

Forges, git, intégration continue et autres outils

Matthieu Boileau, Pierre Gambarotto, Henri Massias, Franck Pérignon

ANF Formation aux outils de versioning et d'automatisation d'intégration et de
déploiement - Marseille, 28/09 - 02/10 2026

This work is licensed under



Le programme ...

Deux sessions pour démarrer cette ANF

- Gérer un projet (sous git) à travers une forge Gitlab
- Pratique des mécanismes d'intégration, de livraison et de déploiement continus avec Gitlab

Organisation

- Cibles : ASR et/ou personnes contribuant à des logiciels.
 Attentes potentiellement différentes ! A vous de piocher ce qui vous intéresse ...
- Alternance de "cours" et de sessions pratiques,
- Travail collectif, en "équipes", pour essayer d'être proche de votre quotidien,
- A emporter  - Tous les supports sont et resteront disponibles ici :
<https://plmlab.math.cnrs.fr/anf-mathrice-2026/forges-git>

Mais qui va vous parler   ...

Objectifs

À la fin de ces deux jours, nous aimerions que vous

- ayez compris ce qu'est (ou n'est pas) une Forge, l'intégration continue, ...,
- soyez convaincu(e)s de l'utilité de ces outils pour votre activité quotidienne,
- soyez autonomes pour un usage standard/basique,
- ayez une bonne maîtrise de Git,
- soyez au fait de bonnes pratiques,
- soyez capables de mettre en place de l'intégration continue,
- ...

Avertissements

- Ce cours est accessible même pour un public (presque ...) novice.
Notre hypothèse : vous avez déjà une pratique forge/git. Oui ? Non ?
- Notre pratique, notre expérience.
- Sessions interactives : n'hésitez pas à nous interrompre et à tout tester !



Session 1

Gérer un projet (sous git) à travers une forge Gitlab

1 Introduction et rappels

Forges - Principe et fonctionnalités

Pourquoi utiliser une forge ?

Quelques intermèdes ...

Quelle(s) forge(s) utiliser ?

Rappels des bases d'utilisation

2 Authentification et accès : clés et tokens

3 L'API Gitlab

4 Git - Rappels des bases

5 Git - Les branches

6 Git avancé : historique, conflits et outils

7 Git, travailler avec des dépôts distants

Introduction et rappels

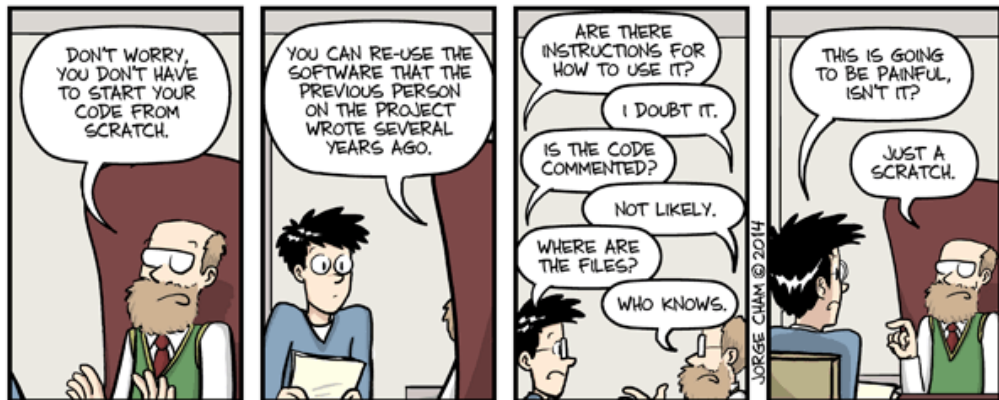
De quoi va t'on parler ?

Forges et gestionnaires de développements collaboratifs

Des plateformes dont le but est de faciliter le travail collaboratif, la gestion de projets de développements (au sens large).



Ça, c'était avant ...



WWW.PHDCOMICS.COM

Aujourd'hui, les Forges ! Systèmes de gestion de projets de développements collaboratifs

Collaborations autour du développements de codes (logiciels, applications)

- Usage connu/standard, à l'origine du développement des forges.
- Aujourd'hui : un outil indispensable et largement utilisé par la communauté des développeurs de logiciels.

Vous confirmez 👁️👁️ ?

🤔 *Logiciel ? On parle de code pouvant aller d'un script shell de quelques lignes à une librairie parallèle de plusieurs centaines de milliers de lignes. Vous êtes donc probablement concerné(e)s !*

Forge : un outil indispensable pour le développement logiciel ...

Le bon cadre et les bons outils pour la mise en place de bonnes pratiques de développement/

Mais aussi pour d'autres utilisations

- Rédaction collaborative (ou pas) d'articles scientifiques, de manuscrit de thèse, ...
- Enseignement (cours, matériel pour les travaux pratiques, ...)
- Génération, publication et maintenance de sites web
- Espace de partage de fichiers, de données ... Oui mais pas trop



une forge n'est pas

- un entrepôt de données type <https://recherche.data.gouv.fr/fr>
- un NextCloud ou DropBox

Forges - Principe et fonctionnalités

Principe :

- Rassembler des utilisateurs ayant des rôles et des profils différents autour de projets dans des groupes .
- Mettre à disposition un ensemble d'outils adaptés pour développer, gérer, suivre, diffuser, valoriser, les projets.

Forges - Principe et fonctionnalités

Principe :

- Rassembler des utilisateurs ayant des rôles et des profils différents autour de projets dans des groupes .
- Mettre à disposition un ensemble d'outils adaptés pour développer, gérer, suivre, diffuser, valoriser, les projets.

En pratique :

- un site web,
- des personnes (login) associées à des projets,
- des outils configurables via l'interface web,
- différents niveaux de droits sur les outils et les projets (visibilité, écriture ...).

Revue (non exhaustive) des outils et fonctionnalités disponibles

- **Structuration** : groupes, projets et utilisateurs authentifiés
 - collaborations possibles avec n'importe qui
 - gestion fine des droits d'accès et d'utilisation de tous les outils
 - gestion de la visibilité des groupes, des projets, des fichiers.

Revue (non exhaustive) des outils et fonctionnalités disponibles

- **Structuration** : groupes, projets et utilisateurs authentifiés
 - collaborations possibles avec n'importe qui
 - gestion fine des droits d'accès et d'utilisation de tous les outils
 - gestion de la visibilité des groupes, des projets, des fichiers.
- **Planification**, ordonnancement, gestionnaire de tâche et de problèmes (**issues/work items**)
 - Constituer des **communautés** (contributeurs, utilisateurs, étudiants ...)
 - **Suivre**, organiser le travail de l'équipe
 - **Collecter**, consigner et suivre les problèmes, les défauts du produit
 - **Gérer** les demandes et les réponses d'utilisateurs, tickets
 - **Partager** de l'information, espace d'échange, wiki

Revue (non exhaustive) des outils et fonctionnalités disponibles

- Gestion de versions

Pratique consistant à conserver/maintenir toutes les versions d'un ensemble de fichiers.

- Centraliser l'activité
- Archiver et suivre les modifications, conserver l'historique, les évolutions
- Permettre/faciliter/gérer les contributions (merge-requests ...)

Revue (non exhaustive) des outils et fonctionnalités disponibles

- Gestion de versions

Pratique consistant à conserver/maintenir toutes les versions d'un ensemble de fichiers.

- Centraliser l'activité
- Archiver et suivre les modifications, conserver l'historique, les évolutions
- Permettre/faciliter/gérer les contributions (merge-requests ...)

- Partage, gestion , de documents et fichiers

- **Sauvegarde** (et donc restauration possible), archivage
- **Edition en ligne**

Revue (non exhaustive) des outils et fonctionnalités disponibles

- CI/CD Intégration, livraison et déploiement continu
 - **CI (continuous integration)** : *pratique consistant à vérifier systématiquement et automatiquement l'impact de toute modification du code source sur le "fonctionnement"* .
 - **CD (continuous delivery)** : *produire et mettre à disposition. Construire, tester, diffuser.*
 - **CD (continuous deployment)** : *livraison de "fonctionnalités" logicielles, par le biais de déploiements automatisés*

Revue (non exhaustive) des outils et fonctionnalités disponibles

- CI/CD Intégration, livraison et déploiement continu
 - Permet de créer du logiciel à partir de son code source (compilation, tests automatiques, assurance qualité, diffusion des livrables)
 - Production de versions diffusables, utilisables
 - Reproductibilité
 - Génération, stockage et fourniture d'images (code prêt à l'emploi, notebooks, ...), "container registries"

Revue (non exhaustive) des outils et fonctionnalités disponibles

- **Gitlab Pages** : publication, hébergement et maintenance de sites web
 - création et déploiement de sites web
 - production de documentation,
 - fait partie du processus CI/CD
- Interaction avec Software Heritage (🤔 ? Nous en reparlerons plus loin ...)

Pourquoi utiliser une forge ?

Un outil fonctionnel et facile d'accès !

- un accès à tous les outils via un seul portail web ;
- des outils bien intégrés, configurables pour chaque projet, une utilisation relativement intuitive ;
- ne nécessite aucune installation préalable ;
- un accès multi-sites, multi-utilisateurs ;
- un outil adapté à
 - des degrés de participation variés : développement, encadrement, tests, diffusion ...
 - des contextes d'utilisation très différents : recherche, collaboration industrielle, enseignement, ...

Pourquoi utiliser une forge?

Un outil utile (indispensable) à plusieurs échelles

Collaboration "interne" au projet, travail collectif aisé

- Espace de travail (git ...) et d'échanges.
- Statistiques, activité, état d'avancement du projet.
- Collaboration inter-organismes
- Pérennisation du projet (sauvegardes, doc ...)

Pourquoi utiliser une forge?

Un outil utile (indispensable) à plusieurs échelles

Collaboration "interne" au projet, travail collectif aisé

Visibilité, diffusion du projet.

Créer/animer une communauté de développeurs, de contributeurs et d'utilisateurs.

- Interface de contacts et d'échanges, communauté (support en ligne, documentation, suivi ...).
- Mise à disposition de documents, de code, d'outils ...
- Vue sur l'activité, la maturité du projet.
- Pour une structure : exposer l'ensemble de la production du site

Pourquoi utiliser une forge?

Un outil utile (indispensable) à plusieurs échelles

Collaboration "interne" au projet, travail collectif aisé

Visibilité, diffusion du projet.

Devops faire collaborer des D v et des ASR

- Code initial : Dev
- code -> livrable : forge par code Dev
- livrable -> d ploiement : forge par code ASR

Cette collaboration  troite peut se faire au sein d'un m me cerveau

Adopter de bonnes pratiques et aller vers la science ouverte

- Favoriser le travail collaboratif, faciliter la vie de tout le monde, collaborateurs ET utilisateurs (présents et futurs)
- Fournir et valoriser des produits de qualité, partageables et réutilisables
- Être dans une démarche de "recherche reproductible"

Les forges et la gestion de version sont au coeur de ces processus



Créer un projet sur une forge devrait être la première étape de tout projet (de développement)

Intermède reproductibilité ...

Ce qui se passe parfois, souvent, ...

- 😱 Mes résultats ont changé ! Pourtant je n'ai touché à rien ...
- Mon code ne marche plus ! Pourtant il fonctionnait la semaine dernière 🤔
- 😞 Impossible de retrouver les résultats de cet article, pourtant j'ai réimplémenté l'algorithme car le code n'était pas disponible
- Impossible d'installer le code qui est référencé dans cet article 💀



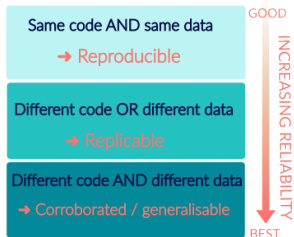
Ce qu'on voudrait (ou qu'on devrait vouloir !)

Donner l'opportunité de vérifier ou ré-obtenir des résultats (maintenant ou plus tard)

- à soi-même
- à d'autres membres de son équipe/labο
- à d'autres personnes (académiques, lecteurs d'une publi ...)



Un sujet crucial que vous ne pouvez pas ignorer !



Reproductibilité : même code (et mêmes données) $\implies$ mêmes résultats

Quels bénéfices à aller vers la reproductibilité ?

- **Augmenter la confiance** dans les produits de la recherche, pour soi, pour les collègues et pour le grand public en général
- **Encourager la réutilisation** : codes et environnements peuvent être repris par d'autres équipes, dans un autre cadre
- **Optimiser l'usage des ressources** : temps, financements et infrastructures mieux utilisés.
- **Garantir la fiabilité** : cela permet de vérifier que les conclusions ne reposent pas sur d'éventuelles erreurs.

Pub !



<https://www.recherche-reproductible.fr>

Soutien du ministère mais on manque de bras !

Un groupe de travail dédié aux "logiciels et environnements computationnels"
`gt-env-logiciel@groupe.renater.fr`

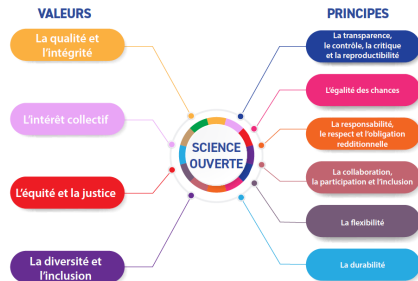
Intermède science ouverte ...

Science ouverte - Définitions et motivations

Diffusion sans entrave des résultats, des méthodes et des produits de la recherche scientifique.

Rendre la production scientifique (issue en majorité de fonds publics) accessibles à tous et dans tous les niveaux de la société

- Reproductibilité
- Science cumulative
- Confiance : transparence, éthique, intégrité
- Visibilité et traçabilité
- Accélération de la circulation des productions scientifiques
- Ouverture de nouveaux champs de recherche, de nouvelles collaborations
- Pérennité d'accès aux travaux scientifiques



Source : Unesco

Quelles obligations et recommandations ?

- Obligations
- Recommandations



Publications



- Rendre disponible en accès ouvert immédiat sous une licence Creative Commons CC-BY ou équivalente, toute publication scientifique issue d'un projet.
- Dépôt du texte intégral dans HAL ou dans une autre archive ouverte.



- Obligation de diffuser en libre accès toutes les formes de publications
- Accès ouvert immédiat sans embargo par le dépôt dans une archive ouverte.
- Frais de publications dans les revues hybrides non éligibles.



Données

- Plan de gestion de données
- « Aussi ouvert que possible, aussi fermé que nécessaire »

- Plan de gestion de données.
- Gestion des données FAIR.
- Partage des données dans des entrepôts de référence selon le principe « aussi ouvert que possible, aussi fermé que nécessaire ».



Codes

- Mise à disposition sous licence libre, archivage des codes dans Software Heritage et référencement dans HAL

- Accès aux résultats nécessaire pour la validation des conclusions scientifiques.
- Accès ouvert aux logiciels

En pratique, que faire ? Diffuser - Archiver - Signaler

Le triptyque Forge-Software heritage-HAL

ou comment utiliser une forge gitlab pour faciliter le référencement de logiciel ?

Forge = brique de base pour la **diffusion** de vos logiciels et outils numériques.

Les autres acteurs :

 Software Heritage (archiver)

 **HAL**
science ouverte (signaler)

Diffuser - Archiver - Signaler

Pourquoi archiver ?

- Le code source est fragile (obsolescence des formats et matériels, dépendances aux outils, perte des codes ...).
- Logiciels : piliers des processus de recherche il est essentiel de les préserver.

Une solution :  Software Heritage

"collecter, préserver et partager tous les logiciels disponibles publiquement sous forme de code source"



archive universelle, collecte de l'intégralité des logiciels disponibles publiquement sous forme de code source.

Diffuser - Archiver - Signaler

Pourquoi signaler ?

- Assurer la description, faciliter la recherche
- Permettre la citation, valoriser les logiciels



Créez une fiche logiciel dans une archive ouverte pluridisciplinaire.



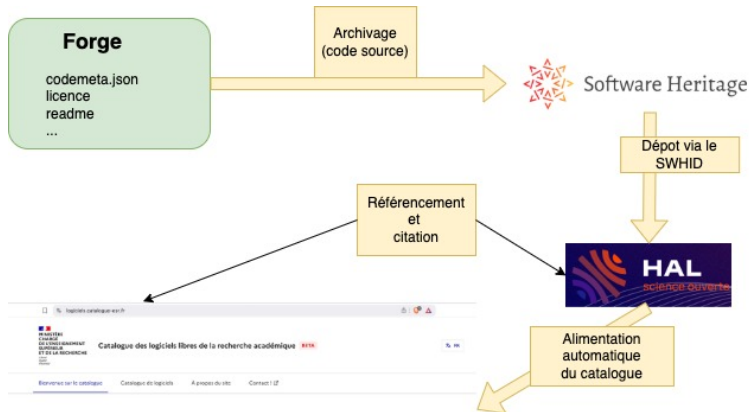
Ajout d'un fichier codemeta.json dans votre repo git pour préparer une notice HAL.

Voir [codemeta-generator](#)

Diffuser - Archiver - Signaler

Catalogue des logiciels libres développés dans la recherche académique.

<https://logiciels.catalogue-esr.fr/>



Comment aller vers la science ouverte et une recherche reproductible ?

Deux points importants/indispensables :

- Disposer des codes sources, qui doivent donc être ouverts/accessibles.
- Pouvoir retrouver les mêmes environnements de travail (compilateurs, ...)



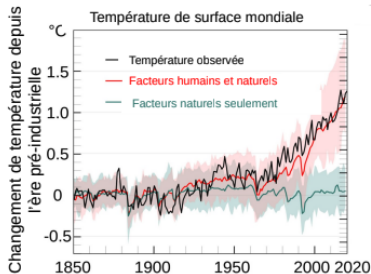
Les outils présentés dans cette ANF sont au coeur de ces processus

Intermède impact environnemental ...

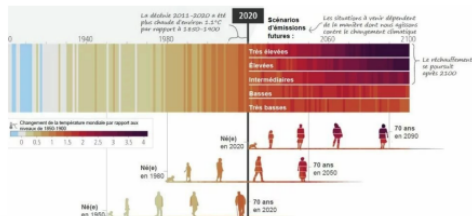
1. Les faits sur le changement anthropique du climat



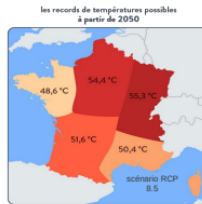
2024 : +1.55°C ; 2019 - 2024 : +1.32°C
(par rapport à l'ère pré-industrielle)



Source : IPCC, AR6 - WGI - 2021



Une forte accélération depuis ~1950

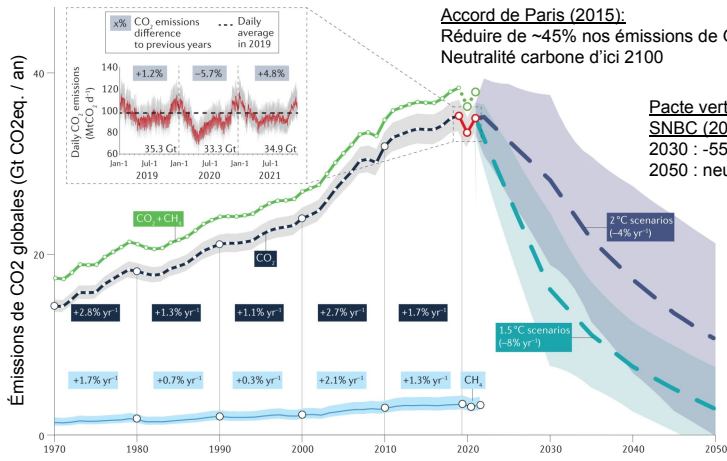


Source : © Copernicus, CMR5 et Météo-France



3. Les scénarios pour demain

(GES = gaz à effet de serre)



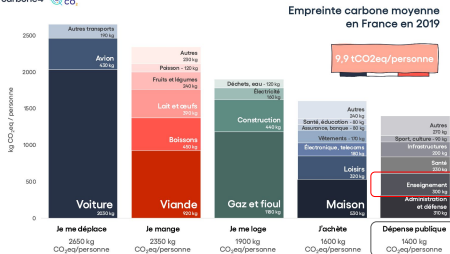
4. Sources des émissions des gaz à effet de serre

Des émissions très inégales : 10% des plus riches au monde sont responsables de **~50% des émissions mondiales**
(10% des plus riches au monde correspond, en France, à un un salaire de 1-2 x le SMIC)



à l'échelle **individuelle** (moyenne nationale)

carbone4 MY CO2



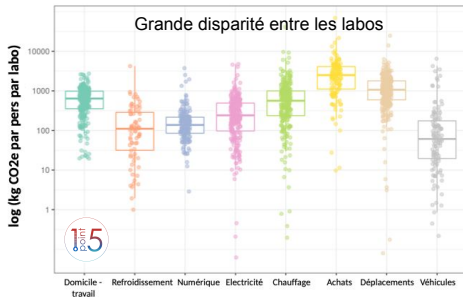
Gaz inclus : CO₂ (hors UTGATF France), CH₄, N₂O, HFC, PFC, H₂O (brûlées de condensation).

Source : MyCO₂ par Carbone 4 d'après le ministère de la Transition écologique, le Haut Conseil pour le Climat, le CITEPA, Agribalyse V3 et INCA 3.

Neutralité carbone d'ici 2050 : 2t CO₂e. / pers. / an
Le service public doit faire sa part

Empreinte individuelle sur <https://nosgestesclimat.fr/>

à l'échelle des **laboratoires** (GES1point5)



Bilan des gaz à effet de serre (BGES) médian des labos en 2019 :
6,6 t CO₂e. / agent / an

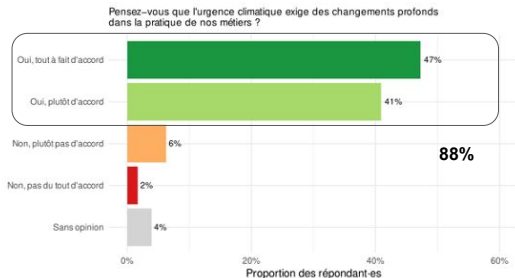
1155 laboratoires en France utilisateurs de GES1point5 (avril 2024)

5. Le rôle de l'enseignement supérieur & la recherche



“La prise en compte des **impacts environnementaux** de la recherche doit être considérée comme relevant de **l'éthique de la recherche**, au même titre que le respect de la personne humaine ou de l'animal d'expérimentation”

(Rapport du COMETS, 5 décembre 2022)



Lecture: 47% des répondant-es pensent que l'urgence climatique exige des changements profonds dans la pratique de nos métiers
Source : enquête « Les personnels de la recherche face au changement climatique », Labos 1point5, 2020
Champ : personnels affiliés à une unité du CNRS (n=6342)

Blanchard et al. (2022) sur la base de plus de 6000 répondant-es parmi de nombreuses disciplines et tous les statuts de recherche

Impact environnemental du numérique



Quelle empreinte écologique ?

... et de l'IA générative en particulier ?

Vaste sujet de réflexion qu'il nous semble indispensable d'évoquer.

Quelques chiffres ...

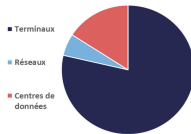


Impacts environnementaux du numérique en France

- L'empreinte carbone du numérique en France : **17 Mt CO₂ eq. soit 2,5 % de l'empreinte nationale**
- **10 % de la consommation électrique française** soit 48,7 TWh par an

407 kg de CO₂ : c'est l'empreinte GES du numérique d'un-e français-e. C'est déjà 25% de son objectif de bilan carbone 2050

Mais l'empreinte d'ici 2050 en France pourrait tripler si aucune action.



L'empreinte carbone provient :

- **des terminaux (79 %) ;**
- puis des centres de données (16 %) ;
- et enfin des réseaux (5 %).

"En 2023, l'ensemble du numérique mondial consomme à lui-seul 27 % du budget soutenable planétaire annuel pour rester en dessous de 1,5°C de réchauffement climatique"

Sources :

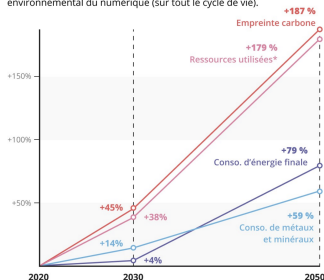
- « Impacts environnementaux du numérique dans le monde, troisième édition », Association Green IT, 2025 (EENM 2025), <https://t.ly/greeniteco.IENM2025>
- BRILLAND Thomas, FANGEAT Erwann, MEYER Julia, WELLHOFF Mathieu, ADEME, 2025. Evaluation de l'impact environnemental du numérique en France. <https://librairie.ademe.fr/>

Quelques chiffres ...



Sans actions pour la réduire, l'empreinte carbone pourrait presque tripler en 2050, la consommation d'énergie doubler

Evolution du scénario tendanciel de 4 indicateurs de l'impact environnemental du numérique (sur tout le cycle de vie).



* Définition MIPS prenant en compte les matériaux utilisés, la biomasse, les déplacements de terre, mécaniques ou par érosion, l'eau, et l'air.

Service Sobriété Numérique / Direction Economie Circulaire

"la consommation électrique des centres de données dans le monde était de 485 TWh en 2025, et pourrait doubler d'ici à 2030 pour atteindre l'équivalent de la consommation du Japon."

Sources :

- « Impacts environnementaux du numérique dans le monde, troisième édition », Association Green IT, 2025 (EENM 2025), <https://t.ly/greeniteco.IENM2025>
- BRILLAND Thomas, FANGEAT Erwann, MEYER Julia, WELLHOFF Mathieu, ADEME, 2025. Evaluation de l'impact environnemental du numérique en France. <https://bibliothèque.ademe.fr/>

Un peu de lecture, d'aide à la réflexion ...

- Le travail du [Groupe ecoinfo](#)
- Le travail du GDR [Labo1point5](#)

Sur l'IA générative en particulier

- CNRS :
<https://www.cnrs.fr/fr/actualite/quand-lia-menace-ledifice-des-connaissances-scientifiques>
- Le travail d'[Atecopol](#): une communauté pluridisciplinaire de scientifiques travaillant ou réfléchissant réfléchissant en interdisciplinarité aux enjeux socio-écologiques et au rôle de l'ESR dans ce contexte.
 - Une [Journée d'étude "Face à l'IAg dans l'ESR"](#)
 - Un "[Manifeste d'objection de conscience face à l'IA générative](#)". Ce document est au minimum l'occasion de **prendre de le temps de s'interroger** sur l'utilisation de l'IA générative et montre qu'une partie de notre communauté la considère comme très problématique.
 - Un document "[Réponses à quelques arguments courants en faveur de l'intelligence artificielle générative](#)" est également disponible

Que faire ?

Je ne sais pas 🙄 ...

Mais nous pouvons au minimum

- réfléchir à notre manière de travailler, la faire évoluer,
- nous interroger sur nos pratiques, sur nos thématiques de recherche ...

Et, pourquoi pas, essayer de répondre à la question proposée dans les ateliers Sens-inria

"Est-ce que les résultats de nos recherches, de notre travail, participent à la construction d'un monde qui correspond à nos valeurs ?"

Revenons à nos forges ...

Quelle(s) forge(s) utiliser (et pourquoi) ?

Beaucoup de possibilités :

- forges “externes” à l'ESR, commerciales, communautaires
- forges académiques (ESR = enseignement supérieur recherche)

Quelques critères de choix

- Fonctionnalités disponibles ?
- Ouverture à votre communauté ?
- Pérennité ?
- Contrôle, sécurité des données, souveraineté ... ?
- ...

Quelle(s) forge(s) utiliser ?

Forges externes



La référence **GitHub** $\approx$ 200 millions d'utilisateurs en 2025, 250 millions de projets publics, rachetée par Microsoft en 2019.

Alternatives équivalentes :

 **Bitbucket**

<https://bitbucket.org>

 **GitLab**

<https://about.gitlab.com>

Quelle(s) forge(s) utiliser ?

Forges externes à l'ESR, commerciales ou communautaires



- très fonctionnelles,
- intègrent de nombreux outils,
- peu de contraintes,
- très utilisées, stables, très régulièrement mises à jour et améliorées.

Quelle(s) forge(s) utiliser ?

Forges externes à l'ESR, commerciales ou communautaires



- très fonctionnelles,
- intègrent de nombreux outils,
- peu de contraintes,
- très utilisées, stables, très régulièrement mises à jour et améliorées.



- commerciales, non académiques
- hébergement non contrôlé (respect RGPD, réglementation européenne ? souveraineté ?)
- parfois payantes 💰
- durée de disponibilité indéterminée (Google code ...) ?

Quelques remarque à propos de l'utilisation de Github (et équivalent)

- Conditions d'utilisation ? Extrait de celles de Github

"You grant GitHub the right to store, parse, and display your content, and make incidental copies as necessary to provide the service."

- Data mining ? IA ? Copilot ?
- Incompatible avec stricte confidentialité/souveraineté numérique (Patriot act, cloud act)
- L'utilisation d'une forge commerciale nécessite que les membres (académiques ou pas) acceptent **individuellement** les conditions d'utilisation

Quelle(s) forge(s) utiliser ?

La bonne solution : les plateformes "ESR" auto-hébergées

≈ 81 forges disponibles¹ dans les structures ou labos

Avec quelques limites potentielles (pas d'accès "universel", frein pour contributions ?
Visibilité internationale ? ...)

¹Forges de l'Enseignement supérieur et de la Recherche - Définition, usages, limitations rencontrées et analyse des besoins - hal-04098702

Quelle(s) forge(s) utiliser ?

Fonctionnalités disponibles
Ouverture à une large communauté
Pérennité
Contrôle, sécurité des données, souveraineté ...

Github et cie Forge ESR



Au final, que choisir ?

🤔 Le choix dépendra de votre projet, des gens impliqués, des habitudes de votre communauté, des tutelles impliquées ...

👍 Privilégiez les plateformes académiques

Quelques bonnes pratiques

- Mettre une licence sur ses projets.
- Rester concentré : ne pas publier des passwords, réfléchir avant d'ouvrir les projets, bien gérer les participants ...
- Répliquer les projets (version publique/privée, sur plusieurs plateformes ...)

Dans ce cours : focus sur **GitLab**

- Un logiciel développé par la compagnie Gitlab Inc.
- Une distribution “open source” gratuite (Gitlab CE) et une version propriétaire payante (Gitlab EE) proposant des fonctionnalités supplémentaires.
- De très nombreux déploiements (des site webs !) : des forges auto-hébergées
 - <https://about.gitlab.com/> : la forge de Gitlab inc
 - $\approx 90\%$ des instances répertoriées dans l'ESR

Pour ce cours, nous utiliserons **plmlab**, la forge

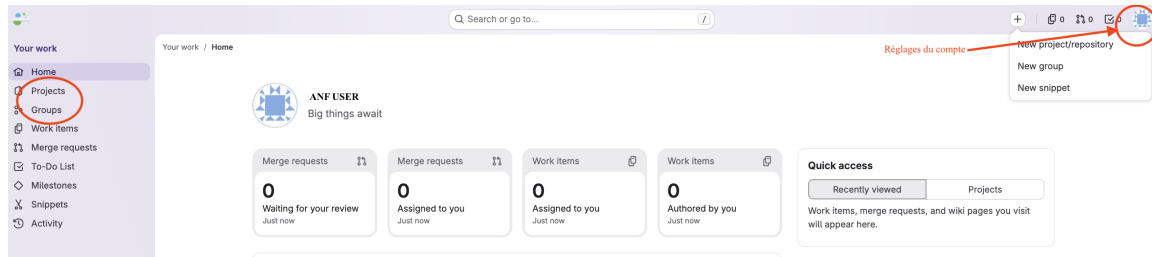


- Plateforme créée en 2017 au sein de la PLM
- A destination de la communauté mathématique et de leurs collaboratrices et collaborateurs
- Accessible via la fédération d'identité Renater à condition d'avoir un compte PLM actif
- Aujourd'hui : 9664 comptes, 9785 projets, 1115 groupes.

<https://plmlab.math.cnrs.fr/>.

Tableau de bord

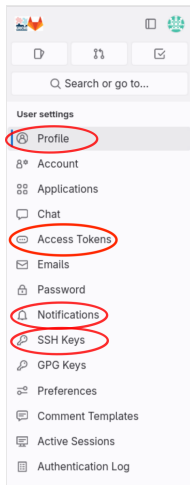
- Configuration du compte.
- Accès aux projets et groupes.
- todo-list, notifications ...



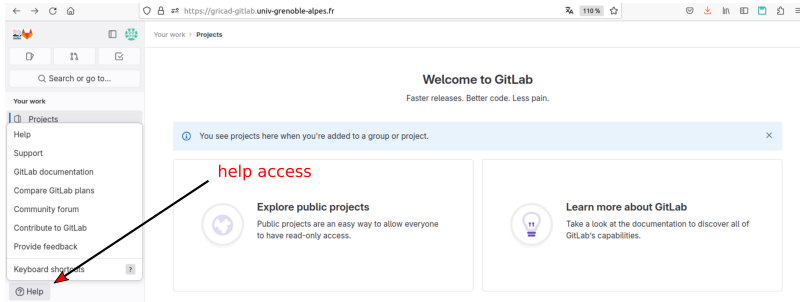
Gestion du compte, le menu Preferences

Premières étapes *indispensables*/bonnes pratiques :

- **Profile** : complétez/vérifiez les différents champs.
(Au minimum “organization”)
- **SSH keys** : déposez une clé ssh (Voir trainings)
Ou comment sécuriser vos échanges avec le serveur ?
- **Notifications** : contrôle du niveau de notifications
Explorez les différentes possibilités et fixez un mode par défaut.
👍 “disabled” pour éviter l’envoi intempestif d’emails
- **Access Tokens** : gestion/automatisation de l’authentification
(API, ...)



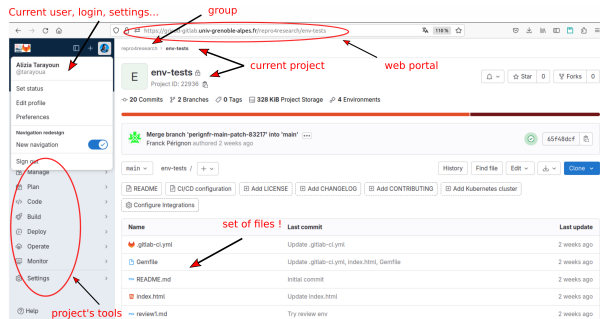
Où trouver de l'aide ?



- La documentation (très complète) intégrée fournie par le logiciel gitlab.

Projets - Un espace pour structurer votre travail

- Héberger, sauvegarder, partager des fichiers,
- Choisir et gérer un ensemble de participants et leurs droits
- Gérer/configurer des outils
- Différents niveaux de visibilité (privé, interne, public)



Création d'un projet

Les informations nécessaires à la création :

- un emplacement (**namespace** $\approx$ dossier), un nom, une description
- un niveau de **visibilité**
 - **Privé** : visible uniquement par les membres du projet (ajoutés par l'admin)
 - **Interne** : visible par tout utilisateur connecté (possibilité de demander à rejoindre le projet)
 - **Public** : visible par n'importe qui

Groupes - Un espace pour organiser vos projets

- Ensemble de projets ou de sous-groupes,
- Associés à un ensemble d'utilisateurs
- Des droits/autorisations par défaut

Informations nécessaires à la création :

- un nom et une description
- un niveau de **visibilité**, comme pour les projets.

 les projets héritent de la visibilité du groupe. Par exemple, un groupe privé ne peut contenir que des projets privés.

Groupes et projets - Namespaces

Namespaces ($\approx$ dossier) : emplacements des projets.

Deux options :

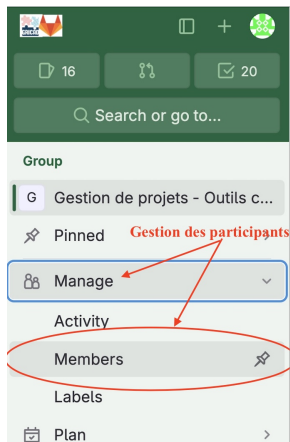
- Espace personnel : votre **username** (login)
 - <https://plmlab.math.cnrs.fr/username/somename>
 - A réserver à vos projets personnels ou temporaires (manuscrit de thèse, forks ...)
 - Non-disponible pour les comptes externes ?
- **Groupe ou sous-groupe.**
<https://plmlab.math.cnrs.fr/grouppname/somename>



Déplacer, renommer, détruire un groupe ou un projet ?

Menu Settings $\rightarrow$ General $\rightarrow$ advanced

Gestion des participants - Rôles et permissions



Les rôles : guest, planner, reporter, security manager, developper, maintenir, owner

Voir le [tableau descriptif des rôles dans la doc Gitlab](#).

- Créateur d'un groupe ou projet : automatiquement "owner"
-  Un utilisateur hérite dans un projet de ses droits sur le groupe
- Un utilisateur peut faire partie d'un projet sans appartenir au groupe.

Groupes et projets, bonnes pratiques

- **Organisez les projets dans des groupes/sous-groupes** thématiques (labo, thèmes de recherche ...)
- Réfléchissez au **nommage des projets**
 - 👉 Impact sur l'affichage/la visibilité du projet (pages web, références etc).
- **⚠ Prenez le temps de configurer** correctement la liste des membres, leurs droits, les rôles, dates d'expiration de participation etc.
- **⚠** aux membres d'un groupe : n'ajoutez que les personnes susceptibles de participer à tous les projets du groupe.

1 *Introduction et rappels*

2 *Authentification et accès : clés et tokens*

3 *L'API Gitlab*

4 *Git - Rappels des bases*

5 *Git - Les branches*

6 *Git avancé : historique, conflits et outils*

7 *Git, travailler avec des dépôts distants*

Authentification et accès : clés et tokens

- Les clés ssh. Voir la page dédiée dans forges-git/trainings.
- Les tokens : suffisamment subtils et complexes pour mériter une section dédiée dans ce cours...

Les tokens GitLab : cas d'usage

Un programme qui agit à la place d'un humain, sans session interactive :

- **API** : scripts, outils (glab, Terraform), automatisation hors CI
- **Git en HTTPS** depuis une machine hors instance : cloner/pousser sans compte interactif (cron, serveur de build)
- **Mirroring de dépôts** : réplication intra ou inter instances
- **Couplage des CI de deux projets** : déclencher un pipeline aval, cloner l'autre projet
- **Registres** : pull/push d'images ou de paquets depuis des machines externes

Les tokens GitLab : cas d'usage

Un programme qui agit à la place d'un humain, sans session interactive :

- **API** : scripts, outils (glab, Terraform), automatisation hors CI
- **Git en HTTPS** depuis une machine hors instance : cloner/pousser sans compte interactif (cron, serveur de build)
- **Mirroring de dépôts** : réplication intra ou inter instances
- **Couplage des CI de deux projets** : déclencher un pipeline aval, cloner l'autre projet
- **Registres** : pull/push d'images ou de paquets depuis des machines externes



Une règle à retenir : quand l'usage est 100 % intra-CI, CI_JOB_TOKEN évite le token à gérer ; sinon, il faut créer un token et gérer son cycle de vie.

Les types de tokens

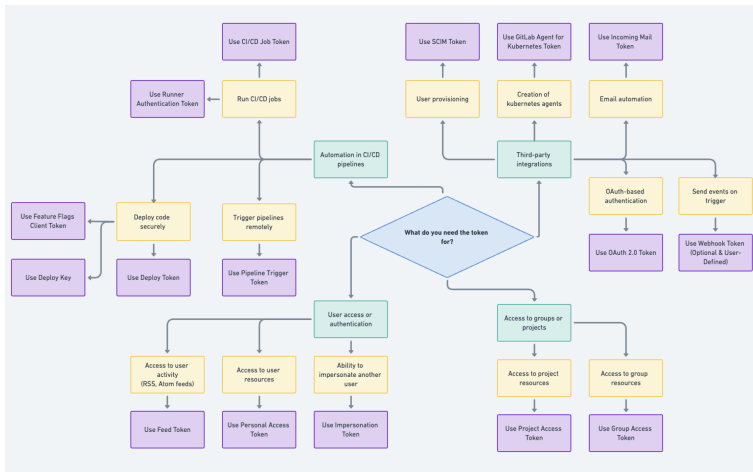
Trois familles principales de tokens :

- les **tokens personnels** (PAT : *legacy* ou *fine-grained*),
- les **tokens de ressource** (projet, groupe),
- le **jeton éphémère de la CI** (CI_JOB_TOKEN).

Les autres tokens, pour mémoire

- **Deploy key / deploy token** : cloner ou déployer du code *read-only* (ou en écriture), **sans compte utilisateur**. Deploy key: clé ssh. Deploy token: couple identifiant/mot de passe.
- **Pipeline trigger token** : déclencher un pipeline **depuis l'extérieur de l'instance** (API trigger).
- **Runner authentication token** : enregistrer un runner auto-hébergé sur un projet, un groupe ou l'instance.
- Autres tokens pour l'administration de l'instance (SCIM, impersonation, OAuth, webhooks ...) : voir <https://docs.gitlab.com/security/tokens/>

Les types de tokens : choisir le bon...



Les tokens personnels : legacy

Jusqu'en 2025, un seul type de token personnel : le *legacy token* (préfixe `glpat-`).

- **Lié à un compte utilisateur** : il fait tout ce que son propriétaire sait faire, sur toute l'instance.
- **Portée par scopes larges** : `api` (tout), `read_api`, `read_repository` ... impossible de le limiter à un projet ou à une action précise.
- **Renommé *legacy* par GitLab** : voué à être remplacé par les *fine-grained tokens*.

Les tokens personnels : fine-grained

- **Double réglage** : une **frontière** (l'utilisateur, un groupe, un projet) + des **permissions par ressource** (create / read / update / delete, ressource par ressource).
- **Double contrôle** à chaque requête : le token doit avoir la permission **et** son propriétaire le droit. Le token ne dépasse jamais le droit de son propriétaire.

Les tokens personnels : fine-grained

- **Double réglage** : une **frontière** (l'utilisateur, un groupe, un projet) + des **permissions par ressource** (create / read / update / delete, ressource par ressource).
- **Double contrôle** à chaque requête : le token doit avoir la permission **et** son propriétaire le droit. Le token ne dépasse jamais le droit de son propriétaire.



En cas de fuite, le rayon d'impact est la frontière du token, et non l'instance entière comme avec un scope api.

Les tokens de ressource : projet et groupe

Le token d'une ressource, pas d'une personne : GitLab crée un **compte bot** membre du projet ou du groupe, avec le rôle choisi à la création.

- **Création** : Settings → Access tokens du projet ou du groupe. Rôle (Guest → Owner) + scopes larges, comme les legacy PAT.
- **Différence clé avec un PAT** : aucun lien avec un humain. Un départ n'y casse rien, plusieurs membres de l'équipe peuvent le gérer, l'inventaire vit dans les réglages du projet/groupe.
-  Date d'expiration obligatoire.

Le jeton éphémère de la CI : `CI_JOB_TOKEN`

Le runner le crée pour **chaque job** (variable masquée `CI_JOB_TOKEN`) et le révoque **à la fin du job**. Pas besoin de gérer la création, la rotation ou la révocation.

- **Permissions** : celles de l'utilisateur qui déclenche le pipeline (`CI_JOB_USER`), bornées au périmètre du job.
- **Portée** : par défaut, le projet lui-même uniquement. Ce n'est pas un PAT : l'API n'accepte qu'une liste d'endpoints (git, registres, paquets).
- **Accès à un autre projet** : liste d'autorisation côté projet cible (Settings → CI/CD → Job token permissions).
-  Limite : uniquement intra-CI sur une instance donnée.

Couplage de CI avec `CI_JOB_TOKEN`

Exemples de `.gitlab-ci.yml`:

```
# Cloner un projet B depuis le job d'un projet A
```

```
build:
  script:
    - git clone https://gitlab-ci-token:${CI_JOB_TOKEN}@${CI_SERVER_HOST}/forge/projet-b.git
```

```
# Déclencher un pipeline d'un projet B depuis le job d'un projet A
# (pas besoin de token : l'utilisateur de A doit avoir le droit de
# déclencher le pipeline de B)
```

```
deploy-b:
  trigger:
    project: forge/projet-b
    branch: main
```

Quels risques ? Quels problèmes ?

Risque

Divulgateion accidentelle de tokens dans le code source

Expiration d'un token non gérée

Token lié à un compte utilisateur supprimé

Solution

→ Utiliser des variables d'environnement ou des secrets

→ Suivre les notifications d'expiration et gérer leur rotation

→ Utiliser des tokens de groupe ou de projet

La vie d'un token : expiration

Quatre actions distinctes, aux conséquences différentes sur les systèmes qui utilisent le token : **expiration, rotation, révocation et suppression.**

Expiration : date limite de validité fixée à la création, le token s'inactive automatiquement, sans intervention.

- Filet de sécurité : un token oublié finit par s'invalider de lui-même.
- Sans expiration : possible sur les comptes de service, mais la responsabilité de la rotation repose alors entièrement sur l'équipe.
-  Un token expiré ne peut plus faire l'objet d'une rotation (GitLab envoie des alertes à l'avance).

La vie d'un token : rotation

Rotation : remplacement d'un token actif par un nouveau, aux mêmes portées et permissions.

- Limite la durée de vie effective du secret sans interrompre le service : à privilégier pour la maintenance préventive.
- Si le token n'est plus utilisé, une simple révocation suffit et est plus rapide.
-  Impossible sur un token expiré ou révoqué. La rotation est proactive, à planifier **avant** l'expiration.

La vie d'un token : révocation

Révocation : désactivation manuelle et immédiate d'un token encore valide ; il reste visible (marqué révoqué) mais inutilisable.

- Invalider un token instantanément en cas de compromission suspectée ou de départ d'un collaborateur.
-  Interrompt immédiatement tous les pipelines et intégrations qui l'utilisent. Si le token est encore légitime, préférer la rotation.
- **Effet de bord** : demander la rotation d'un token déjà révoqué est interprété comme une tentative frauduleuse : GitLab révoque alors le token actif de sa famille, celui que vous utilisez.  Ne jamais relancer une rotation avec l'ancien token.

La vie d'un token : suppression

Suppression : destruction du token et de son enregistrement ; il disparaît de l'inventaire.

- Nettoyer l'inventaire des tokens obsolètes.
-  Pour les tokens de groupe/projet, GitLab conserve les enregistrements inactifs 30 jours après expiration pour l'audit. Une suppression prématurée empêche ce suivi.

Bonnes pratiques : moindre privilège

- Identifier le type de token sur la base de l'organigramme
- Ajuster la portée (est-ce que la lecture seule est suffisante ?)
- Éviter des tokens avec des privilèges d'administrateur
- Révoquer les tokens devenus inutiles
- Stocker leurs valeurs dans des zones protégées (hors du code source !)

Bonnes pratiques : rotation

- Un token par usage : petit rayon d'impact, rotation sans crainte
- Expiration systématique dès la création
- Faire la rotation tôt : à l'alerte (60/30/7 jours), pas au bord de l'expiration
- La valeur vit à un seul endroit (variable CI/CD, vault) : faire la rotation, c'est mettre à jour cette unique donnée
- L'ancien token est révoqué immédiatement : propager puis tester aussitôt
- Tenir un inventaire : token, usage, date d'expiration, responsable

Bonnes pratiques : rotation automatisée

- Automatisation possible (job CI planifié via l'API, ou Terraform), mais ne jamais retenter avec l'ancien token , sinon le token est révoqué par sécurité
- Un pipeline planifié appartient à son créateur et tourne sous son identité : s'il quitte le projet, le schedule s'arrête silencieusement. Utiliser plutôt un **compte de service** (ci-bot)

Références

- Article sur le blog gitlab.com :

*[https://about.gitlab.com/fr-fr/blog/
the-ultimate-guide-to-token-management-at-gitlab/](https://about.gitlab.com/fr-fr/blog/the-ultimate-guide-to-token-management-at-gitlab/)*

- Documentation officielle :

<https://docs.gitlab.com/security/tokens/>

- ① *Introduction et rappels*
- ② *Authentification et accès : clés et tokens*
- ③ *L'API Gitlab*
- ④ *Git - Rappels des bases*
- ⑤ *Git - Les branches*
- ⑥ *Git avancé : historique, conflits et outils*
- ⑦ *Git, travailler avec des dépôts distants*

L'API Gitlab

Une API (Application Programming Interface) permettant d'interagir avec GitLab via la ligne de commande, des scripts, sans passer par l'interface web.

Quels intérêts ?

- Automatiser des tâches d'administration et de développement
- Interagir (créer, modifier, monitorer ...) avec vos ressources : projets, groupes, utilisateurs ...
- Intégrer GitLab avec d'autres outils (éditeurs ...)

Comment ?

- Requêtes http (curl ...)
- Python
- emacs, vscode
- ...

Présentation des tps

Les énoncés des tps sont disponibles dans le projet

plmlab.math.cnrs.fr/anf-mathrice-2026/forges-git/trainings



Ce projet contient également des **compléments du cours**.

Une zone expérimentale, bac à sable 

Le groupe plmlab.math.cnrs.fr/anf-mathrice-2026/forges-git/sandbox



N'hésitez pas à faire n'importe quoi dans ce groupe, (sauf supprimer le travail de quelqu'un d'autre) !



TP 1 - Gestion du compte, clés ssh, tokens et API

Une première étape pour vérifier que tout est fonctionnel pour vous sur plmlab ...

- 1 Connectez vous sur la plate-forme via votre identifiant plm
(Vous n'en avez pas ? Faites nous signe !) et vérifiez votre accès au groupe [anf-mathrice-2026/forges-git](#)
- 2 Visitez le menu "settings" de votre compte et complétez tout ce qui vous paraît utile. Assurez vous notamment d'avoir un clé ssh valide et associez cette clé à votre compte plmlab.

Utilisation de l'API

- 3 Visitez le projet [anf-mathrice-2026/forges-git/trainings](#), organisez votre équipe.
- 4 Faites le tp [gitlab api](#).



A partir d'ici : tous les exercices, tutos et démos seront décrits et disponibles dans ce groupe [anf-mathrice-2026/forges-git](#).

Gestion de version avec Git

- 1 Introduction et rappels
- 2 Authentification et accès : clés et tokens
- 3 L'API Gitlab
- 4 Git - Rappels des bases
 - A propos des gestionnaires de version
- 5 Git - Les branches
 - Git, principes de fonctionnement
 - Git en ligne de commande
 - Ajouter, supprimer des fichiers
 - Examiner l'état, l'historique ...
- 6 Git avancé : historique, conflits et outils
- 7 Git, travailler avec des dépôts distants

Gestion de version

Pratique qui consiste à conserver/maintenir toutes les versions d'un ensemble de fichiers.

Deux fonctions principales :

- permet le **travail simultané** de plusieurs personnes sur un ensemble de fichiers :
 - synchronisation des modifications, fusion automatique des fichiers,
 - détection et résolution (plus ou moins automatique ...) des conflits,
- une **gestion de l'historique** du projet :
 - accès à n'importe quelle version archivée,
 - information sur qui a fait une modification, quand, où, pourquoi ...
 - notifications automatiques aux participants

Bref, un outil indispensable.

Quelques notions de base

- **Repository/dépôt** : les données et méta-données concernant l'ensemble des répertoires et fichiers "versionnés".
⇒ Une référence.
- Des **utilisateurs** i.e. des personnes référencées susceptibles de lire/modifier le dépôt.
- **Commit**/Révision : **enregistrement** dans le repository d'un ensemble de modifications.
Par extension, désigne un état d'un repository, une version, un instantanée de l'ensemble des fichiers.

Quelques exemples de logiciels de gestion de versions :

- cvs (1990) : <http://cvs.nongnu.org>
- subversion (svn, 2000) : <http://subversion.apache.org/>
- git (2005): <http://git-scm.com/>
- mercurial (2005): <http://mercurial.selenic.com/>
- bazaar (2005): <http://bazaar.canonical.com/en/>

Tous sont libres et en général disponibles facilement.

Quel gestionnaire utiliser ?

Git !

- Rapide, efficace, performant ...
- Intégré à Gitlab.
- Le standard actuel. Le plus utilisé, donc difficile à ignorer.

Mais, un petit conseil ...

- Ne soyez pas bornés à l'utilisation stricte d'un seul outil,
- Utilisez ce qui répond à votre besoin
- Evitez quand même les outils (trop) anciens !

La suite ...

- Git : principes de fonctionnement
- Git en ligne de commande : rappel (?) des principales commandes.
- Git : branches et dépôts remotes, lien avec Gitlab.

Git, principes de fonctionnement

Empilement (dans le temps) de versions dans une base de données.

Chaque version d'un repository est un **instantané (snapshot)** de l'état de l'ensemble des fichiers, sauvé dans une base de donnée.

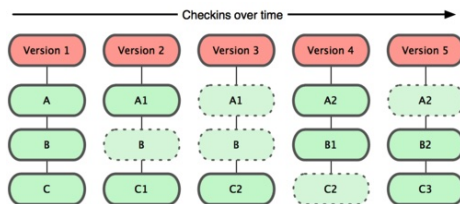


Figure: source, git-scm.com

Nouvelle version :

- copie de l'état pour les fichiers nouveaux ou changés,
- référence ($\approx$ lien) vers les fichiers inchangés.

Git, les différents états d'un fichier

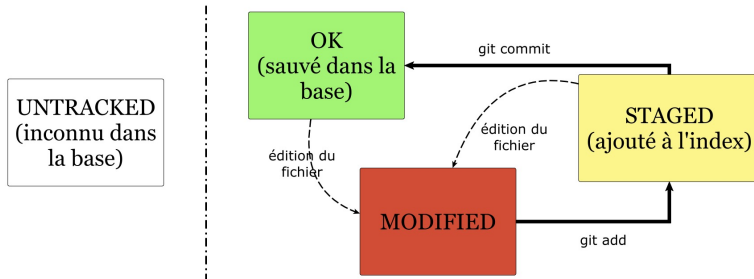
Le cycle simplifié de vie des fichiers d'un projet :

- 1 extraction de la base de donnée d'un "instantané" du projet
⇒ une version de l'ensemble des fichiers : le répertoire de travail .
- 2 modification(s) /éditions des fichiers du répertoire de travail,
- 3 indexage des modifications $\approx$ liste de ce qui a vocation à être versionné, candidats au commit
- 4 validation (commit) : création d'un nouvel instantané, sauvegardé dans la base de données

Git, les différents états d'un fichier

Repository/working directory : extraction d'une version de l'ensemble des fichiers

Du point de vue de git, les fichiers de ce répertoire passent par différents états :



Zone d'index (staging) : stockage des informations concernant ce qui fera partie de la prochaine version.

La commande git

Git s'utilise en général via la ligne de commande (terminal, PowerShell). Le fonctionnement est identique sous Windows, mac ou Linux.

```
git <command> <arguments>
```

Pour un aperçu des différentes commandes, essayez

```
git help
```

ou, pour obtenir la documentation d'une commande particulière :

```
git help <command>
```

Configuration de git

Quelques paramétrages sont nécessaires, à la première utilisation de git :

```
git config --global user.name "Franck Perignon"  
git config --global user.email "..."
```

Tout est enregistré dans un fichier `.gitconfig` à la racine de votre compte,

```
[user]  
  email = ...  
  name = Franck Perignon
```

Configuration de git

Vous pouvez notamment choisir un éditeur par défaut (pour la gestion des conflits par exemple) :

```
git config --global core.editor emacs
```

ou définir des alias en éditant le fichier .gitconfig

```
[alias]  
df = diff origin/main
```

Configuration de git

Niveau	Portée	Fichier	Option
System	Tous les utilisateurs de la machine	/etc/gitconfig	--system
Global	Votre compte utilisateur	~/.gitconfig	--global
Local	Un seul dépôt	.git/config	--local

- En pratique, vous utiliserez principalement `--global` pour vos paramètres personnels et `--local` pour les exceptions spécifiques à un projet.
- Une commande pratique pour afficher toute votre configuration et savoir d'où vient chaque valeur :

```
git config --list --show-origin
```


Configuration de git

- Appliquer une configuration spécifique aux dépôts dans un répertoire.

Dans le fichier ~/.gitconfig ajouter **à la fin** :

```
[includeIf "gitdir:~/repo/plmlab/"]  
  path = .gitconfig-plmlab
```

[Un lien pour plus d'informations.](#)

Créer un repertoire de travail 'git'

Git, première étape : création du répertoire de travail

Méthode 1 : à partir d'un dépôt vide

```
# Création d'un répertoire et/ou cd vers un répertoire existant
cd workdir
ls
fic.md   truc.cxx # Le répertoire n'est pas nécessairement vide
# Initialisation du repo git
git init
```

Résultat

```
ls -a
.git
# ==> La base de données
```

Git, première étape : création du répertoire de travail

Méthode 2 : copie (**clone**) d'un dépôt git existant (**remote**)

```
git clone target  
# exemples :  
# git clone /chemin/vers/depot_local WorkingDir  
# git clone username@server:/chemin/vers/depot
```

Le "dépôt existant" peut être

- un autre répertoire sur votre machine (contenant un .git)
- l'adresse d'un dépôt sur un serveur (accès réseau)

Git, première étape : création du répertoire de travail

Conséquences

👉 Extraction à partir de la base de données de la **dernière version** du repository, vers un répertoire sur votre machine.

👉 Avec un répertoire `.git` contenant toutes les infos sur votre projet.

- Les méta-données et la base de données git
- Les différentes versions de l'ensemble des fichiers de votre projet.

⚠ N'essayez pas d'éditer le contenu du `.git` ...

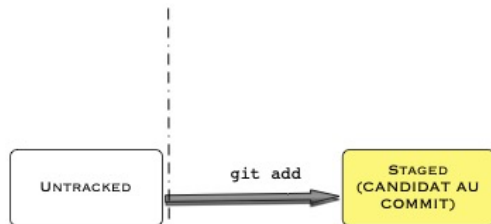
Ajouter, supprimer des fichiers

Git, ajouter un fichier à la base

Comment prendre en compte un fichier inconnu (UNTRACKED) et le sauvegarder dans la base ?

- 1 Marquer le fichier comme suivi par git ("STAGED")

```
git add nomFichier
```



Git, ajouter un fichier à la base

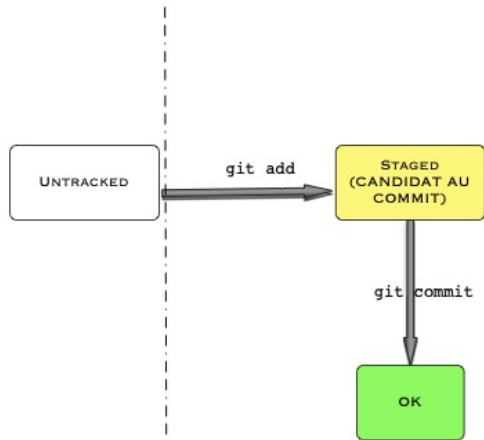
Comment prendre en compte un fichier inconnu (UNTRACKED) et le sauvegarder dans la base ?

- 1 Marquer le fichier comme suivi par git ("STAGED")

```
git add nomFichier
```

- 2 Commit (STAGED → OK)

```
git commit -m "message"
```



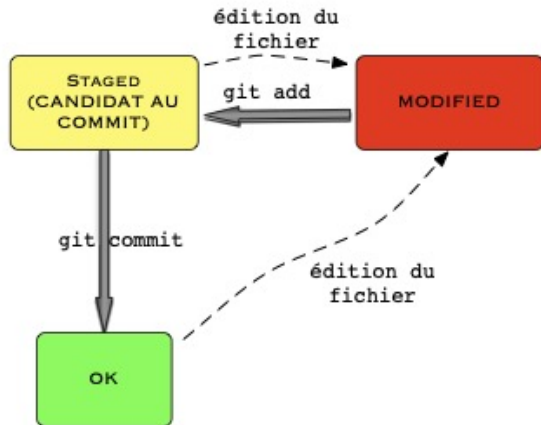
Git, ajouter un fichier à la base

Comment prendre en compte un fichier modifié et le sauvegarder dans la base ?

- 1 Ajout du fichier à l'index

MODIFIED → **STAGED**

```
git add nomFichier
```



Git, ajouter un fichier à la base

Comment prendre en compte un fichier modifié et le sauvegarder dans la base ?

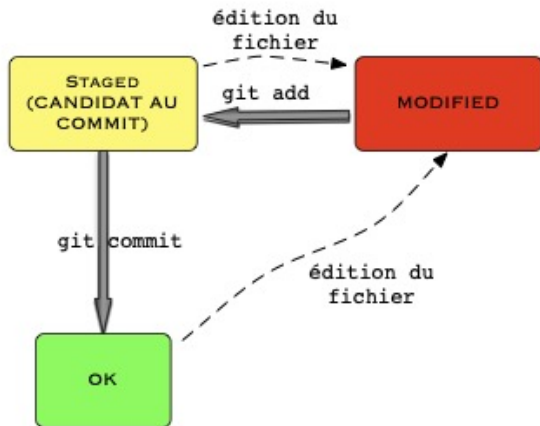
- 1 Ajout du fichier à l'index

MODIFIED → STAGED

```
git add nomFichier
```

- 2 Commit (STAGED → ok)

```
git commit -m "message"
```



Git, ajouter un fichier à la base

Comment prendre en compte un fichier modifié et le sauvegarder dans la base ?

- 1 Ajout du fichier à l'index

MODIFIED → STAGED

```
git add nomFichier
```

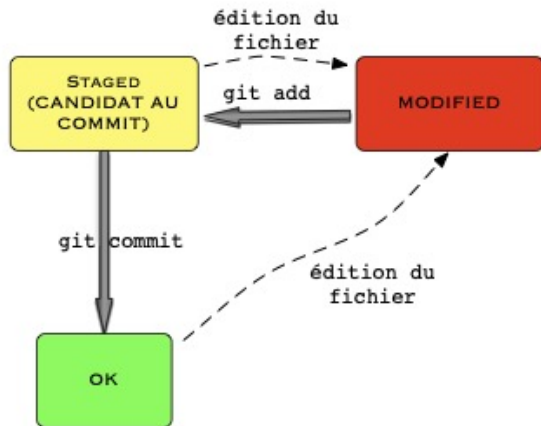
- 2 Commit (STAGED → ok)

```
git commit -m "message"
```

- 3 1 et 2 en une seule commande

MODIFIED → ok

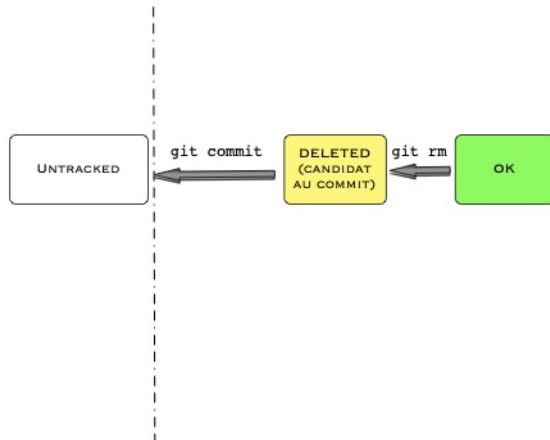
```
git commit -a -m "message"
```



Git, supprimer un fichier

- 1 Marquer le fichier "à supprimer" dans l'index **OK** → **STAGED**

```
git rm nomFichier
```



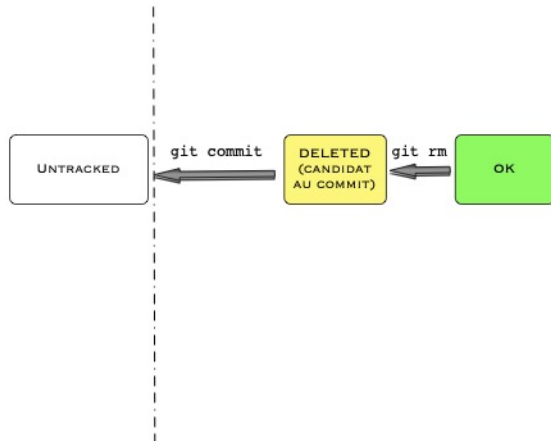
Git, supprimer un fichier

- 1 Marquer le fichier "à supprimer" dans l'index **OK** → **STAGED**

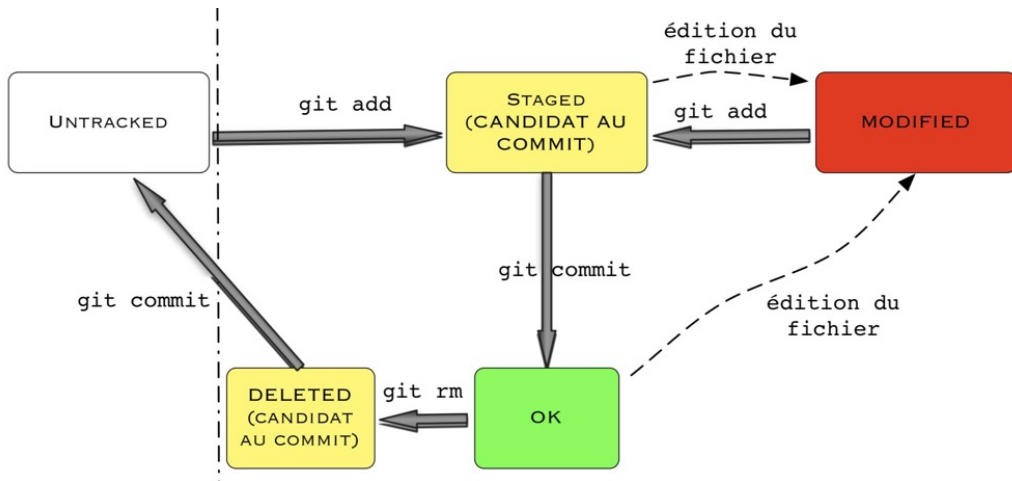
```
git rm nomFichier
```

- 2 Commit (**STAGED** → UNTRACKED)

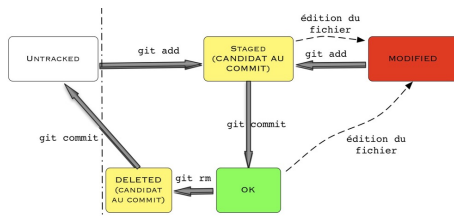
```
git commit -m "Nettoyage"
```



Git, les différents états d'un fichier



Git, les différents états d'un fichier



- **Committed** Etat correspondant à un commit donné
- **Index** Une liste des “candidats au commit”
- Working directory : tous les fichiers du répertoire courant, untracked, **modifiés**, **staged** ou **committed**.

Examiner l'état, l'historique ...

git status, état du répertoire de travail

Permet d'obtenir des infos sur l'état de tous les fichiers du répertoire de travail.

Exemple - Modification d'un fichier

```
vim fic.md # édition du contenu de fic.md
git status
```

On branch main

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: fic.md

Notez au passage l'aide fournie par git status sur les commandes à taper pour l'étape suivante.

git status, état du répertoire de travail

Permet d'obtenir des infos sur l'état de tous les fichiers du répertoire de travail.

Exemple - Staging

```
git add fic.md # On valide les modifications
git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   fic.md
```

git status, état du répertoire de travail

Permet d'obtenir des infos sur l'état de tous les fichiers du répertoire de travail.

Exemple - Commit

```
git commit -m "message"  
git status
```

```
On branch main  
nothing added to commit, working tree clean
```

git status, état du répertoire de travail

Version courte

```
git status -sb

## main...origin/main
 M fic1.md      # modifié, pas staged
M  truc.tex.    # staged
MM main.c.      # staged ET modifié
?? truc.log.    # inconnu
```

colonne de gauche :
état de l'index (stage)
colonne de droite :
état du répertoire de
travail

git status, état du répertoire de travail



Cacher les fichiers non suivis : le fichier `.gitignore`

```
# contenu de .gitignore
```

```
*.nav
```

```
*.log
```

Pour les afficher quand même :

```
git status --ignored
```

`.gitignore` est versionné -> partagé avec l'équipe

`.git/info/exclude` fait la même chose, mais en local uniquement. -> ne pas partager des fichiers/environnements locaux uniquement

git log, historique des commits

- identifiant (hash) unique de chaque commit
- date, auteur, message

```
git log
commit 871da133938c8e8a423d1010ab969625078b4b79 (HEAD -> main)
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Fri Jan 11 17:27:47 2019 +0100

message du dernier commit

commit d18e05e80fe11c6206e2cb5a0624b477a6c31226
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Fri Jan 11 16:26:05 2019 +0100

bonjour
```

git log, historique des commits

Et tout un panel d'options, que nous vous invitons à tester

```
# La version à utiliser au quotidien :  
git log --oneline --decorate --graph --all  
# Liste des commits ayant modifiés le fichier  
git log -- nomfichier  
# Commits d'une personne donnée :  
git log --author="..."  
# Commits contenant une chaîne de caractère :  
git log --grep=" ..."
```



à ajouter dans votre .gitconfig :

```
[alias]
```

```
mylog = log --oneline --decorate --graph
```

git show obtenir des infos sur un commit

```
git show 04b4936
```

```
Author: Henri Massias <henri.massias@unilim.fr>
```

```
Date:   Wed Sep 16 17:16:38 2026 +0200
```

```
Ajout du fichier blablabla ...
```

```
diff --git a/fic1 b/fic1
```

```
... les diffs ...
```


git show obtenir des infos sur un commit

```
# Le contenu du fichier tel qu'il était 2 commits en arrière
```

```
git show HEAD~2 nomfichier.tex
```

```
# Les modifs apportées 3 commits en arrière
```

```
git show HEAD~3
```

git diff, comparaison de fichier

- Diff entre la version **staged** et la version **modified**

```
git diff filename
```

- Diff entre la version **staged** et la **version référence**

```
git diff --staged filename
```

- Diff entre la version courante et celle d'un commit précédent

```
git diff d18e05e80fe11c6206e2cb5a0624b477a6c31226 filename
```



Sans l'argument filename, diff liste toutes les différences pour TOUS les fichiers.

git diff, comparaison de fichier

Exemple - Différence relativement à un commit de référence (hash)

```
git diff d18e05e80fe11c6206e2cb5a0624b477a6c31226
```

```
diff --git a/fic.md b/fic.md
```

```
index e69de29..8810795 100644
```

```
--- a/fic.md
```

```
+++ b/fic.md
```

```
@@ -0,0 +1 @@
```

```
+some new line added in fic.md
```

git diff, comparer les différentes versions (historique) d'un fichier

Le même en version courte (option `--stat`)

```
git diff d18e05e80fe11c6206e2cb5a0624b477a6c31226 --stat
```

```
fic.md | 10 ++-----
```

```
1 file changed, 2 insertion(+), 8 deletions(-)
```

Une option très pratique :

```
git diff --ignore-space-change
```

Résumons ...

- **git status** : quel est l'état de mon répertoire de travail ?
- **git log** : quels commits existent (+toutes les infos associés) ?
- **git show** : quel contenu pour un commit donné ?
- **git diff** : comparer

1 *Introduction et rappels*

2 *Authentification et accès : clés et tokens*

3 *L'API Gitlab*

4 *Git - Rappels des bases*

5 *Git - Les branches*

6 *Git avancé : historique, conflits et outils*

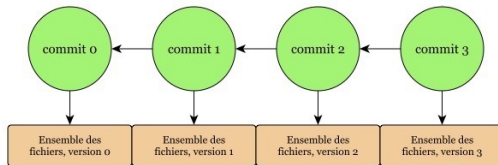
7 *Git, travailler avec des dépôts distants*

Les branches

Git, les branches

Rappel : **Commit** = lien (un pointeur) vers un instantanée (snapshot) de l'état de l'ensemble des fichiers.

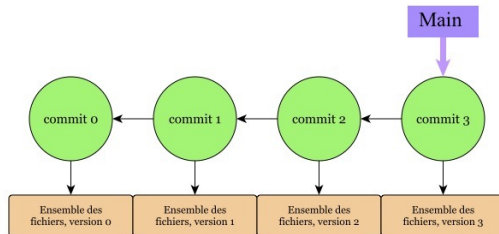
Git empile les commits au fil des validations pour construire l'historique du projet.



Git, les branches

Rappel : **Commit** = lien (un pointeur) vers une **instantanée (snapshot)** de l'état de l'ensemble des fichiers.

Git empile les commits au fil des validations pour construire l'historique du projet.

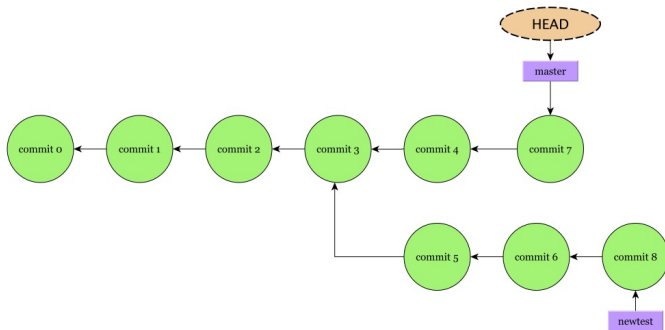


Branche = pointeur vers un commit particulier + historique

Une branche par défaut : **main** (ou **master** si la version de git est ancienne).

Git, les branches

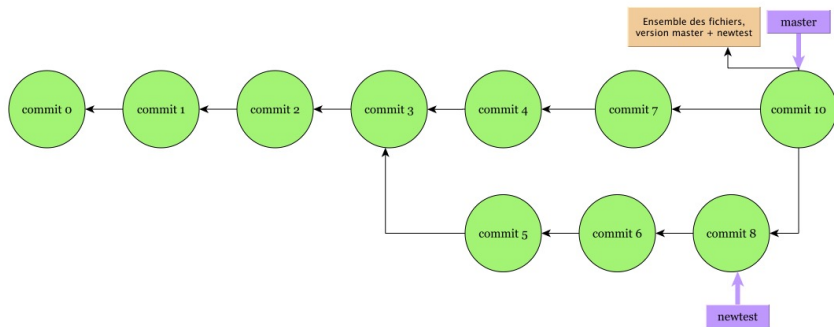
Nouvelle branche = divergence par rapport à la ligne principale de développement



Identification de la branche en cours : le pointeur **HEAD**.

Git, les branches

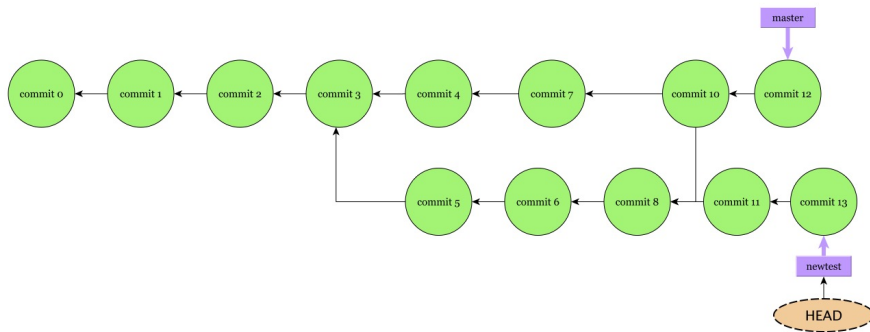
Nouvelle branche = divergence par rapport à la ligne principale de développement



Fusion ... Création d'un nouveau commit !

Git, les branches

Nouvelle branche = divergence par rapport à la ligne principale de développement



Evolution en parallèle ...

Git, les branches

Pourquoi créer des branches?

- Tester une nouvelle fonctionnalité
- Corriger un bug
- Revenir en arrière sans perturber l'existant
- ...

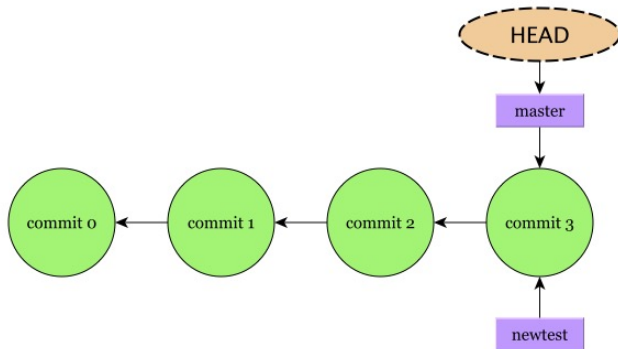
Git, créer et utiliser une branche

Créer une nouvelle branche

```
> git branch newtest
```

newtest : un nouveau pointeur vers le dernier commit.

⚠ on (HEAD) reste sur master !



Git, créer et utiliser une branche

Créer une nouvelle branche

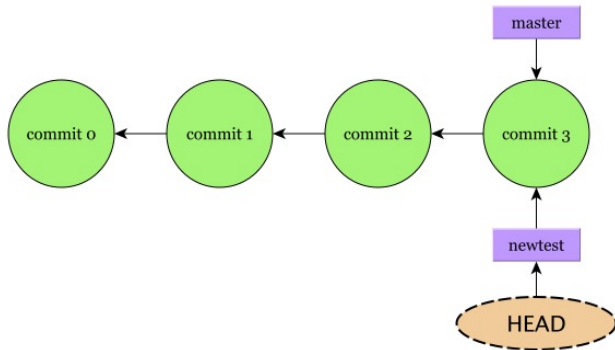
```
> git branch newtest
```

newtest : un nouveau pointeur vers le dernier commit.

Basculer vers une branche

```
> git switch newtest  
# ou git checkout newtest  
Switched to branch 'newtest'
```

⚠ HEAD bouge vers newtest



Identification de la branche en cours : le pointeur **HEAD**.

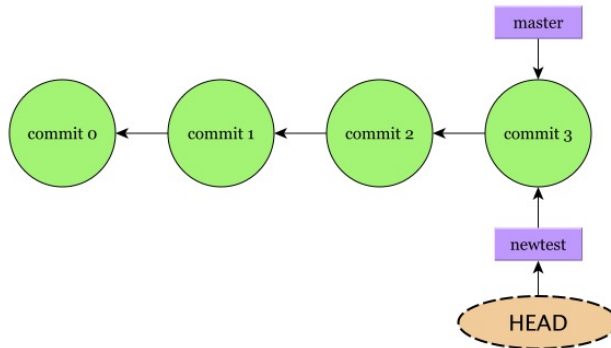
Git, créer ET basculer vers une branche

Création + switch

```
> git switch -c newtest
# ou git checkout -b newtest
Switched to branch 'newtest'
# Liste des branches existantes
> git branch
  master
* newtest
```

Liste des branches existantes

```
> git branch
  master
* newtest
```



Notez le '*' devant la branche en cours (HEAD) dans la liste des branches.

Git, créer ET basculer vers une branche

Création d'une branche à partir d'un commit donné (issu de n'importe quelle branche)

```
> git switch -c newtest cb1070ae82
```

Git, créer et utiliser une branche

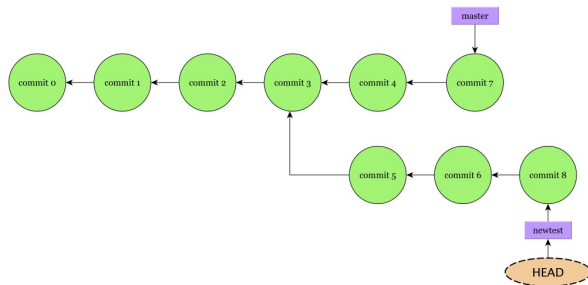
- Changement de branche $\Rightarrow$ modification des fichiers du répertoire de travail.
- Conservation de l'état de chaque branche dans le repository (.git)

```
git branch
* master
  branch_dev1
ls
bin documentation examples include src

git switch branch_dev1
Switched to branch 'branch_dev1'
ls
bin documentation examples include python_tools src visu_3D
```

Git, fusionner les branches

Intégrer une branche (source) à une autre (cible)

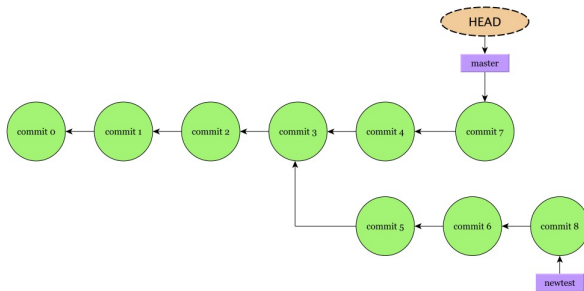


Git, fusionner les branches

Intégrer une branche (source) à une autre (cible)

1. Basculer vers la branche cible
(ici master)

```
git switch master  
Switched to branch 'master'
```



Git, fusionner les branches

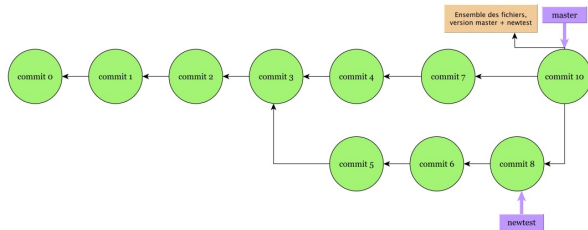
Intégrer une branche (source) à une autre (cible)

1. Basculer vers la branche cible
(ici master)

```
git switch master
Switched to branch 'master'
```

2. Fusionner

```
# Fusion de newtest
# dans master
git merge newtest
```



Git, fusionner les branches

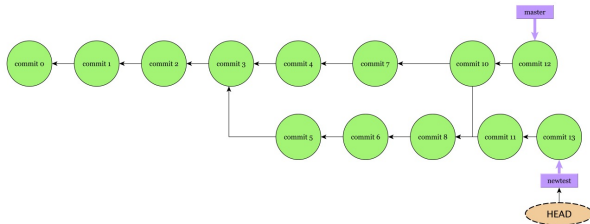
Intégrer une branche (source) à une autre (cible)

1. Basculer vers la branche cible
(ici master)

```
git switch master
Switched to branch 'master'
```

2. Fusionner

```
# Fusion de newtest
# dans master
git merge newtest
```



Chaque branche peut ensuite continuer à évoluer ...

Git, fusionner les branches

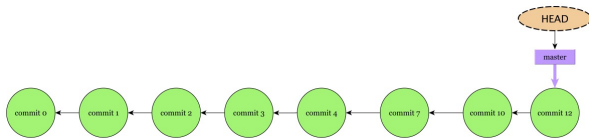
Intégrer une branche (source) à une autre (cible)

1. Basculer vers la branche cible
(ici master)

```
git switch master  
Switched to branch 'master'
```

2. Fusionner

```
# Fusion de newtest  
# dans master  
git merge newtest
```



Ou être supprimée

```
git branch -d newtest  
Deleted branch newtest (was 87469b4)
```



Trainings - Git

Rendez-vous dans le projet habituel pour la première partie des **tps Git**.

1 Introduction et rappels

2 Authentification et accès : clés et tokens

3 L'API Gitlab

4 Git - Rappels des bases

5 Git - Les branches

6 Git avancé : historique, conflits et outils

Revenir en arrière, modifier l'historique

Gérer les conflits

Quelques autres commandes utiles

7 Git, travailler avec des dépôts distants

Revenir en arrière

Git, comment revenir à un état antérieur : restore

Objectifs : rétablir le contenu d'un fichier à partir d'une "référence"

Comment ? La commande **git restore**

Avant d'entrer dans les détails, un petit rappel de vocabulaire :

- **HEAD** état correspondant au commit courant
- **Index** liste/états des "candidats au commit"
- **Working directory** : tous les fichiers du répertoire courant, untracked, modifiés, staged ou committed.

Git restore

restore : rétablir le contenu d'une cible à partir d'une source.

- Restaurer le **working directory** à partir du contenu de **l'index**

```
git restore filename
```

```
# ou git checkout -- filename pour les anciennes versions de git
```

Contenu du fichier filename ...

Avant :

- **HEAD** : A
- **Index** : B
- **Working directory** : C



Le contenu C est perdu !

Après :

- **HEAD** : A
- **Index** : B
- **Working directory** : B

Git restore

restore : rétablir le contenu d'une cible à partir d'une source.

Cible par défaut = working directory, mais on peut la modifier.

- Restaurer l'index à partir de HEAD.

```
# --staged : ajout de l'index dans la cible  
git restore --staged filename
```

Le working directory n'est pas impacté !

- Restaurer l'index ET le working directory à partir de HEAD.

```
# --worktree : ajout du working directory dans la cible  
git restore --staged --worktree filename
```

Git restore

On peut également modifier la source.

Restaurer le **working directory** à partir de **main**.

```
git restore --source=main filename
```

Autres possibilités : source=<commit-id>, tag ...

Ou combiner tout ça ...

```
git restore --source=main --staged --worktree filename
```

Avant :

Après :

- **HEAD** : A
- **Index** : B
- **Working directory** : C
- **main** : D

?

Git restore

On peut également modifier la source.

Restaurer le **working directory** à partir de **main**.

```
git restore --source=main filename
```

Autres possibilités : source=<commit-id>, tag ...

Ou combiner tout ça ...

```
git restore --source=main --staged --worktree filename
```

Avant :

- **HEAD** : A
- **Index** : B
- **Working directory** : C
- **main** : D

Après :

- **HEAD** : A
- **Index** : D
- **Working directory** : D
- **main** : D

Git checkout

! Une commande historique qui faisait beaucoup (trop ?) de choses.

A éviter ? (Bien que très présente dans les docs, tutos, ...)

Remplacée par

- **switch** pour la gestion des branches $\Rightarrow$ Où suis-je / Où est-ce que je veux me placer ?
- **restore** pour la gestion de la restauration de contenu $\Rightarrow$ manipuler le contenu du working directory et de l'index.

Git, comment revenir à un état antérieur ?

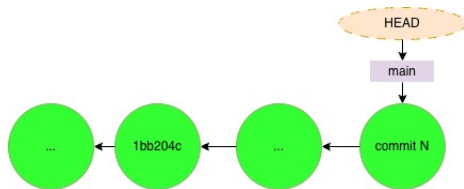


Exploration d'une ancienne branche, d'un ancien commit, recherche de bug ...

Objectifs

- Retour en arrière complet : l'intégralité des fichiers ramenée à son état dans un commit donné.
- Vérifier, tester, explorer ...
- Modifier les sources à partir de ce point ($\Rightarrow$ nouvelle branche)

Mais que se passait-il en 1bb204c 🤔 ?



Git, comment revenir à un état antérieur ?



Exploration d'une ancienne branche, d'un ancien commit, recherche de bug ...

Objectifs

- Retour en arrière complet : l'intégralité des fichiers ramené à son état dans un commit donné.
- Vérifier, tester, explorer ...
- Modifier les sources à partir de ce point ($\Rightarrow$ nouvelle branche)

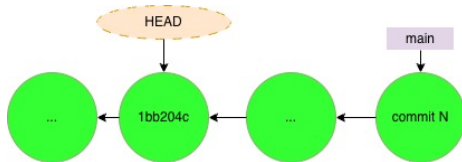
Ancienne mode :

```
git checkout 1bb204c  
# HEAD "detached"
```

Nouvelle mode :

```
git switch --detach 1bb204c  
# tests divers ...
```

$\Rightarrow$ on déplace HEAD vers le commit 1bb204c



Git, comment revenir à un état antérieur ?



Exploration d'une ancienne branche, d'un ancien commit, recherche de bug ...

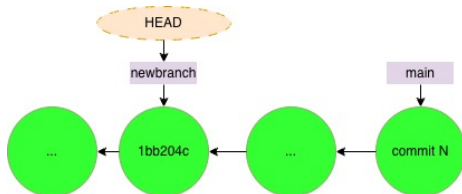
Objectifs

- Retour en arrière complet : l'intégralité des fichiers ramenée à son état dans un commit donné.
- Vérifier, tester, explorer ...
- Modifier les sources à partir de ce point ($\Rightarrow$ nouvelle branche)

Et pour poursuivre le travail (démarrer une branche ...)

```
git switch -c newbranch 1bb204c  
# modifs, commits etc ...
```

$\Rightarrow$ HEAD n'est plus "detached" et pointe sur la nouvelle branche.



Git commit amend : modification du dernier commit

Remplace le dernier commit par un nouveau commit construit à partir de l'index.
Cas d'usage ?

- Modification du dernier message commit

```
git commit --amend  
# ...
```

- Ajout d'un fichier oublié

```
git commit -m "blabla"  
git add jaioublie.txt  
git commit --amend
```

L'ancien commit existe toujours mais n'est plus référencé par la branche ...

Git, comment revenir à un état antérieur : reset

Objectifs : annuler des commits, possiblement sans perdre de contenu.

Intérêt : annulation/réécriture d'une partie des commits

Comment ? La commande **git reset**, réinitialisation de HEAD à un état donné

Options : soft, mixed ou hard.

Git reset --soft

```
# Branche = main
# Commit actuel = ref-id
git reset --soft previous-id
# ou
git reset --soft HEAD~2 # 2 commits en arrière
```

Conséquences

- HEAD et la branche courante vers previous-id
- Les commits entre previous-id et ref-id ne font plus partie de l'historique de la branche courante
-  Aucun changement dans le contenu des fichiers (working directory)
- Les modifications des commits annulés apparaissent comme "staged"

Intérêt : annuler puis réécrire/modifier des commits récents.

Git reset --mixed

```
# Commit actuel = ref-id  
git reset --mixed previous-id
```

Conséquences

- HEAD et la branche courante vers previous-id
- Les commits entre previous-id et ref-id ne font plus partie de l'historique de la branche courante
- Index = celui de previous-id
- ⚠️ Aucun changement dans le contenu des fichiers
- Les modifications post previous-id (celles des commits annulés entre autres) apparaissent comme "modified"

Intérêt : annuler puis réécrire des commits récents + les sortir de l'index

Git reset --hard

```
# Commit actuel = ref-id  
git reset --hard previous-id
```

Conséquences

- HEAD et la branche courante vers previous-id
- Les commits entre previous-id et ref-id ne font plus partie de l'historique de la branche courante
- Index = celui de previous-id
-  working dir = celui de previous-id

Intérêt : annulation complète de commits récents sans conservation.

Git reset

 pour utilisation strictement locale - Ne pas faire de reset si la branche “remote” a été modifiée (voir plus loin)

Autres options non détaillées ici :

—merge ou —keep $\approx$ hard mais sécurisé (blocage si risque de perte de données)

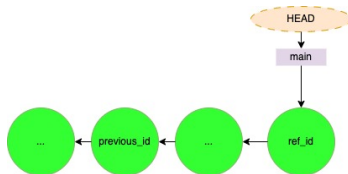
Annuler des modifications : revert

Objectif : annuler uniquement les effets d'un ou plusieurs commits

On ne perd rien, pas de destruction de l'historique

```
# Etat courant = ref_id
git revert previous_id
# ou pour annuler une plage de commits :
git revert start_range..end_range
```

- Création d'un nouveau commit
- Nouvel état = celui de HEAD (*ref_id*) moins les effets de *previous_id*
- Potentiels conflits à gérer



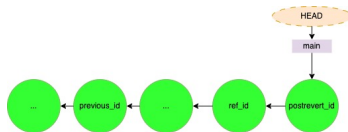
Annuler des modifications : revert

Objectif : annuler uniquement les effets d'un ou plusieurs commits

On ne perd rien, pas de destruction de l'historique

```
# Etat courant = ref_id
git revert previous_id
# ou pour annuler une plage de commits :
git revert start_range..end_range
```

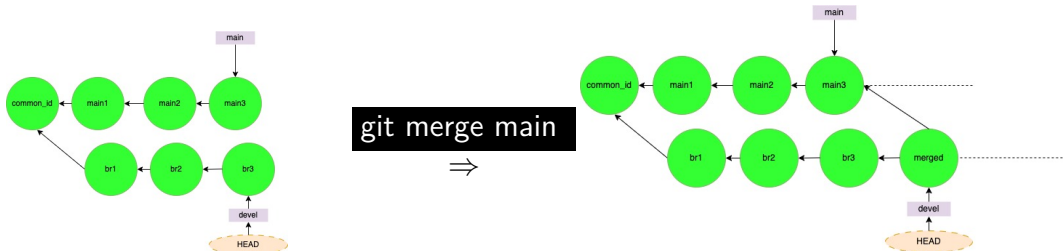
- Création d'un nouveau commit
- Nouvel état = celui de HEAD (*ref_id*) moins les effets de *previous_id*
- Potentiels conflits à gérer



git rebase

Rejouer/appliquer des commits **après** intégration d'une autre branche

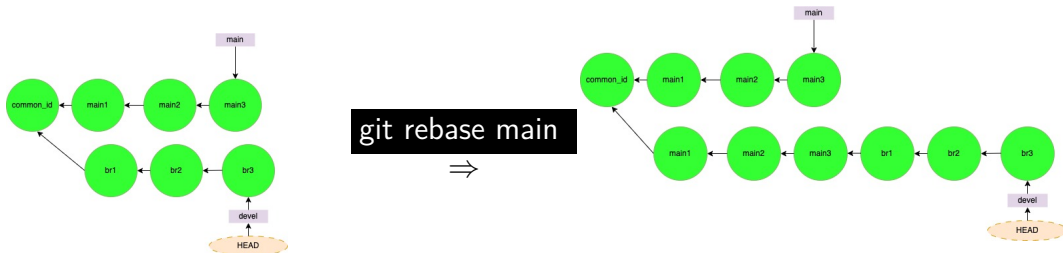
Comparaison avec 'git merge'



git rebase

Rejouer/appliquer des commits **après** intégration d'une autre branche

Comparaison avec 'git merge'



Historique + propre ... mais plus de conflits potentiels à gérer

Fusionner (squash) des commits

Utilisation de rebase interactif + squash

```
git rebase -i HEAD~3
```

```
commit b908cffb9931277e2c751da1f8034537f285c0a2 (HEAD -> main)
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Wed Jun 4 11:25:59 2025 +0200

    bouquet final

commit 84cd013229e56e0306589b7eaddf6d6aa61db1b8
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Wed Jun 4 11:25:42 2025 +0200

    autre chose ...

commit 4c5763c5699e050ef68a63f6d7a9ca9cc6d0882e
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Wed Jun 4 11:25:28 2025 +0200

    des trucs ...

commit fbd1701ab7377dd4282aad09b1406025c1b6e3ae
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Tue Jun 3 17:42:46 2025 +0200

    v8
```

Fusionner (squash) des commits

Utilisation de rebase interactif + squash

```
git rebase -i HEAD~3
```

```
pick 4c5763c des trucs ...
pick 84cd013 autre chose ...
pick b908cff bouquet final

# Rebase fbd1701..b908cff onto fbd1701 (3 commands)
#
```

Fusionner (squash) des commits

Utilisation de rebase interactif + squash

```
git rebase -i HEAD~3
```

```
pick 4c5763c des trucs ...
s 84cd013 autre chose ...
s b908cff bouquet final

# Rebase fbd1701..b908cff onto fbd1701 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
```


Fusionner (squash) des commits

Utilisation de rebase interactif + squash

```
git rebase -i HEAD~3
```

```
commit 31d2f571027e1ad903c6624bae44c93a80189325 (HEAD -> main)
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Wed Jun 4 11:25:28 2025 +0200
```

version finale complète

```
commit fbd1701ab7377dd4282aad09b1406025c1b6e3ae
Author: Franck Pérignon <franck.perignon@univ-grenoble-alpes.fr>
Date:   Tue Jun 3 17:42:46 2025 +0200
```

v8

Gestion des conflits

Lors du merge, tout ne se passe pas nécessairement toujours très bien ... 🙄

```
git merge newtest
...
CONFLICT
...
git status

unmerged : somefile
```

git ajoute des chevrons autour des zones de conflit :

```
vim somefile
```

```
<<<<<< HEAD:somefile
version de master
=====
version newtest en conflit
>>>>>> newtest:somefile
```

Gestion des conflits

Pour corriger les problèmes, il faudra :

- 1 résoudre les conflits à la main ou via un outil dédié, par exemple :

```
git mergetool
```

L'outil graphique ouvert par mergetool dépendra de votre choix d'éditeur dans `.gitconfig`.

- 2 basculer les fichiers corrigés vers l'index (`git add somefile`),
- 3 valider (`commit`).

Quelques autres commandes utiles

Gestion des tags (étiquettes) : mise en valeur d'un état dans l'historique

```
git tag -a v1.3 -m "Version 1.3 - bla bla bla"
```

Cacher les modifications en cours (modified ou staged)

```
git stash # on cache ...  
git pull  
git stash pop # on ramène ce qui était sous le tapis
```

Quelques autres commandes utiles

Chercher un coupable ... Pour chaque ligne, info (auteur, date ...) au sujet de la dernière modif

```
git blame nomfichier
```

Recherche de bugs par dichotomie : git bisect

```
git bisect start
git bisect bad # bug dans la version courante
git bisect good 18f77f6a6d3d3de0 # id d'un commit ok
# good / bad par dichotomie jusqu'au commit coupable
git bisect run script.sh
# Si script renvoie 0 le commit est vu comme "good". Si non, "bad"
git bisect reset
# puis résolution du problème
```

Quelques autres commandes utiles

nettoyage du repository local (tout ce qui n'est pas versionné)

```
git clean -n # dry-run  
Would remove ...  
git clean -fd # suppression (récursif)
```

Appliquer les modifications d'un ou plusieurs commits existants par ailleurs (autres branches ...)

```
git cherry-pick commit_id  
# ou  
git cherry-pick <start>..  
<end>
```

git reflog : pour retrouver et récupérer des commits qui ne sont plus référencés par aucune branche

Petit résumé

- **Démarrer** clone, init
- **Gérer ses fichiers** add, rm, mv, commit, restore
- **Monitorer** status, log, diff
- **Bifurquer, fusionner, corriger, naviguer dans l'historique ...** branch, switch, merge, revert, rebase, reset



Trainings - Git

Rendez-vous dans le projet habituel pour la suite des **tps Git**

① *Introduction et rappels*

② *Authentification et accès : clés et tokens*

③ *L'API Gitlab*

④ *Git - Rappels des bases*

⑤ *Git - Les branches*

⑥ *Git avancé : historique, conflits et outils*

⑦ *Git, travailler avec des dépôts distants*

Travailler avec des dépôts distants (Forge !)

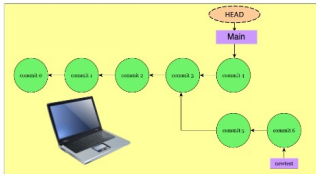
Git : travailler avec plusieurs repositories

Git : un gestionnaire de version **décentralisé**

- pas de notion de dépôt/serveur central de référence,
- coexistence possible de plusieurs dépôts (locaux, distants ...), désynchronisés, indépendants,
- des repositories privilégiés sur une forge

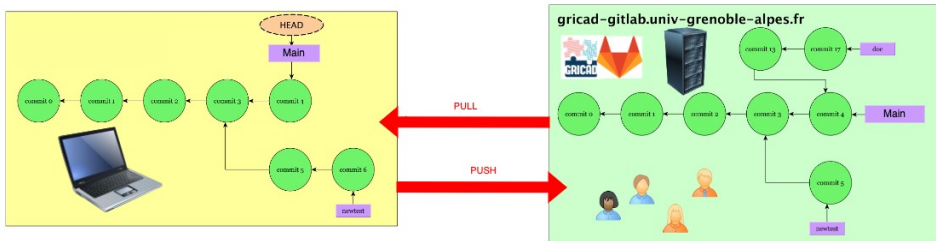
Git : travailler avec plusieurs repositories

- **Local** avec les fonctionnalités classiques d'un gestionnaire de version,
👉 Gestion de l'historique 👉 Branches (plusieurs lignes de développement)

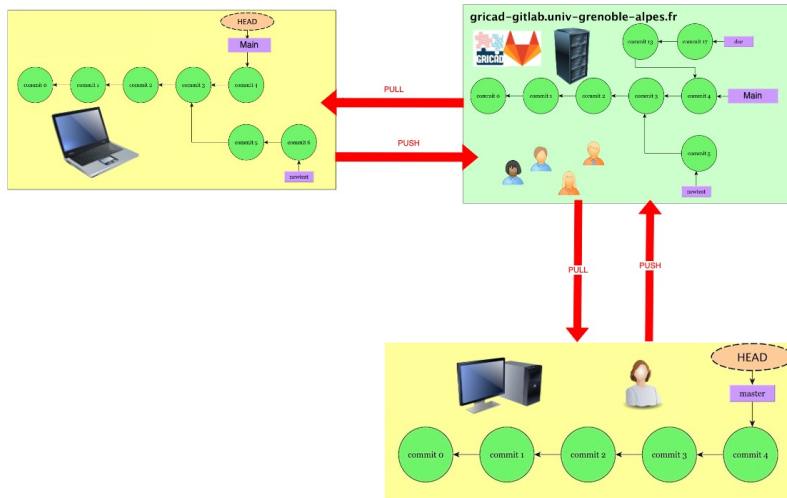


Git : travailler avec plusieurs repositories

- **Local** avec les fonctionnalités classiques d'un gestionnaire de version,
 - 👉 Gestion de l'historique
 - 👉 Branches (plusieurs lignes de développement)
- **Distants** : avec la possibilité de synchroniser votre repository avec d'autres dépôts, dits remote/distants
 - 👉 Collaboration avec d'autres utilisateurs

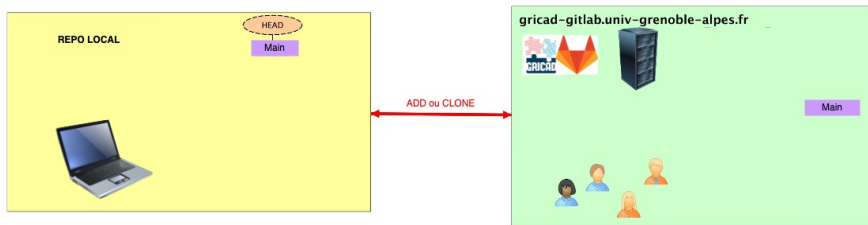


Et des connexions avec d'autres utilisateurs / repositories



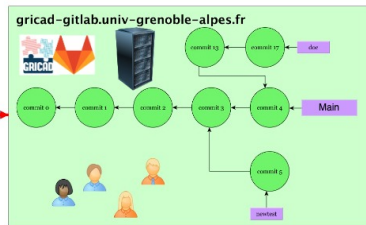
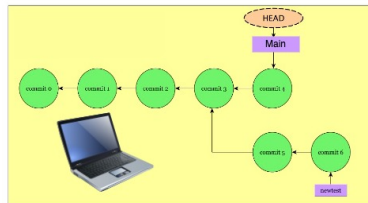
Dépôt remote, principe et mise en place

- 1 **Connexion** : ajout ou copie (**clone**) d'un dépôt distant (une seule fois, au début du projet),



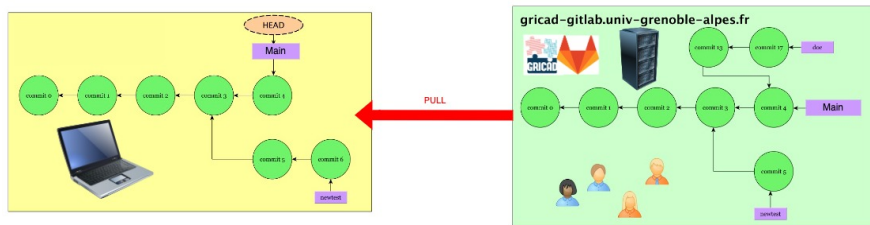
Dépôt remote, principe et mise en place

- 1 **Connexion** : ajout ou copie (**clone**) d'un dépôt distant (une seule fois, au début du projet),
- 2 Développements (indépendants) dans chaque repository (plusieurs branches ...),



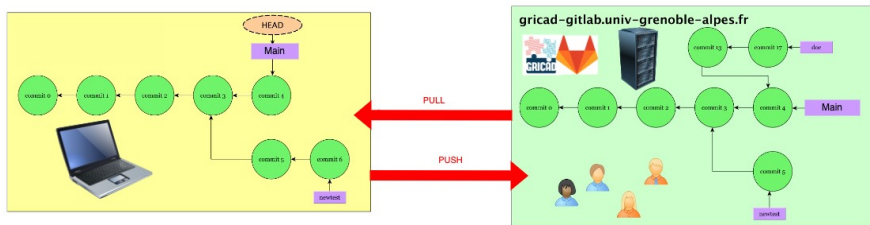
Dépôt remote, principe et mise en place

- 1 **Connexion** : ajout ou copie (**clone**) d'un dépôt distant (une seule fois, au début du projet),
- 2 Développements (indépendants) dans chaque repository (plusieurs branches ...),
- 3 Intégration des modifications du dépôt remote (**pull**),



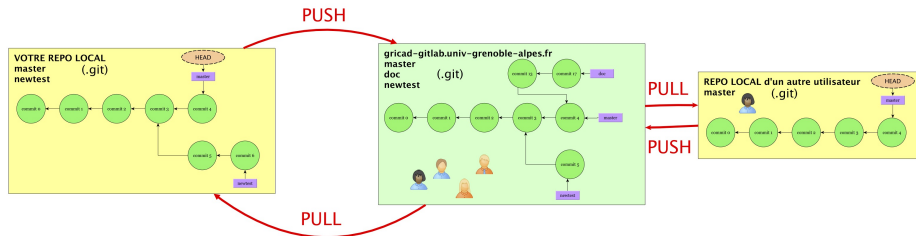
Dépôt remote, principe et mise en place

- 1 **Connexion** : ajout ou copie (**clone**) d'un dépôt distant (une seule fois, au début du projet),
- 2 Développements (indépendants) dans chaque repository (plusieurs branches ...),
- 3 Intégration des modifications du dépôt remote (**pull**),
- 4 Transfert de vos modifications vers le remote (**push**),



Dépôt remote, principe et mise en place

- 1 **Connexion** : ajout ou copie (**clone**) d'un dépôt distant (une seule fois, au début du projet),
- 2 Développements (indépendants) dans chaque repository (plusieurs branches ...),
- 3 Intégration des modifications du dépôt remote (**pull**),
- 4 Transfert de vos modifications vers le remote (**push**),



Ajout d'un dépôt distant : *git remote*

Déclaration d'un dépôt distant "connecté", **git remote add** :

```
# Vers un repo local
```

```
git remote add local_name /path/to/some_git_dir
```

```
# Vers un serveur
```

```
git remote add another_name git@plmlab.math.cnrs.fr:vide/rien.git
```

Liste des dépôts remote

```
# Liste des remotes  
git remote -v # -v pour le mode verbeux  
some_name /path/to/some_git_dir  
another_name git@plmlab.math.cnrs.fr:vide/rien.git
```

👉 Possibilité de synchroniser votre repository courant avec `some_name` ou `another_name`.

👉 Aucune limite au nombre de dépôt remotes ...

⚠ Il est préférable que ces dépôts aient un ancêtre (une version) en commun.

Copie d'un dépôt remote : git clone

- Création et initialisation.
- Le serveur cloné est automatiquement ajouté comme remote.
Nom par défaut : **origin**

```
git clone git@plmlab.math.cnrs.fr:vide/rien.git
cd rien
git remote -v
origin git@plmlab.math.cnrs.fr:vide/rien.git
```

Dépôt distant - Compléments

Renommer (localement) un dépôt distant :

```
git remote rename current_name new_name
```

Supprimer (localement) un dépôt distant :

```
git remote rm local_name
```

 Suppression **locale** de la liste des *remotes*

Le dépôt sur le serveur distant ne sera pas physiquement supprimé mais simplement ignoré par votre repository local.

Ajout d'un dépôt distant - Résumé

cas 1 : from scratch

```
# Création d'un dépôt vide
git init
# 'connexion' avec un dépôt remote
git add remote some_name repository_address
```

cas 2 : copie

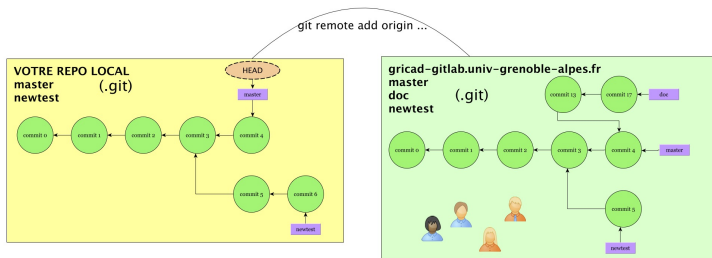
```
git clone git@plmlab.math.cnrs.fr:vide/rien.git monprojet
# --> lien automatique avec le dépôt remote, nommé localement 'origin'
```


Git, travailler avec des dépôts distants

Visualiser (*git branch*) la liste des branches remotes

Pour obtenir la liste complète des branches des dépôts distants :

```
git branch -r
# r for remote
git branch -a
# a for all,
# local + remote
* master
newtest
remotes/origin/master
remotes/origin/newtest
remotes/origin/doc
```



Git, travailler avec des dépôts distants

Comparer n'importe quelle branche de votre dépôt à celles du dépôt distant :

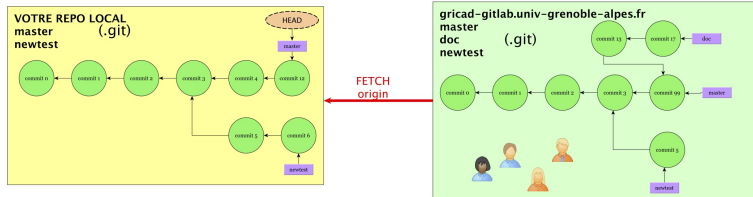
```
git diff some_name/branch_name  
git diff origin/main  
# avec les options habituelles de git diff
```

Git, travailler avec des dépôts distants

Récupérer (*git fetch*) le contenu d'un dépôt remote

👉 Collecter toutes les infos (données, branches ...) du dépôt distant.

```
git fetch nom_depot
# par exemple
git fetch origin
```



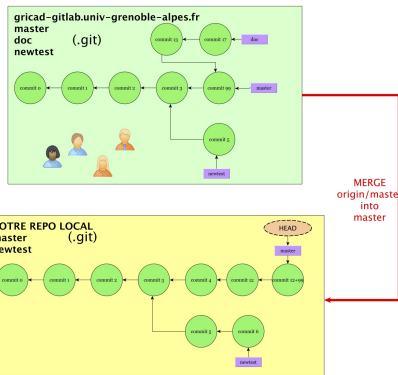
⚠️ `fetch` importe les données remote mais ne modifie pas vos branches locales.

Git, travailler avec des dépôts distants

Fusionner (*git merge/pull*) votre branche locale avec une branche remote

👉 Fusionner une branche distante et une branche locale

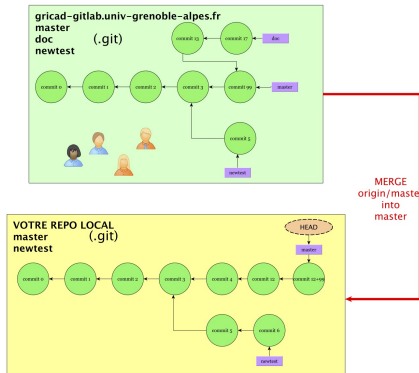
```
git merge NomDepot/NomBranche
# par exemple
git merge origin/master
```



Git, travailler avec des dépôts distants

Fetch et merge peuvent être combinés en une seule opération **pull**

```
git pull NomDepot NomBranche
# par exemple
git pull origin master
```



pour un fonctionnement correct, il faut un “ancêtre” (commit) commun aux différentes branches.

Git, travailler avec des dépôts distants

Transférer (*git push*) votre branche vers un dépôt remote

```
git push remote_name local_branch_name:remote_branch_name
```

par exemple : transfert de la branche locale devel vers la branche remote machin

```
git push origin devel:machin
```

- Il faut être explicite !
- Mais il est possible (et fortement recommandé) d'associer/connecter (**tracking**) chaque branche locale à une branche remote.
- Cette dernière sera dénommée branche “**upstream**” de la branche locale.



Permet de clarifier et simplifier les synchronisations avec git push/pull sans arguments

Le tracking ou comment gérer les liens entre branches distantes et locales ?

- la branche main locale est associée (suit) automatiquement à origin/main.

```
git clone ...  
...  
git pull # ok main suit origin/main par défaut  
git push
```

- Les branches remote sont elles aussi automatiquement suivies

```
git clone ... # hypothèse : devel existe dans origin  
git switch devel # => devel suit automatiquement origin/devel  
...  
git pull # ok  
git push
```



Mais pas de comportement par défaut pour les autres branches locales !

Git, travailler avec des dépôts distants - Tracking

Si on a créé localement une nouvelle branche : pas de tracking par défaut

```
git switch -c newtest
git pull # ERROR
git pull origin truc # ok si truc existe dans origin
git push # ERROR
```

Solution :

```
git branch --set-upstream origin newtest
git push # ok
```

ou directement :

```
git branch --set-upstream origin newtest
git push -u origin newtest # ok => fixe tracking
...
git push # ok
```


Git, travailler avec des dépôts distants, résumé

Cycle classique de travail dans un dépôt git.

1. Démarrage du suivi d'une branche distante (une seule fois en général)

```
git clone git@plmlab.math.cnrs.fr:vide/rien.git
```

2. Développements locaux, vie du repository local

```
git add, commit, status, diff, branch ...
```

3. Synchronisation (**fetch/merge/pull**) avec la branche distante

```
git pull
```

4. Transfert de votre travail vers le dépôt remote

```
git push
```



Trainings - Git

Rendez-vous dans le projet habituel pour les tps Git, dépôts remotes

Session 2

Pratique des mécanismes de déploiement et d'intégration continus sous Gitlab

8 *Gitlab workflows*

Workflow

Merge-requests

9 *Les hooks*

10 *Intégration, livraison et déploiement continu*

Workflow git/gitlab ?

Organisation de l'ensemble des tâches autour de votre projet de développement

- Choix d'une stratégie d'utilisation de git (Branches ? Forks ? ...)
- Détermination des rôles et responsabilités (Qui peut merger ? Où ?)
- Gestion des interactions entre développeurs
- Processus de review ? Gestion des erreurs ?
- ...

Pourquoi choisir un workflow ?

- Différents intervenants dans plusieurs contextes, avec leurs habitudes propres,
- Eviter une gestion compliquée voir chaotique et donc probablement inefficace.
- Risques de perte de temps, de données !
- ...

Quelques outils sur Gitlab

- work items (ex-issues)
- Merge-requests
- Intégration continue

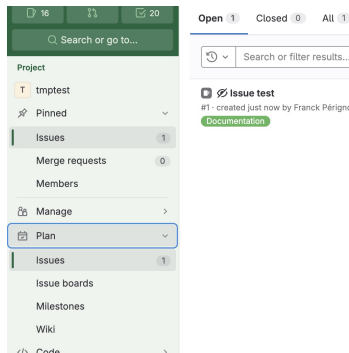
Gestionnaire de problème : le menu *Plan* - *Work items* , *milestones* ...

Work items

- **Suivre l'évolution** des problèmes (du signalement à la résolution),
- **Discuter** de nouvelles idées,
- Suivre l'ajout de **nouvelles fonctionnalités**, leur évolution

List Board, Milestones

- Visualiser, classer, étiqueter ...
- Définir, planifier les étapes de développement



Gestionnaire de problème - *Work items*

Quelques conseils et fonctionnalités utiles

- Créez des labels explicites pour classer vos issues (documentation, bugs, ...).
- Utilisez le markdown pour rédiger vos issues, vos commits

- **@username** ⇒ email + ajout dans todo-list

- **#id** : référence à une issue

- **Fix #id** : close automatiquement une issue via un message de commit

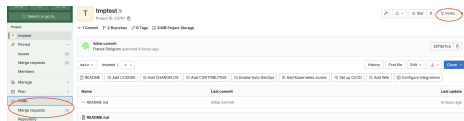
Voir la doc [Closing issues automatically](#).



Les merge-requests

Un mécanisme pour gérer proprement les contributions

- Réserver la branche main pour la version principale et stable du projet,
- Isoler les essais/implémentations de nouvelles fonctionnalités, corrections d'un bug etc.



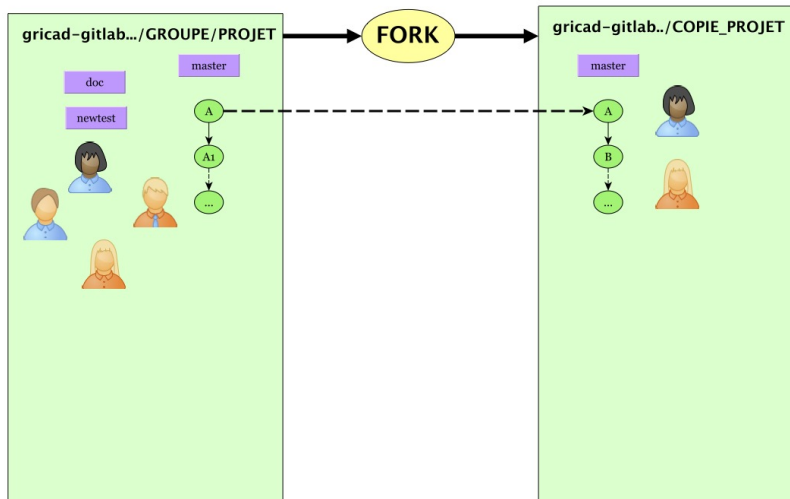
Merge-request : soumission d'une demande de fusion d'une branche dans une autre

- Informer les membres du projets, demander une revision, un avis
- Fusion dans la branche principale

Fork : duplication d'un projet (et donc copie du repository git)

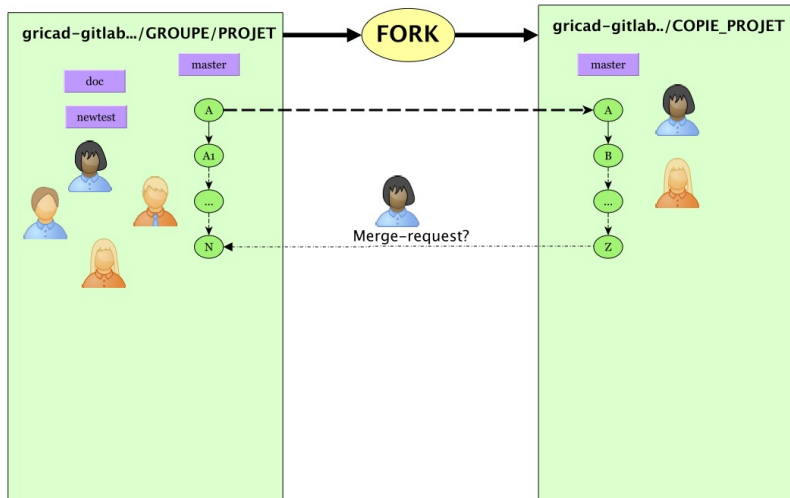
Les merge-requests, principe

Copie du projet gitlab vers un autre namespace, **Fork** $\Rightarrow$ création d'un nouveau projet.



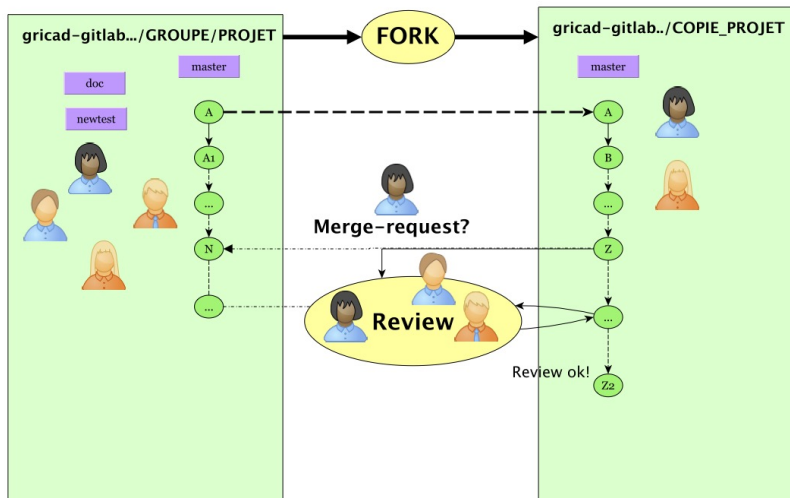
Les merge-requests, principe

Soumission d'une merge-request



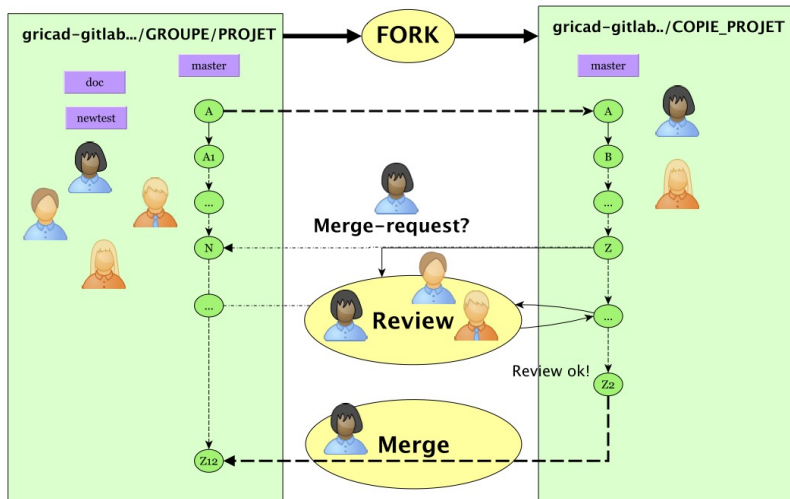
Les merge-requests, principe

Review, jusqu'à obtention d'une solution (un commit) acceptable et fusionnable



Les merge-requests, principe

Merge de la branche du fork vers le projet principal, cloture éventuelle de la MR.



Quel "workflow" choisir ?

Beaucoup de possibilités et pas de réponse unique ...

- Workflow centralisé : tout le monde travaille sur la même branche.
 - Nécessite une petite équipe et beaucoup de discussions
 - Difficile de gérer les releases, les versions instables ...
- Git workflow
- GitHub flow
- OneFlow
- ...

Un exemple plus détaillé, Gitflow

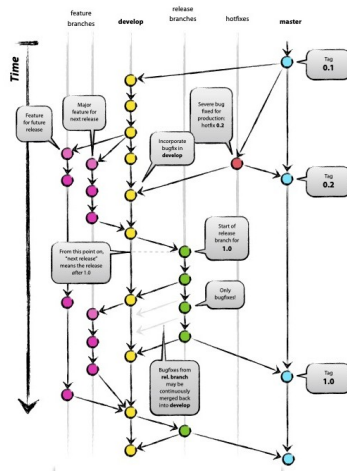
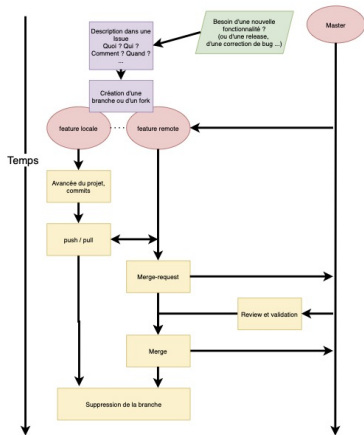


Figure: Gitflow

Gitlab, git, workflow, ... Quelques recommandations



- 1 Utilisez les **issues**. Déclarer chaque problème, nouveau développement souhaité, etc
- 2 Créez une nouvelle **branche** (ou un **fork**) pour chaque nouvelle “feature”, release, résolution de bug ...
- 3 Utilisez les **merge-requests** et profitez du processus de review.
- 4 Synchronisez régulièrement avec le serveur et la branche principale (pull). Plus une branche vit longtemps, plus le merge sera pénible ...
- 5 Merge ou rebase ? Faites un choix pour tous les utilisateurs du projet !

Gitlab, travailler avec des dépôts distants - Mirroring

Mirroring : un mécanisme de "synchronisation" automatique entre deux dépôts.

Mise en place sur Gitlab : voir [mirroring](#) dans le projet trainings.

8 *Gitlab workflows*

9 *Les hooks*

10 *Intégration, livraison et déploiement continu*

Passons la parole à Pierre ...

Rendez-vous ici : [Des hooks Git à la CI](#)

Mathys Astruc (IMT Toulouse)

Sensibilisation à la rédaction de messages de commit



Trainings - Workflows Gitlab

Rendez-vous dans le projet habituel pour les [tps workflow](#)

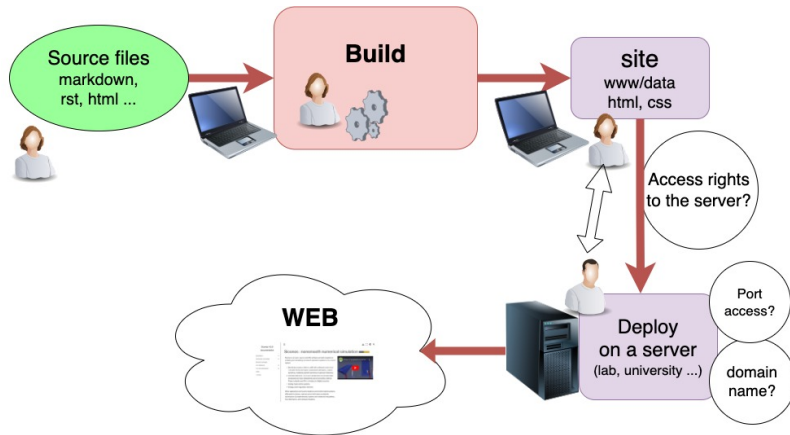
8 Gitlab workflows

9 Les hooks

10 Intégration, livraison et déploiement continu

Un exemple pour commencer

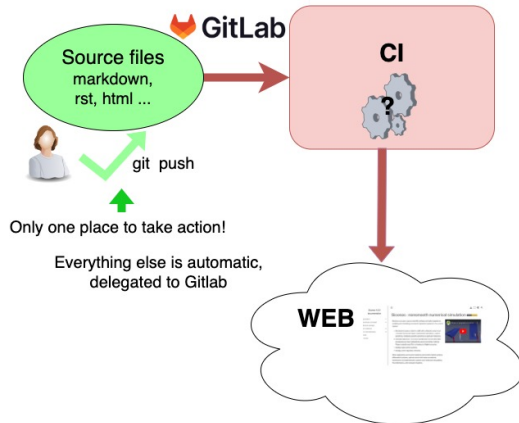
Objectif : créer un site web et le publier.



Sans gitlab et l'intégration continue 🙄

Un exemple pour commencer

Objectif : créer un site web et le publier.



Avec gitlab et l'intégration continue 😊

Continuous Integration (CI)

Concept : une pratique consistant à vérifier systématiquement et automatiquement que les modification du code source ne produise pas de régression dans le fonctionnement, les performances ...

👉 Un outil "devops"

👉 Essentiellement pour les développeurs et la production de logiciels

⚠ Vérifier, valider $\Rightarrow$ définition préalable de références, d'un cahier des charges ...
Très dépendant du contexte !

Continuous Integration, Delivery, Deployment (CI/CD)

Livraison continue : *"produire/construire des logiciels dans des cycles courts, pour les mettre à disposition à n'importe quel moment."*

- Produire et mettre à disposition, construire, tester, diffuser.
- Notion de "livrable" non présente dans la CI.
- Pas nécessairement de déploiement (mise en production)

Déploiement continu : *"livraison de "fonctionnalités" logicielles, par le biais de déploiements automatisés."*

- Notion de "livrable" non présente dans la CI.
- La mise en production est régulière et automatisée.

Continuous Integration, Delivery, Deployment (CI/CD)

En résumé: des pratiques consistant à associer à chaque modification des sources une série d'opérations qui seront exécutées systématiquement et automatiquement.

Avec des objectifs divers

- Valider et tester un produit (pendant le process de développement par exemple) : configuration, compilation, tests unitaires ...
- Fournir des livrables (pdf, documentation, release, images Docker ...)
 - Génération et publication de site web
 - Création de documents (markdown, latex ...)
 - Déploiement d'applis Shiny (R), Voila (Python)
 - ...
- Mettre en production

Continuous Integration, Delivery, Deployment (CI/CD)

En résumé: des pratiques consistant à associer à chaque modification des sources une série d'opérations qui seront exécutées systématiquement et automatiquement.

En pratique :

- Un projet Gitlab
- Une série de scripts associés au projet
- À chaque “push” (modification des sources), le code est construit, installé, testé, déployé ...
- Un rapport est publié sur la page du projet

Pourquoi utiliser la CI/CD ?

La version longue ...

- Facilite les développements au quotidien, les collaborations
- Identifier et corriger les erreurs plus facilement et plus rapidement
- Produire un code plus stable, plus robuste, plus portable
- S'assurer que le résultat de nouvelles modifications n'introduit pas de régression du code
- Anticiper, planifier, tester différents environnements (debug/release, OS, paramètres ...)
- Déployer le code, fournir des "images" prêtes à l'emploi (Docker, Singularity ...)
- Economiser du temps pour les développeurs, réduire le "time-to-release" ou le "time-to-new-features"

Pourquoi utiliser la CI/CD ?

En résumé ...

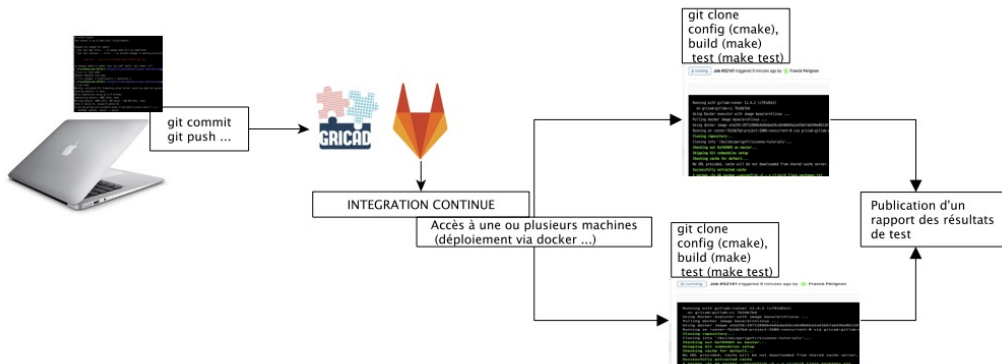
👍 Facilite la vie des développeurs et utilisateurs

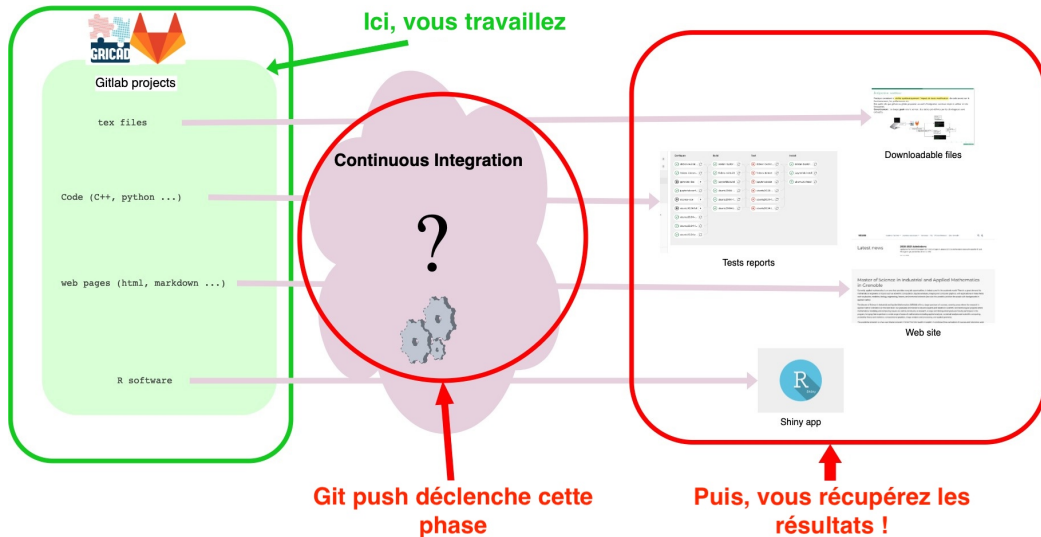
👍 Mettre en place un process de CI/CD vous obligera à tout scripter, automatiser, vérifier ...

C'est donc un outil fondamental pour la qualité des logiciels , la reproductibilité

CI - Principe et fonctionnement

Concrètement : à chaque **push** vers le serveur, des tâches pré-définies par les développeurs sont exécutées.





CI - Principe et fonctionnement



Job : une suite d'actions à exécuter

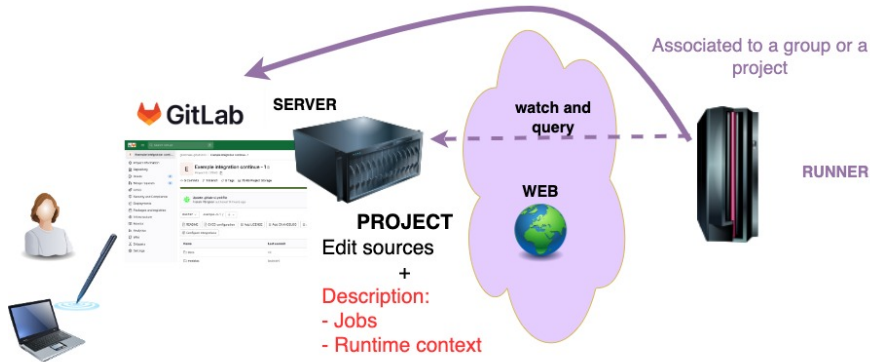
- Chaque job est indépendant
- Avec un **contexte d'exécution** propre

CI - Principe et fonctionnement



Runner : une machine hôte en charge d'exécuter votre job (via une **executor**, e.g. Docker)

CI - Principe et fonctionnement



Le **runner** interroge Gitlab pour détecter toute activité dans le projet (git push)

CI setup

Avec  **GitLab**, un outil adapté et performant : `gitlab-ci`
<https://docs.gitlab.com/ee/ci/>

Setup?

- 1 Créer un fichier `.gitlab-ci.yml` à la racine du projet
 - La CI est active !
 - Ce fichier liste toutes les tâches qui doivent être exécutées (les jobs !): quoi, où, quand, dans quel contexte ...

CI setup

Avec  **GitLab**, un outil adapté et performant : `gitlab-ci`
<https://docs.gitlab.com/ee/ci/>

Setup?

- 1 Créer un fichier `.gitlab-ci.yml` à la racine du projet
 - La CI est active !
 - Ce fichier liste toutes les tâches qui doivent être exécutées (les jobs !): quoi, où, quand, dans quel contexte ...
- 2 Définir et enregistrer des `runners` : les hôtes des jobs

CI setup

Avec  **GitLab**, un outil adapté et performant : `gitlab-ci`

<https://docs.gitlab.com/ee/ci/>

Setup?

- 1 Créer un fichier `.gitlab-ci.yml` à la racine du projet
 - La CI est active !
 - Ce fichier liste toutes les tâches qui doivent être exécutées (les jobs !): quoi, où, quand, dans quel contexte ...
- 2 Définir et enregistrer des `runners`: les hôtes des jobs
- 3 😞 Décrire (yaml) les jobs dans le fichier `.gitlab-ci.yml`

Conseil: profitez des nombreux templates. "copy and paste" est votre ami.

Vocabulaire

- **Job** : une séquence d'actions à exécuter dans un contexte prédéfini, sur un runner
 - Chaque job est exécuté indépendamment des autres
 - Tout est supprimé lorsque le travail se termine, à l'exception des artefacts
- **Artifacts** : répertoires et fichiers à conserver et à transférer entre les jobs.
- **Runner** : machine, hébergeant et exécutant un job
- **Executor** : utilisé pour exécuter votre job sur le runner (shell, Docker, ...)
- **Pipeline** : une séquence de jobs (éventuellement dépendants).
Chaque pipeline correspond à un seul commit
- **Stage** : du pipeline, peut contenir plusieurs jobs, exécution parallèle

A partir d'ici, toutes les exercices seront décrits dans le projet [anf-mathrice-2026/forges-git/trainings](#) habituel !

Et à emporter  : [les projets du groupe Demos](#),
des tutos/projets 'clé en main' incluant tout ce sur quoi nous allons travailler
aujourd'hui.

- Alternance entre cours et démos
- Dans les projets : description détaillée et complète (pour retravailler plus tard)
- Des fichiers modèles dans les répertoires "templates"

Premier exemple de job

```
my_job:  
  script:  
    - ls  
    - whoami  
    - uname -a  
    - source scripts/do_something.sh
```

- Un (nouveau) langage, yaml
- Un nom : my_job
- Des instructions : le mot-clé **script** comme elles seraient exécutées dans un terminal

👉 Demo 1.1

- Le "pipeline editor"
- Le menu Build
- Ce qui se passe au lancement d'un job

Un pas de plus ...

👉 **Demo 1.2:** construire et rendre disponible un pdf

Notions abordées :

- Artifacts
- Image et Docker executor

Construire et deployer des pages web - Gitlab-pages

Un outil pour publier automatiquement un site web *statique*, associé à votre projet

- Pages privées ou publiques
- Publication et hébergement délégués à Gitlab
- Doc : <https://about.gitlab.com/features/pages/>

Comment?

- 1 Choisir un générateur de site moderne (par exemple Pelican, Sphinx, Mkdocs ..., plus [ici](#)).
Bref, un outil capable de générer des fichiers html.
- 2 Sauvegarder les sources dans un projet gitlab.
- 3 Ecrire un job nommé "pages".

Contrôle de la CI: rules, conditions et tags

CI : facile à mettre en place.

Attention à l'empreinte écologique et à la consommation de ressources 😞

👉 Bonnes pratiques :

- Réfléchir 😊
- Ajouter des règles pour n'exécuter que ce qui est vraiment nécessaire.
- Réduire le nombre et la durée des jobs, la taille des images stockées

Contrôle de la CI: rules, conditions et tags

Contrôler la CI est facile, avec l'une ou l'autre des options suivantes

- Messages de Commit

e.g. : ne pas exécuter la CI pour chaque commit → ajouter [skip CI] dans le message de commit

- Règles en yaml

e.g. : un pipeline réduit pour les branches de développement (et tout le reste pour les branches principales ou les versions), contrôle manuel des jobs ...

- Tags sur les runners

e.g. : à la configuration du runner, ajout d'un tag : le runner ne prendra que les jobs contenant ce tag. Alternative : cocher "Run untagged jobs".

Contrôle de la CI: rules, conditions et tags

Quelques mot-clés et options utiles

- **if** : pour tester une condition.
- **when** : pour déterminer quand le job doit être exécuté.
- **rules** : pour lister les règles à appliquer (globales ou pour un job).
- **workflow** : pour fixer des règles au niveau du pipeline.
- **allow_failure: true** : le pipeline peut continuer même en cas d'échec.
- **when: on_success** : seulement si les jobs précédents (dans le pipeline) ont réussi
- **when: manual** : exécution en mode manuel (un bouton dans le navigateur)

Plus d'infos : voir les démos et [la doc Gitlab](#).

Contrôle de la CI: rules, conditions et tags

Exemples

```
mon-job:
  ...
  rules:
    - if: $CI_COMMIT_MESSAGE =~ /\[publish\].*/i
      when: always
    - when: never
```

Si la condition est vérifiée $\Rightarrow$ le job est créé.
Si non, pas de job dans le pipeline.

Contrôle de la CI: rules, conditions et tags

Exemples

```
mon-job:
  ...
  rules:
    - if: $CI_COMMIT_MESSAGE =~ /\[publish\].*/i
      when: always
    - when: never
```

Si la condition est vérifiée $\Rightarrow$ le job est créé.
Si non, pas de job dans le pipeline.

```
monjob1:
  ...
  when: manual
  # allow_failure: true par défaut

monjob2:
  ...
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
      # allow_failure: false par défaut
```

 le comportement par défaut de manual dépend de l'endroit où le when est appliqué.

- job1 : optionnel et ne bloque pas le pipeline.
Si il échoue $\Rightarrow$ "success with warning (- job2 : le pipeline est bloqué tant que job2 n'a pas été déclenché. Pipeline failed () en cas d'échec.

A propos des variables (pré-définies ou pas)

Des variables d'environnement pour contrôler (entre autres) le comportement de vos jobs et pipelines.

- **Variables prédéfinies.** e.g. `$CI_PROJECT_DIR`, `$CI_COMMIT_MESSAGE` ...



ajoutez “env” dans vos jobs. Voir la liste complète dans [la doc Gitlab](#).

- Définies dans votre script CI (niveau global ou job)

```
variables:  
  name: value
```

- Définies avec l'interface utilisateur de votre groupe/projet :
Settings → CI/CD → variables.
Peuvent être masquées ou protégées.

Gitlab Pages demo

Demos 1.3

- Stage
- Pipeline
- Pages
- Jobs dependencies
- mots-clés "when" et "rules",
- variables CI,
- badges.



Retrouvez la démo complète dans le projet [Exemple intégration continue - Bases et utilisation de gitlab-pages](#).



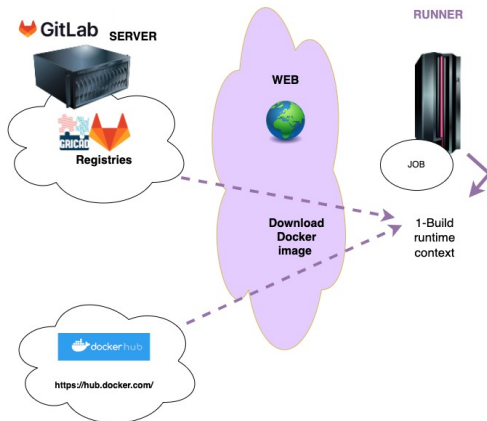
docker - *Quel intérêt ?*

- Accès à une large gamme de systèmes et d'outils potentiellement indisponibles sur votre système d'exploitation
- Facilité de reproduction des environnements utilisateurs
- Docker est disponible en tant qu'exécuteur pour gitlab-ci
- Vous pouvez utiliser le CI pour créer vos propres images et les sauvegarder dans un projet Gitlab (**registries** Gitlab, plus tard, soyez patients).



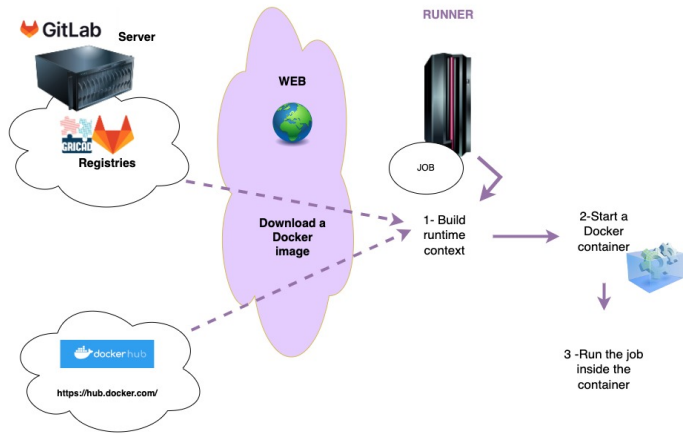
Un outil essentiel pour (essayer d')assurer des environnements reproductibles

Docker executor pour Gitlab CI



Registries : un ensemble d'images téléchargeables

Docker executor pour Gitlab CI



Container : un contexte d'exécution isolé pour chaque job

Retour sur les runners

Runner : une machine, un hôte sur lequel vont être exécutées les tâches d'intégration continue.



- Des **shared runners** disponibles (⚠ variable selon la plateforme) pour tous les projets

Retour sur les runners

Runner : une machine, un hôte sur lequel vont être exécutées les tâches d'intégration continue.



- Des **shared runners** disponibles (⚠ variable selon la plateforme) pour tous les projets
- Des **runners privés "self-managed"**, associés à un groupe ou un projet
 - 👉 *N'importe quelle machine à votre disposition : votre poste, un serveur, une machine virtuelle ...*
 - 👉 *Possiblement dans un réseau privé. Pas d'ouverture vers l'extérieur nécessaire.*
 - Req : capable de contacter (ping) le serveur Gitlab et de cloner un projet*

Retour sur les runners

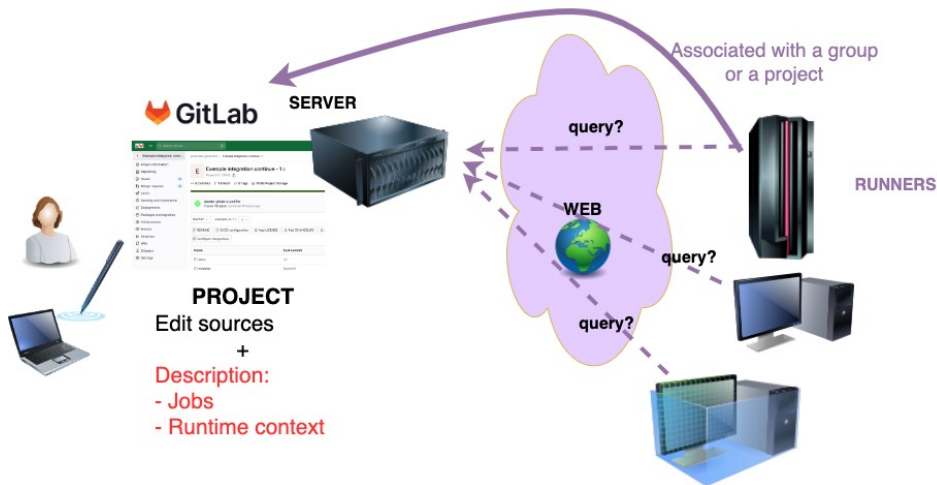
Runner : une machine, un hôte sur lequel vont être exécutées les tâches d'intégration continue.



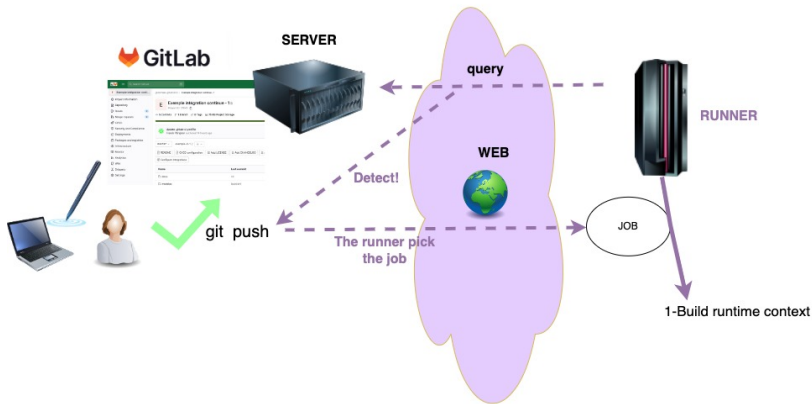
- Des **shared runners** disponibles (⚠ variable selon la plateforme) pour tous les projets
- Des **runners privés "self-managed"**, associés à un groupe ou un projet
 - 👉 *N'importe quelle machine à votre disposition : votre poste, un serveur, une machine virtuelle ...*
 - 👉 *Possiblement dans un réseau privé. Pas d'ouverture vers l'extérieur nécessaire.*
 - Req : capable de contacter (ping) le serveur Gitlab et de cloner un projet*

Où voir/trouver les runners ?

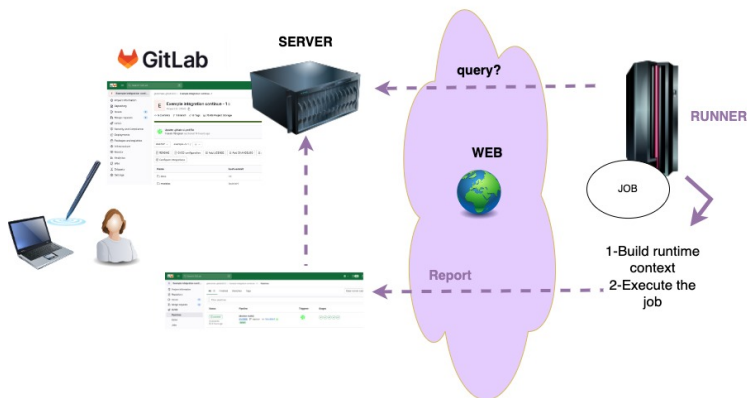
- Menu Build/runners (groupes), menu Settings/CI CD (projets)



Plusieurs runners peuvent être reliés à un seul groupe ou projet.

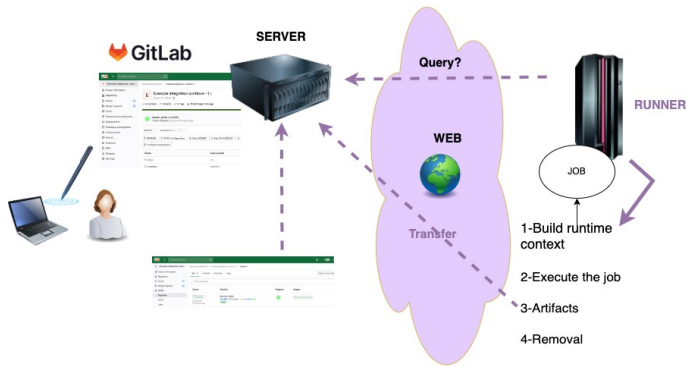


Git push $\Rightarrow$ job "pris" par un runner (💡 Des tags pour contrôler la portée du runner)



Sur le runner

- creation du contexte (executor ...),
- clone du project dans le contexte et exécution des opérations requises (job),
- envoi d'un rapport au projet propriétaire du job.



Artifacts : fichiers ou répertoires, résultats du job, devant être conservés

- Sauvegardés sur le serveur Gitlab. ⚠ Attention au coût en mémoire.
 - Définir une date d'expiration.
 - Sauvegarder seulement ce qui est nécessaire.
- Possiblement transférés d'un job à l'autre.

Comment déclarer/installer un runner ?

- 1 Disposer d'une machine et installer gitlab-runner (et Docker)
Installation standard et sans difficulté - Exemple: gitlab runner pour ubuntu/debian

Comment déclarer/installer un runner ?

- ① Disposer d'une machine et installer gitlab-runner (et Docker)
Installation standard et sans difficulté - Exemple: [gitlab runner pour ubuntu/debian](#)
- ② Demander un runner auprès du projet/groupe (Interface web Gitlab)
 - Projet: Settings → CI/CD → New project runner
 - Groupe: Build → Runner → New group runner

Comment déclarer/installer un runner ?

- 1 Disposer d'une machine et installer gitlab-runner (et Docker)
Installation standard et sans difficulté - Exemple: [gitlab runner pour ubuntu/debian](#)
- 2 Demander un runner auprès du projet/groupe (Interface web Gitlab)
 - Projet: Settings → CI/CD → New project runner
 - Groupe: Build → Runner → New group runner
- 3 Enregistrer la machine en tant que runner auprès du projet ou groupe (ligne de commande, sur l'hôte)
 établir la communication entre le runner et le serveur gitlab

```
gitlab-runner register \  
--url https://plmlab.math.cnrs.fr \  
--token <SOME-TOKEN>
```

Comment configurer son runner ?

Après les étapes précédentes, le runner est opérationnel, avec une configuration par défaut. Toute la conf est décrite dans le fichier `/etc/gitlab-runner/config.toml`.

```
# Combien de jobs peuvent tourner en parallèle sur ce runner
```

```
concurrent = 4
```

```
...
```

```
[[runners]]
```

```
  name = "anf-runner-2"
```

```
  url = "https://plmlab.math.cnrs.fr"
```

```
  token = "..."
```

```
  executor = "docker"
```

```
  [runners.docker]
```

```
    image = "alpine:3.18.4"
```

```
    privileged = false
```

```
    ...
```

```
[[runners]]
```

```
  name = "anf-runner-3"
```

```
  ...
```


Comment configurer son runner ?

- Plusieurs runners peuvent être enregistrés sur le même hôte.
→ ils apparaissent tous dans le fichier config.toml.
- Ces runners peuvent correspondre à des projets sur des forges différentes.
- Le mode "privileged=true" est donne accès à des droits élevés pour la construction d'images (voir plus loin).
- Attention à la gestion des images Docker sur l'hôte ... La volumétrie peut vite devenir importante !
 Utilisez /etc/docker/daemon.json pour choisir le chemin de stockage des images et containers Docker.

Demos 1.4,

- Runners registration
- Runner tags
- Executors

Où en sommes nous ?

- ✓ Un projet avec un fichier `.gitlab-ci.yml`. La CI est active.
- ✓ Un ou plusieurs runners disponibles, prêts et associés au projet/groupe.
- ✓ Mots-clés de base connus et compris (image, artefacts, script, ...). Ok ?

Où en sommes nous ?

- ✓ Un projet avec un fichier `.gitlab-ci.yml`. La CI est active.
- ✓ Un ou plusieurs runners disponibles, prêts et associés au projet/groupe.
- ✓ Mots-clés de base connus et compris (image, artefacts, script, ...). Ok ?

👉 Passons à un "vrai" projet de logiciel.

Démo : CI/CD pour un environnement logiciel "réel"

De quoi a t'on besoin (en général) pour un projet logiciel ?

- Construire un exécutable (et plus) : configure, build et tests du logiciel (e.g. cmake, make, make test).
 - Pour différents systèmes d'exploitation (Ubuntu, Debian ...).
 - Pour différentes configurations (Debug, release ...).
- Générer de la documentation et publier la page web du logiciel.
- Contrôler et automatiser des publications de versions, de releases.
- Le tout en collaborant entre développeurs et avec des utilisateurs.
- ...

👉 Nous allons utiliser ce cas pour illustrer l'utilisation d'un process CI/CD plus complexe dans les **démos partie 2**

⚠ Les bases de l'intégration continue ont été abordées dans les démos précédentes. Les notions de cette seconde partie sont d'un usage plus avancé et nous n'aurons pas forcément le temps de tout aborder en détail. L'idée est de vous fournir une base de travail, des modèles et idées à éventuellement exploiter plus tard dans vos projets.

Les trainings / [demos partie 2](#) :

- 1 revue (non exhaustive) / description de quelques notions
- 2 illustration dans le projet [Demos - Exemple intégration continue - CI CD pour un code de calcul](#)

Un premier pipeline

Configure, build et tests d'un logiciel, pour plusieurs systèmes d'exploitation

Demos 2.1.

Notions abordées :

- liens de dépendances entre jobs : mot-clé `needs`
- gestion des dépendances du logiciel, mot-clé `before_script`
- rapports de tests via les artifacts, mot-clé `reports`



Que pensez-vous des scripts CI précédents ? Des points problématiques ?

2 problèmes dans les démos précédentes (2.1)

- des répétitions ennuyeuses (mêmes opérations pour des OS différents)
- before script :
 - la manière d'installer les dépendances peut dépendre de l'OS $\Rightarrow$ des before_script différents
 - on répète la même chose pour chaque job (long, coûteux ...)

👉 inutile : nous n'avons pas besoin de tester les OS (apt ... ou outils équivalents), seulement notre logiciel !

2 problèmes dans les démos précédentes (2.1)

- des répétitions ennuyeuses (mêmes opérations pour des OS différents)



utilisez les templates !

- before script :

- la manière d'installer les dépendances peut dépendre de l'OS $\Rightarrow$ des before_script différents
- on répète la même chose pour chaque job (long, coûteux ...)



inutile : nous n'avons pas besoin de tester les OS (apt ... ou outils équivalents), seulement notre logiciel !



construisez vos propres images !

Réutilisation de “modèles” pour gitlab-ci - extends

Il est possible de créer des configurations de référence utilisables comme base pour plusieurs jobs.

Déclaration du “template”

```
.soft-build:
  image: $IMAGE_NAME
  stage: build
  script:
    - env
    - python -m build $CI_PROJECT_DIR/python
```

- 👉 factorisation des instructions
- 👉 utilisation des variables CI

Utilisation

```
python:ubuntu:build:
  variables:
    IMAGE_NAME: ubuntu:24.04
  extends:
    - .soft-build

python:debian:build:
  variables:
    IMAGE_NAME: debian:buster
  extends:
    - .soft-build
```

(Bonnes) pratiques :

- Définir des 'extends' pour factoriser les opérations
- Un fichier dédié pour déclarer tout ce qui est commun (templates, variables ...)

```
# Fichier ci-templates.yml

variables:
  BUILD_DIR: $CI_PROJECT_DIR/build
  ...

stages:
  - build
  - test

.soft-build:
...
```

```
# fichier .gitlab-ci.yml
include:
  - ci/ci-templates.yml

python:debian:build:
  variables:
    IMAGE_NAME: debian:buster
  extends:
    - .soft-build

...
```

Construire et stocker des images Docker via la CI

Rappel du problème :

- il faut gérer le contexte dans lequel on veut construire, tester notre code,
- **before_script** est une solution mais peu satisfaisante.



solution : construire et utiliser vos propres images !

Etapes :

- 1 Fabriquer une image Docker 📌 recette **reproductible**.
- 2 La sauvegarder quelque part 📌 registries du projet (contrôle de contenu, de l'accès ...).
- 3 Utiliser cette image dans vos jobs.

Comment construire et stocker son image docker ?

Construction d'une image : voir démo

En attendant, reprenez simplement qu'il suffit de fournir un fichier **Dockerfile** et d'appliquer une commande.

- Faisable directement sur votre machine.
- Mieux : laisser faire la CI !

👍 automatique. Votre seul travail : écrire le Dockerfile,

👍 reproductible (construction ET utilisation),

👍 images dédiées, appartenant au projet (privées, publiques, à vous de choisir).

Ajouter une images dans les registries d'un projet

Etapes :

- 1 Construire une image localement (docker build ...)
- 2 La nommer (docker tag ...)
registry.plmlab.math.cnrs.fr/<NOM_GROUPE>/.../<NOM_PROJET>/somename
- 3 La pousser dans les registries du projet

```
docker login registry.plmlab.math.cnrs.fr # username/password ou token
docker push registry.plmlab.math.cnrs.fr/.../somename
```

👉 **Démo 2.2 et 2.3**

- Utilisation d'extends (2.2)
- Kaniko et CI pour construire des images docker et les stocker dans les registries du projet (2.3)

A propos des registries et des runners



Attention à l'empreinte mémoire, les images docker tiennent rapidement de la place.



Ne produire que ce qui est nécessaire

(voir intro impact environnemental, réfléchir tout ça ...)



Registries. Mettre en place une "cleanup policy"

Sur la page du projet, menu Deploy/Container registries : "Set up cleanup".

Définition de règles de conservation/suppression d'images (selon age, regexp ...)



Runners. Nettoyer régulièrement ou automatiquement (cron).

docker prune, rm, rmi ...

Indispensable !

Pour les admin de plateforme : garbage collector.

Performance des pipelines : où passe-t-on du temps ?

Durée d'un job = attente du runner + pull de l'image + clone du projet
+ exécution du script + envoi des artifacts.

👉 Avant d'optimiser : n'exécuter que ce qui est nécessaire (rules, workflow ...)

👉 Les leviers :

- Paralléliser : les jobs d'un même stage s'exécutent en parallèle, le mot-clé needs permet même de démarrer un job sans attendre la fin du stage
- Interrompre ce qui est périmé : interruptible: true $\Rightarrow$ un nouveau push annule les pipelines de la branche devenus inutiles
- Réduire le coût de mise en place de l'environnement du job (diapo suivante)

Installer ses dépendances : cache ou image ?

Le **cache** Gitlab est une archive (tar.gz) téléchargée puis extraite au début du job, re-téléversée à la fin. On ne gagne du temps que si **reproduire son contenu est plus long**. Exemple pour un projet Python :

- pip : souvent utile (installation lente)
- uv : installation rapide $\Rightarrow$ gain souvent marginal
- Dépendances lourdes (torch, scipy ...) ou bande passante limitée $\Rightarrow$ utile

 Pièges :

- les chemins du cache doivent être dans \$CI_PROJECT_DIR
- un contenu modifié à chaque exécution est re-téléversé à chaque pipeline.



Plus efficace d'embarquer les dépendances dans l'image Docker (l'extraction par le runtime de conteneur est bien plus efficace qu'une archive).

Contrôler le workflow

Ok, vous avez choisi un workflow git, bien. Mais comment le gérer correctement avec le CI ?

Même comportement pour toutes les branches, pour MR, ... ?

Non !

- Ajouter des règles
- Utilisez le mot-clé **Workflow**: contrôler quand les pipelines sont créés (entre autres)

👉 **Voir démos 2.4**, combiner branches, MR et issues. Utiliser un workflow et des règles de CI.

Passons au "Continuous Delivery"

- ✓ CI pour configurer, construire, tester nos logiciels
- ✓ Capable de basculer entre différents contextes (OS, paramètres ...)
- ✓ Contrôler le workflow (règles, ...)

Plus ?

- Release
- Installer le logiciel et le rendre disponible
- Produire de la documentation ...

CI/CD pour rendre son logiciel disponible

👉 Démon 2.5

- CI/CD pour livrer des images Docker "prêtes à l'emploi", contenant notre logiciel.
- Comment écrire dans les registres d'autres projets.

Pré-requis: un **token**, pour gérer les droits (écrire dans les registries d'un autre projet).
Comment ?

- Être propriétaire des groupes ou admin des projets concernés
- Créer un token dans le projet cible (ou mieux dans un groupe contenant les projets source et cible)
→ Settings → Access tokens
- Utilisez ce token pour renseigner des variables CI/CD (\$REGISTRY_USER et \$REGISTRY_PASSWORD dans notre démo) dans le groupe ou projet source (celui qui lance la CI)

Release

Démonos 2.6

- CI pour produire une Release à chaque création d'un tag dans le dépôt git.

Triggered jobs

Démonos 2.6

- CI pour déclencher des jobs dans un autre projet

Quelques items que nous n'avons pas eu le temps d'aborder

Mais qui sont potentiellement très intéressants

- Gitlab components
- git annex

Ce cours collectif est inspiré de plusieurs sources issues d'autres formations impliquants les intervenants d'aujourd'hui mais aussi d'autres personnes que nous tenons à citer et remercier !

- un cours du collège des écoles doctorales à Grenoble, construit et donné avec Alizia Tarayoun. [Gestion de projet et développements collaboratifs - \(CED Grenoble depuis 2018\)](#)

Merci !