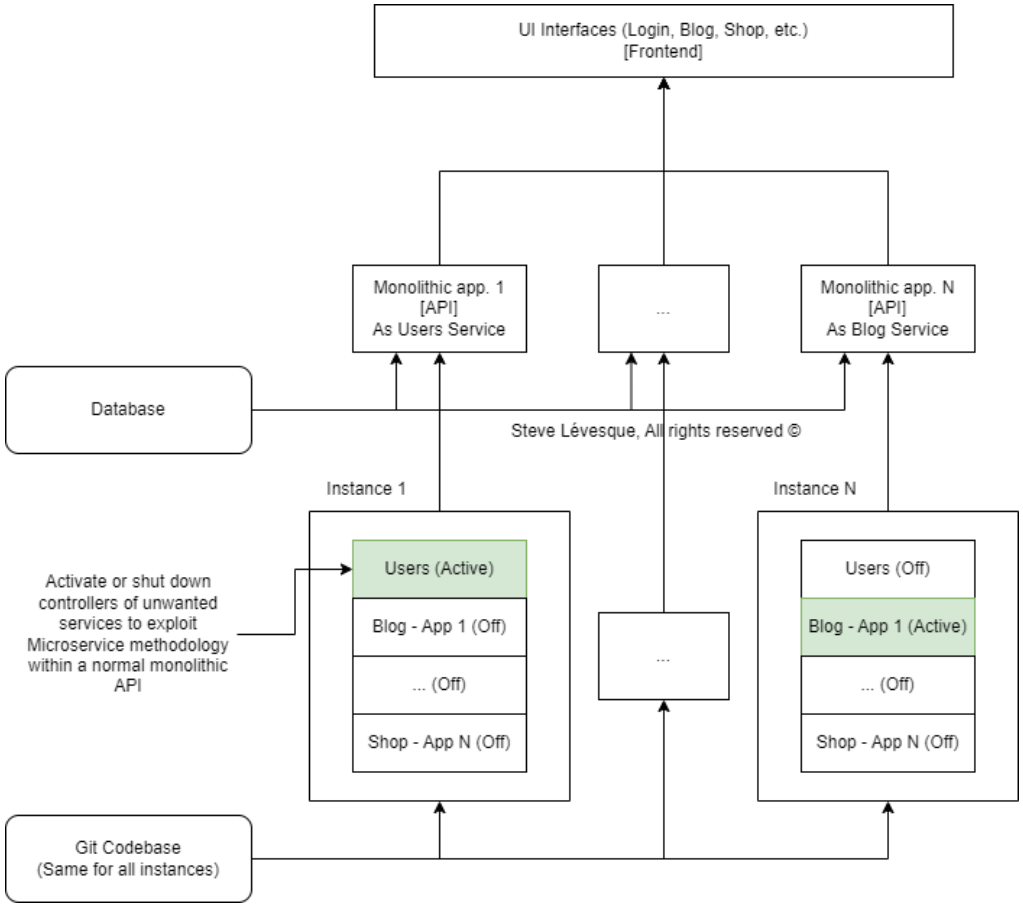


Figure: Steve Lévesque

Conception

D'une vue globale, un système complexe peut avoir plusieurs applications inter-connectées.

Par rapport à une seule application, il est possible de choisir la méthodologie de conception :

- Singleton (Monolithique)
- Client-Serveur
- Model-View-Controller (MVC)
- Model-View-ViewModel (MVVM)

Conception - Singleton (Monolithique)

Une application contenant toute la logique dans un code inter-relié et fortement dépendant l'un à l'autre.

Avantages et désavantages sont les mêmes que l'architecture monolithique.

Par contre, à noté que c'est acceptable pour une petite application Web avec un petit but précis et aucune vision future au niveau de l'amélioration et l'augmentation de la clientèle (utlisateurs finaux).

Conception - Singleton (Monolithique)

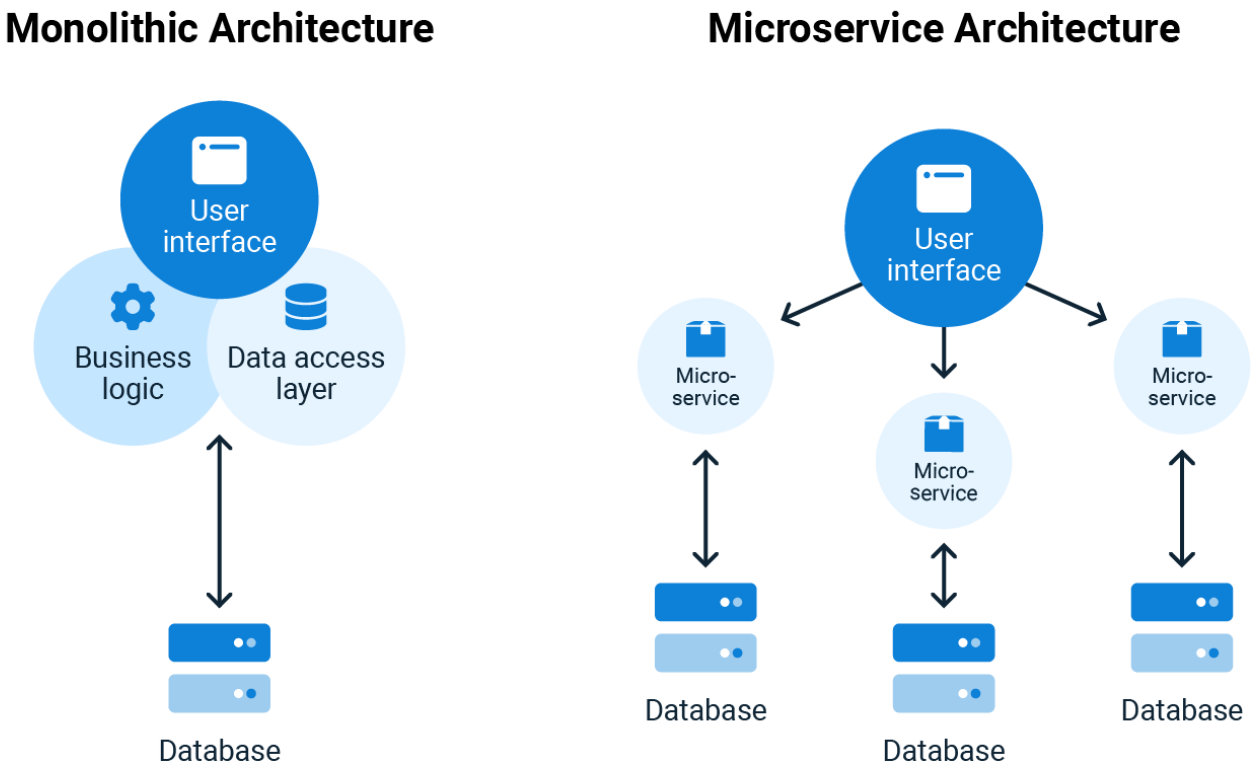


Figure: `https://blog.coderco.io/p/monoliths-vs-microservices-a-guide`

Conception - Client-Serveur

Une application séparée en deux : le client (App Web) et le serveur (API).

Avantages :

- Simple et propose une séparation claire du Frontend et Backend.
- Développement rapide et par responsabilité.

Désavantages :

- Pas de typage en temps normal entre le Frontend et Backend, si oui duplication des modèles pour les 2 langages des deux séparations.
- Nécessite la création d'un API, peut être inutile pour une application Web simple (monolithique).

Conception - Client-Serveur

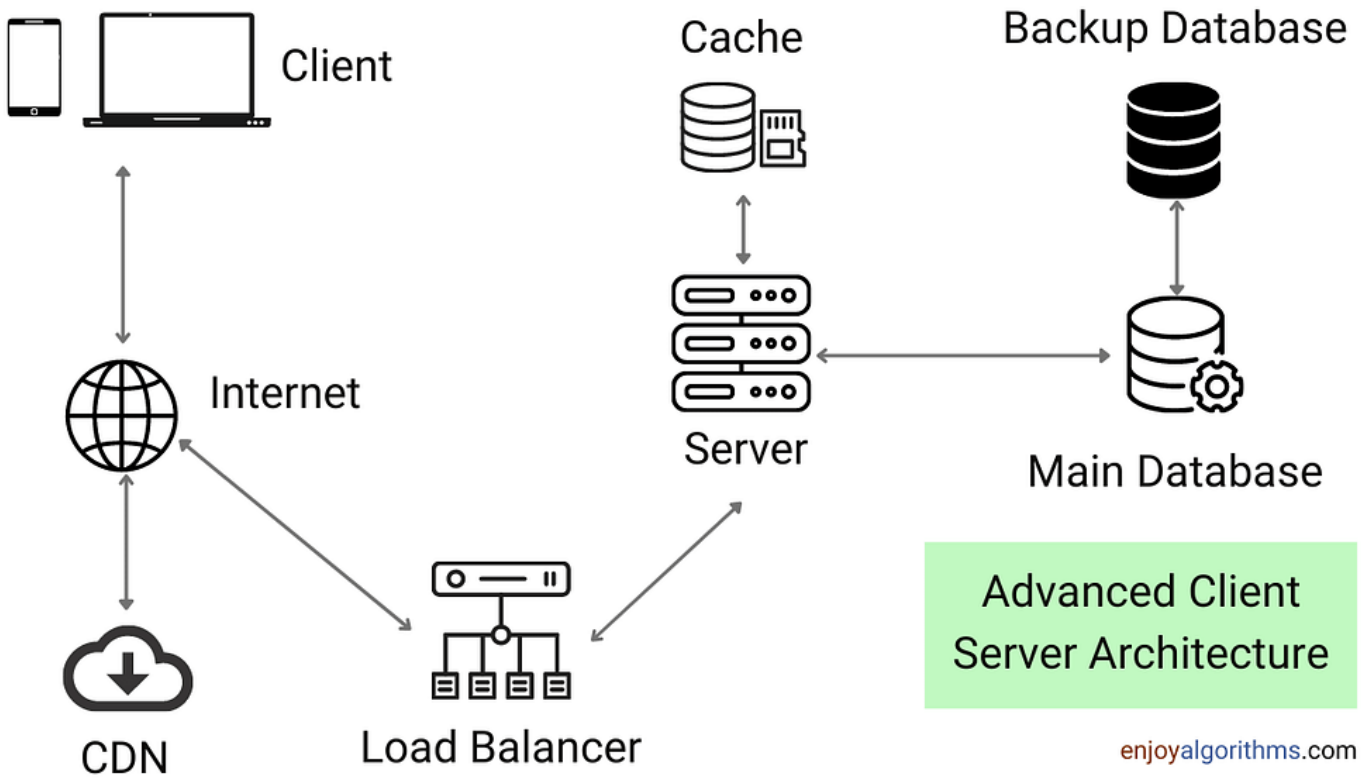


Figure: `https://www.enjoyalgorithms.com/blog/client-server-architecture`

Conception - Model-View-Controller (MVC)

Application séparée en trois responsabilité distinctes : Modèle, Vue et Contrôlleur.

Avantages :

- La logique d'affaire est séparée dans le modèle de manière explicite et complètement distrincte.
- Peu de duplication de code et typage fort facile à appliquer avec les objets (modèles).

Désavantages :

- Plus complexe qu'un modèle client-serveur, demande une meilleure connaissance et adaptabilité d'apprentissage
- Une certaine perte de typage avec la vue et le modèle si un Cadriciel Web client est utilisé avec un langage typé (i.e. React + .Net).

Conception - Model-View-Controller (MVC)

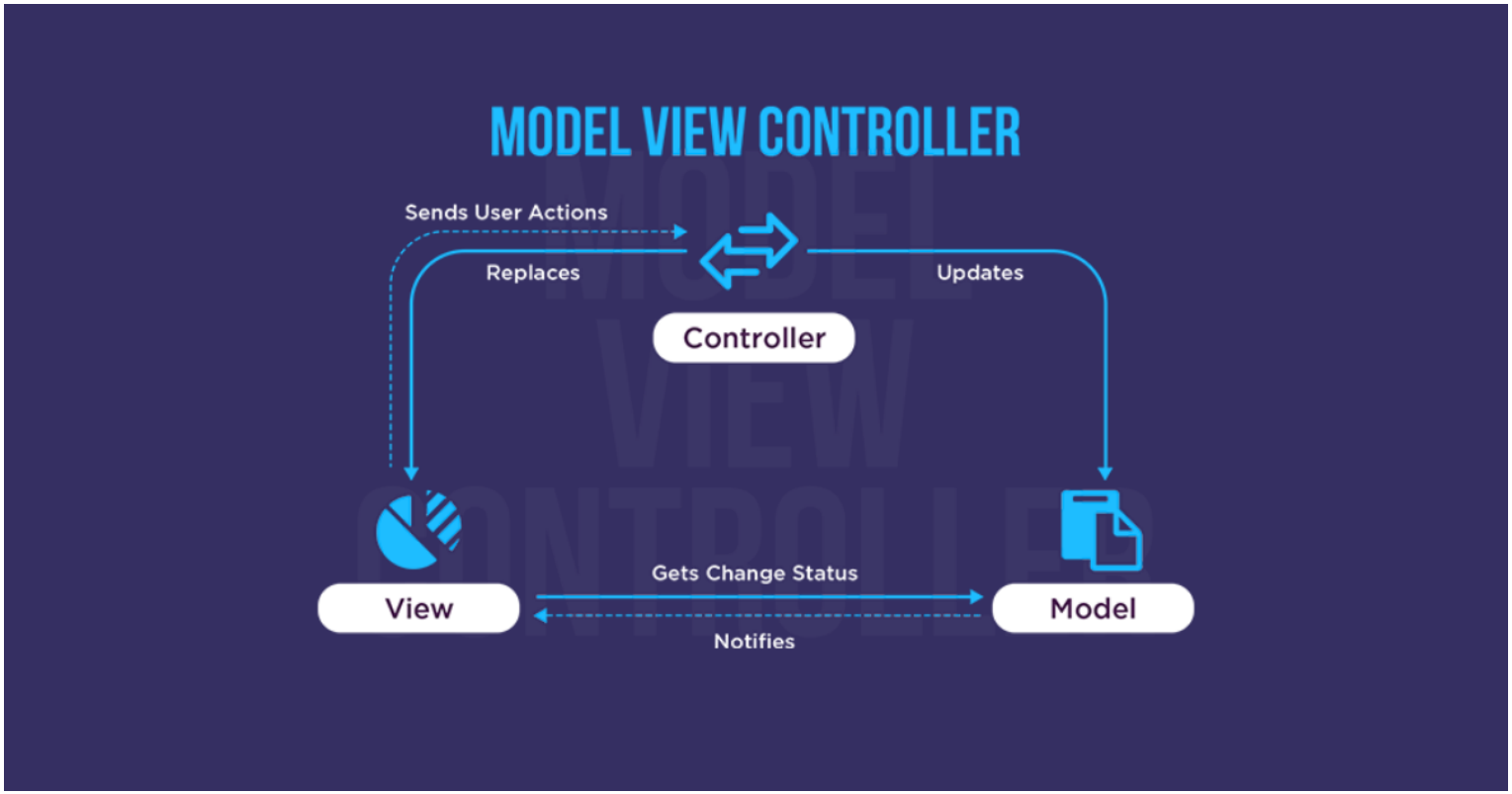


Figure:
https://www.thirdrocktechkno.com/blog/architecture-presentation-patterns-mvc-vs-mvp-vs-mvvm/

Conception - Model-View-ViewModel (MVVM)

Le contrôlleur est remplacé par une composante “vue modèle” pour améliorer les applications qui ont besoin de beaucoup de manipulations entre la vue et le modèle (i.e. applications mobiles).

Avantages :

- La vue en XAML pour les applications mobiles n’a pas besoin d’être codé à nouveau, très bon pour mettre à jour les interfaces sans MAJ dans l’app store.
- Séparation claire entre le modèle et la vue.

Désavantages :

- Communication deux côtés (two-way data binding), demande plus de ressources.
- Plus complexe s’il y a des sous-vue dans une vue spécifique

Conception - Model-View-ViewModel (MVVM)

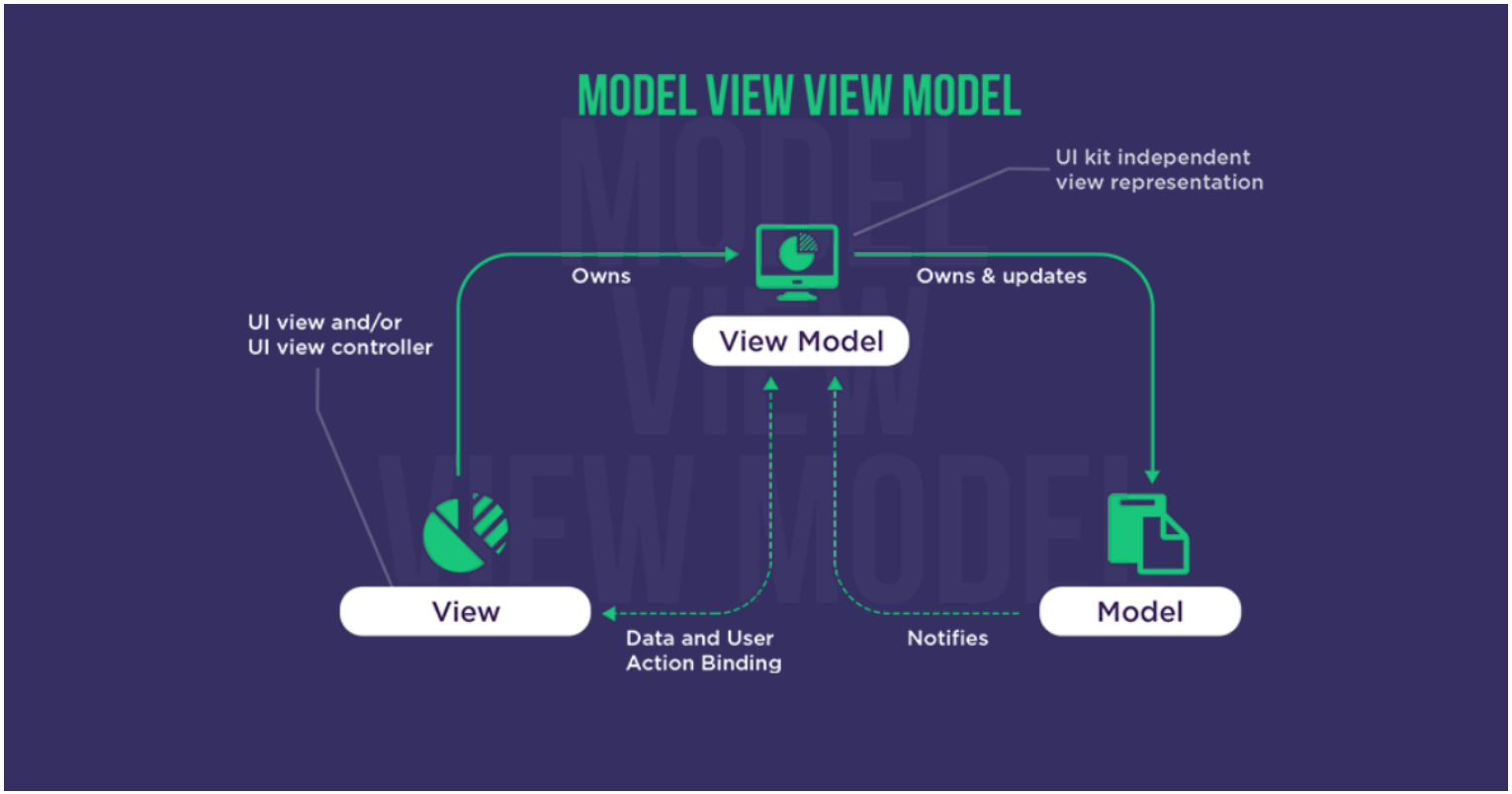


Figure:
https://www.thirdrocktechkno.com/blog/architecture-presentation-patterns-mvc-vs-mvp-vs-mvvm/

Tests

Il existe plusieurs méthodologies de tests pour valider le code des applications.

Dépendamment des équipes, de la culture de l'entreprise, les architectures, et conceptions d'applications, les méthodologies employées sont différentes.

Voici des méthodes que nous allons aborder dans ce cours (il en existe plus) :

- Traditionnel
- Test Driven Development (TDD)

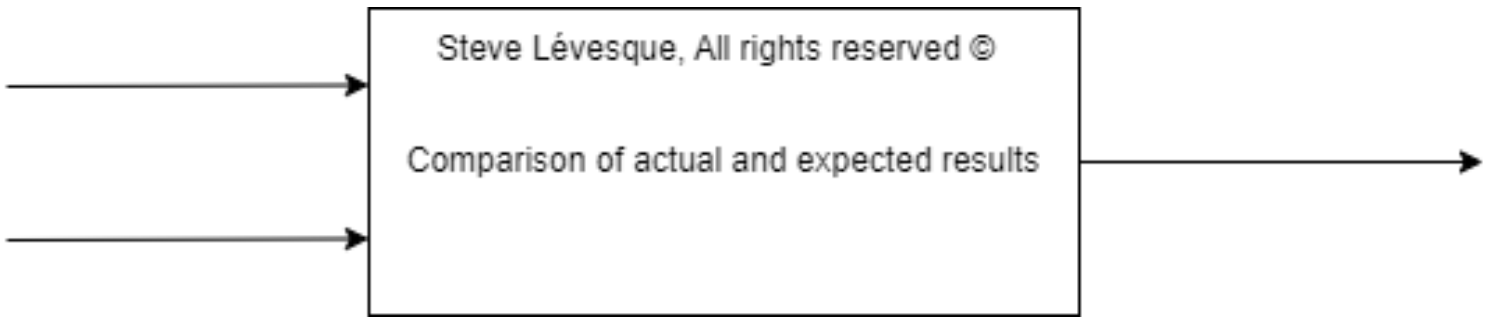
Tests - Traditionnel

De la manière traditionnelle, le développeur créer une suite de tests unitaires après avoir terminé son programme pour pouvoir le tester.

Méthode simple, mais introduit du biais, puisqu'il va instinctivement essayer de faire des tests qui passent pour pouvoir pousser son code en production.

Cette situation arrive souvent de manière naturelle (instinctive), sans qu'on le veuille puisque c'est la solution la plus rapide.

Tests - Traditionnel

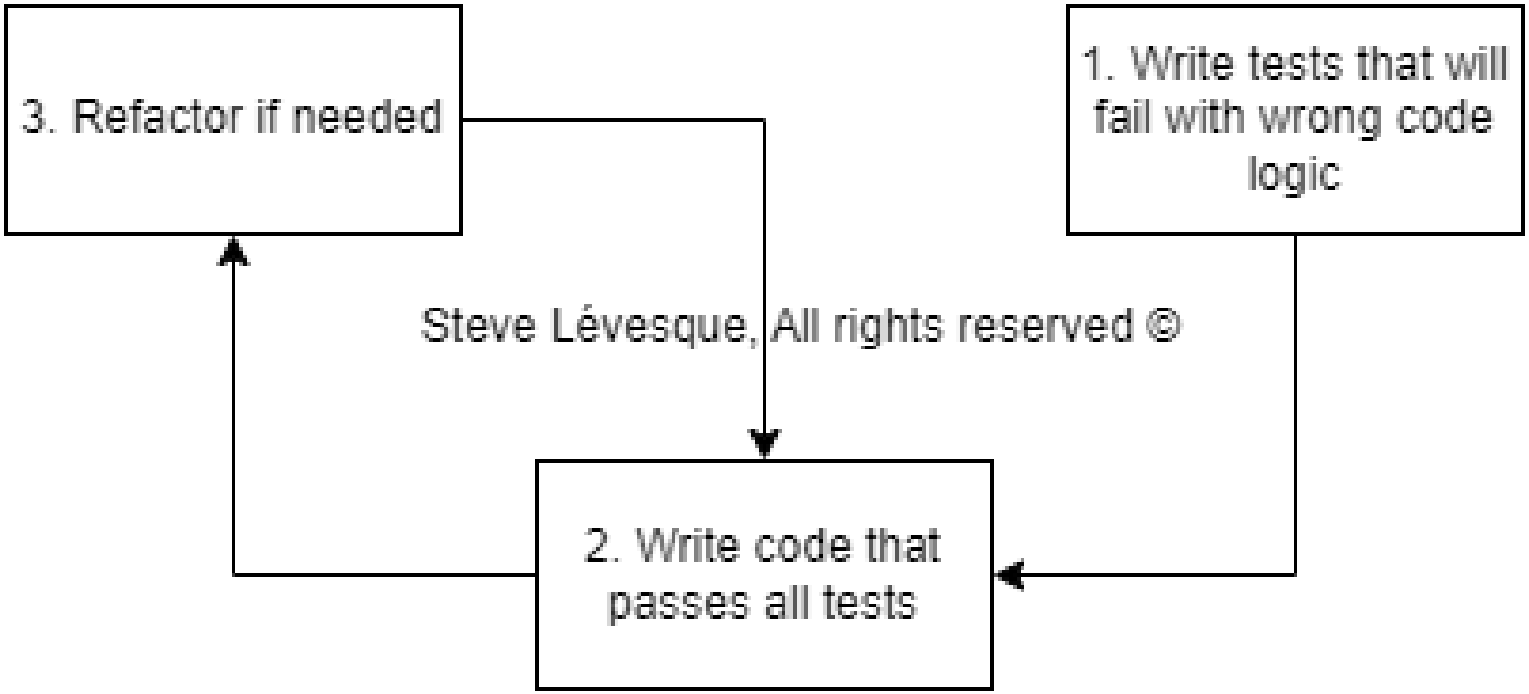


Tests - Test Driven Development (TDD)

De cette manière, les tests sont créés **avant** de développer le programme ou l'application.

Moins de biais est introduit puisque le développeur à plus de chances de penser au résultat pure et intègre dès le début.

Tests - Test Driven Development (TDD)



Bibliographie

- https://aws.amazon.com/fr/compare/the-difference-between-monolithic-and-microservices-ar?nc1=h_ls
- https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures
- https://www.thirdrocktechkno.com/blog/architecture-presentation-patterns-mvc-vs-mvp-vs-mvvm/
- https://www.enjoyalgorithms.com/blog/client-server-architecture