

Git : Utilisation et Collaboration

⌚ Date de création	@28 octobre 2024 09:29
☰ Étiquettes	Workshop
👤 Crée par	 Pierre Fontaine

[Introduction](#)

[Historique](#)

[Une structure bien définie](#)

[Partie 1 : Préparation de l'environnement](#)

[Partie 2 : Configuration de GIT](#)

[Partie 3 : Premier projet, initialisation et utilisation de GIT](#)

[3.1 Création d'un répertoire de travail](#)

[3.2 Initialisation de GIT](#)

[3.3 Créons notre premier fichier Markdown](#)

[3.4 Ajoutons et Validons notre premier fichier dans Git](#)

[3.5 Une nouvelle version en devenir](#)

[3.6 Vérifier le statut du répertoire local](#)

[3.7 Consultons l'historique](#)

[3.8 Travailons avec les branches](#)

[3.9 Annulons une modification](#)

[3.10 Une fusion de branche](#)

[Partie 4 : Jeu de Rôle : Collaboration sur le Projet "Documentation de Projet"](#)

[4.1 Participants](#)

[4.2 Objectif](#)

4.3 Étapes du Jeu de Rôle

- Étape 1 : Configuration Initiale
- Étape 2 : Protection de la Branche Principale
- Étape 3 : Ajout de Contenu par Dev1
- Étape 4 : Ajout de Contenu par Dev2
- Étape 5 : Création de Pull Requests
- Étape 6 : Examen par le Lead
- Étape 7 : Fusion des Pull Requests
- Étape 8 : Résolution des Conflits

Partie 5 : Bonus

- 5.1 Un portfolio en ligne ?
- 5.2 Créons le dépôt
- 5.3 Créons et envoyons notre première page
- 5.4 Configuration du dépôt en mode Page
- 5.5 Et maintenant la suite ?

Annexes

- 1. Git et GitHub
- 2. Commandes Git
- 3. Markdown
- 4. Gestion des branches
- 5. Conflits de fusion
- 6. GitHub Pages
- 7. HTML et CSS (pour votre portfolio)
- 8. Frameworks et outils
- 9. Commandes Shell

Introduction

Si tu lis ce mot d'introduction, félicitations ! Cela signifie que tu es entré dans le monde du développement informatique et dans la sphère de la tech.

Depuis des années, les développeurs écrivent des centaines de milliers de lignes de code chaque jour. Mais comment gèrent ils cette quantité de données ? Comment collaborent ils sur un projet ? Tant de questions auxquelles nous allons tenter de répondre dans ce workshop.

Historique

L'histoire du versionnement de fichiers remonte à 1972, avec la création du **Source Code Control System (SCCS)**. À cette époque, tout se faisait exclusivement en ligne de commande, et certaines interfaces étaient très

compliquées. De nombreux autres systèmes de gestion de versions ont suivi. Voici une liste pour votre curiosité, que je vous invite à explorer plus tard :

- **Revision Control System (RCS)** - 1982
- **Concurrent Versions System (CVS)** - 1986
- **Perforce (P4)** - 1995
- **Visual SourceSafe (VSS)** - 1994
- **BitKeeper** - fin des années 1990

Passons maintenant au dernier-né : **GIT**. Son créateur, Linus Torvalds, se heurte en 2005 à des problèmes liés aux licences de BitKeeper, alors utilisé pour le développement du noyau de Linux. Il en vient rapidement à se dire qu'il lui faudra autre chose. Fidèle à l'esprit de Linux, axé sur la liberté et l'accessibilité, il décide de développer son propre outil, avec pour objectifs qu'il soit libre, rapide et efficace.

Maintenant que les bases sont posées, nous allons pouvoir passer à la structure de GIT.

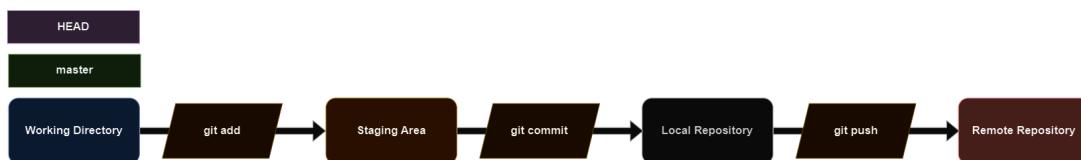
Une structure bien définie

Git utilise une structure de type graphe acyclique dirigé (DAG) pour gérer les versions et les branches. Chaque commit a un identifiant unique (SHA-1 à l'origine), et chaque commit garde une référence vers son prédecesseur, ce qui facilite la gestion des versions et des branches.

Voici une liste des termes spécifiques à Git avec leurs descriptions :

- **Répertoire de travail (Working Directory)** : C'est là où tu modifies et crées tes fichiers, sur ta machine. Les modifications que tu fais dans le répertoire de travail ne sont pas encore suivies par GIT tant que tu ne les ajoutes pas.
- **Index (Staging Area)** : C'est une étape intermédiaire où tu ajoutes les modifications que tu veux sauvegarder. Les fichiers ajoutés ici sont marqués pour le prochain commit, sans pour autant être encore dans l'historique Git.
- **Dépôt local (Local Repository)** : Une fois les modifications prêtes dans l'Index, tu les enregistres dans le dépôt local par un commit. Le dépôt contient l'historique des commits de ton projet.

- **HEAD** : C'est un pointeur indiquant l'endroit où tu te trouves dans l'historique des commits. La plupart du temps, HEAD pointe vers le dernier commit de la branche active.
- **Commit** : C'est un instantané de ton projet dans le dépôt local. Chaque commit conserve les modifications d'un ensemble de fichiers, avec un identifiant unique et un message descriptif.
- **Branch (Branche)** : Une branche est une ligne de développement dans Git. Par défaut, il y a une branche principale souvent nommée `main` ou `master`, et d'autres branches peuvent être créées pour expérimenter ou travailler sur des fonctionnalités spécifiques.
- **Dépôt distant (Remote Repository)** : C'est une version du dépôt hébergée sur un serveur (comme GitHub, GitLab ou Bitbucket). Le dépôt distant permet de sauvegarder et de partager le code avec d'autres collaborateurs.



Mais attention, ce type d'outil ne fait pas tout à votre place. Une rigueur sera nécessaire pour apprivoiser pleinement toute sa puissance.

Partie 1 : Préparation de l'environnement

Commençons par installer git à travers le lien du site officiel : <https://git-scm.com/downloads>

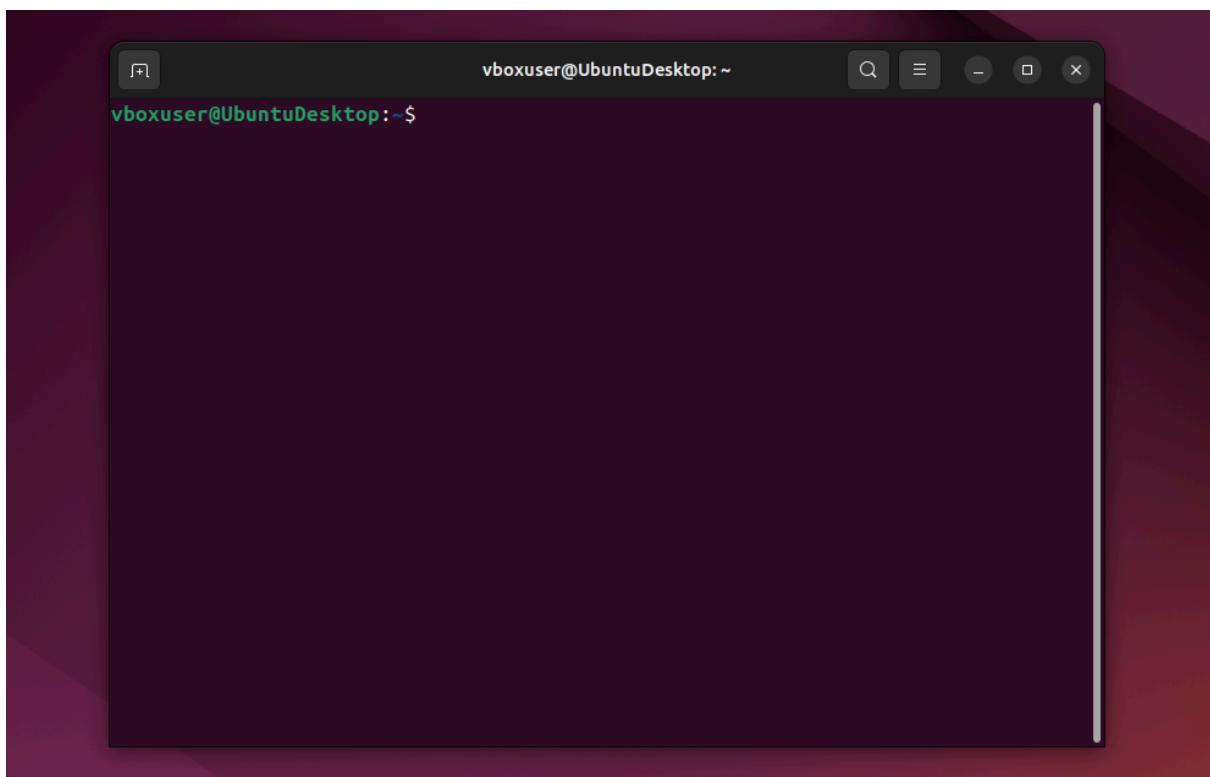
Faites attention à bien prendre le logiciel adapté à votre environnement. Dans mon cas je serais sous Linux, mais cela ne vous empêchera pas de suivre le workshop sauf pour quelque commande spécifique.

Pour le bon déroulement de ce workshop c'est pour cela que je vous recommande d'être sous Linux, soit à travers WSL sur Windows pour simuler un environnement Linux en console : <https://learn.microsoft.com/fr-fr/windows/wsl/install>

Ou bien une machine virtuelle Ubuntu à travers le logiciel de virtualisation VirtualBox ou VMWare peut-être aussi une bonne solution afin d'isoler votre environnement de test/développement du reste de votre machine.

Si vous êtes déjà sous un système Linux ou MacOS pas d'inquiétude les commandes restent les mêmes.

Une fois l'installation terminée, ouvrons notre terminal de commande et si tout ce passe bien nous devrions retrouver cette fenêtre (Si vous avez choisi l'option machine virtuelle sous Ubuntu).



Partie 2 : Configuration de GIT

Ouvrez votre terminal et configurez votre nom et email (Git enregistre ces informations pour chaque commit) :

```
git config --global user.name "VotreNom"  
git config --global user.email "votre.email@example.com"
```

Partie 3 : Premier projet, initialisation et utilisation de GIT

3.1 Crédation d'un répertoire de travail

Créez un dossier pour votre projet Git à l'aide des commandes shell propres à l'environnement Linux. Si vous souhaitez en apprendre davantage sur ces commandes, je vous invite à consulter les documentations disponibles en annexes.

Les commandes:

- `mkdir` (Make Directory) : Cette commande est utilisée pour **créer un nouveau dossier** dans le système de fichiers.
- `cd` (Change Directory) : Cette commande permet de **naviguer vers un dossier spécifique**.

```
mkdir projet-git  
cd projet-git
```

3.2 Initialisation de GIT

En exécutant la commande suivante cela initialisera un nouveau **dépôt Git** dans le dossier courant, permettant ainsi de commencer à suivre les modifications des fichiers dans ce dossier. Elle crée un sous-dossier caché nommé `.git` qui contient toutes les informations nécessaires pour suivre les versions. Cette commande ne doit être réalisée qu'une seule fois par répertoire de travail.

```
git init
```

3.3 Crédons notre premier fichier Markdown

Markdown est un langage de balisage léger que nous devons à John Gruber, qui l'a développé en 2004 avec l'aide d'Aaron Swartz. L'objectif de Markdown était de rendre le formatage de texte simple et lisible, même sous sa forme brute, en opposition au HTML, souvent complexe à lire et à écrire pour des non développeurs. Grâce à Markdown, il devient possible de créer facilement des

documents formatés (titres, listes, liens, images, etc.), tout en restant accessible et clair, même sans logiciel spécialisé.

Aujourd'hui, Markdown est largement employé pour la rédaction de documentation, notamment sur GitHub, ainsi que dans des forums, blogs et autres plateformes collaborative.

Avec la commande shell `touch`, nous allons pouvoir **créer un fichier vide** directement depuis le terminal ou encore **mettre à jour la date de modification** d'un fichier déjà existant. C'est une commande très utile, surtout quand on veut rapidement ajouter des fichiers texte, des fichiers de configuration, ou des fichiers de code dans le dossier où l'on travaille.

```
touch README.md
```

Maintenant, afin de vérifier au sein de notre terminal si le fichier s'est bien créé, nous allons utiliser la commande shell `ls`. Cette commande nous permet d'afficher la liste des fichiers et dossiers dans le répertoire actuel, nous montrant ainsi si le fichier que nous venons de créer y figure bien.

```
ls
```

Si tout se passe bien, vous devriez voir votre fichier récemment créé dans la liste affichée. Cela confirmera que le fichier a bien été ajouté au dossier.

Maintenant passons à la modification de celui-ci, pour le modifier plusieurs options s'offre à nous :

- **Utiliser un Éditeur de Texte en Ligne de Commande :**

- Nous pouvons utiliser des éditeurs comme `nano` ou `vim`. Par exemple, pour modifier `README.md` avec `nano`, vous pouvez exécuter :

```
nano README.md
```

- Cela ouvrira le fichier dans l'éditeur, où vous pourrez effectuer vos modifications. Une fois que vous avez terminé, vous pouvez sauvegarder et quitter (dans `nano`, appuyez sur `CTRL + X`, puis sur `Y` pour confirmer la sauvegarde).

- **Utiliser un Éditeur de Texte Graphique :**

- Si vous préférez un environnement graphique, vous pouvez ouvrir le fichier avec des éditeurs comme **VS Code** ou **Sublime Text**. Vous pouvez faire cela depuis le terminal en tapant :

```
code README.md # pour Visual Studio Code
```

```
subl README.md # pour Sublime Text
```

Au sein de ce workshop, nous considérons que Visual Studio Code est installé et peut être utilisé pour l'édition de fichiers.

Une fois mon fichier ouvert je lui écris donc un titre comme ceci :

```
# Hello World !
```

Puis, enregistrez à l'aide de **Ctrl + S** ou dans le menu **Fichier > Sauvegarder** .

3.4 Ajoutons et Validons notre premier fichier dans Git

Ajoutons et validons notre premier fichier dans Git. Voici les étapes à suivre :

1. Ajouter le fichier à l'index :

Pour commencer, nous devons ajouter le fichier que nous souhaitons suivre à l'index de Git. Utilisez la commande suivante :

```
git add README.md
```

Cela indique à Git que vous souhaitez inclure ce fichier dans le prochain commit.

2. Valider les changements :

Ensuite, vous devez valider les modifications que vous avez apportées.

Cela crée un point de sauvegarde dans l'historique de votre dépôt. Utilisez la commande suivante :

```
git commit -m "Création de mon fichier readme + un titre au document"
```

Ici, le texte entre les guillemets est le message de commit, qui décrit les modifications que vous avez effectuées.

Il sert à **décrire les modifications** que vous avez apportées au projet. Voici pourquoi cela est important :

1. **Clarté** : Un bon message de commit permet de comprendre rapidement ce qui a été modifié sans avoir à examiner chaque ligne de code. Cela aide non seulement vous-même, mais aussi d'autres développeurs qui pourraient travailler sur le projet plus tard.
2. **Historique** : Les messages de commit créent un historique clair des changements apportés au projet. Cela peut être crucial lors de la recherche de bugs ou de la compréhension de l'évolution du code.
3. **Collaboration** : Dans un environnement de travail collaboratif, des messages de commit informatifs aident les membres de l'équipe à suivre le travail des autres et à savoir ce qui a été fait et pourquoi.
4. **Meilleures pratiques** : Utiliser des messages de commit significatifs est une bonne pratique dans le développement de logiciels. Cela montre un souci du détail et un respect pour le travail de l'équipe.

3.5 Une nouvelle version en devenir

Faisons un autre commit pour enregistrer des modifications supplémentaires dans notre projet :

- **Modifier le fichier `README.md`** :

Ouvrez le fichier `README.md` et ajoutez une première section sous le titre. Par exemple :

```
# Hello World !  
  
## Description  
Ceci est une description
```

- **Ajouter le fichier à l'index** :

Une fois que vous avez enregistré vos modifications, ajoutez le fichier à l'index de Git avec la commande suivante :

```
git add README.md
```

- **Faire votre commit** :

Ensuite, effectuez un nouveau commit avec un message décrivant les

modifications apportées. Par exemple :

```
git commit -m "Ajout de la section Description dans README.md"
```

3.6 Vérifier le statut du répertoire local

Après l'ajout de notre deuxième version GIT, un état de lieu s'impose. Pour cela une commande GIT vous sera utile :

```
git status
```

Elle permet de voir les modifications en attente, les fichiers suivis et non suivis, ainsi que les fichiers prêts pour le prochain commit.

3.7 Consultons l'historique

Nous allons maintenant utiliser la commande `git log` afin de consulter l'historique de nos actions dans le dépôt. Cette commande permet d'afficher tous les commits effectués, accompagnés de la date, de l'auteur et du message de chaque commit.

Pour afficher l'historique des commits, utilisez :

```
git log
```

Pour une vue simplifiée, utilisez :

```
git log --oneline
```

3.8 Travailons avec les branches

Il existe aussi dans GIT la notion de branche. Pensez à une branche comme à une **copie** de votre projet où vous pouvez travailler librement sans affecter le travail des autres.

Pourquoi devons nous les utiliser ?

1. **Travailler sans risque** : Imaginez que vous essayez une nouvelle recette. Vous n'allez pas modifier votre plat principal tout de suite, n'est-ce pas ? Vous feriez d'abord un essai. Les branches fonctionnent de la même

manière : vous pouvez faire des modifications ou ajouter des fonctionnalités sans toucher à la version principale de votre projet.

2. **Collaboration facile** : Lorsque plusieurs personnes travaillent ensemble, chaque personne peut créer sa propre branche pour ses idées. Cela permet à chacun de contribuer sans perturber le travail des autres.
3. **Tester avant de finaliser** : Avant de servir votre plat à des invités, vous voudriez le goûter d'abord. Avec les branches, vous pouvez tester vos modifications et vous assurer qu'elles fonctionnent bien avant de les ajouter au projet principal.

Testons à présent l'utilisation de celle-ci :

1. Créons une nouvelle branche pour développer notre nouvelle fonctionnalité :

```
git branch feature/ajout-liste
```

2. Maintenant que notre nouvelle branche est créée, basculons dessus :

```
git checkout feature/ajout-liste
```

Une fois que la bascule est faite nous allons rééditer notre `README.md` et ajouter notre liste :

1. Premier élément
2. Deuxième élément
3. Troisième élément

Une fois l'ajout effectué, sauvegarder le fichier et fermer votre éditeur. Puis ajoutons nos modifications à notre dépôt local.

```
git add README.md  
git commit -m "Ajout d'une liste au fichier."
```

Une fois cela fait je vous propose de re basculer sur la branche principal `master`, et ouvrez à nouveau le fichier `README.md`. Vous devriez constater que la liste à disparu, ce qui est normal car nous avons basculé sur le projet en cours et les modifications ce trouve encore sur la branche nouvellement créée.

3. Assurons nous que notre branche nouvellement créé est toujours disponible, cette commande vous permet d'explorer l'ensembles des branches existante sur votre projet GIT :

```
git branch
```

4. Puis dans le contexte de notre workshop je vous propose de supprimer notre branche récente :

```
git branch -d feature/ajout-liste
```

Vous devriez normalement avoir une erreur, qui vous précise qu'elle n'a pas été complément fusionné donc il ne peut pas là supprimer. Cela est tout à fait normal car cela permet de la protéger au cas où vous ferez une mauvaise manipulation. Cependant toujours dans le cadre de notre workshop nous allons forcer la commande car nous savons que ce n'est pas grave.

```
git branch -D feature/ajout-liste
```

Dans la commande, le paramètre `-d` à juste été mis en majuscule afin d'indiquer que nous forcions la suppression.

Bien sûr dans la nous sommes dans le cadre d'un workshop dans une entreprise toujours demander l'accord d'un responsable avant toute action sur l'environnement git d'un projet.

3.9 Annulons une modification

Parfois, il est nécessaire d'annuler une modification que nous avons apportée à un fichier dans votre dépôt Git. Voici plusieurs méthodes pour le faire, selon la situation :

1. Je vous propose donc de modifier le fichier Markdown est de mettre en gras un mot en dessous de la section précédemment créé :

```
# Hello World !
```

```
## Description  
Ceci est une **description**.
```

Une fois le fichier enregistré je vous invite à effectuer la commande suivante :

```
git checkout -- README.md
```

Nous constatons qu'après avoir réouvert le fichier, cette commande a rétabli le fichier à son dernier état enregistré, perdant ainsi toutes les modifications non ajoutées.

2. Maintenant nous allons remettre dans notre fichier la mise en gras de notre texte puis l'ajouter à la Staging area afin de l'indexer :

```
git add README.md
```

Dans certains cas il peut être nécessaire d'exclure un fichier avant d'effectuer le commit, essayons maintenant la commande suivante.

```
git reset HEAD README.md
```

En réouvrant le fichier vous devriez constater que nous avons pas perdu nos modifications mais il a été enlevé de l'indexation. Vous pouvez vérifier avec la commande :

```
git status
```

3. Si vous avez déjà fait un commit et souhaitez l'annuler, utilisez la commande suivante pour revenir au commit précédent :

```
git reset --soft HEAD
```

Cela supprimera le dernier commit tout en conservant les modifications dans votre répertoire de travail.

4. Si vous souhaitez également supprimer les modifications, utilisez :

```
git reset --hard HEAD
```

Attention : Cette commande efface définitivement les modifications, alors assurez vous de ne pas perdre de travail important.

Maintenant je vous propose d'essayer chaque commande afin d'en voir les effets.

3.10 Une fusion de branche

Comme vu dans la section 3.8, lors de la tentative de suppression de la branche, il nous a été indiqué que la suppression n'était pas possible car la branche que nous souhaitions supprimer n'était pas entièrement fusionnée. Maintenant, voyons comment procéder.

Mais d'abord, pourquoi devrions nous fusionner ?

- **Intégrer les changements** : Lorsque vous avez terminé de travailler sur une fonctionnalité ou un correctif dans une branche, il est important de l'intégrer à la branche principale pour que tout le monde puisse en bénéficier.
- **Collaborer** : Cela permet aux membres de l'équipe de partager leurs contributions et d'assurer une harmonisation de tout le travail.

Passons maintenant à la pratique : reprenons la section 3.8 jusqu'au commit de notre liste. Une fois notre branche prête, basculons sur celle qui doit recevoir les modifications. Ensuite, exécutez la commande suivante :

```
git merge feature/ajout-liste
```

Dans le cas où la fusion de branche doit être annulé, la commande qui peut être utilisé est :

```
git merge --abort
```

Partie 4 : Jeu de Rôle : Collaboration sur le Projet "Documentation de Projet"

4.1 Participants

1. **Dev1** : Développeur 1, responsable de l'ajout de contenu.

2. **Dev2** : Développeur 2, responsable de l'ajout de contenu.
3. **Lead** : Responsable de merge, en charge de l'examen et de la fusion des contributions des développeurs.

4.2 Objectif

Collaborer sur un fichier Markdown nommé `README.md`, en ajoutant et en modifiant le contenu pour améliorer la documentation d'un projet. Le Lead examinera les modifications avant de les fusionner dans la branche principale.

4.3 Étapes du Jeu de Rôle

Étape 1 : Configuration Initiale

1. Créer un Dépôt GitHub

- Ouvrez GitHub et créez un nouveau dépôt nommé `documentation-projet`.

2. Cloner le Dépôt

- Chaque participant clone le dépôt sur sa machine locale :

```
git clone https://github.com/votre_nom_utilisateur/documentation-projet.git  
cd documentation-projet
```

3. Créer une Branche pour Chaque Développeur

- Dev1 crée une branche pour ajouter du contenu :

```
git checkout -b feat/add-content-part1
```

- Dev2 crée une branche pour ajouter du contenu lui aussi :

```
git checkout -b feat/add-content-part2
```

Étape 2 : Protection de la Branche Principale

1. Configurer la Protection de la Branche sur GitHub

- Sur GitHub, allez dans les **Settings** du dépôt.

- Cliquez sur **Branches** dans le menu latéral.
- Sous **Branch protection rules**, cliquez sur **Add rule**.
- Dans le champ **Branch name pattern**, entrez `main`.
- Cochez les options suivantes :
 - **Require pull request reviews before merging** : Cela nécessite une revue de code avant de pouvoir fusionner.
 - **Require status checks to pass before merging** : Cela garantit que les tests passent avant la fusion (si configuré).
- Cliquez sur **Create** pour enregistrer la règle.

Étape 3 : Ajout de Contenu par Dev1

- **Dev1** ouvre le fichier `README.md` et ajoute une nouvelle section :

```
# Documentation de Projet

## Section Fonctionnalités du Projet
```

Cette section a été ajoutée par Dev1 pour expliquer les fonctionnalités du projet.

- **Valider et Pousser les Modifications** :

```
git add README.md
git commit -m "Ajout d'une nouvelle section par Dev1"
git push origin feat/add-content-part1
```

Étape 4 : Ajout de Contenu par Dev2

- **Dev2** ouvre le même fichier et met à jour une section existante :

```
# Documentation de Projet

## Section Informations
```

Cette section a été mise à jour par Dev2 pour améliorer la clarté des informations.

- **Valider et Pousser les Modifications :**

```
git add README.md  
git commit -m "Mise à jour de la section existante par Dev2"  
git push origin feat/add-content-part2
```

Étape 5 : Création de Pull Requests

1. **Dev1** crée une Pull Request (PR) pour sa branche :

- Sur GitHub, allez à l'onglet **Pull Requests** et cliquez sur **New Pull Request**.
- Sélectionnez `feat/add-content-part1` comme branche source et `main` comme branche de destination. Cliquez sur **Create Pull Request**.
- **Ajoutez une description** expliquant les modifications apportées.

2. **Dev2** fait de même pour sa branche :

- Créez une PR pour `feat/add-content-part2` en suivant le même processus.

Étape 6 : Examen par le Lead

- **Lead** examine les deux Pull Requests :
 - Ouvrez chaque PR sur GitHub.
 - Lisez les modifications apportées par Dev1 et Dev2.
 - **Laissez des commentaires** si nécessaire pour poser des questions ou demander des modifications supplémentaires.

Étape 7 : Fusion des Pull Requests

1. **Lead** fusionne les modifications après examen :

- Pour Dev1, cliquez sur **Merge pull request** et confirmez la fusion.
- Pour Dev2, répétez le même processus pour fusionner sa PR.

2. **Rappels :**

- Grâce à la protection de la branche, la fusion ne pourra se faire que si les Pull Requests ont été examinées et validées.
- Si des conflits apparaissent, le Lead doit les résoudre localement.

Étape 8 : Résolution des Conflits

- Si les modifications de Dev1 et Dev2 ont créé des conflits, le Lead doit les résoudre avant de fusionner.

1. Mettre à jour la branche principale :

```
git checkout main  
git pull origin main
```

2. Fusionner la branche de Dev1 :

```
git merge feat/add-content-part1
```

- Résoudre les conflits dans `README.md`.

3. Valider les changements :

```
git add README.md  
git commit -m "Résolution des conflits"
```

4. Pousser les modifications sur GitHub :

```
git push origin main
```

Partie 5 : Bonus

5.1 Un portfolio en ligne ?

Un portfolio en ligne est un site web ou une page dédiée où un individu, souvent un professionnel ou un créatif, présente ses travaux, compétences et réalisations. Il sert de vitrine numérique pour montrer ses projets, son expérience, et son expertise, permettant aux employeurs, clients potentiels, et partenaires de découvrir ses capacités en un coup d'œil.

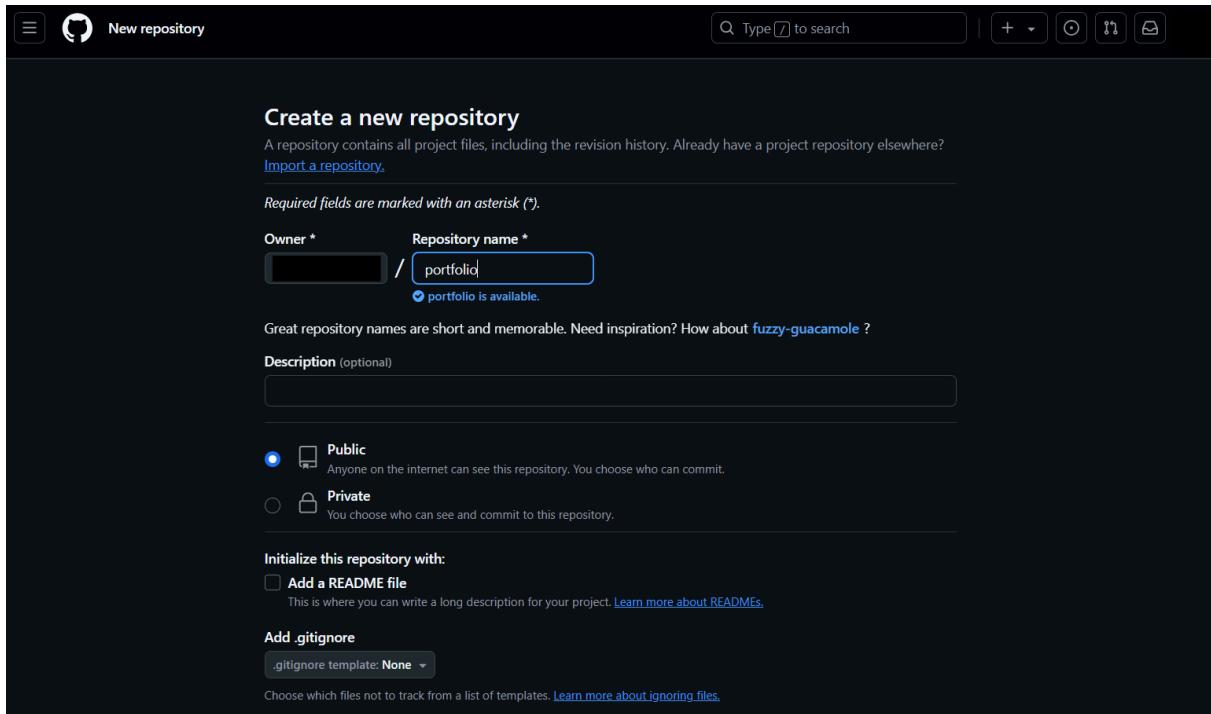
L'intérêt d'un portfolio en ligne réside dans sa capacité à renforcer la crédibilité et la visibilité d'un profil professionnel, surtout dans des domaines comme le design, le développement web, l'écriture ou la photographie. Il offre une manière dynamique et interactive de se démarquer, facilite le partage de son travail, et permet d'attirer de nouvelles opportunités tout en centralisant les informations importantes sur un même espace accessible à tout moment.



5.2 Créons le dépôt

GitHub va te permettre de créer et d'héberger gratuitement une page internet en ligne, accessible à tous grâce à un service proposé par GitHub, appelé **GitHub Pages**. C'est un outil très pratique pour publier un site web simple, comme un portfolio, une documentation, ou un projet scolaire.

Commence par créer un dépôt que nous laisserons en public afin de le rendre accessible depuis l'extérieur :



The screenshot shows the GitHub interface for creating a new repository. The repository name is set to 'portfolio'. The 'Public' option is selected. There are fields for adding a README file and a .gitignore template.

Une fois cela fait nous allons pouvoir cloner notre répertoire sur notre ordinateur et venir y créer deux nouveau fichier `index.html` et `style.css`.

5.3 Créons et envoyons notre première page

Commençons par créer notre `index.html` dans notre répertoire et rentrer le code suivant :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Portfolio Étudiant Informatique</title>
  <link rel="stylesheet" href="style.css">
```

```

</head>
<body>
    <header>
        <h1>Mon Portfolio</h1>
        <nav>
            <ul>
                <li><a href="#about">À propos de moi</a></li>
                <li><a href="#projects">Projets</a></li>
                <li><a href="#contact">Contact</a></li>
            </ul>
        </nav>
    </header>

    <section id="hero">
        <h2>Bonjour, je suis [Nom de l'étudiant]</h2>
        <p>Étudiant en informatique, passionné par le développement et la technologie.</p>
    </section>

    <section id="about">
        <h2>À propos de moi</h2>
        <p>Je suis étudiant en informatique, actuellement en [nom du programme ou université]. Je m'intéresse particulièrement à [vos domaines d'intérêt, ex: le développement web, l'IA, la cybersécurité, etc.].</p>
    </section>

    <section id="projects">
        <h2>Mes Projets</h2>
        <div class="project">
            <h3>Projet 1</h3>
            <p>Description du projet 1. Ce projet se concentre sur [technologies ou concepts utilisés].</p>
        </div>
        <div class="project">
            <h3>Projet 2</h3>
            <p>Description du projet 2. Ce projet met en avant mes compétences en [compétences techniques, ex: Java, C++, etc.].</p>
        </div>
    </section>

```

```

<!-- Ajoutez plus de projets selon vos besoins →
</section>

<section id="contact">
    <h2>Contact</h2>
    <p>Vous pouvez me contacter par email : <a href="mailto:email@example.com">email@example.com</a></p>
</section>

<footer>
    <p>&copy; 2024 [Nom de l'étudiant]. Tous droits réservés.</p>
</footer>
</body>
</html>

```

Puis on fait la même chose avec le fichier `style.css` :

```

* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
    color: #333;
    background-color: #f4f4f4;
}

header {
    background-color: #333;
    color: #fff;
    padding: 1rem;
    text-align: center;
}

header h1 {

```

```
    font-size: 2rem;
}

nav ul {
    list-style: none;
    padding: 0;
}

nav ul li {
    display: inline;
    margin-right: 1rem;
}

nav ul li a {
    color: #fff;
    text-decoration: none;
    font-weight: bold;
}

#hero {
    background-color: #4CAF50;
    color: #fff;
    padding: 2rem 1rem;
    text-align: center;
}

#about, #projects, #contact {
    padding: 2rem 1rem;
    margin: 1rem 0;
    background-color: #fff;
    border-radius: 5px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
}

#projects .project {
    margin: 1rem 0;
    padding: 1rem;
    border-bottom: 1px solid #ddd;
```

```

}

#projects .project:last-child {
    border-bottom: none;
}

footer {
    text-align: center;
    padding: 1rem;
    background-color: #333;
    color: #fff;
    margin-top: 1rem;
}

/* Responsive Design */
@media (min-width: 768px) {
    header h1 {
        font-size: 2.5rem;
    }

    #hero {
        padding: 3rem 2rem;
    }

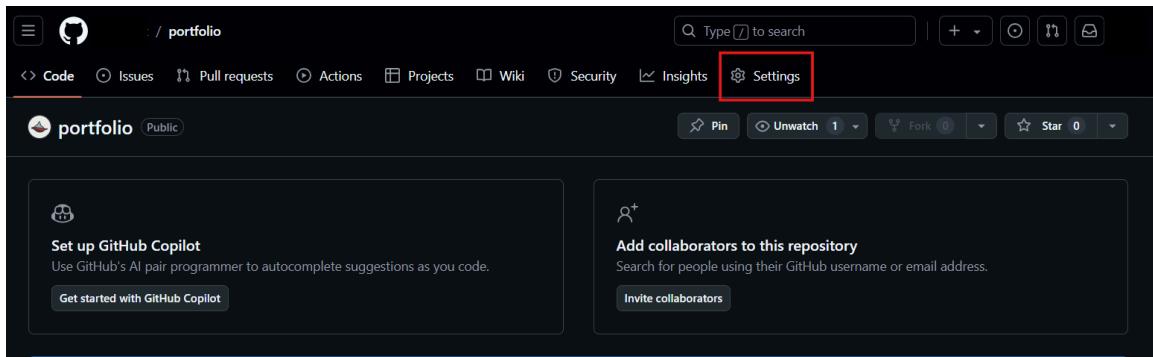
    #about, #projects, #contact {
        width: 60%;
        margin: 1rem auto;
    }
}

```

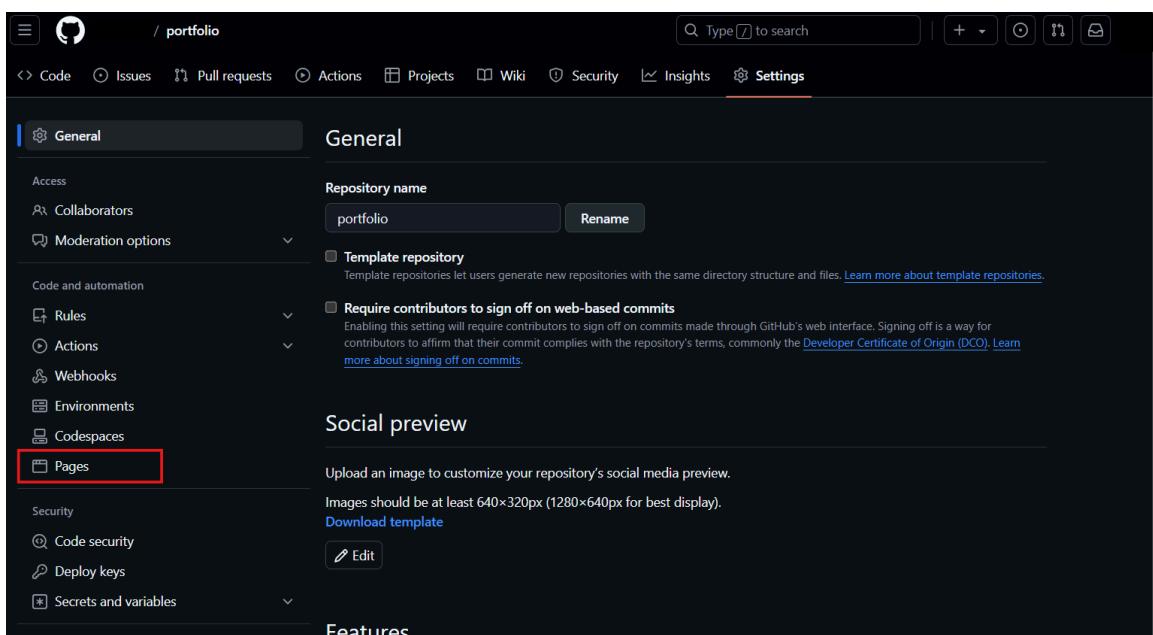
Bien sûr un code de base est fourni mais vous êtes libre de le modifier entièrement et de l'adapter à vos besoins.

5.4 Configuration du dépôt en mode Page

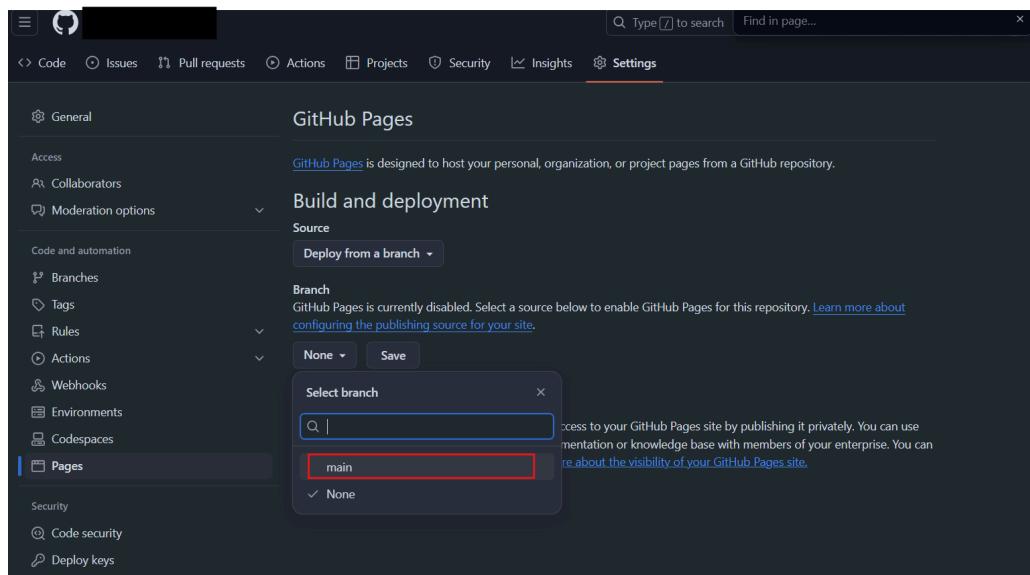
1. Une fois dans votre répertoire distant sur GitHub, dans les onglets supérieurs, dirigez vous vers **Settings** :



2. Maintenant que vous êtes sur cette page, dans la colonne de gauche, accédez à **Pages** :



3. Une fois arrivé comme ci-dessous ,nous allons à présent sélectionner la branche principale, celle sur laquelle se trouve notre fichier `index.html` :



Après avoir cliqué sur le bouton **Save**, et attendu quelques minutes, vous devriez voir le lien apparaître en haut pour accéder à votre page.

Pensez à protéger la branche **main**, comme vu dans la section précédente lors du jeu de rôle, car sinon, n'importe qui pourrait injecter du contenu non désiré dans votre branche principale.

5.5 Et maintenant la suite ?

Maintenant que vous avez créé votre portfolio de base, vous êtes libre de le personnaliser et de l'améliorer selon vos goûts et besoins. N'hésitez pas à :

- Modifier le design en ajustant le CSS pour refléter votre style personnel
- Ajouter du contenu supplémentaire, comme vos projets récents ou vos compétences techniques
- Intégrer des liens vers vos profils sur les réseaux sociaux professionnels (LinkedIn, GitHub, etc.)
- Inclure votre CV en ligne ou un lien vers celui-ci

Une fois que vous êtes satisfait de votre portfolio, n'hésitez pas à le partager sur les réseaux sociaux et à l'inclure dans vos candidatures. Cela démontrera non seulement vos compétences techniques acquises lors de ce workshop, mais aussi votre initiative et votre capacité à présenter votre travail de manière professionnelle.

Rappelez-vous que ce portfolio est un excellent moyen de mettre en valeur vos compétences en développement web et en gestion de version avec Git, des atouts précieux pour votre future carrière dans l'informatique.

Annexes

1. Git et GitHub

- **Documentation officielle de Git** : git-scm.com
- **Guide Git de Atlassian** : [Atlassian Git Tutorial](https://www.atlassian.com/git/tutorials/introduction-to-git)
- **GitHub Guides** : guides.github.com

2. Commandes Git

- **Liste des commandes Git** : [Git Command Cheat Sheet](https://git-scm.com/docs/git-command-cheatsheet)

3. Markdown

- **Guide Markdown de Daring Fireball** : [Daring Fireball Markdown](https://daringfireball.net/projects/markdown/)
- **Markdown Cheatsheet** : [Markdown Cheatsheet](https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet)

4. Gestion des branches

- **Branching Strategies in Git** : [Atlassian Branching Strategies](https://www.atlassian.com/git/tutorials/branch-strategies)
- **Git Branching Documentation** : [Git Branching](https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-Strategies)

5. Conflits de fusion

- **Understanding Merge Conflicts** : [Atlassian Merge Conflicts](https://www.atlassian.com/git/tutorials/merging-conflicts)
- **Resolving Merge Conflicts in Git** : [Git Merge Conflict Resolution](https://git-scm.com/book/en/v2/Git-Tools-Merge-Conflicts-and-Three-Way-Merges)

6. GitHub Pages

- **Documentation GitHub Pages** : [GitHub Pages](https://help.github.com/categories/github-pages/)
- **Using GitHub Pages to Host Your Portfolio** : [GitHub Pages for Portfolios](https://help.github.com/categories/hosting-your-portfolio/)

7. HTML et CSS (pour votre portfolio)

- **W3Schools HTML Tutorial** : [W3Schools HTML](https://www.w3schools.com/html/)
- **W3Schools CSS Tutorial** : [W3Schools CSS](https://www.w3schools.com/css/)
- **MDN Web Docs on HTML** : [MDN HTML](https://developer.mozilla.org/en-US/docs/Web/HTML)
- **MDN Web Docs on CSS** : [MDN CSS](https://developer.mozilla.org/en-US/docs/Web/CSS)

8. Frameworks et outils

- **Bootstrap Documentation** : getbootstrap.com
- **Jekyll Documentation** : jekyllrb.com

9. Commandes Shell

- **Introduction aux commandes Shell** : [Shell Command Cheat Sheet](#)
- **Commandes de base pour naviguer dans le terminal** :
 - `ls` : Lister les fichiers et répertoires.
 - `cd` : Changer de répertoire.
 - `mkdir` : Créer un nouveau répertoire.
 - `touch` : Créer un nouveau fichier vide.
 - `rm` : Supprimer un fichier ou répertoire.
- **Guide des commandes Shell** : [Linux Command Line Basics](#)