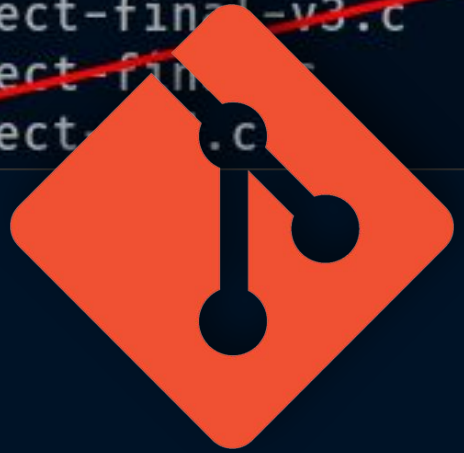
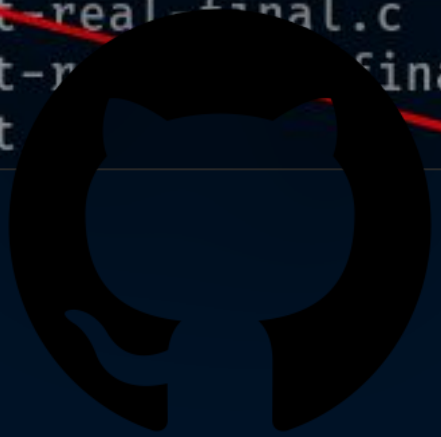
Git & GitHub

From Scratch

```
litone@kali ~/final_files> ls
my_project-final-v0.c   my_project-old2.c
my_project-final-v1.c   my_project-real-final-v1.c
my_project-final-v2.c   my_project-real-final-v2.c
my_project-final-v3.c   my_project-real-final.c
my_project-fin          my_project-r          final.c
my_project-     .c      my_project
```





$ whoami

Alexandre GRAU - grau.alexandre@gmail.com

- École ESGI Spécialisation Cybersécurité
- SysAdmin / Développeur Cybersécurité / Consultant Cybersécurité
- Freelance : (Audit, Dev, Reversing, Analyse de Malware, Formation, …)

Git & GitHub

Déroulement du cours :

- Test de Positionnement / QCM (non noté)
- TPs (non notés)
- TPs (notés)
- Projet de groupe (noté)
- Exam machine (noté)

Git & GitHub

Objectif du cours :

- Créer, configurer, participer et gérer un projet Git et GitHub.
- Créer des branches sur des projets.
- Faire des Pull Requests sur des projets GitHub
- Forker un projet sur GitHub

Git

Qu’est ce que Git ?

- Outil open-source de versionning de code créé en 2005 par Linus Torvalds, le créateur de Kernel Linux.

- Successeur des solutions - Subversion (SVN), BitKeeper (SVC) et Mercurial

Git

Pourquoi utiliser Git ?

- Suivi de l'historique des modifications et des versions d’un code source.
- Facilite la collaboration.
- Historique complet du projet.
- Gestion des branches et des fusions.

GitHub

Qu’est ce que GitHub ?

GitHub créé en 2008 est une plateforme web qui permet d'héberger des dépôts Git et de faciliter la collaboration entre développeurs et communautés.

Elle offre des fonctionnalités comme les pull requests, les issues, et l'intégration continue, facilitant le travail en équipe, la revue de code et le suivi des problèmes.

GitHub

Pourquoi utiliser GitHub ?

- Hébergement de dépôts Git
- Gestion des pull requests
- Suivi des problèmes (issues)
- Intégration continue (CI/CD)
- Visbilité et partage du code
- Communauté

Git & GitHub from Scratch

Test de positionnement / QCM :

Qui s’est déjà servi de git ?
Régulièrement ?

Qui s’est déjà servi de GitHub ?
Régulièrement ?

https://x0ne.training

Git

Installation :

Depuis un système Linux (Debian/Ubuntu) :

```
$ sudo apt update
$ sudo apt install git
```

Git

Configuration :

Avant d'utiliser Git, il est recommandé de configurer votre nom et votre adresse email, ces information sont utilisées pour identifier les auteurs des commits.

```
$ git config --global user.name “<YOUR NAME>”
$ git config --global user.email “<YOUR EMAIL
```

Git

Configuration :

Le fichier de configuration globale se trouve ~./gitconfig
Et celui de la configuration du projet se trouve .git/config

```
$ git config --list
```

Git Concepts clefs

Dépôt / Repository :

Un dépot contient le code source, les branches, les commits, l'historique, les méta-données du projets, etc.

local repository est le dépôt Git sur ta machine, où tu fais des commits et modifies ton code.

remote repository est un dépôt hébergé sur un serveur distant (comme GitHub) et accessible à d'autres utilisateurs, permettant la collaboration.

```
Création d'un repo
$ mkdir my_repo
$ cd my_repo
$ git init
```

Git Concepts clefs

Clone :

Un clone est une copie d'un dépôt distant. Il permet de récupérer le code source sur votre machine locale.

```
# Clone d’un remote repository
$ git clone <SSH_LINK / HTTP_LINK>
# Ou clone d’un local
$ git clone <PATH_SOURCE> <PATH_DEST>
```

Git Concepts clefs

Pull :

Un pull est l'action de récupérer les modifications depuis un dépôt distant. Il permet de mettre à jour votre dépôt local depuis le dépôt distant (remote).

```
$ git pull
```

Git Concepts clefs

Status :

Un status représente l'état de ton espace de travail, les fichiers modifiés, ajoutés ou supprimés, et leur statut par rapport à l'index (staging area) et au dépôt (repository).

Untracked : Fichiers de ton répertoire de travail (Working directory) non staged, non commités.

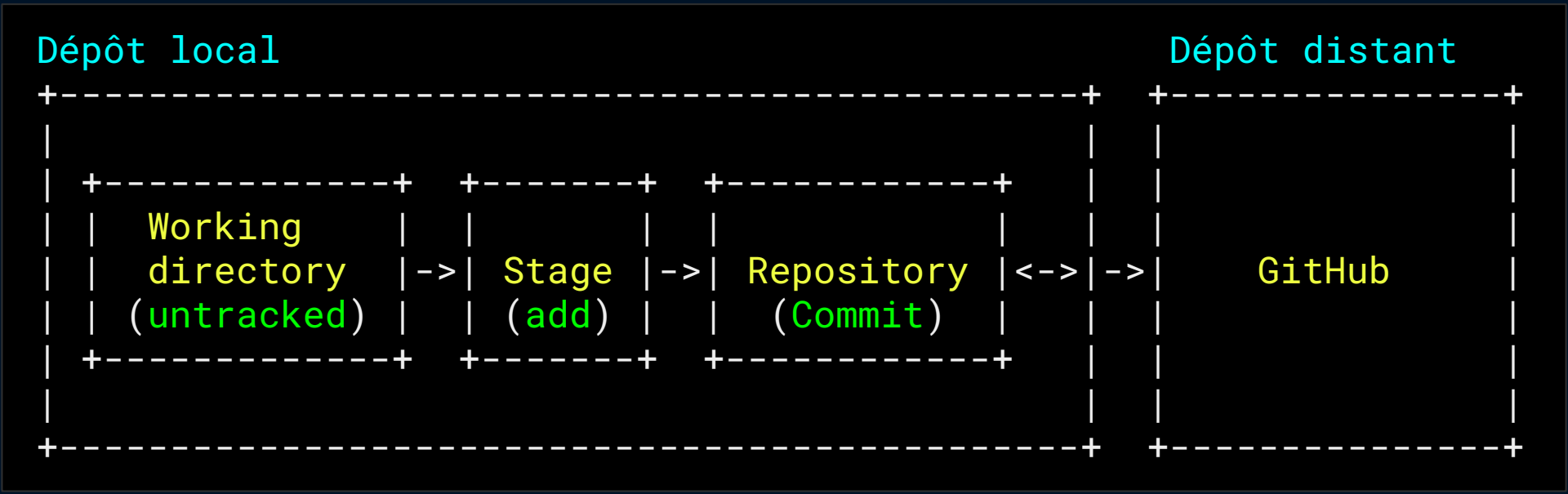
Staged or Indexed : Fichiers ajoutés à l'index, staged

Modified or Commited : Fichiers modifiés, ajoutés au staged et commited

```
$ git status
$ git status -s
```

Git Concepts clefs

Status :



Git Concepts clefs

Diff :

La commande git diff permet d'afficher les différences entre les fichiers modifiés et les fichiers suivis. Elle permet de voir les lignes ajoutées, modifiées ou supprimées.

```
Différences de tous les fichiers (untraked vs tracked or commited)
$ git diff
Différences d'un fichiers (untracked vs tracked or commited)
$ git diff <FILE>
Différence entre deux commits
$ git diff <COMMIT1> <COMMIT2>
Différences entre deux branches
$ git diff <BRANCH1> <BRANCH2>
```

Git Concepts clefs

Staging :

Le staging est l'étape qui précède le commit. Il permet de préparer les fichiers à inclure dans le commit. C’est un commit en cours de confirmation.

```
$ git add <MON_FICHIER>
```

Git Concepts clefs

Commit :

Un commit est un snapshot du code à un moment donné. Il contient les modifications apportées, incluant les auteurs, les dates, les messages, etc. Il officialise les modifications.

```
$ git commit -m “<MA_DESCRIPTION>”
```

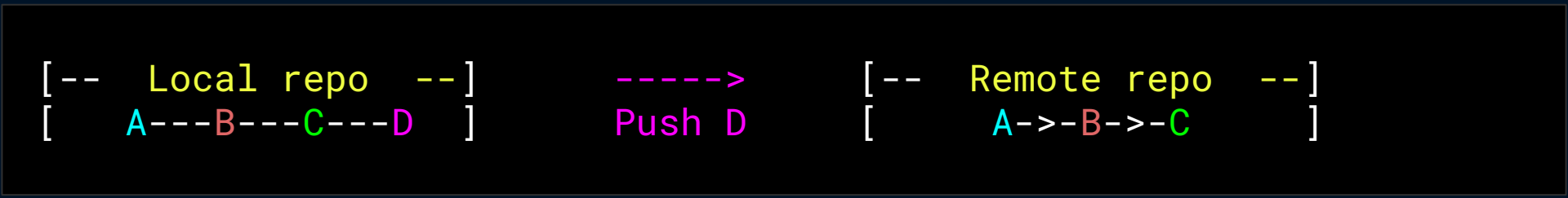
```
[Début du projet] A->-B->-C->-D [Fin du projet]
                      Commits
```

Git Concepts clefs

Push :

Un push est l'action de pousser les commits locaux vers un dépôt distant. Il permet de partager les modifications avec les autres développeurs.

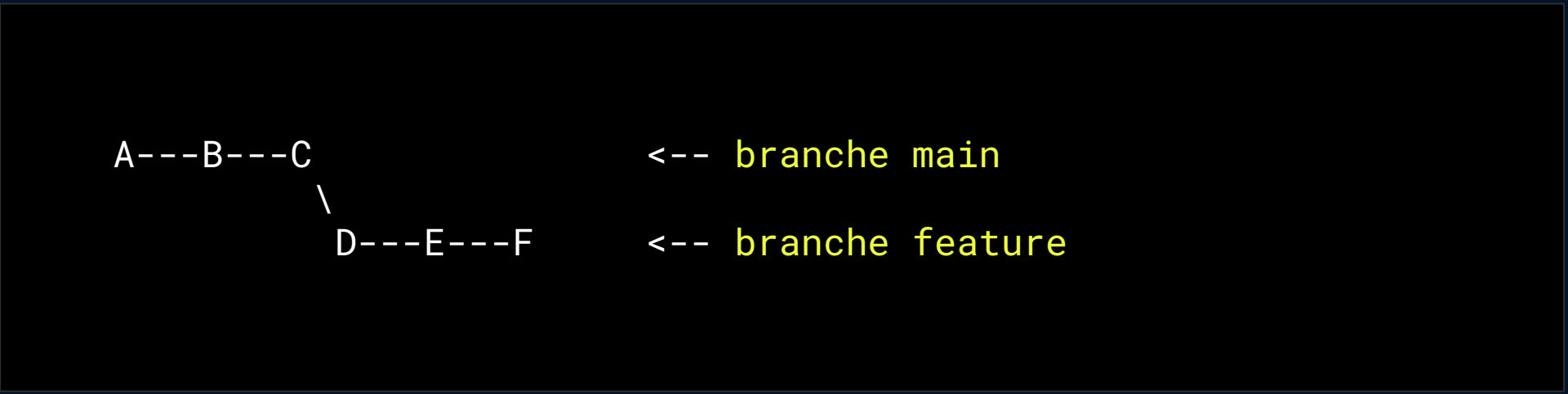
```
$ git push
```



Git Concepts clefs

Branche :

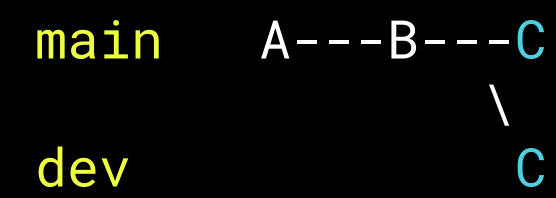
Une branche dans Git est une ligne indépendante de développement qui te permet de travailler sur des changements ou des fonctionnalités sans affecter la branche principale est généralement main ou master, tout en gardant la possibilité de fusionner tes modifications plus tard.



Git Concepts clefs

Branche :

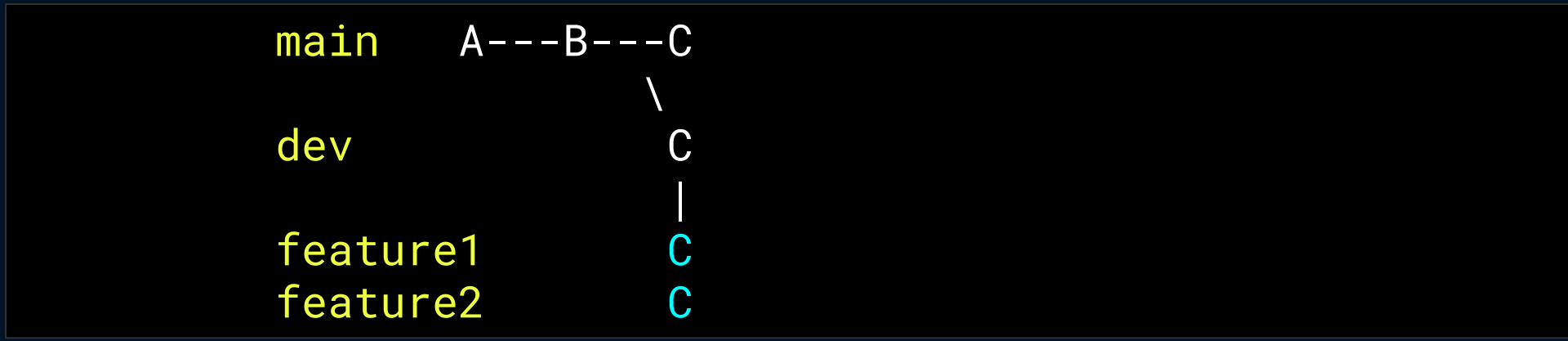
```
Création de la branche dev
$ git branch dev

Bascule sur la branche dev
$ git checkout dev
```



Git Concepts clefs

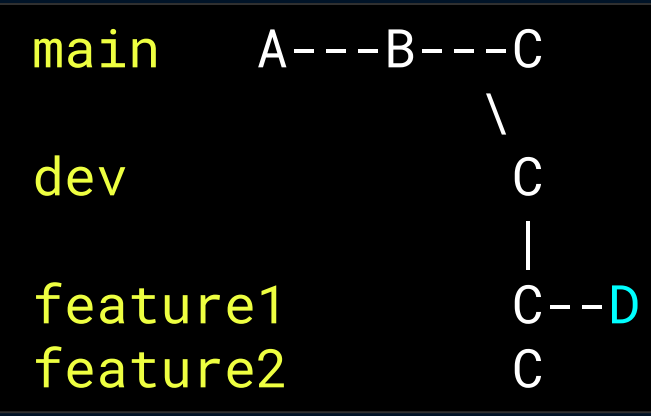
Branche :

```
Depuis la branche dev je crée deux branches features
$ git checkout -b feature1
Je bascule sur dev
$ git checkout dev
$ git checkout -b feature2
```



Git Concepts clefs

```
Feature1 développée dans mon working directory je la stage et commit sur le repo
$ git status
$ git add <FILES>
$ git commit -m “add feature1”
Si le repo origin est remote, je publie mon commit
$ git push
```

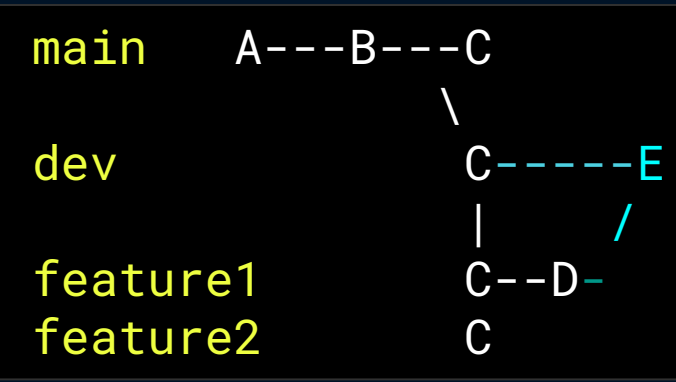


Git Concepts clefs

Merge :

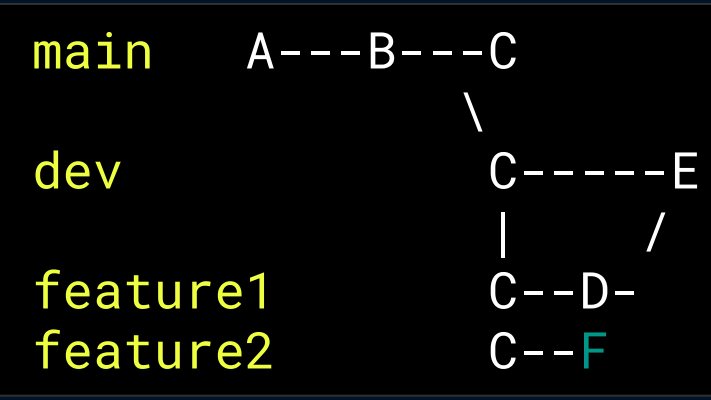
Un merge est une fusion de deux branche, cela crée un commit qui intègre l'historique des deux branches.

```
Je bascule sur dev
$ git checkout dev
Ou
$ git switch dev
Je merge feature1 dans dev
$ git merge feature1
```



Git Concepts clefs

```
Feature2 développée, je la stage, je la commit et la publie
$ git switch feature2
$ git add <FILES>
$ git commit -m “add feature2”
$ git push
```

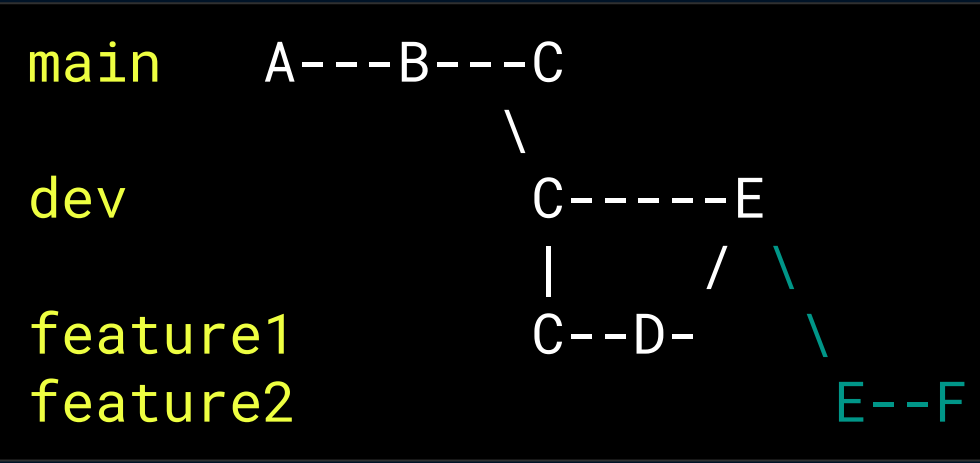


Git Concepts clefs

Rebase :

Le rebase applique les commits de la branche de départ, donnant l'illusion que vous avez créé vos commits à partir d'un point différent.

```
$ git switch feature2
$ git rebase dev
```

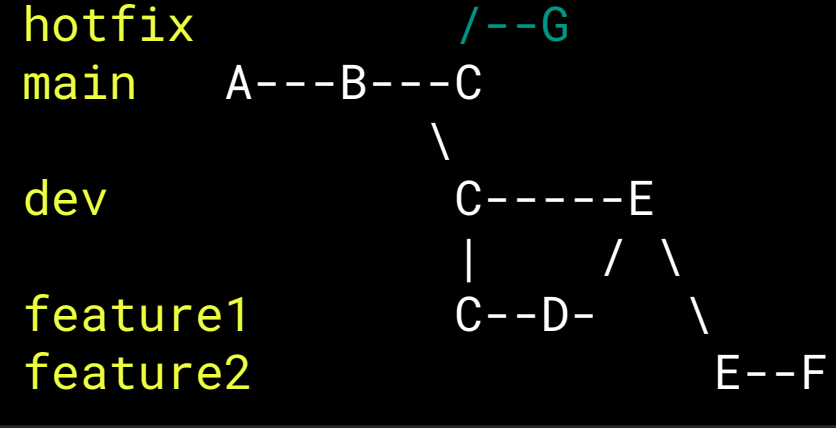


Git Concepts clefs

Hotfix :

Si par exemple une vulnérabilité est découverte sur la branch main, un hotfix est une branche qui va partir directement de la branche main

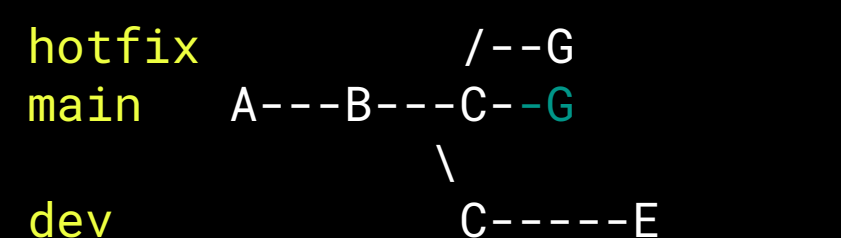
```
$ git switch main
$ git branch hotfix
```



Git Concepts clefs

Hotfix :

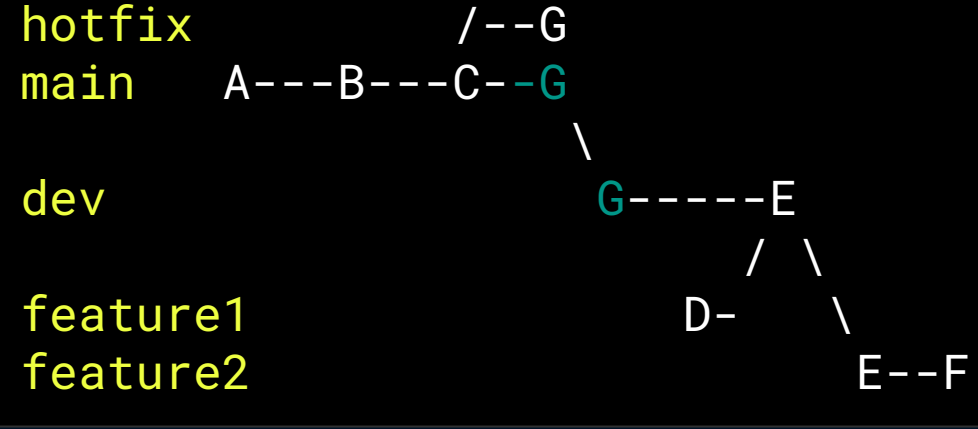
```
Merge du hotfix avec main
$ git switch main
$ git merge hotfix
```



Git Concepts clefs

Hotfix :

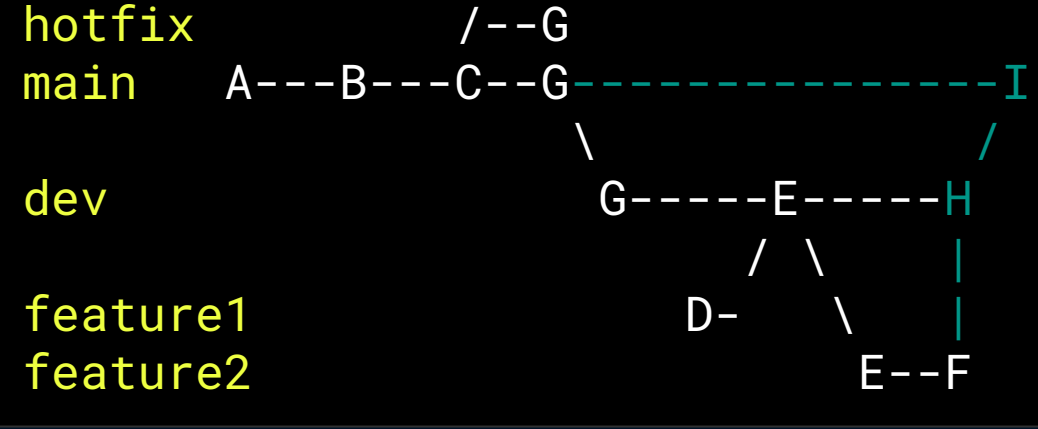
```
Rebase de toutes les branches
$ git switch dev
$ git rebase main
$ git switch feature1
$ git rebase dev
$ git switch feature2
$ git rebase dev
```



Git Concepts clefs

Intégration :

```
$ git switch dev
$ git merge feature2
$ git switch main
$ git merge dev
```



Git Concepts clefs

Log :

```
Affiche l'historique des commits
$ git log
$ git log --graph --oneline
```

Git Concepts clefs

Attention :

Regression : Ces opérations, merge, rebase, hotfix, rebase en cascade, peuvent générer ce que l’on appelle des régressions (altération des fonctionnalités existante).
C’est pourquoi chaque modification, correction, intégration de nouvelles fonctionnalités doivent déclencher des tests de régression.

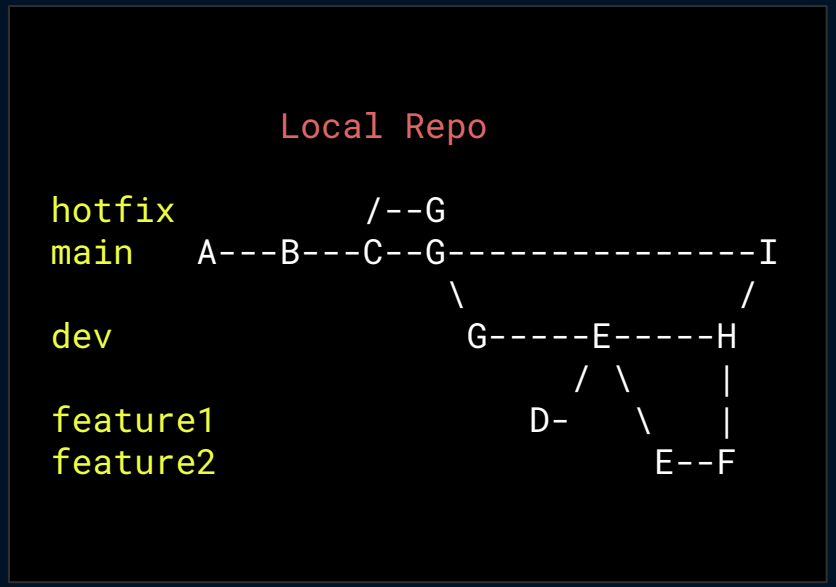
Activité

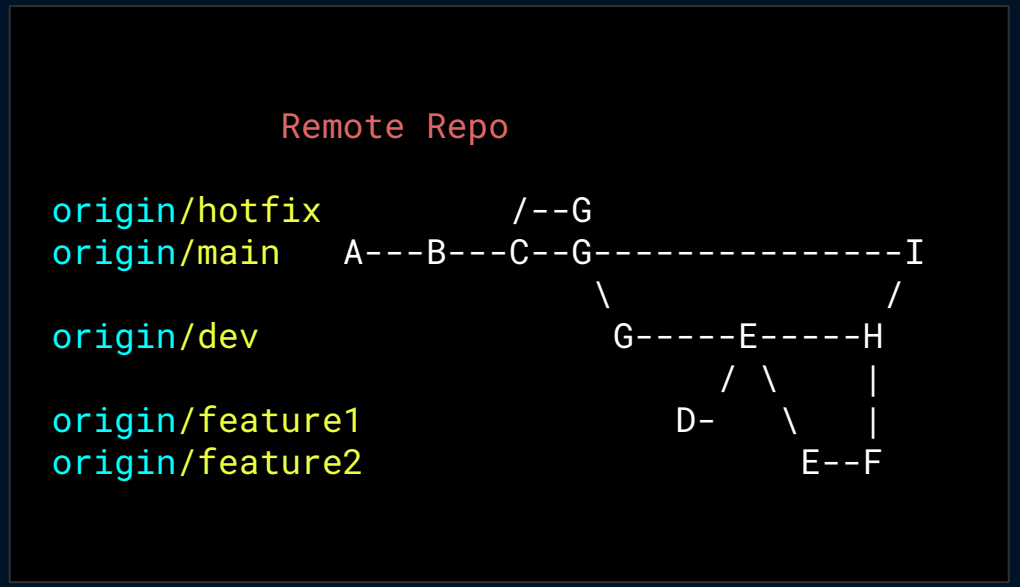
https://x0ne.training/git

Git Concepts clefs

Clone :

```
$ git clone <PATH_SRC> <PATH_SRC>
```





Git Concepts clefs

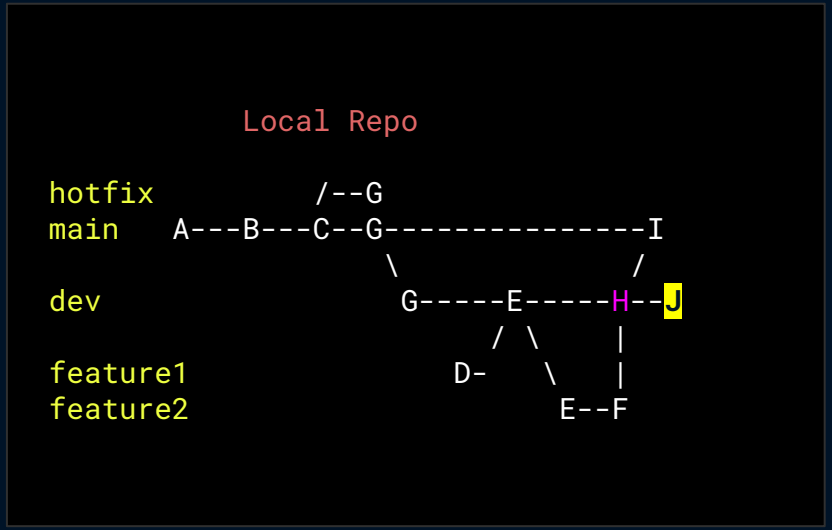
Attention :

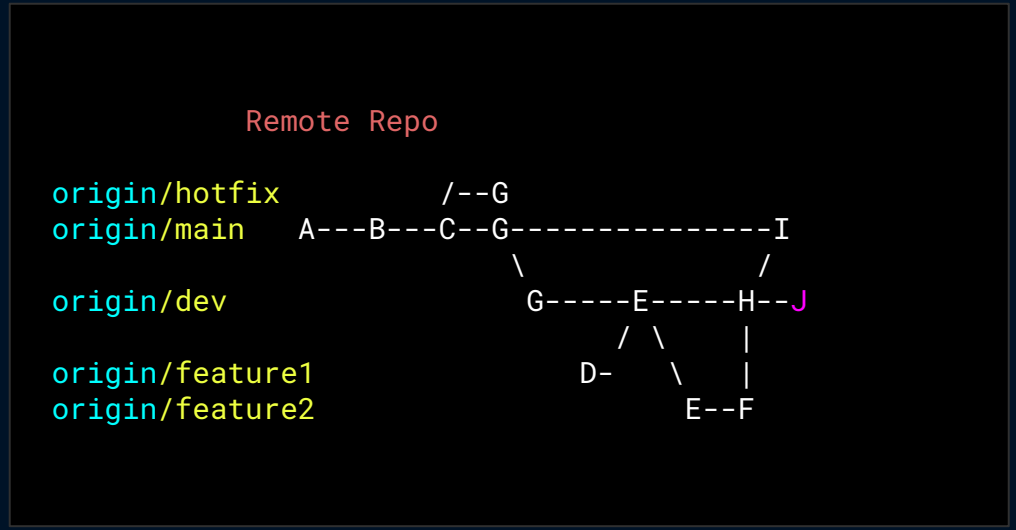
Un dépôt "non-bare" a une copie de travail qui peut entrer en conflit avec les pushes directs, alors qu'un dépôt "bare" est conçu spécifiquement pour servir de point central de synchronisation.

Git Concepts clefs

Fetch :
Permet de récupérer les modifications des branches distantes sans les fusionner avec les branches locales.

```
$ git fetch
$ git diff origin/<BRANCH>
$ git diff origin/dev
```





Git Concepts clefs

Gitignore :

Le fichier .gitignore permet d’inclure le nom des fichier que l’on souhaite garder dans son working directory mais sans les stage ou les commit :

- Fichiers de configuration spécifiques à l'environnement local
- Fichiers temporaires ou de cache
- Fichiers de build ou de compilation
- Dépendances externes (comme le dossier node_modules pour les projets JavaScript)
- Fichiers de logs
- Fichiers contenant des informations sensibles (comme les clés API)
- Fichiers spécifiques au système d'exploitation (.DS_Store pour macOS)
- Fichiers spécifiques aux IDE ou éditeurs de texte (.vscode, .idea)

```
$ cat .gitignore
tmp
build
.env
```

Conflits dans Git

Un conflit survient quand Git ne peut pas fusionner automatiqument les modifications sur les mêmes parties d’un fichier.

Causes courantes :

- Modifications des mêmes lignes dans différentes branches.
- Suppression d’un fichier dans une branche alors qu’il est modifié dans une autre.

Conflits dans Git

Identifier un conflit :

Lors d'une fusion ou rebase, Git affiche un message d'erreur indiquant les fichiers en conflit.

Les fichiers concernés contiennent des marqueurs spéciaux

```
<<<<<<< HEAD
Votre modification
=======
Modification de la branche fusionnée
>>>>>>> nom_de_la_branche
```

Conflits dans Git

Résoudre un conflit :

- Éditez le fichier pour intégrer la solution choisie
- Ajouter le fichier au staging
- Pour un rebase “git rebase --continue”
- Pour un commit “git commit”

Les workflows

Les workflows sont une série de pratiques et de conventions pour organiser et gérer les branches, commits, et collaborations dans un projet. Il définit comment les développeurs créent, fusionnent et déploient les modifications de code.

Au cours de vos projet de groupe je vous encourage à utiliser le “Git Flow”.

Les workflows

Au cours de vos projet de groupe je vous encourage à utiliser le “Git Flow”.

Branches principales :

- main : version stable, prête pour la production
- Dev : version de développement, où sont fusionnés les features avant d’être validées pour la production.

Branches de travail :

- feature/*
- hotfix/*

GitHub Concepts clefs

Issues :

Les Issues sont utilisées pour le suivi des tâches, des améliorations et des bugs. Caractéristiques importantes

- Peuvent être catégorisées avec des labels
- Assignables à des membres spécifiques de l'équipe
- Peuvent être organisées en milestones (jalons)
- Permettent des discussions et le suivi de l'avancement

GitHub Concepts clefs

Projects :

GitHub Projects est un outil de gestion de projet flexible.

- Tableaux kanban personnalisables
- Intégration avec les issues et les pull requests
- Possibilité de créer des vues personnalisées (tableaux, listes, etc.)
- Automatisation de certaines tâches (par exemple, déplacer une carte quand une PR est fusionnée)

GitHub Concepts clefs

Pull Requests (PR) :

Les Pull Requests sont un moyen de proposer des modifications à un projet. Voici les points clés :

- Permettent de discuter des changements avant de les fusionner dans la branche principale
- Facilitent la revue de code par les collaborateurs
- Peuvent être liées à des issues
- Offrent un espace pour les commentaires, les suggestions et les approbations

Monorepo

Un seul repository pour plusieurs projets ou services, Google, Facebook, Microsoft utilisent des monorepos pour gérer leurs énormes projets.

Quand l’utiliser :

- Projet avec beaucoup de code partagé.
- Besoin de versions synchronisés.
- Grosses équipes qui travaillent sur plusieurs parties du projet.

Quand l’éviter :

- Projets complètement indépendants.
- Si il y a beaucoup de binaires.
- Si la taille est trop conséquente.