Chapitre 3
Repos : gestion du code source

1. Les deux modes de gestion

Lorsque vous créez votre premier projet, par défaut Azure DevOps présélectionne le gestionnaire de code source Git. Bien que Git soit aujourd'hui le gestionnaire de code source le plus utilisé, Azure DevOps propose tout de même la possibilité d'utiliser TFVC (*Team Foundation Version Control*).

Avant d'étudier concrètement l'aspect pratique de ces deux modes de gestion du code source, il convient d'expliquer la différence fondamentale entre leurs deux fonctionnements.

1.1 Centralisé vs décentralisé

Ce qui distingue majoritairement Git de TFVC est la notion de centralisation.

1.1.1 Contrôle de code source centralisé

Un code source centralisé dépend d'une autorité unique, généralement un serveur. En conséquence, le gestionnaire de code source dialogue systématiquement avec le serveur à chaque action que l'on souhaite réaliser (envoyer du code, travailler sur un fichier, etc.).

Cette centralisation impose des contraintes majeures :

– Le lien avec le serveur doit être permanent, car c'est lui qui autorise ou refuse les demandes. Certains contrôles de code source permettent de travailler en mode déconnecté de façon temporaire, mais in fine, le lien avec le serveur doit être effectué.
– Toutes les actions réalisées par le développeur sont tracées en temps réel. Ainsi, chaque extraction de fichier pour une modification est notée, et chaque événement important est historisé.

Bien qu'il soit compliqué de travailler de façon déconnectée avec un contrôle de source centralisé, la centralisation offre des possibilités non négligeables :

– Du fait que le serveur est l'autorité principale, il existe un point unique de configuration pour gérer la façon dont l'équipe travaille avec le code source. Il est possible, par exemple, de définir les modes d'accès à un fichier (concurrent ou exclusif).
– La centralisation permet une sécurité allant jusqu'aux fichiers. Il est ainsi possible de définir, sur le serveur, des droits spécifiques à chaque utilisateur, de façon très fine.
– Étant donné que tout est centralisé et historisé, les administrateurs disposent d'un suivi conséquent pour voir tout ce qui a été entrepris.

Un contrôle de code source centralisé offre donc une grande flexibilité sur la sécurité et les droits des utilisateurs, en imposant des contraintes de communication avec le serveur à chaque développeur.

1.1.2 Contrôle de code source décentralisé

À l'inverse, un contrôle de code source décentralisé offre la possibilité au développeur de récupérer l'intégralité du dépôt sur sa machine et d'être totalement autonome dans la gestion de son travail.

Étant donné qu'il n'existe pas d'autorité unique, la seule contrainte est le serveur commun, qui contient le code source et permet de synchroniser les modifications entre les différents développeurs de l'équipe. De ce fait, il faut que chaque développeur travaille de façon intelligente afin que son travail puisse être facilement intégré dans le code source commun (nous verrons une technique de travail permettant cela à la section Git-flow de ce chapitre).

Le fait que chaque développeur ait sur sa machine une version du dépôt commun indique également que le serveur n'a connaissance du travail du développeur que lorsque ce dernier décide d'envoyer son travail. Ainsi, même s'il est possible de sécuriser le serveur commun dans une certaine mesure, il est tout simplement impossible de limiter ce que fait le développeur sur sa machine, une fois que le dépôt a été cloné.

Naturellement, un contrôle de code source décentralisé est pensé pour un travail en mode déconnecté.

1.2 Zoom sur TFVC

TFVC est le contrôle de code source centralisé proposé par Azure DevOps. Assez célèbre aux débuts du développement avec .NET, il est aujourd'hui largement moins utilisé, car trop contraignant pour les usages modernes et le nomadisme.

Néanmoins, certaines entreprises continuent de l'exploiter pour garantir une bonne intégrité de leur code source et gérer de façon fine la sécurité de ce dernier.

Parmi les avantages de TFVC, citons :

– Le suivi des changements facilité, car comme tout contrôle de code source centralisé, il suffit de consulter l'activité sur le serveur pour voir en temps réel le travail des développeurs (qui a extrait quel fichier, etc.).
– La sécurité centralisée, qui permet un niveau de paramétrage extrêmement fin.
– La gestion des dépôts de tailles diverses, avec des fichiers de tailles variées, sans aucune difficulté (le serveur doit bien entendu être assez bien dimensionné, ce qui est le cas des serveurs Azure DevOps).
– La maturité et la stabilité de TFVC. Cet outil ancien est considéré comme terminé par les équipes de Microsoft, ce qui implique qu'il n'y aura plus de modifications pouvant impacter son utilisation. Il est aujourd'hui largement éprouvé.

Cependant, TFVC reste un contrôle de code source centralisé, ce qui implique notamment que le support du mode déconnecté, bien qu'existant partiellement, peut rendre difficile le développement en équipe. Comme cela a été dit, le produit est considéré comme mature et stable. Le corollaire de cette affirmation est que rien ne sera entrepris pour mieux prendre en charge le mode déconnecté. TFVC impose donc un mode de travail également centralisé, où l'équipe tout entière est en mesure de dialoguer de façon constante avec le serveur.

Un autre point qui peut être délicat est la gestion des branches avec TFVC. En effet, le système de fichiers est utilisé intégralement pour gérer les branches. Cela veut dire que lorsqu'une branche est créée (pour isoler son travail ou créer une nouvelle version), l'intégralité du dossier comportant le code source est dupliquée dans un dossier. L'opération est suffisamment rapide sur un bon serveur, sans pour autant être instantanée, car cette duplication prend du temps. De surcroît, cela entraîne une consommation bien plus importante de l'espace disque pour gérer les multiples branches.

Finalement, étant donné que TFVC n'est pas prévu pour travailler en mode déconnecté, le processus de mise à disposition du code produit pour être revu n'est pas optimal. En effet, chaque utilisateur procède à l'extraction du fichier sur lequel il veut travailler, ce qui implique que si quelqu'un d'autre veut travailler dessus, il lui est déjà notifié qu'un utilisateur est en train de le modifier. Il est possible de configurer TFVC pour fonctionner en mode exclusif, ce qui implique qu'une seule personne à la fois peut modifier un fichier donné. Néanmoins, TFVC offre une fonctionnalité permettant à l'utilisateur de mettre ses changements « sur étagère » (c'est le terme utilisé par l'outil) afin qu'un membre de l'équipe récupère ces derniers pour les vérifier et, éventuellement, les intégrer dans le tronc commun. Cette gestion, bien qu'existante, n'est cependant pas optimale, car elle n'offre pas d'interface centralisée et ne facilite pas davantage l'intégration du code dans le tronc commun, qui nécessite de toute manière des droits et les dernières mises à jour.

En conclusion, TFVC a déjà été éprouvé et il a montré sa robustesse, mais nous n'utiliserons pas cet outil, car il ne répond plus aux problématiques modernes. La documentation officielle de Microsoft est assez fournie pour répondre à vos éventuelles questions, notamment sur son utilisation dans Azure DevOps.

1.3 Zoom sur Git

Git est, de loin, le contrôle de code source le plus utilisé dans l'industrie informatique de nos jours. S'il a réussi à s'imposer face aux solutions qui étaient disponibles initialement, comme TFVC ou SVN (*Subversion*), c'est pour sa légèreté et sa décentralisation.

En effet, Git est totalement décentralisé. Il est tout à fait possible d'utiliser le contrôle de code source sur sa machine sans communiquer avec un serveur distant. En cela, Git est l'outil parfait pour travailler hors connexion car, même en présence d'un serveur commun, toutes les modifications et toutes les manipulations classiques (extraction de code, création de branches, etc.) peuvent être réalisées sans l'accord préalable du serveur distant. Bien entendu, ces manipulations nécessitent un formalisme un peu particulier pour éviter les pertes de code et les conflits en cas de travail en équipe (cf. section Git-flow de ce chapitre).

De ce fait, les avantages de travailler avec Git sont assez évidents :

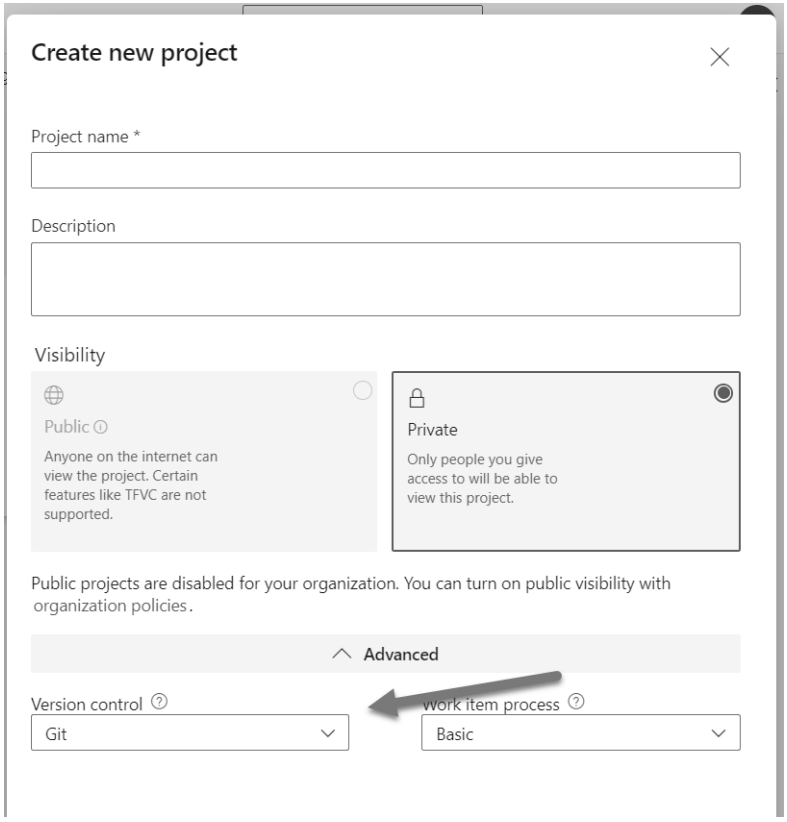
– L'absence de connexion permanente avec le serveur facilite un mode de travail libre (que ce soit de façon nomade, ou sédentaire sans connexion).
– La gestion des branches est extrêmement rapide.
– L'approche décentralisée crée par nature de multiples branches, et le processus de réunification, intégré nativement, facilite les revues de code.
– Chaque membre de l'équipe possède une copie du dépôt de code source, incluant tout l'historique.
– Étant donné qu'il s'agit de l'outil le plus utilisé actuellement, il existe une importante communauté et de nombreuses ressources sur le Web.

Le reproche principal que l'on pourrait faire à Git est la quasi-impossibilité d'exercer des contraintes de sécurité aussi fines que celles qui existent sur TFVC. Il est bien entendu possible de sécuriser le serveur à cet effet (et nous expliquerons comment mettre en place plusieurs stratégies au niveau d'Azure DevOps dans la section Définition des attributs des branches principales de ce chapitre), mais la granularité fine qu'offre TFVC n'est pas atteignable.

Cependant, l'approche Agile met en avant la confiance dans les équipes, et il est assez courant aujourd'hui de voir des équipes fonctionner avec une transparence totale sur ce que fait chaque membre, ce qui va dans le sens de l'outil.

2. Initialiser le dépôt de code source

Lorsque vous créez un projet avec Azure DevOps, il vous est demandé de choisir le contrôle de code source associé. Git est sélectionné par défaut. Pour faire un éventuel changement vers TFVC, dépliez la partie **Advanced**, et dans le champ **Version control**, choisissez **TFVC** :



Fenêtre de création de projet avec le choix du contrôle de code source

Choisir Git à la création du projet indique simplement que le contrôle de code source sera Git, sans pour autant créer d'éléments.

Si vous ouvrez le projet **MonPremierProjet** et cliquez sur l'élément de menu **Files** dans la section **Repos**, une page d'assistant s'affiche, vous permettant d'initialiser votre dépôt Git. Il y a ici plusieurs options pour initialiser le dépôt :

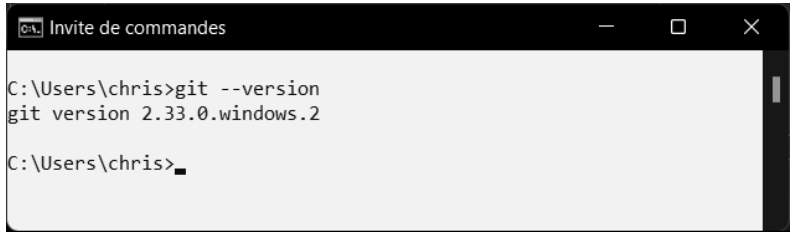
– Cloner le dépôt vide sur votre ordinateur local pour l'initialiser. L'avantage de cette approche est que le lien avec le serveur distant, ici Azure DevOps, est déjà initialisé.
– Créer et initialiser un dépôt Git sur votre ordinateur, puis l'envoyer vers Azure DevOps afin de l'utiliser.
– Importer un dépôt Git depuis un autre projet Azure DevOps.
– Demander à Azure DevOps d'initialiser le dépôt Git avec un fichier readme, et éventuellement un fichier gitignore, puis le cloner sur votre ordinateur local.

Dans la section suivante, nous allons voir les quelques concepts de base autour de Git. Si vous êtes utilisateur de Git et connaissez les commandes essentielles, passez à la section Git-flow.

2.1 Les commandes de base de Git

Pour pouvoir travailler avec Git, il est nécessaire d'installer l'outil localement. Le site officiel de Git (https://git-scm.com) indique les étapes à suivre afin d'installer Git sur votre ordinateur. Sur la partie droite du site, dans une image représentant un écran, un bouton permet de télécharger Git selon votre système d'exploitation. Selon le cas, vous pourrez télécharger l'installeur ou avoir accès aux lignes de commandes permettant d'installer l'outil.

Suivez la procédure proposée par votre environnement. Une fois Git installé, ouvrez une invite de commande, écrivez la commande `git --version`, puis appuyez sur [Entrée]. La version de Git installée s'affiche :



Affichage de la version de Git

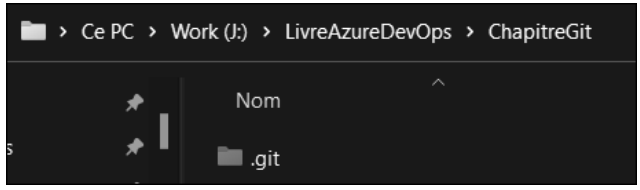
2.1.1 Créer le premier dépôt

Pour créer un dépôt, il faut se placer dans un dossier vide, et taper la commande `git init`.

D'une part, le résultat sur la console indique que Git a bien créé le dépôt :

```
Initialized empty Git repository in J:/LivreAzureDevOps/ChapitreGit/.git/
```

D'autre part, un dossier caché appelé .git a été créé dans le dossier où vous avez exécuté la commande :



Dossier contenant les informations du dépôt

Dans ce dossier se trouve la totalité des objets dont Git a besoin pour travailler, notamment tout l'historique du dépôt ainsi que les branches existantes. Dans le cas présent, étant donné que le dépôt est vide, la taille de ce dossier est relativement petite.