IUT ANNECY UNIVERSITÉ SAVOIE MONT BLANC

INFO

R3.04 - QUALITE DE DEVELOPPEMENT

CM1 : Méthodes de développement, Architectures, versioning

Vincent COUTURIER

Bonnes pratiques

1. Choisir la bonne méthode de développement (**CM1**)
2. Modéliser (analyse, conception) (**CM1**)
3. Choisir la bonne architecture de l'application (et appliquer des patrons d'architecture) (**CM1**)
4. Appliquer des patrons logiciels (software patterns) (**CM2**)
5. Développement (**CM1**) :
   a. Créer des vues accédant aux tables de la base de données
   b. Eventuellement des procédures/fonctions stockées
   c. ORM (Object-Relational Mapping)
   d. Utiliser un framework
   e. Gestion des exceptions
   f. Documenter
6. Versioning (**CM1**)
7. Intégration continue / Déploiement / Usine logicielle / DevOps (**CM1**)
8. Tests (**CM2**)

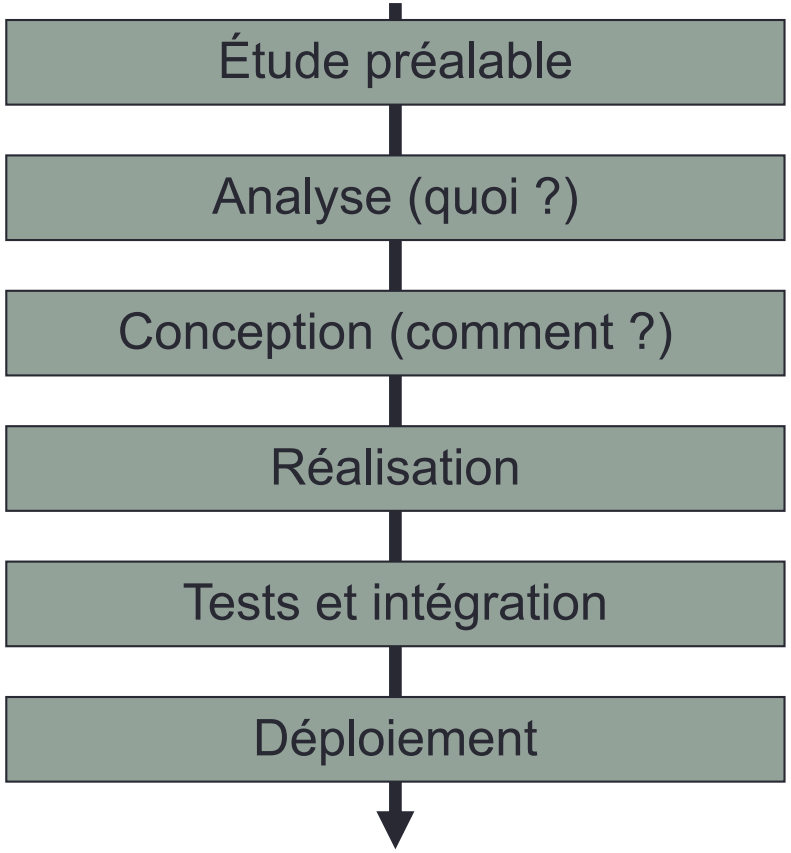
CHOISIR LA BONNE METHODE DE DEVELOPPEMENT

Bonne méthode ?

- Celle qui permet de diminuer les risques tout au long du projet
- Risques proviennent :
  - Utilisateurs défavorables
  - Méconnaissance des besoins (client)
  - Incompréhension des besoins (maitrise d'ouvrage, prestataire)
  - Instabilité des besoins
  - Technologiques trop complexes
  - Mouvement de personnel
  - Intégration au SI trop complexe (interopérabilité)
  - Excès de bureaucratie

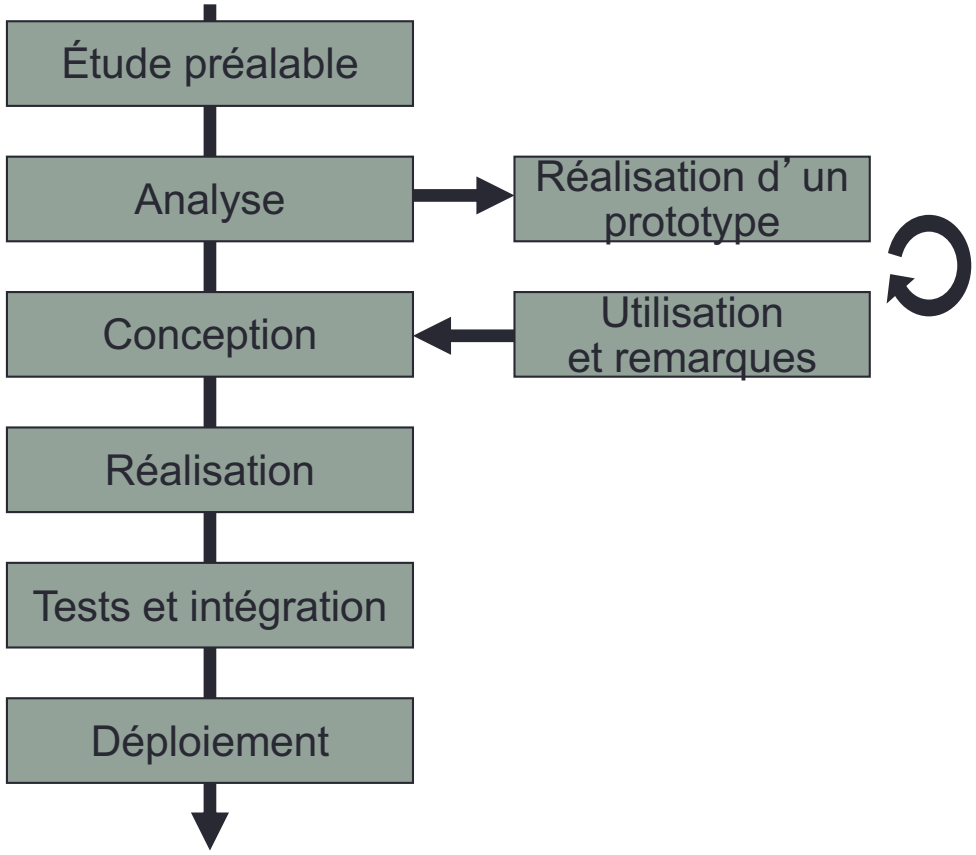
Règles

- Eviter le cycle en cascade :
  - Engendre un dérapage des coûts et des délais
  - Le client ne voit son produit qu'à la fin et si analyse fausse => on recommence TOUT depuis le début !



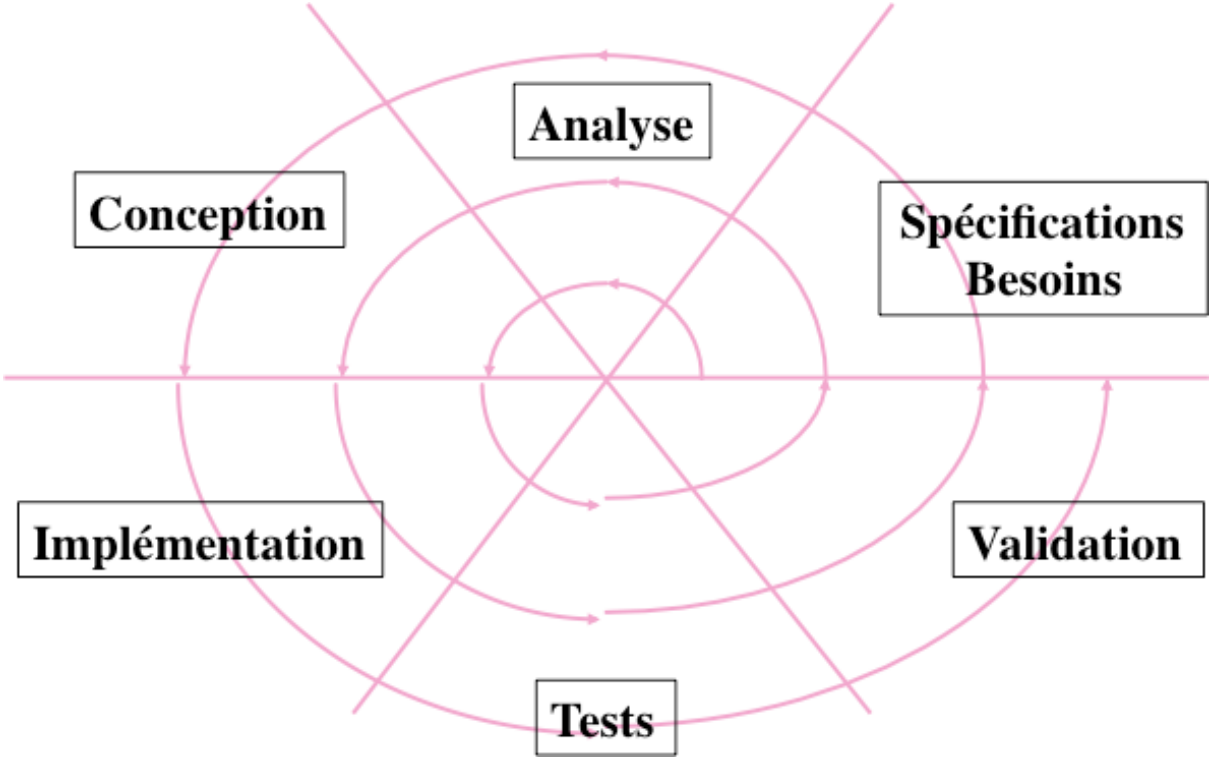
Règles

- Utiliser au moins un cycle intégrant un prototype
  - Prototype = image simplifiée (non opérationnelle, aspects sécurité et fiabilité non traités) du futur système. Fait l'objet de remarques des utilisateurs et d'adaptations avant sa réutilisation lors de la phase de conception.
  - ATTENTION cependant à ne pas considérer que version finale = prototype



Règles

- **OU MIEUX,** une méthode itérative et incrémentale
  - Chaque itération (ou lot) fait l'objet d'un cycle de développement complet.
  - Est à la base des méthodes agiles
  - SCRUM ou XP (Extreme Programming) intègre ce type ce cycle en spiral



Règles

- **OU ENCORE MIEUX,** une méthode agile
  - Vise à intégrer le client très tôt
  - SCRUM :
    - + : intègre le client tout au long du projet
    - + : les itérations, appelées sprints, sont courtes => permet de gérer et réduire le risque. D'autant plus vrai que l'application est importante.
    - - : plus difficile si le client est éloigné ou manque de temps.
  - EXTREME PROGRAMMING (XP) :
    - Méthode orientée sur le développement, même si une partie sur la gestion de projet
    - Adapté aux équipes réduites avec des besoins changeants
    - 7 Règles :
      1. « Puisque la revue de code est une bonne pratique, elle sera faite en permanence (par un binôme) » -> **Pair programming**
      2. « Puisque les tests sont utiles, ils seront faits systématiquement avant chaque développement » -> **Tests / TDD**
      3. « Puisque la conception est importante, elle sera faite tout au long du projet (refactoring) » -> **Conception nécessaire et suffisante**
      4. « Puisque la simplicité permet d'avancer plus vite, nous choisirons toujours la solution la plus simple » -> **Refactoring + Conception**
      5. « Puisque la compréhension est importante, nous définirons et ferons évoluer ensemble des métaphores »
      6. « Puisque l'intégration des modifications est cruciale, nous l'effectuerons plusieurs fois par jour » -> **Intégration continue / Usine logicielle**
      7. « Puisque les besoins évoluent vite, nous ferons des cycles de développement très rapides pour nous adapter au changement »

ANALYSE & CONCEPTION : MODELISER

Règles

- Un modèle est toujours préférable à une longue description !!!
- **NE PAS OUBLIER DE COMMENTER CHAQUE MODELE.**
  - Exemple : sous le MCD, lister les règles de gestion :
    - « Une commande ne peut être passée que par un client »
    - « Une facture peut faire l'objet de plusieurs règlements »
    - etc.

- Définir les besoins est essentiel => Diagramme des UC
  - Bien comprendre / reformuler le besoin si nécessaire
  - **Ne pas sous-estimer la tâche – primordiale – de réalisation d'un CDC**
- Définir le QUOI est essentiel => MCD ou diagramme de classes d'analyse, diagramme de séquences système, diagramme d'activités.
- Définir le COMMENT est essentiel pour les applications de taille importante =>
  - Diagrammes des classes participantes
  - Diagrammes de séquences (multi-couches)
  - Diagrammes d'état-transition

Règles

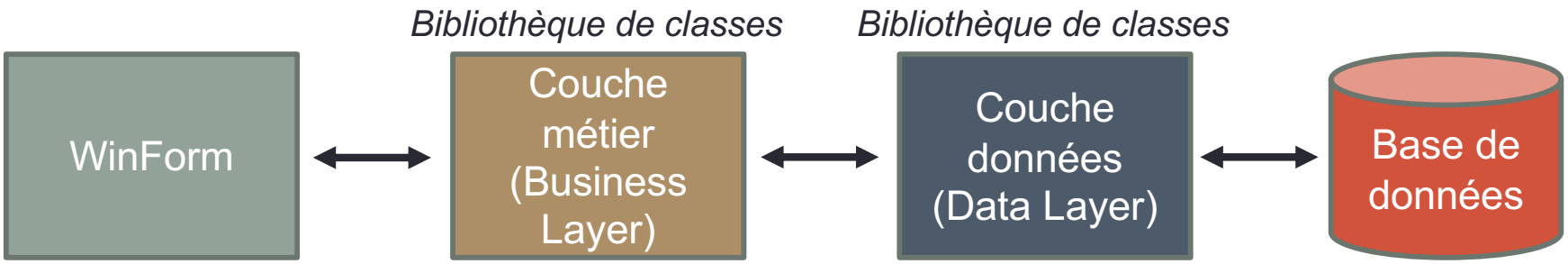
- Au fur et à mesure de la programmation, il est vital de générer le diagramme de classes de l'application et/ou une carte de code de façon à :
  - Garder une vision claire de l'application
  - Comparer / au diagramme de classes de conception

- *Truc et astuce : avoir à disposition une impression A3 du diagramme de classes implanté n'est pas absurde !*

CHOISIR LA BONNE ARCHITECTURE

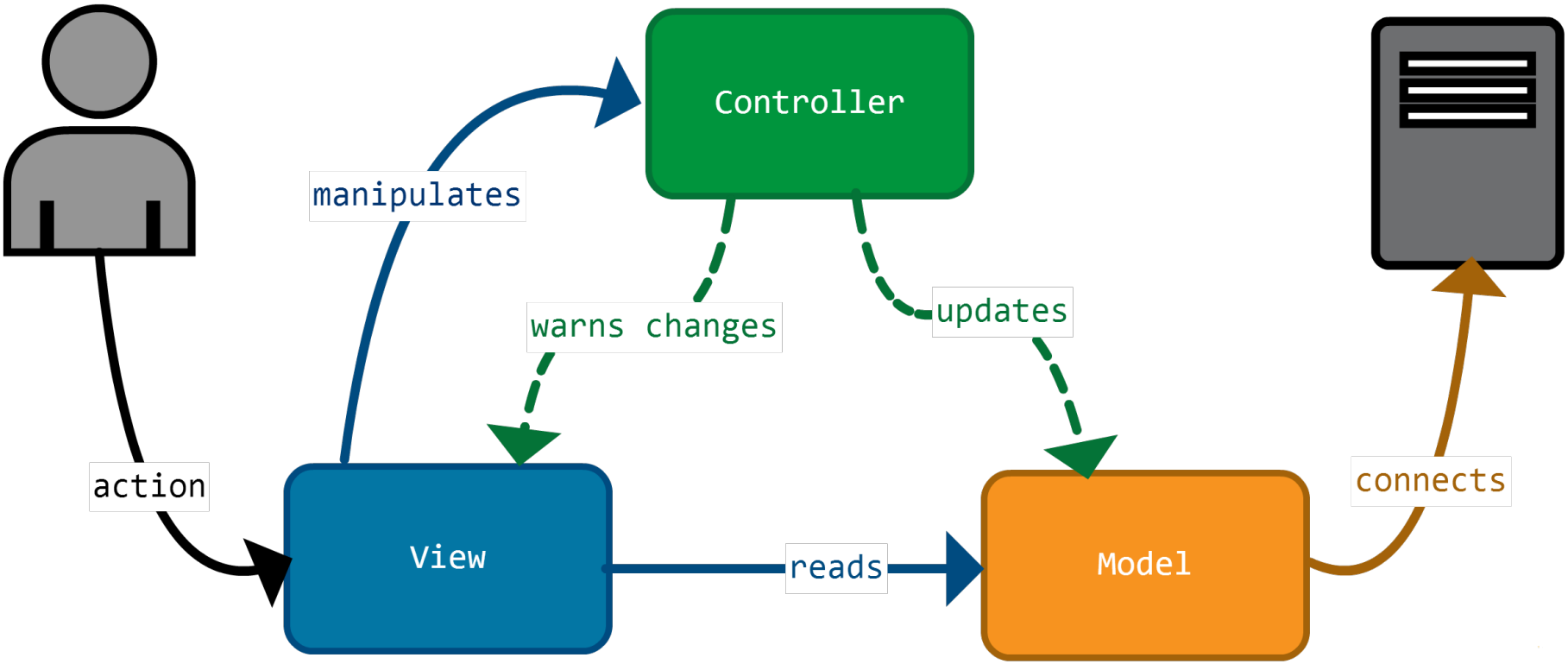
Règles

- Penser à **décomposer votre application en au moins 3 couches**, même pour les applications client-serveur lourdes (i.e. dont le client intègre des traitements, ex. Windows Forms, Web Forms ASP.Net) :
  - Exemple : Windows Forms



- Pour les technologies fonctionnant avec un client XAML ou XML décrivant l'interface + du code behind (ex. WPF ou UWP), certains frameworks JS (Knockout, Angular/Ionic, Ext, Vue.js), Android (Java, Kotlin), Flutter/Xamarin, Swift, il est possible de respecter le **pattern d'architecture MVVM**.
- Pour les autres frameworks de développement ou langages, le **pattern d'architecture MVC** est conseillé.

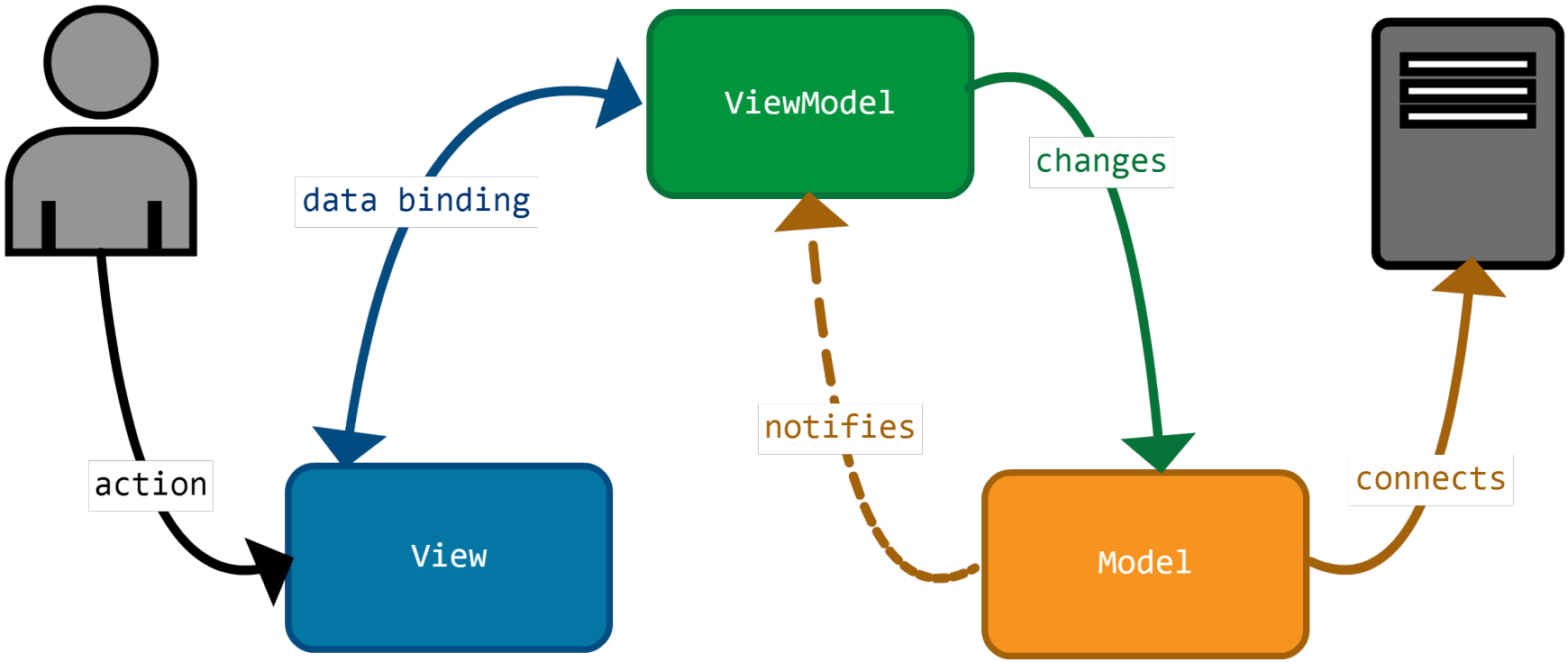
Pattern MVC



- Le *Modèle* définit la structure des données et communique avec le serveur.
- La *Vue* affiche les informations du *Modèle* et reçoit les actions de l'utilisateur.
- Le *Contrôleur* gère les évènements et la mise à jour de la *Vue* et du *Modèle*.

https://www.docdoku.com/blog/2015/02/17/architecturer-ses-applications-js-pattern-mvvm/

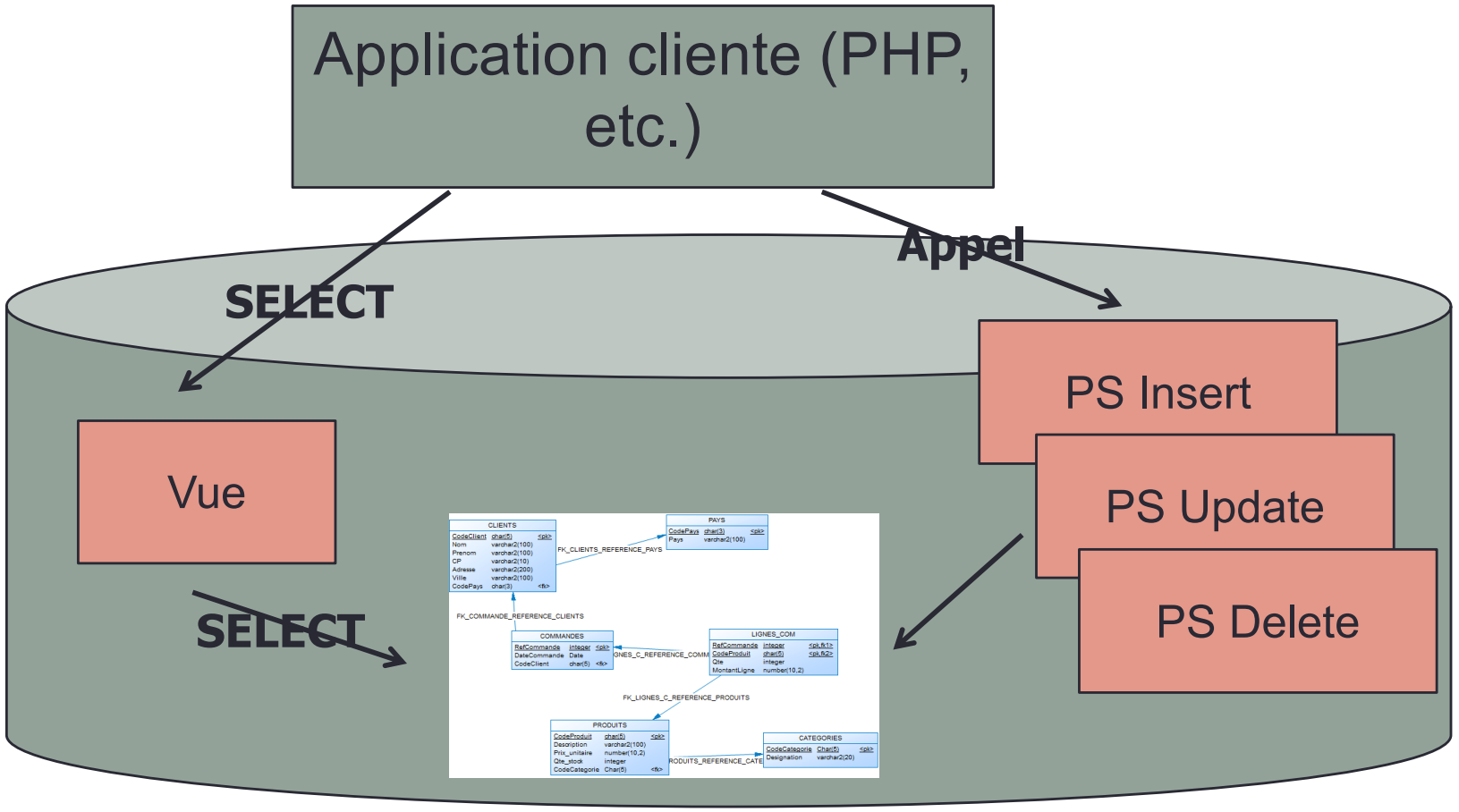
Pattern MVVM



- La *Vue* reçoit toujours les actions de l'utilisateur et interagit seulement avec le *ViewModel*.
- Le *Modèle* communique avec le serveur et notifie le *ViewModel* de son changement.
- Le *ViewModel* s'occupe de :
  - présenter les données du Model à la Vue,
  - recevoir les changements de données de la Vue,
  - demander au Model de se modifier.
- **Nous y reviendrons en CM3.**

DEVELOPPEMENT

Base de données



1 vue par table (pour SELECT) ; 3 fonctions ou procédures stockées par table (INSERT, UPDATE, DELETE)

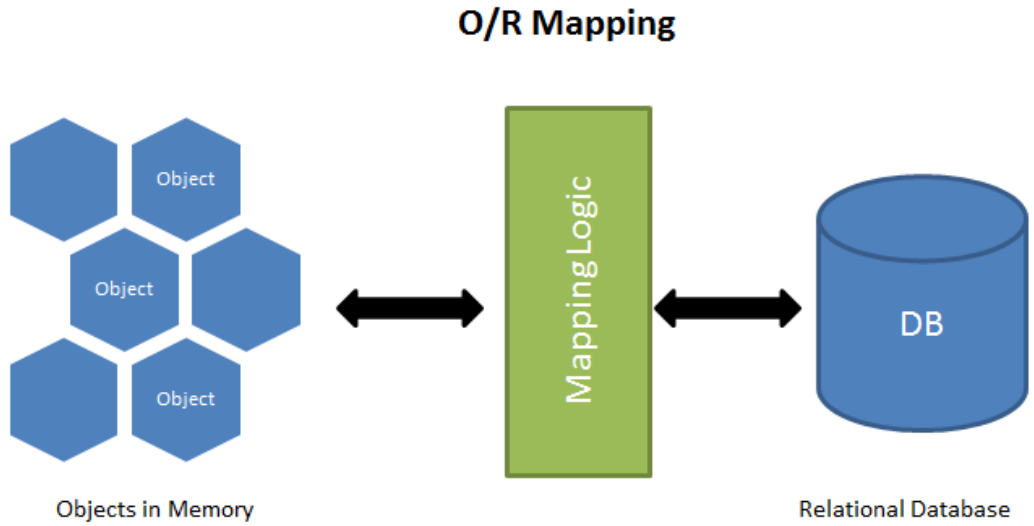
Base de données

- Une vue est créée pour chaque table, intégrant tous ou partie des champs de celle-ci.
  - Permet d'utiliser seulement des vues pour l'accès aux données en lecture et donc de limiter le risque de verrou qui pourrait poser des problèmes pour un accès fluide aux données de la table.
- Procédures pour INSERT, UPDATE et DELETE :
  - Au moins 1 procédure de chaque, pour chacune des tables de la base de données.
- Les procédures réalisant les mises à jour utilisent en paramètre d'entrée l'identifiant de la ligne à mettre à jour, ainsi que tous les autres champs qui peuvent être modifiés.
- Les procédures réalisant les ajouts retournent l'ID de l'enregistrement inséré.

Base de données

- Si pas d'ORM :
  - Vues + procédures stockées
- Si ORM :
  - S'il gère des vues (Doctrine, Hibernate, Entity Framework / Entity Framework Core, Dapper, Eloquent…) => utiliser les vues
  - S'il gère les procédures stockées (Doctrine, Hibernate, EF / EF Core, Dapper, Eloquent…) => utiliser les procédures stockées.
  - S'il gère ni les vues, ni les procédures stockées =>
    - soit on se passe de l'ORM
    - soit on utilise quand même l'ORM (directement sur les tables).

ORM ?

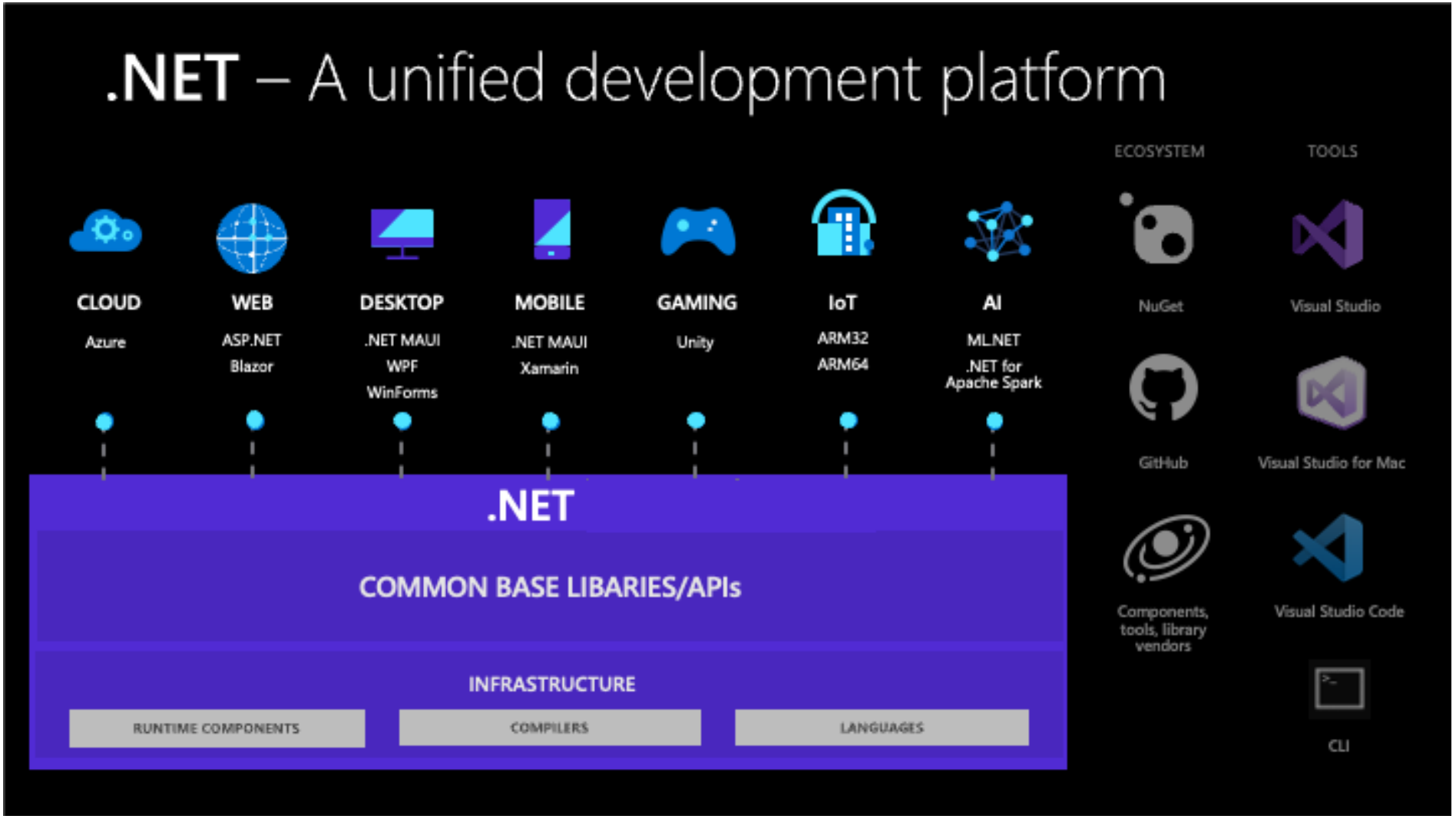


- Mapping objet-relationnel (en anglais *object-relational mapping* ou *ORM*) :
  - « Programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet ».
  - Il définit des correspondances entre les schémas de la base de données et les classes du programme applicatif.
  - Ex. d'ORM : Entity Framework/Entity Framework Core ou Dapper en .NET, Hibernate en Java, Doctrine ou Eloquent en PHP, Sequelize ou Prisma en JS, etc.
  - 2 modes : *Code first* ou *Database first*

Codage : framework

- Utiliser un **framework** est BEAUCOUP mieux que de tout redévelopper à la main
  - PHP4 à la main : BEURK !
  - PHP 5 : MIEUX
  - Symfony, Laravel, etc. : BIEN !!!!!!

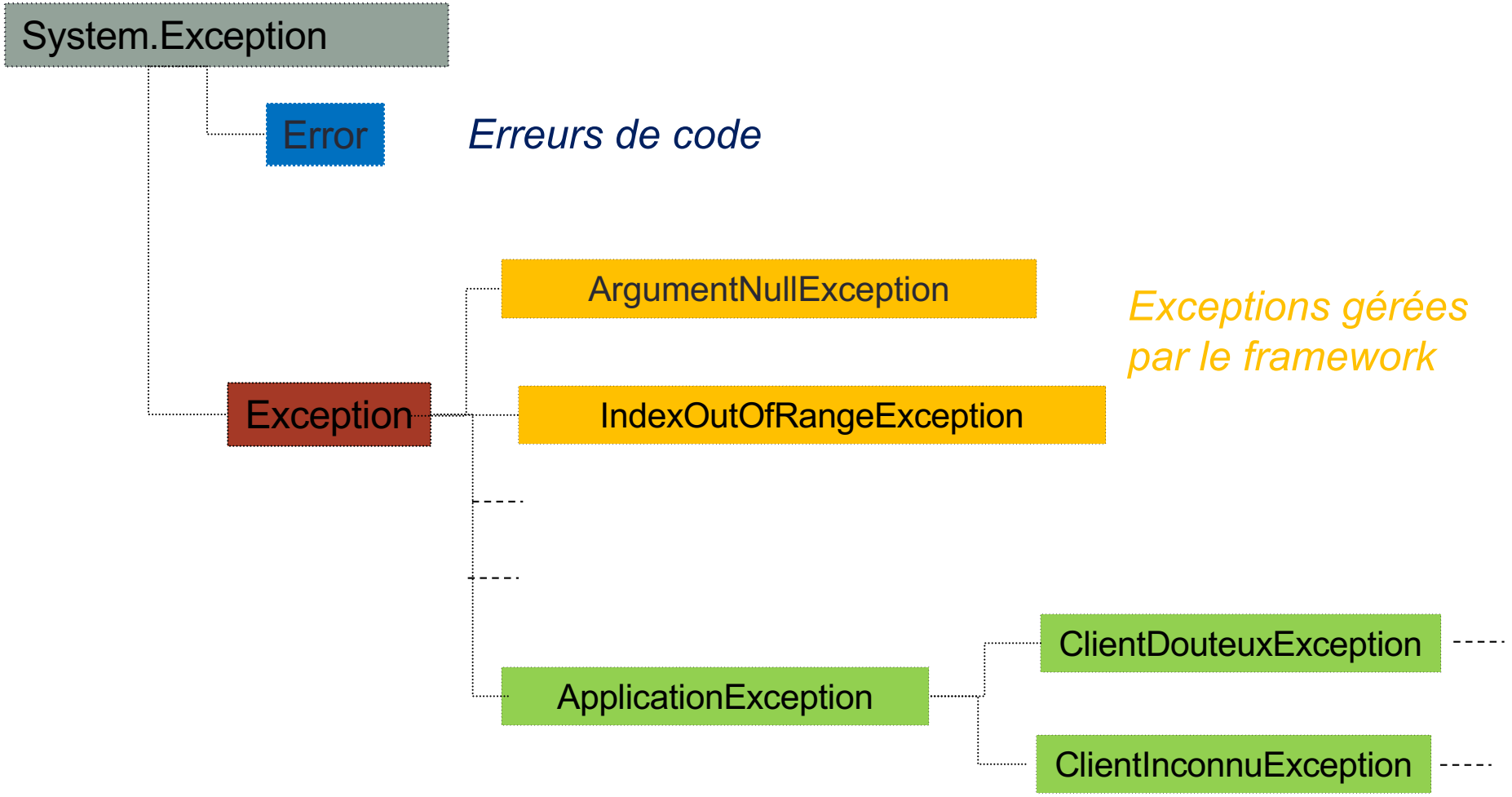
Framework .NET 8



Framework .NET ?

- .NET Framework :
  - Fonctionne uniquement sous Windows
  - Framework propriétaire
  - Version 1.0 (2002) à Version 4.8 (2019)
- .NET Core :
  - Fonctionne sur tous les OS (Windows, Linux, MacOS)
  - Open-source
  - Version 1.0 (2016) à Version 3.1 (2019)
- .NET :
  - Fusion de .NET Framework et .NET Core
  - Fonctionne sur tous les OS
  - Open-source
  - Version .NET 5 (2020) à .NET 6 (2021)
  - .NET 7 (version standard non LTS) sortie en novembre 2022
  - .NET 8 sortie en novembre 2023

Codage : Exceptions



Codage : créer/lancer des exceptions

- 1. Créer une exception :
  - Créer sa propre classe héritant de `Exception`
- 2. Instancier notre classe ou créer une exception déjà existante dans le framework :
  - `new`
- 3. Lever notre exception :
  - `throw`

try … catch … finally

- Ex :

```
try {
  Console.WriteLine("try");
}
catch (ArgumentNullException e) {
  Console.WriteLine("caught null argument");
}
catch (Exception e) {
  throw(e);
}
finally {
  Console.WriteLine("finally");
}
```

- Snipets : `try` et `tryf`

Codage : Commentaires

Règle : **Commenter maintenant et non plus tard !!!!**

Sert à expliquer votre code.

A l'intérieur de votre code : // ou /* */

```
//Affichage
Console.WriteLine("Hello !");

/*
 * Si le commentaire est conséquent,
 * Utilisez le /*
 */
```

Codage : Commentaires

- .NET permet d'écrire les commentaires en XML
  - Début du commentaire : `///`
  - Fin du commentaire : `fin de ligne`
- Un document XML est généré lors de la compilation du code avec l'argument `/doc`
  - S'appuie sur un schéma XML prédéfini
  - Il est aussi possible d'ajouter ses propres balises
  - Certains commentaires sont vérifiés : paramètres, exceptions, types

Codage : Commentaires

- Avant la méthode / classe :

```
/// <summary>
///  Description complète de la classe. Généralement on donne la fonction de cette classe ainsi que ces particularités
/// </summary>
public class MaClasse
{
    /// <summary>
    /// Une méthode qui ne fait rien ;-)
    /// </summary>
    /// <param name=a>parametre qui ne sert à rien du tout.</param>
    /// <returns>Valeur passée en argument non modifiée car la fonction ne fait rien .</returns>
    public int UneMethode(int a)
    {
        return a;
    }

}
```

- Dans la méthode :

```
public void Switch(ref int a, ref int b)
{
    //Effectue l'échange de a et b
    a *= b;
    b = a/b;
    a = a/b;


}
```

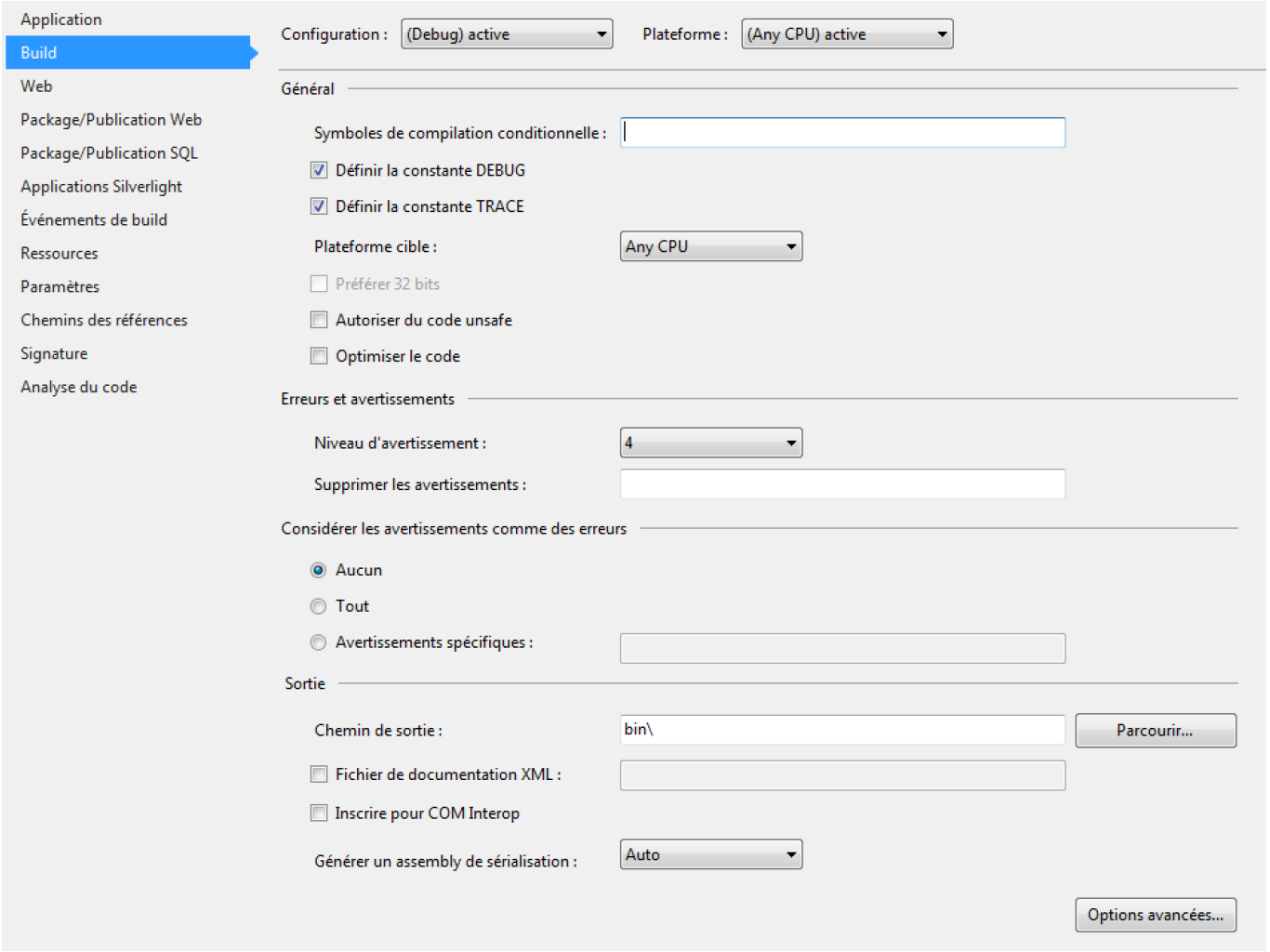
Codage : Commentaires

- Ces tags sont utilisables en fonction de l'entité documentée

Entité	Tags utilisables
class	<summary>, <remarks>, <seealso>
struct	<summary>, <remarks>, <seealso>
interface	<summary>, <remarks>, <seealso>
delegate	<summary>, <remarks>, <seealso>, <param>, <returns>
enum	<summary>, <remarks>, <seealso>
constructor	<summary>, <remarks>, <seealso>, <param>, <permission>, <exception>
property	<summary>, <remarks>, <seealso>, <value>, <permission>, <exception>
method	<summary>, <remarks>, <seealso>, <param>, <returns>, <permission>, <exception>
event	<summary>, <remarks>, <seealso>

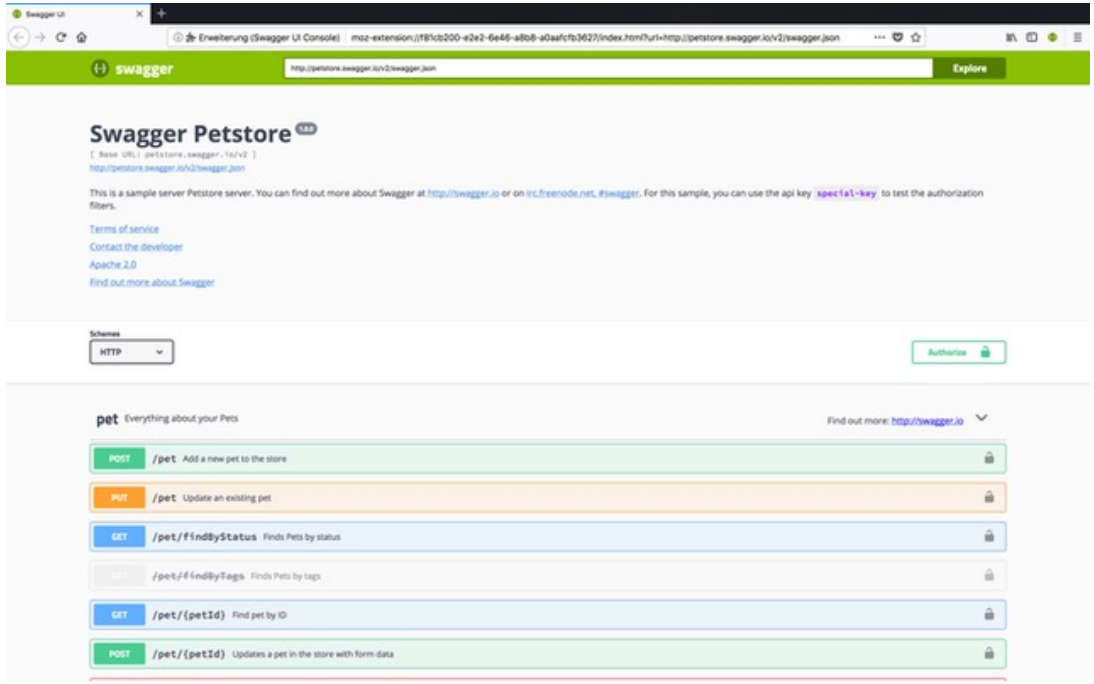
Codage : Commentaires

- Dans les propriétés du projet, sélectionner « Build » et modifier la propriété « Fichier de documentation XML»
- Inconvénient : une fois le fichier xml de documentation précisé pour un projet, il faut définir un **commentaire de documentation pour toutes les entités du projet ainsi que leur membre**, sinon un avertissement est signalé pour chaque élément qui n'en possède pas.



Codage : documentation

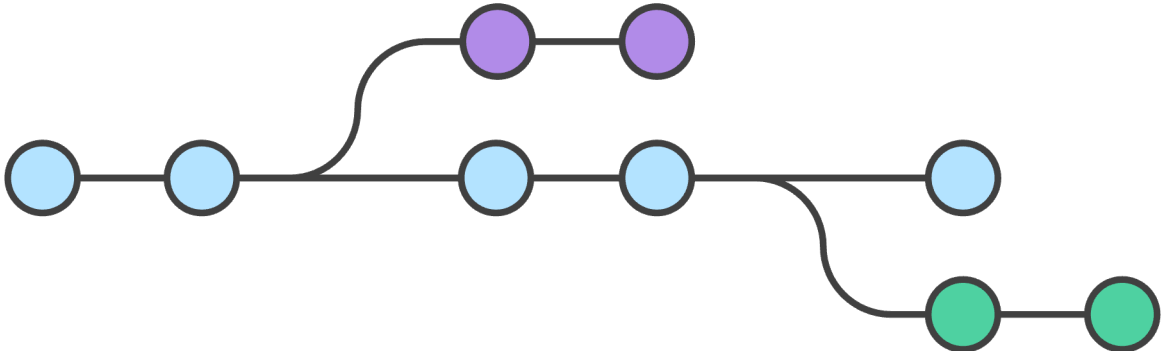
- Documenter !!!!!!
  - Générer ensuite la doc :
    - Avec Doxygen (Format « Doxygen » Java)
    - Avec SandCastle (Format « Microsoft »)
    - Avec Swagger pour les API (web services)



VERSIONING

Gestionnaire de versions et du code source (versioning)

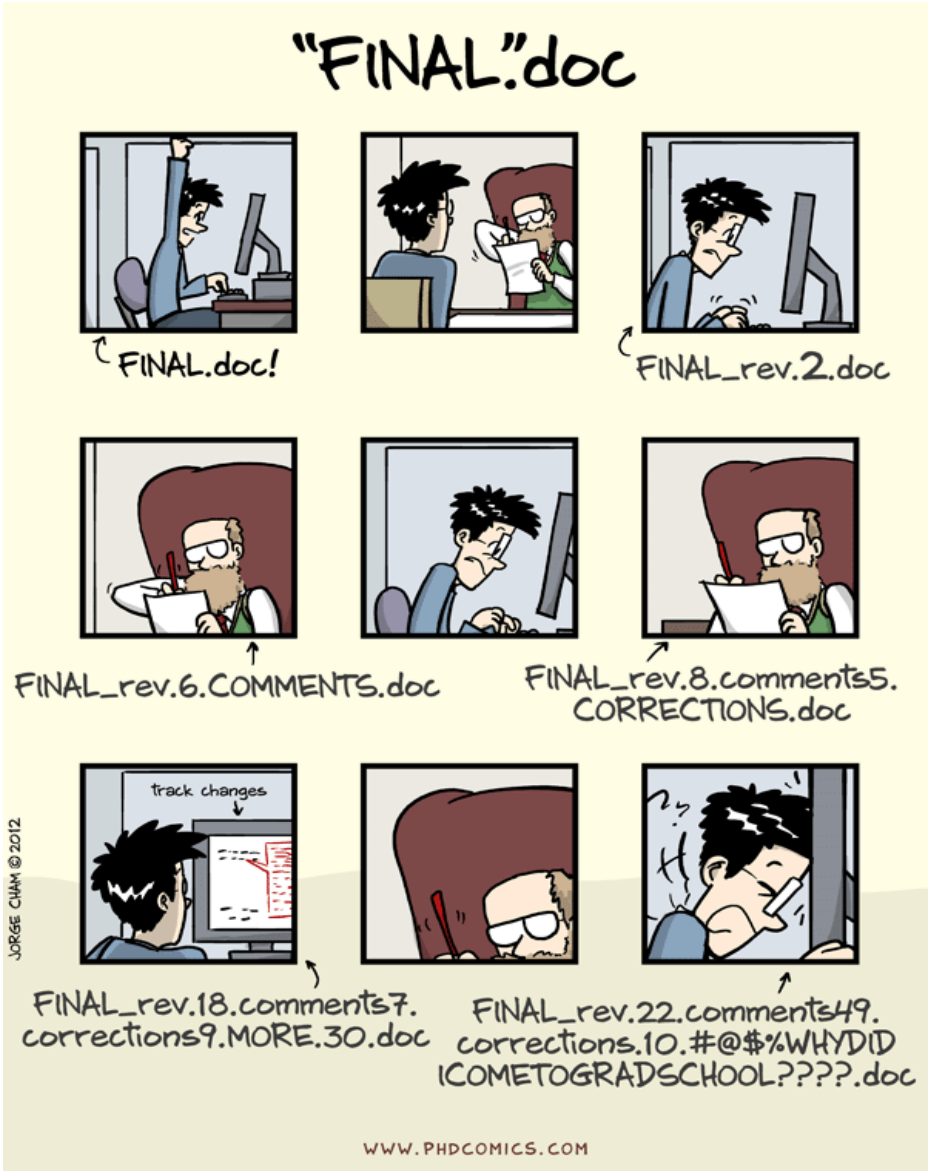
- Pour tout projet informatique, il faut une **stratégie de backup** du code source.
- On ajoute souvent une **gestion des versions permettant** de sauvegarder les différentes versions du code source et de les comparer.
- Un développeur peut proposer plusieurs **révisions** par jour
- Objectif du versioning :
  - **Travailler à plusieurs sans se marcher dessus** : indispensable pour les projets en équipe
  - **Garder un historique propre de toutes les modifications effectuées** : on organise son travail sous forme de “commits” documentés
  - Supporter des **branches** de développement :



Gestionnaire de versions et du code source (versioning)

- Utiliser un gestionnaire de version, **même si vous développez seul !**
- Il permet de :
  - Sauvegarder le code source
  - Partager le développement (travail de groupe)
  - Gérer les conflits en cas de codage sur les mêmes parties de fichiers
  - Garder une trace de tout ce qui a été fait
  - Pouvoir revenir en arrière (annuler les modifications)
- Exemples :
  - Git et sa version cloud Github (ou ses concurrents : Gitlab, Gitea, Framagit, Bitbucket)
  - SVN. Exemple : Microsoft Team Foundation Service (TFS) installé sur un serveur ou utilisable online sur *Azure DevOps*

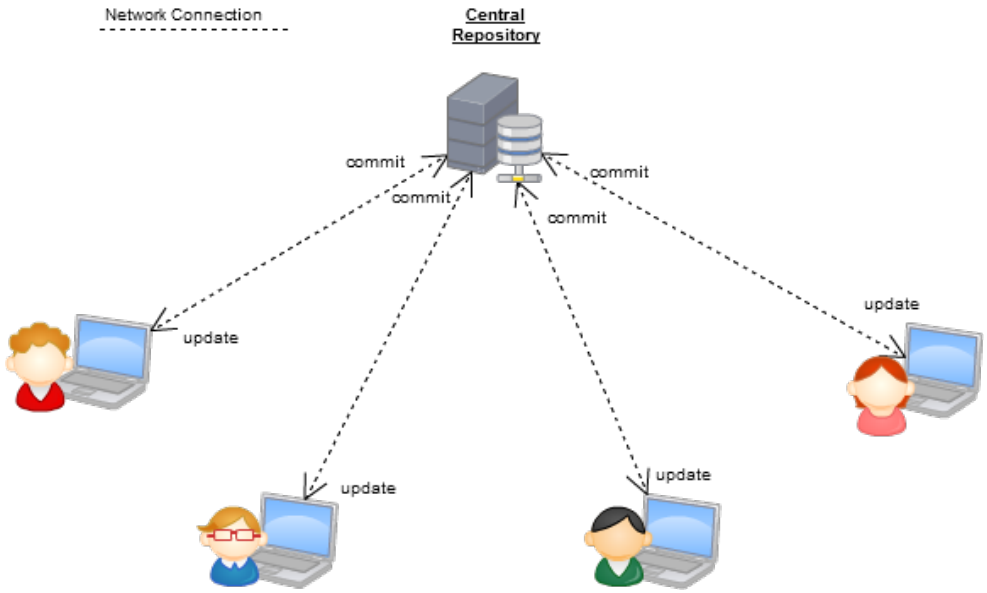
Gestionnaire de versions et du code source (versioning)



Gestionnaire de versions et du code source (versioning)

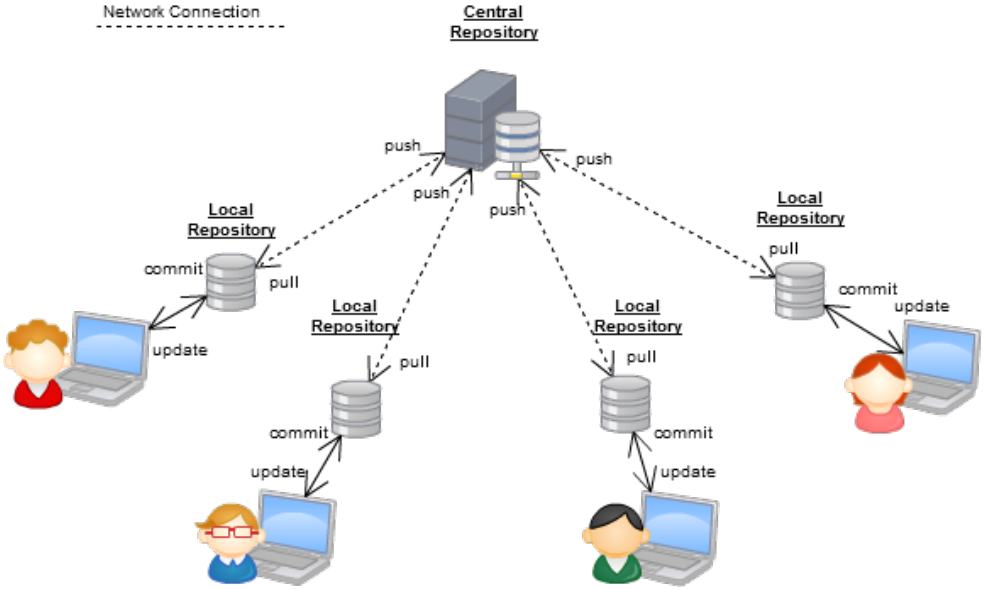
Ne pas se limiter à copier le dossier du code source !

2 types de gestionnaires de versions



SVN (Subvertion)

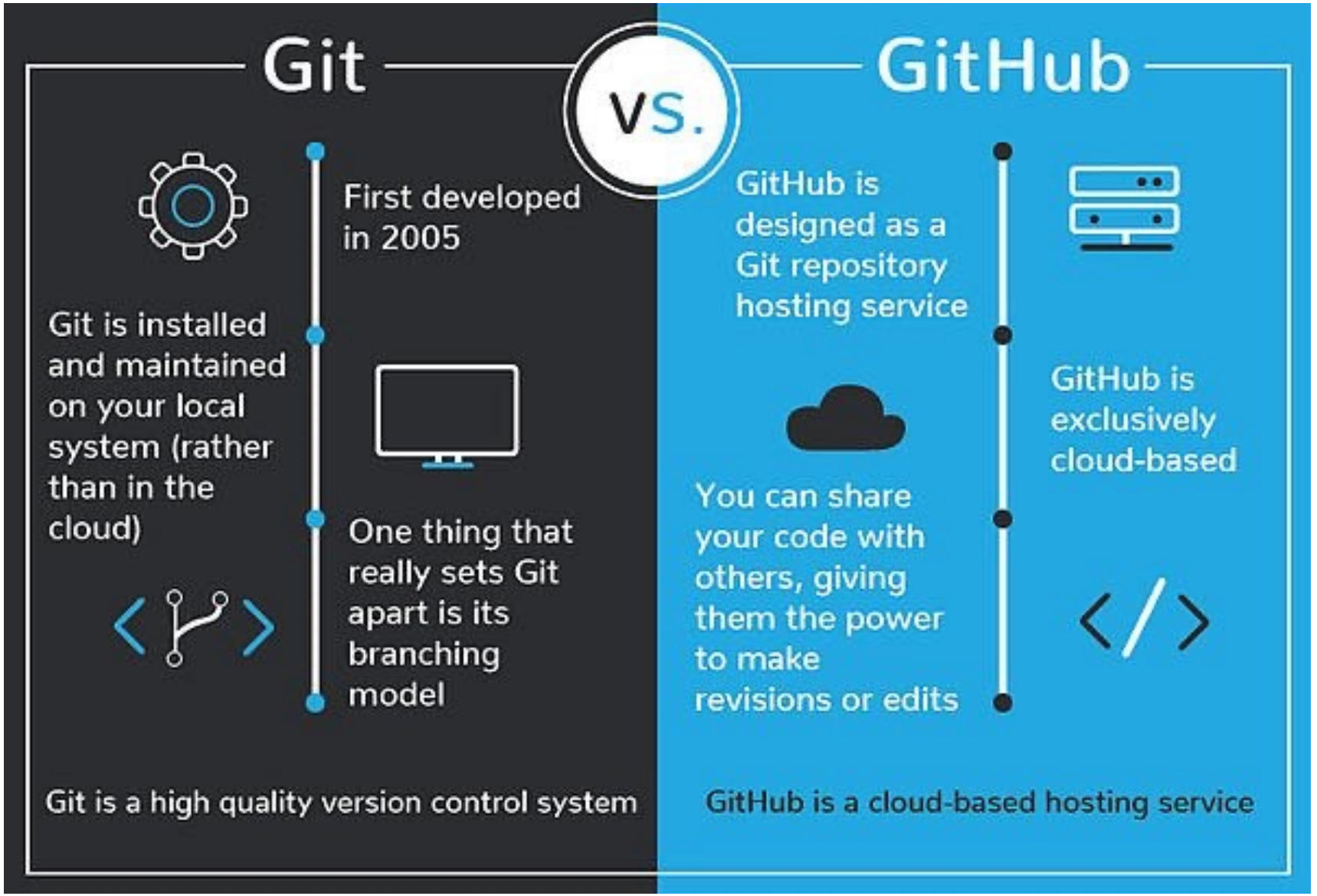
Git



Git

- Logiciel le plus couramment utilisé pour suivre les modifications apportées à un ensemble de fichiers
- Généralement utilisé pour coordonner le travail des programmeurs qui développent en collaboration le code source pendant le développement d'un logiciel
- Git peut être installé sur ses propres serveurs (*on premise*) ou possibilité d'utiliser une version cloud :
  - Github (propriétaire (Microsoft) mais gratuit)
  - Gitlab (licence MIT, i.e. partiellement libre)
  - Framagit (libre)
  - Gitea (licence MIT, i.e. partiellement libre)
  - Bitbucket (propriétaire)

Git vs. Github



Comment fonctionne Git ?

- 3 zones, 3 ambiances
- Les modifications sont sauvegardées 3 fois

Dossier de travail (Working directory)	**Index (Staging area)**	**Dépôt local (Local repository)**
C'est la zone de travail : les fichiers tout juste modifiés sont ici	Zone qui permet de stocker les modifications sélectionnées en vue d'être commitées	Code commité (validé), prêt à être envoyé sur un serveur distant

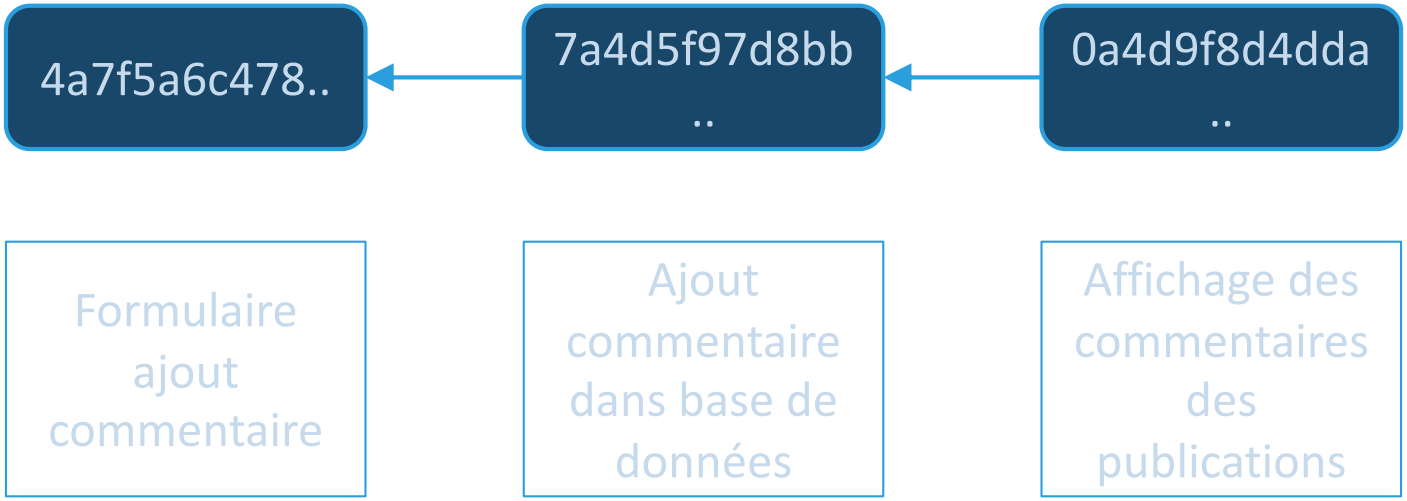
Commit ?

- **Commit** : ensemble de modifications cohérentes du code

- Un bon commit est un commit :
  - Qui ne concerne qu'une seule fonctionnalité du programme
  - Le plus petit possible tout en restant cohérent
  - Idéalement qu'il compile seul

- C'est quoi concrètement un commit ?
  - une différence (ajout / suppression de lignes)
  - des métadonnées (**titre**, hash, auteur)

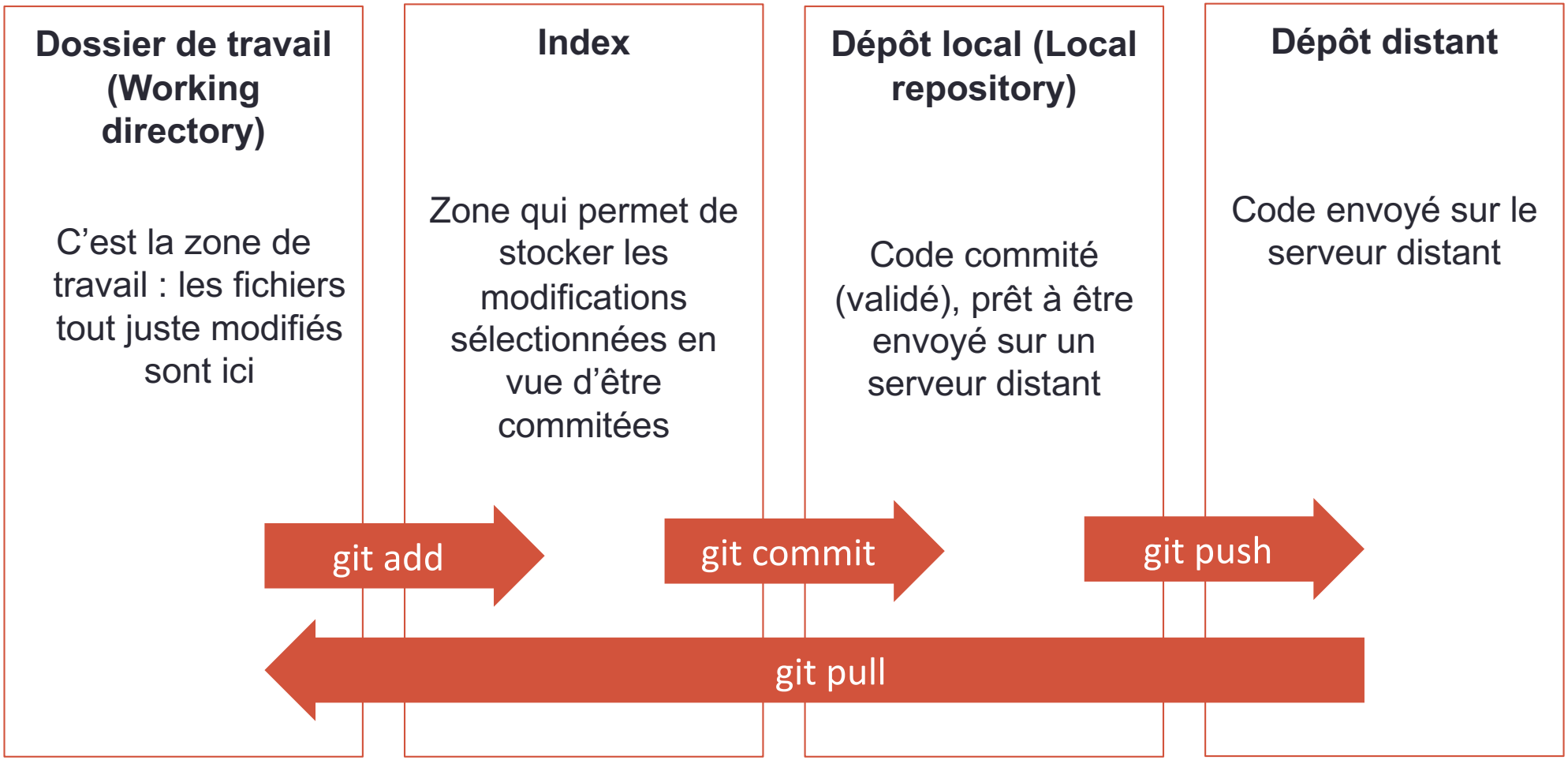
Arbre de commit



Principales commandes Git

Commande	Description
`clone` ***`url [dir]`***	Créé un dossier et récupère tous les fichiers d'un dépôt git pour pouvoir y ajouter du code
`add` ***`files`***	Ajoute le contenu du fichier à la zone d'index
`commit`	Enregistre un instantané (snapshot) de la zone d'index.
`diff`	Montre les différences de code entre 2 versions du même fichier
`pull`	Récupère les commits du dépôt distant et essaye de fusionner les fichiers dans le dépôt local
`push`	Envoi le dépôt local sur le dépôt distant

Principales commandes Git

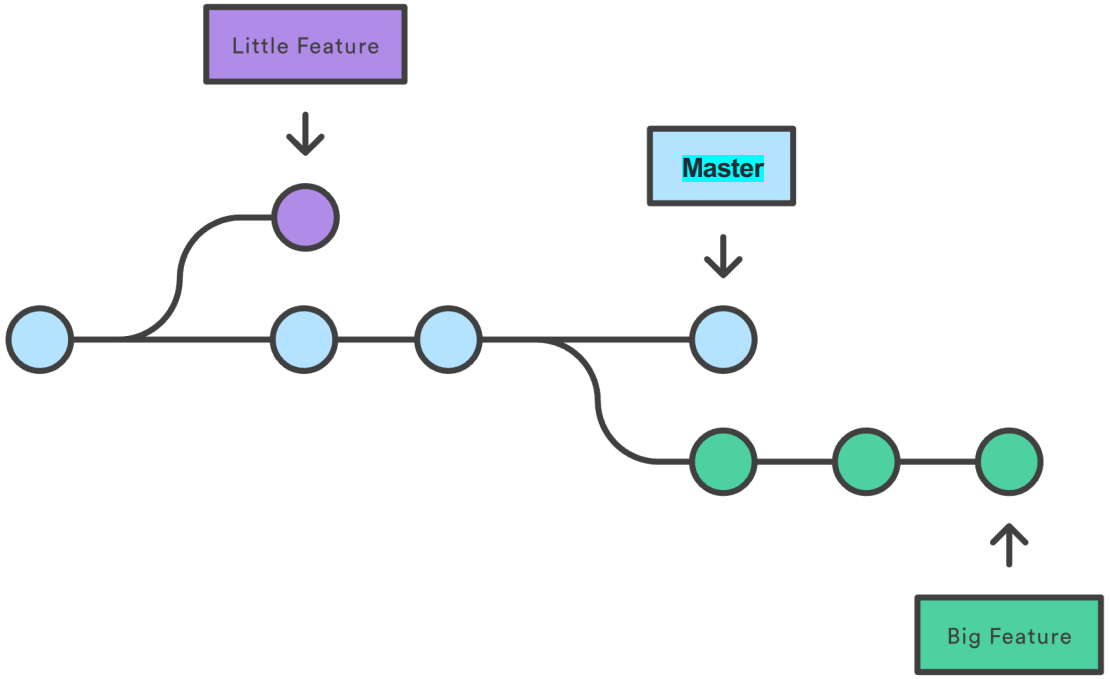


Importance du commentaire lié au commit

- **IMPORTANT : quand vous commitez, il est indispensable de saisir un commentaire pertinent !!!!!**

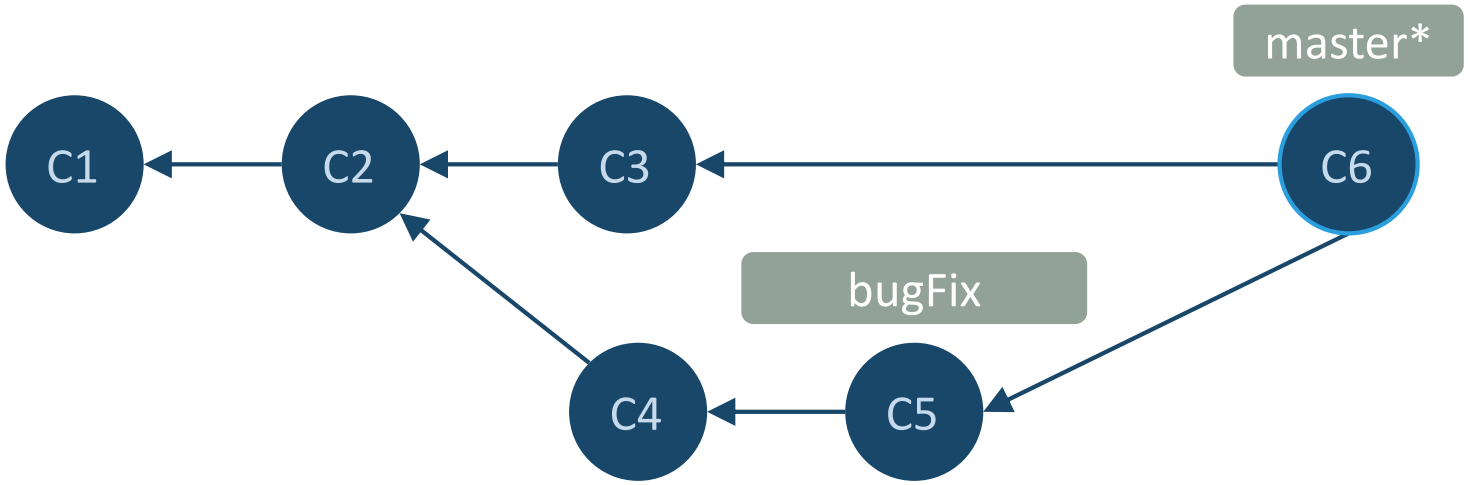
Branche

- Une branche représente une **ligne de développement indépendante**.
- Lorsque vous souhaitez ajouter une nouvelle fonctionnalité ou correction de bug (quelle que soit sa taille), vous créez une branche pour encapsuler vos changements.
- Permet de travailler sur **plusieurs fonctionnalités en parallèle** et évite de **polluer la branche principale (master) avec du code douteux**.
- **Branche : pointeur vers un commit**
- 1 branche principale : master
- En général, une branche par fonctionnalité en cours de développement



Branches : merge

- Une fois la fonctionnalité codée et validée ou le bug corrigé, il faut fusionner (merge) la branche actuelle avec la branche master.
- Merge : intégration des modifications d'une branche dans la branche master
- **`git merge`** `ma_branche:` *`merge ma branche dans la branche courante`*



La zone de bordel : Stash

- Le stash sert à :
  - Sauvegarder les modifications du dossier de travail (working directory) dans une zone tampon pour rendre le working directory propre.
  - Possibilité d'appliquer les modifications stashées n'importe où
  - Peut être vu comme une zone de brouillons
- Plutôt adapté à des gros développements

Fichier .gitignore

- Permet de configurer Git pour ignorer des fichiers spécifiques souvent liés à l'environnement de développement
- Dans le cas de Visual Studio, il faut toujours définir le **modèle .gitignore** à **Visual Studio**

Quels services cloud Git utiliser avec Visual Studio ?

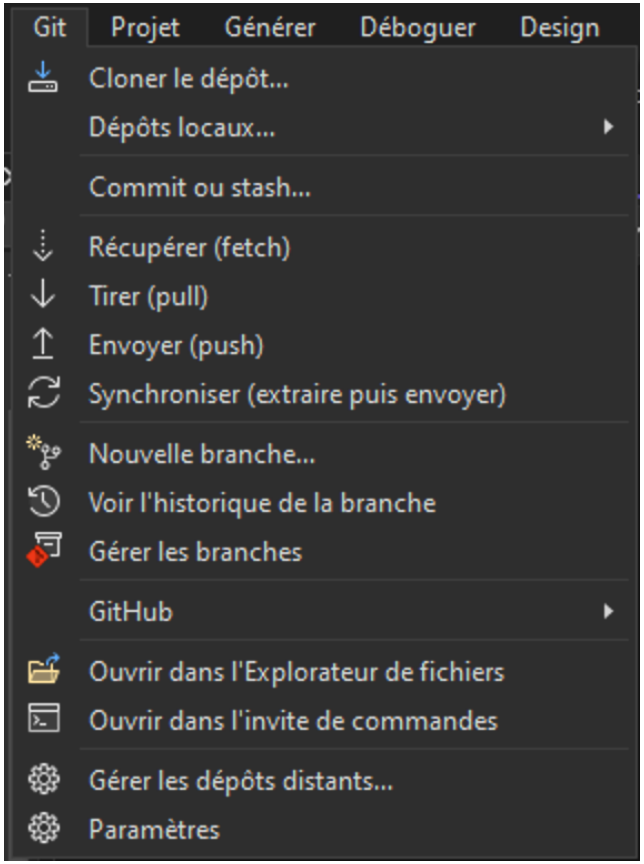
- Tous les services cloud Git (Github, Gitlab, etc.) sont utilisables avec Visual Studio mais depuis la version 2022, il est plus simple d'utiliser **Github** car directement intégré à l'environnement de développement (pas besoin d'installer d'extension).

Visual Studio & Github ?

- Visual Studio 2019 + Github (vidéo en français) :
  - https://www.youtube.com/watch?v=p4Y0WvpEGgU
  - Attention en version 2022, il n'est plus utile d'installer une extension.
  - Les fenêtres et menus ont également changé (cf. slides suivants)
- Visual Studio 2022 + Github (vidéos US) :
  - https://www.youtube.com/watch?v=yDIncg-5c8Y
  - https://www.youtube.com/watch?v=BWqpTpo1kfw

Modifications VS 2022 par rapport à la vidéo en français

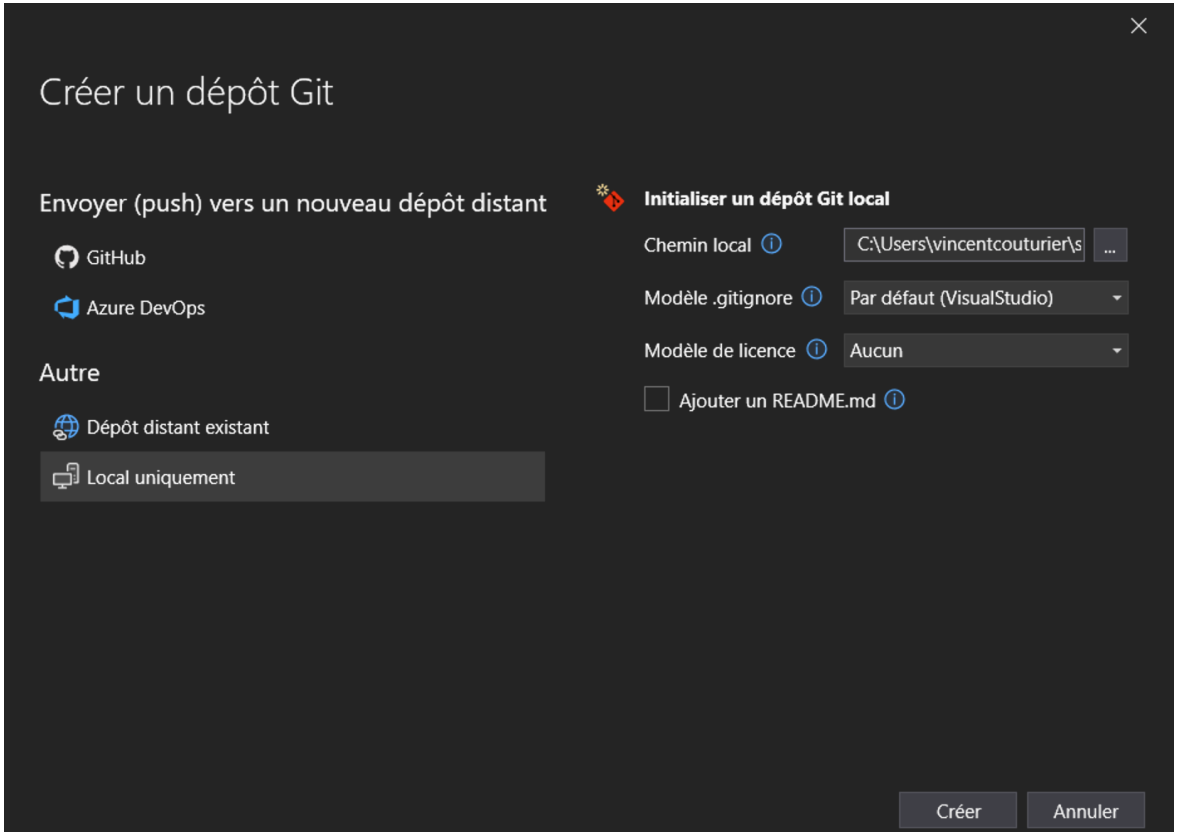
- Menu Git :



Modifications VS 2022 par rapport à la vidéo en français

- Création d'un dépôt (bouton en bas à droite de la fenêtre VS) :

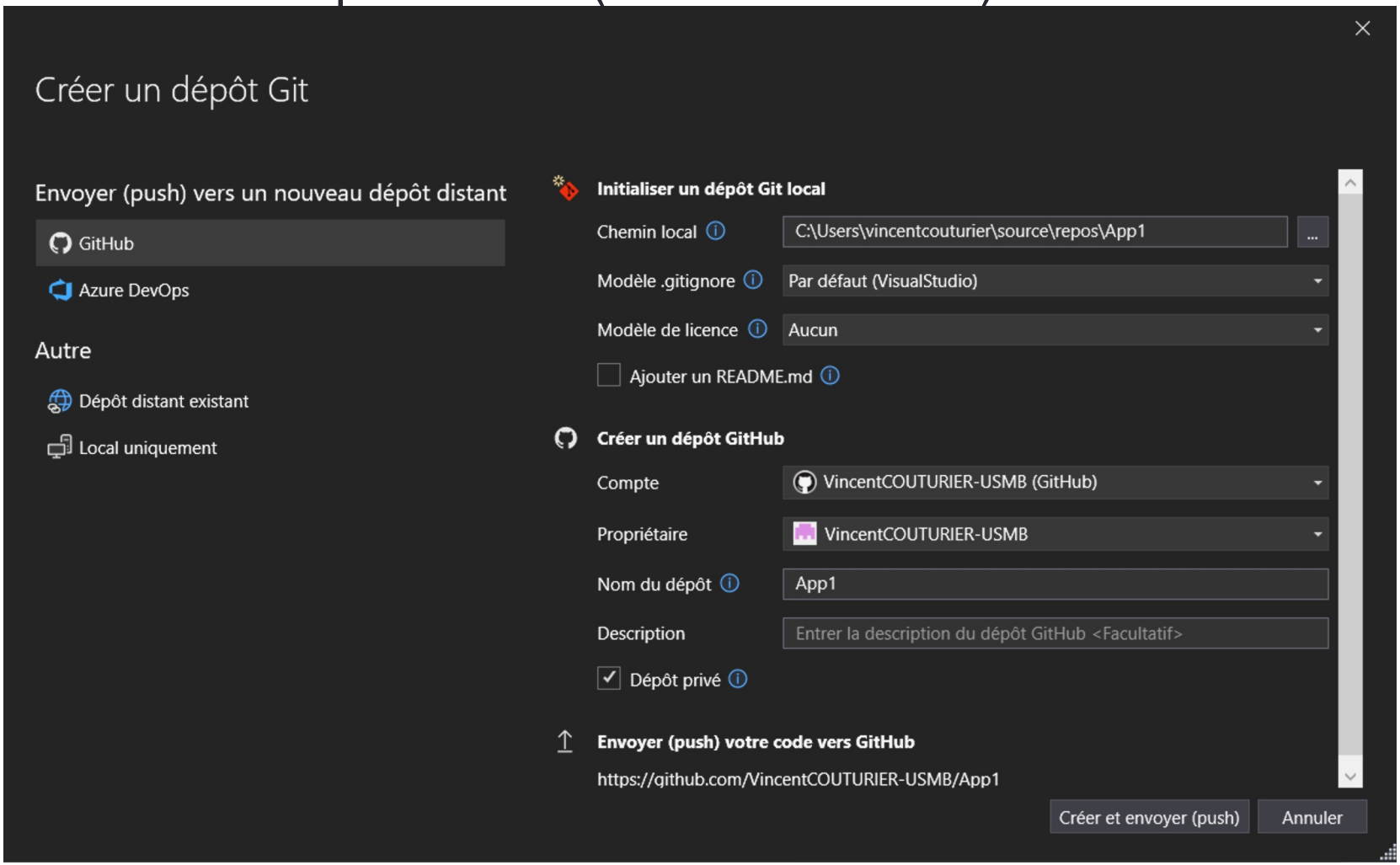
- Création d'un dépôt local :



Attention à bien choisir le modèle .gitignore « Visual Studio »

Modifications VS 2022 par rapport à la vidéo en français

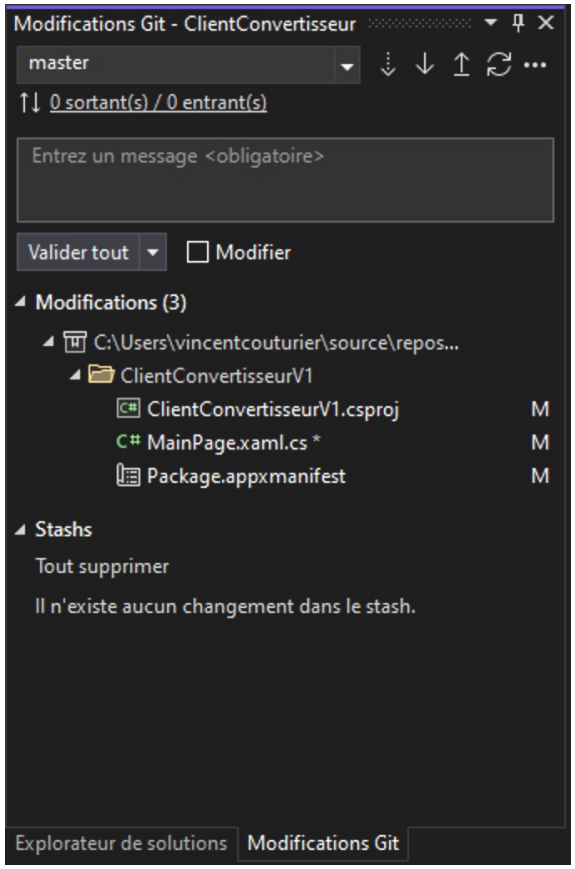
- Création d'un dépôt distant (mais aussi local) :

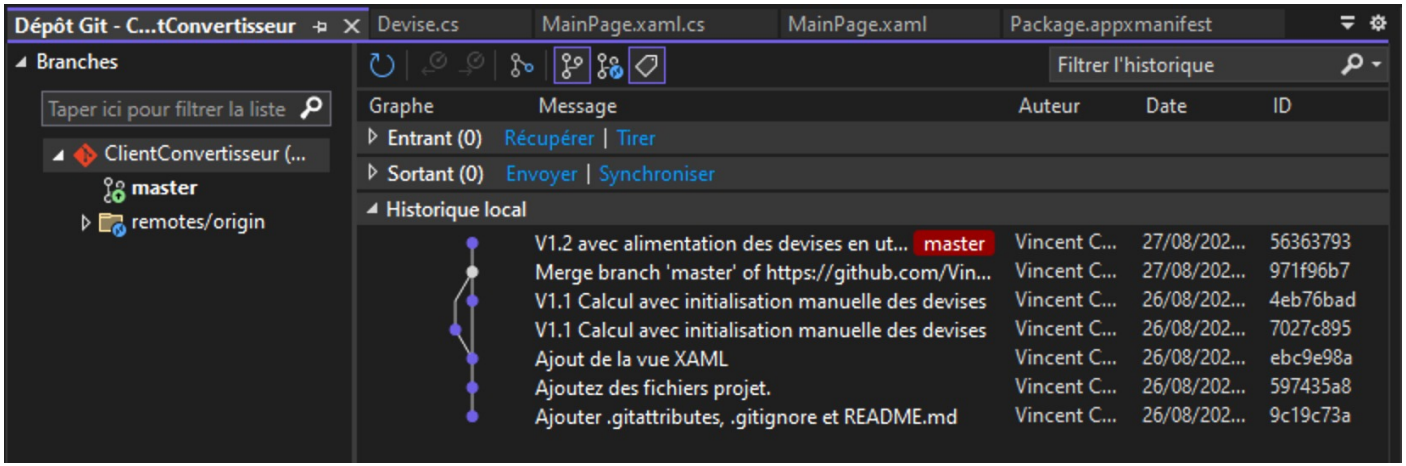


Attention à bien choisir le modèle .gitignore « VisualStudio »

Modifications VS 2022 par rapport à la vidéo en français

- Fenêtres « Modification Git » et « Dépôt Git »





La fenêtre Team Explorer existe toujours mais n'a plus d'intérêt

INTEGRATION CONTINUE / DEPLOIEMENT CONTINU

Intégration continue (CI)

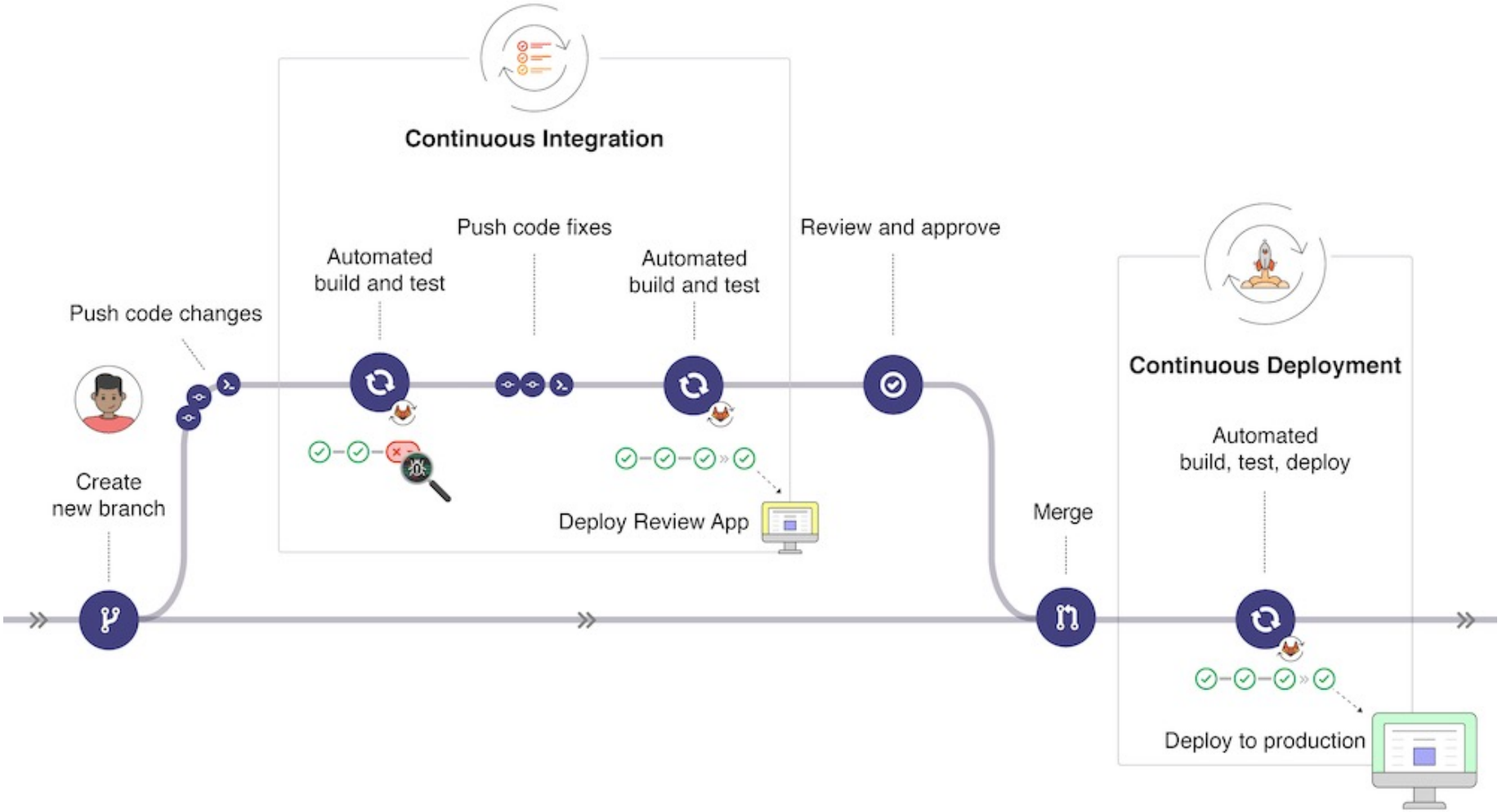
- Intégration : consiste à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée (intégration de chaque composant).
- De nombreuses options existent, entre le déclenchement manuel et la build automatique intégrée qui se déclenche à chaque commit (intégration continue).
- Si intégration continue, l'outil de build va exécuter les étapes du processus de génération :
  1. Obtention des fichiers (autres composants) depuis le contrôle de code source (par exemple, Github)
  2. Configuration de l'espace de travail local
  3. Compilation du code source
  4. Exécution des tests (UT, IT, E2E, tests de performance)
  5. Enfin, fusion des fichiers dans le contrôle de code source, si pas de code incompatible ou manquant
- => builds d'intégration continue compilent et testent votre application automatiquement après toute modification de code + Reporting
- IC est une des bonnes pratiques de l'Extreme Programming (XP)
- Outils : Jenkins (principalement utilisé avec Java), CruiseControl, Gitlab CI, Github Actions, Microsoft Azure DevOps.

Déploiement continu (CD)

- Les fonctionnalités logicielles sont livrées fréquemment par le biais de déploiements automatisés.
- = Intégration continue + déploiement automatisé sur un serveur (de test, pré-prod ou prod).
- Exemple : Utilisation d'Azure DevOps pour déployer sur le cloud Azure (App Engine).
- Déploiement continu != livraison continue :
  - Approche similaire dans laquelle des fonctionnalités logicielles sont également livrées fréquemment et considérées comme pouvant potentiellement être déployées
  - MAIS le processus de déploiement reste manuel

https://blog.octo.com/5-bonnes-raisons-de-deployer-en-continu/

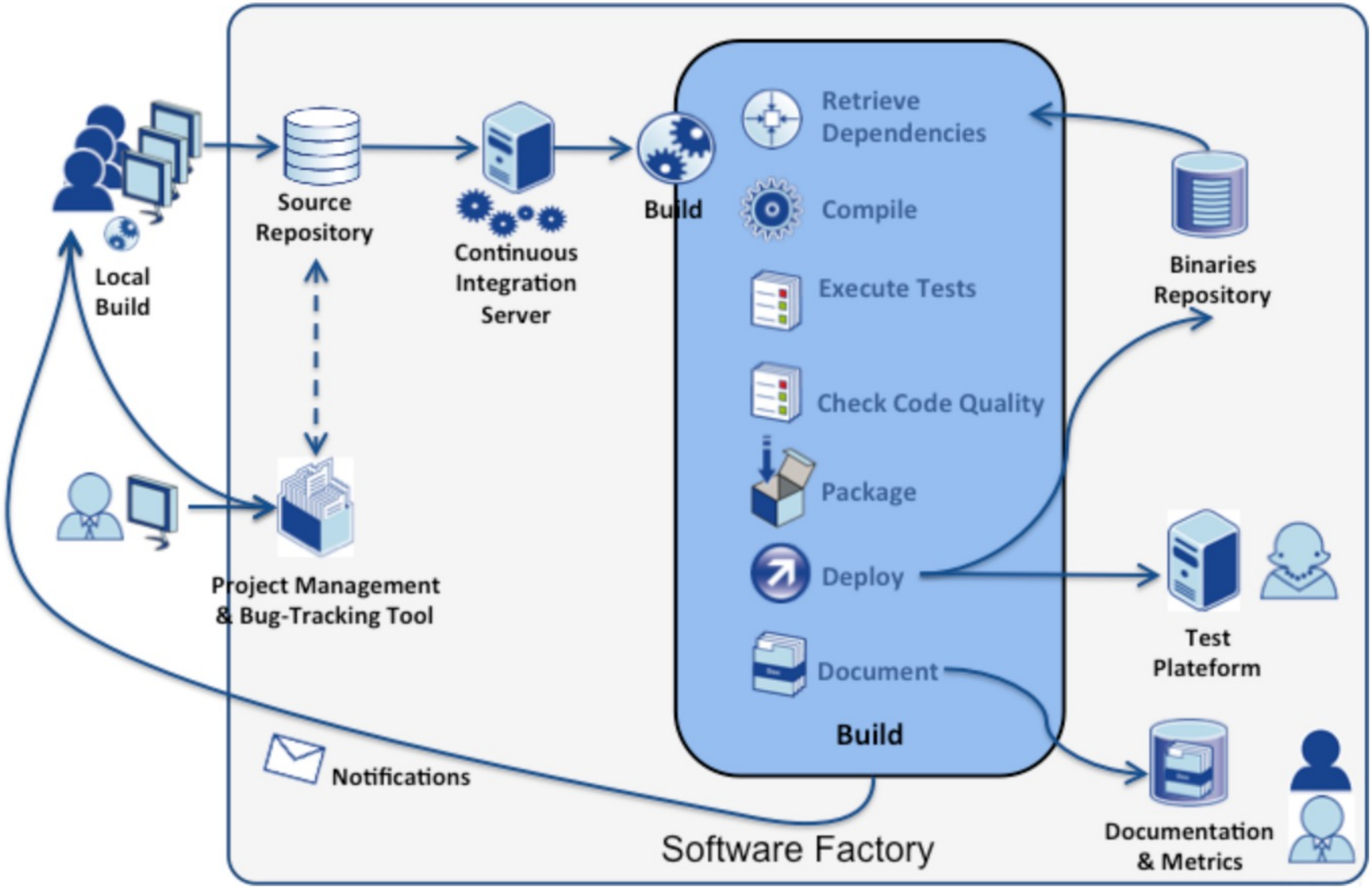
CI/CD


Continuous Integration
Push code fixes
Review and approve
Automated
build and test
Automated
build and test
Push code changes
Create
new branch
Deploy Review App
Merge
Continuous Deployment
Automated
build, test, deploy
Deploy to production

Usine logicielle (*software factory*)

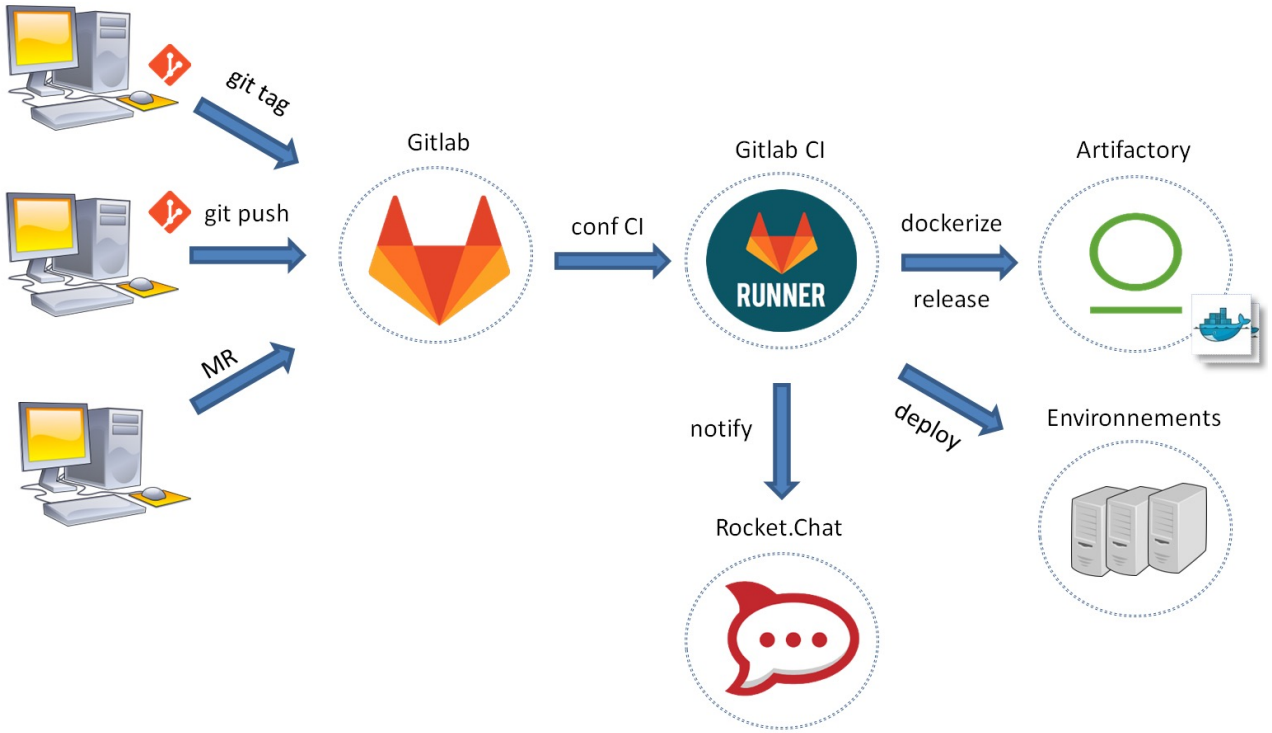
- Principe : l'organisation est découpée comme une chaîne de production où les tâches répétitives sont automatisées comme le lancement régulier de la compilation, des tests unitaires, du déploiement.
- Gère :
  - La gestion de projet (SCRUM, Kanban)
  - L’expression des besoins
  - Le contrôle de version et du code source
  - La gestion des plans de tests
  - L’intégration continue (CI)
  - Le déploiement continu (CD)
  - Le reporting (tableaux de bord)

Usine logicielle (*software factory*)



Usine logicielle (*software factory*)

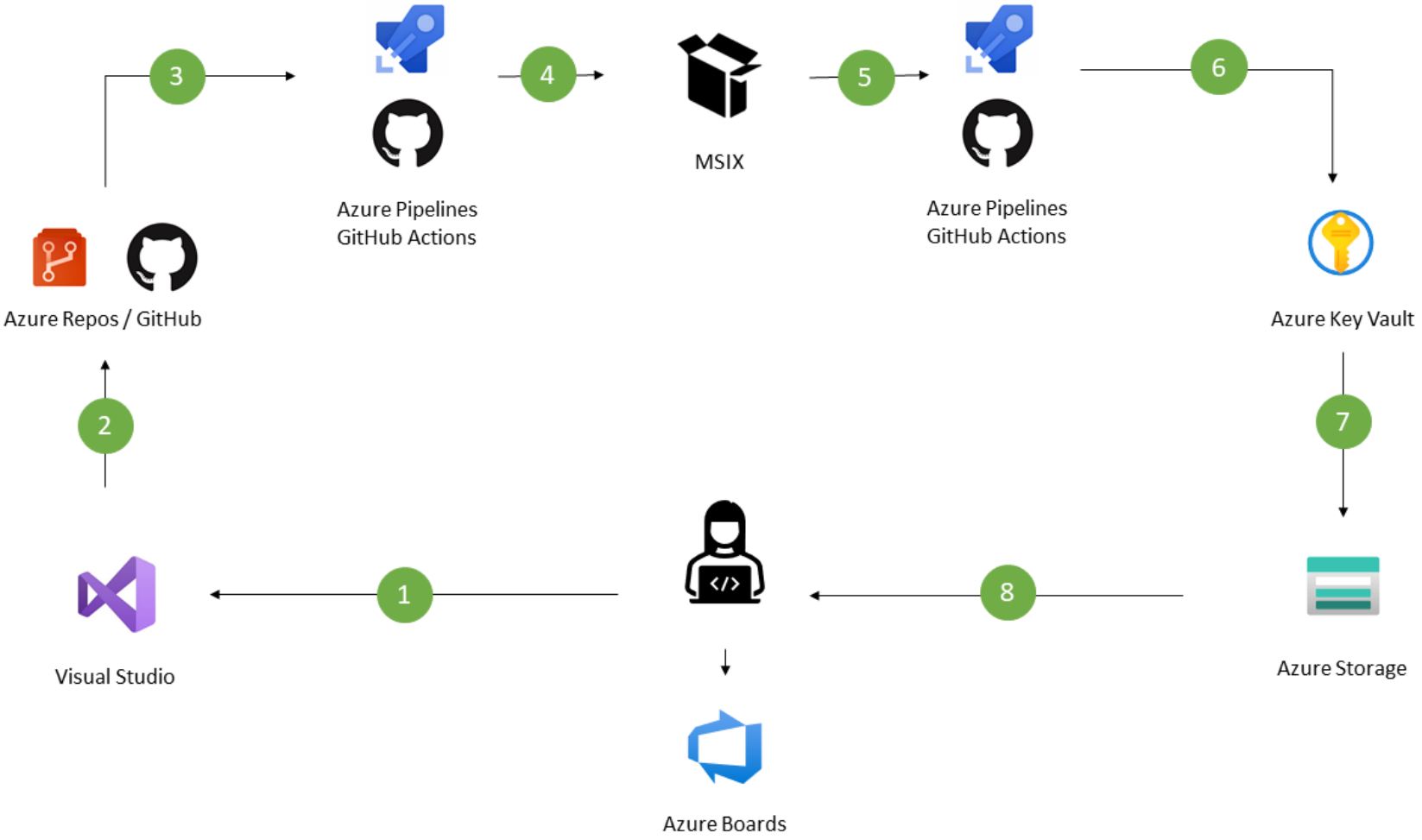
- Exemple : GitLab



JFrog Artifactory : gestionnaire de dépôts binaires, i.e. outil dédié à l'optimisation du stockage et du téléchargement de fichiers utilisés dans un projet de développement.
Rocket.Chat : Plateforme de collaboration d'équipe de type chat. Equivalent de Slack.

Usine logicielle (*software factory*)

- Exemple : Azure DevOps :
  - https://lesdieuxducode.com/blog/2018/11/monter-une-usine-logicielle-avec-azure-dev-ops
  - https://docs.microsoft.com/fr-fr/azure/architecture/example-scenario/apps/devops-dotnet-webapp

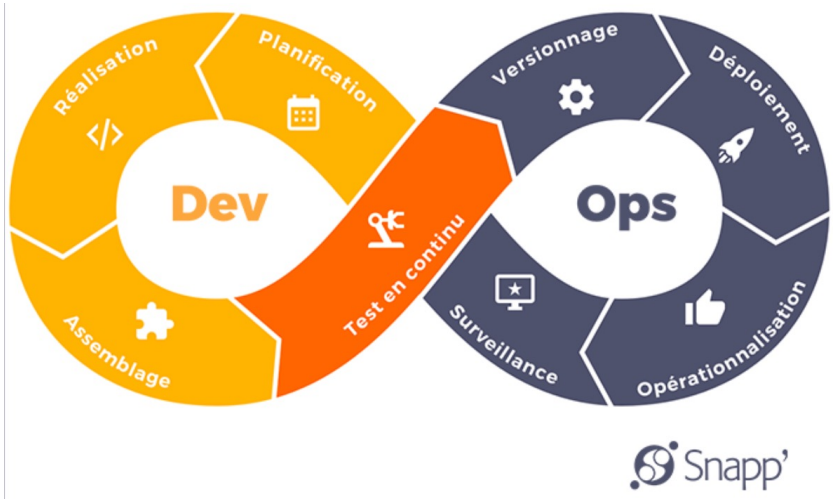


Usine logicielle (*software factory*)

- Exemple : Azure DevOps :

1. Un développeur modifie le code de l'application.
2. Le code est validé dans un référentiel de code source, qui peut être hébergé sur Azure Repos ou GitHub.
3. L'intégration continue est déclenchée par un pipeline qui exécute des tests et qui génère la solution. Le pipeline peut être hébergé sur Azure Pipelines ou GitHub Actions.
4. Le pipeline génère un package MSIX non signé.
5. Le déploiement continu est déclenché par une autre étape du pipeline. Le package MSIX est déployé sur votre plateforme de distribution.
6. Le package MSIX est signé avec un certificat approuvé, et le certificat est stocké dans Azure Key Vault. Un package MSIX doit être signé avec un certificat approuvé, sinon Windows ne pourra pas installer l'application. La signature doit être effectuée pendant le déploiement du package. Key Vault protège votre certificat de signature de code en veillant à ce que les développeurs malveillants ne puissent pas le voler et l'utiliser pour signer d'autres applications.
7. Le package MSIX signé est déployé sur un site web statique, hébergé sur Stockage Azure, avec un fichier Programme d'installation d'application. Ce fichier décrit l'application et les options de mise à jour.
8. Chaque fois qu'un développeur valide de nouvelles mises à jour dans le référentiel, un nouveau package MSIX est généré et envoyé (push) vers le même emplacement Stockage Azure. La technologie Programme d'installation d'application garantit que ce nouveau déploiement déclenche le processus de mise à jour automatique, de sorte que vos clients disposent toujours de la version la plus récente de votre application.

DevOps ?



- = Pratique visant à l'unification du développement logiciel (*dev*) et de l'administration des infrastructures informatiques (*ops*)
- Référentiel de bonnes pratiques en constante évolution basées sur les pratiques agiles et Lean (« Gestion sans gaspillage » ou « Gestion allégée »)
- L'objectif est de livrer des logiciels plus rapidement, de façon plus efficace, tout en garantissant la meilleure qualité et ce en adéquation avec les besoins de l'entreprise et de ses clients.
- Repose sur l'automation et le suivi (monitoring) de toutes les étapes de la création d'un logiciel, depuis le **développement**, **l'intégration**, les **tests**, la **livraison** jusqu'au **déploiement**, **l'exploitation** et la **maintenance** des infrastructures => **cycles de développement plus courts**, une **augmentation de la fréquence des déploiements** et des **livraisons continues**