Manuel utilisateur

Gitrust — forge Git self-hosted écrite en Rust

Équipe Gitrust

© Gitrust

Sommaire

1. Manuel utilisateur 3
1.1 Manuel utilisateur 3
1.2 Tutoriels 6
1.3 Guides pratiques 33
1.4 Référence 69
1.5 Explications 95

© Gitrust

1. Manuel utilisateur

1.1 Manuel utilisateur

Ce manuel t'accompagne depuis ta première connexion à une instance gitrust jusqu'à l'autonomie complète : pousser du code, collaborer via les pull requests, gérer les issues et automatiser tes workflows avec la CI intégrée. Tu n'as pas besoin d'administrer ni de comprendre l'architecture interne — ce manuel suppose seulement que tu sais utiliser Git en ligne de commande.

1.1.1 Parcours d'apprentissage

Le manuel est structuré comme un **parcours progressif** de 4 tutoriels. Chaque tutoriel te laisse dans un état stable, vérifié, avant de passer au suivant. L'aide fournie recule d'un cran à chaque étape (du copier-coller intégral vers la composition autonome).

```
graph LR
A[Compte créé] --> B[01-premier-pas
20 min]
B --> C[02-cloner-pousser
25 min]
C --> D[03-collaborer
25 min]
D --> E[04-automatiser-ci
20 min]
E --> F[Utilisateur autonome]
```

> ***Checkpoint de parcours** : avant de passer au tutoriel 02, tu dois pouvoir te connecter à gitrust, voir ton dépôt dans l'interface et l'accès SSH configuré. Si ce n'est pas le cas, relis le tutoriel 01.*
>
> ***Checkpoint de parcours** : avant le tutoriel 03, tu dois avoir poussé au moins un commit sur ta branche via SSH. Si ce n'est pas le cas, relis le tutoriel 02.*
>
> ***Checkpoint de parcours** : avant le tutoriel 04, tu dois avoir ouvert et mergé une pull request. Si ce n'est pas le cas, relis le tutoriel 03.*

© Gitrust

1.1.2 Tutoriels — apprendre en faisant

Les tutoriels sont guidés pas-à-pas avec des sorties verbatim attendues. Suis-les dans l'ordre — chacun suppose le précédent complété.

- 01 — Premiers pas : 2FA, clé SSH, premier dépôt — ~20 min
- 02 — Cloner et pousser du code — ~25 min
- 03 — Collaborer : équipes, issues, pull requests — ~25 min
- 04 — Automatiser avec la CI — ~20 min

1.1.3 How-to — recettes pour les tâches courantes

Les how-to répondent à « comment faire X » sans explication de fond. Tu peux les consulter dans n'importe quel ordre selon ton besoin immédiat.

Ces guides supposent que tu as suivi le parcours tutoriels. Ils ne répètent pas les étapes de base.

- Gérer ses clés SSH
- Configurer la 2FA
- Créer un Personal Access Token
- Gérer les équipes et permissions
- Ouvrir une pull request
- Fusionner : fast-forward, squash, merge commit
- Utiliser les labels hiérarchiques
- Importer un dépôt externe
- Consulter le SBOM de ton projet

1.1.4 Référence — informations techniques exactes

La référence est destinée à la consultation ponctuelle, pas à la lecture linéaire. Elle documente les formats, schémas et comportements exacts.

- Modèle de permissions (4 rôles)
- API REST v1 — authentification, pagination, codes d'erreur
- Schéma YAML `.gitrust-ci.yml` (Easy Mode)
- Syntaxe Markdown supportée
- Notifications : SSE, e-mail, préférences

© Gitrust

1.1.5 Explication — comprendre le pourquoi

Les explications t'aident à construire des modèles mentaux durables. Lis-les quand tu veux comprendre *pourquoi* gitrust fonctionne ainsi, pas seulement *comment*.

- Modèle de collaboration gitrust
- Sécurité utilisateur : 2FA, PAT, audit log
- Cycle de vie d'une pull request

© Gitrust

1.2 Tutoriels

1.2.1 Premiers pas : active la 2FA, ajoute ta clé SSH et crée ton premier dépôt

Objectifs

À la fin de ce tutoriel, tu sauras :

- **O1. Activer** l'authentification à deux facteurs (2FA) sur ton compte gitrust
- **O2. Enregistrer** une clé SSH publique dans ton profil pour te connecter sans mot de passe
- **O3. Créer** un dépôt vide et vérifier qu'il est accessible à l'URL attendue

Pré-requis

- **Technique** : un compte gitrust existant (e-mail + mot de passe), Git installé (`git --version` répond), une paire de clés SSH générée ou à générer (`ssh-keygen`)
- **Pédagogique** : aucun tutoriel préalable requis — c'est le point de départ du parcours
- **Temps estimé** : ~20 minutes

Vue d'ensemble

Avant de taper la première commande, prenons 2 minutes pour comprendre ce que tu vas faire et pourquoi.

Un dépôt (*repository*), c'est comme un dossier de projet avec tout l'historique de ses modifications — une boîte d'archives vivante que Git surveille. Sur gitrust, ce dossier vit sur le serveur et peut être partagé avec ton équipe.

Une clé SSH fonctionne comme une serrure et une clé : tu gardes la clé privée sur ta machine (ne la partage jamais), et tu déposes la clé publique dans ton profil gitrust. Ensuite, Git se connecte automatiquement sans te redemander de mot de passe — gitrust reconnaît ta machine.

La 2FA (authentification à deux facteurs) ajoute un second verrou à ton compte : même si quelqu'un vole ton mot de passe, il ne peut pas se connecter sans le code de ton téléphone. C'est une protection essentielle pour tout compte qui héberge du code.

Voici le flux que tu vas configurer :

```
sequenceDiagram
participant T as Ton navigateur
participant G as gitrust (HTTPS :4000)
participant S as Ton terminal (SSH :2222)

T->>G: Connexion avec e-mail + mdp + code 2FA
T->>G: Dépôt de la clé SSH publique
```


© Gitrust

```
S->>G: git push via SSH (authentifié par clé)
G-->>T: Dépôt visible dans l'interface web
```

Étape 1 : Connecte-toi et accède aux paramètres de sécurité

Ouvre ton navigateur et va sur l'URL de ton instance gitrust (par exemple `https://demo.gitrust.eu`). Connecte-toi avec ton e-mail et ton mot de passe.

Une fois connecté, clique sur ton avatar en haut à droite, puis sur **Paramètres** (`/settings`). Dans le menu de gauche, sélectionne **Sécurité**.

Checkpoint : tu dois voir la page *Sécurité* avec une section « Authentification à deux facteurs » et le statut « Désactivée ». Si tu ne la vois pas, vérifie que tu es bien connecté avec le bon compte.

Étape 2 : Active la 2FA avec une application d'authentification

Sur la page *Sécurité*, clique sur **Activer la 2FA**. gitrust affiche un QR code.

1. Ouvre ton application d'authentification (Google Authenticator, Authy, ou tout équivalent compatible TOTP).
2. Scanne le QR code avec l'application.
3. L'application affiche un code à 6 chiffres qui change toutes les 30 secondes.
4. Saisis ce code dans le champ **Code de vérification** et clique **Confirmer**.

Sortie attendue (message dans l'interface) :

```
La 2FA a été activée avec succès.
Conservez vos codes de secours dans un endroit sûr.
```

gitrust affiche également **10 codes de secours** (format `XXXX-XXXX`). Copie-les et conserve-les hors ligne — ils te permettent de récupérer l'accès si tu perds ton téléphone.

Checkpoint : retourne sur *Paramètres → Sécurité*. La section 2FA doit maintenant afficher « **Activée** » en vert. Si elle affiche toujours « Désactivée », recommence l'étape en t'assurant que le code à 6 chiffres est saisi avant qu'il expire.

© Gitrust

Étape 3 : Génère ou localise ta clé SSH

Si tu n'as pas encore de paire de clés SSH, génère-en une maintenant dans ton terminal :

```
ssh-keygen -t ed25519 -C "mon-compte@example.com"
```

Sortie attendue :

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/toi/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/toi/.ssh/id_ed25519
Your public key has been saved in /home/toi/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:xXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXx mon-compte@example.com
```

Appuie sur Entrée pour accepter le chemin par défaut. Tu peux choisir une passphrase ou laisser vide (appuie deux fois sur Entrée). Une passphrase ajoute une couche de sécurité.

Affiche maintenant ta **clé publique** (le fichier `.pub` — celui-ci est fait pour être partagé) :

```
cat ~/.ssh/id_ed25519.pub
```

Sortie attendue (une seule ligne) :

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx m
```

Copie cette ligne entière dans le presse-papier.

Checkpoint : la commande `cat` doit afficher une ligne commençant par `ssh-ed25519` (ou `ssh-rsa` si tu as une clé plus ancienne). Si elle affiche « No such file or directory », la génération n'a pas abouti — relance `ssh-keygen`.

Étape 4 : Enregistre la clé SSH dans ton profil gitrust

Dans ton navigateur, retourne sur *Paramètres* et clique sur **Clés SSH** dans le menu de gauche.

© Gitrust

Clique sur **Ajouter une clé SSH**. Un formulaire apparaît :

- **Titre** : donne un nom descriptif à cette clé (ex. `laptop-perso-2026`, `poste-bureau`)
- **Clé** : colle la ligne copiée à l'étape précédente

Clique **Enregistrer la clé**.

Sortie attendue (message dans l'interface) :

```
Clé SSH ajoutée avec succès.
```

Teste maintenant la connexion SSH depuis ton terminal :

```
ssh -T git@demo.gitrust.eu -p 2222
```

Sortie attendue :

```
Bonjour tonpseudo ! Vous êtes authentifié, mais gitrust ne fournit pas d'accès shell.
```

Checkpoint : si tu obtiens ce message, ta clé SSH est correctement enregistrée. Si tu obtiens `Permission denied (publickey)`, vérifie que tu as bien copié la clé **publique** (`.pub`) et non la clé privée. Consulte la section « Et si ça ne marche pas » ci-dessous.

Étape 5 : Crée ton premier dépôt vide

Dans ton navigateur, clique sur le + en haut à droite (ou navigue vers `/repo/new`).

Remplis le formulaire :

- **Propriétaire** : ton pseudo
- **Nom du dépôt** : `mon-premier-depot` (lettres, chiffres, tirets uniquement)
- **Visibilité** : Privé (ou Public selon ta préférence)
- **Initialiser le dépôt** : coche « Ajouter un README »

Clique sur **Créer le dépôt**.

Sortie attendue : gitrust te redirige vers la page du dépôt à l'URL :

```
https://demo.gitrust.eu/tonpseudo/mon-premier-depot
```


© Gitrust

La page affiche un fichier `README.md` avec le nom du dépôt, et la barre latérale montre 1 commit, 1 branche (`main`).

Checkpoint : note l'URL affichée dans la barre d'adresse de ton navigateur. Elle doit correspondre au patron `https://<instance>/<pseudo>/<nom-depot>` . Si tu vois une erreur 404 ou 500, relis le formulaire de création.

Récapitulatif

- √ **O1 accompli** : la 2FA est activée sur ton compte — tu l'as vérifiée sur la page *Sécurité* qui affiche « Activée »
- √ **O2 accompli** : ta clé SSH publique est enregistrée — tu l'as confirmée avec `ssh -T git@... -p 2222` qui a répondu « Vous êtes authentifié »
- √ **O3 accompli** : ton premier dépôt est créé et visible à l'URL `/<pseudo>/mon-premier-depot`

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`Permission denied (publickey)` lors du test SSH	Tu as collé la clé privée au lieu de la clé publique, ou le fichier `.pub` ne correspond pas à la clé active	Dans gitrust, supprime la clé ajoutée. Retourne au terminal et exécute `ssh-add -l` pour voir les clés actives. Recopie la bonne clé `.pub` avec `cat ~/.ssh/id_ed25519.pub`
`Connection refused` lors du test SSH	Le port SSH de l'instance est différent de 2222, ou l'instance est inaccessible	Vérifie avec l'administrateur le port SSH réel. Essaie `ssh -T git@<instance> -p 22` en alternative
Le code 2FA est refusé alors qu'il semble correct	L'horloge de ton téléphone est désynchronisée (décalage > 30 s)	Active la synchronisation automatique de l'heure sur ton téléphone (*Paramètres* → *Date et heure* → *Automatique*). Réessaie immédiatement après
Le dépôt redirige vers une erreur 422 lors de la création	Le nom contient des caractères interdits (espaces, accents, `/`)	Utilise uniquement des lettres non accentuées, chiffres et tirets. Évite les points en début ou fin de nom

Prochaine étape

→ 02 — Cloner et pousser du code : clone ton dépôt en local avec SSH, fais ton premier commit et pousse-le sur gitrust (~25 min)

© Gitrust

1.2.2 Cloner un dépôt et pousser ton premier commit

Objectifs

À la fin de ce tutoriel, tu sauras :

- **O1. Générer** une clé SSH ed25519 et la déclarer dans ton profil gitrust (`/settings/keys`)
- **O2. Cloner** un dépôt gitrust en local via SSH (`git clone git@…`)
- **O3. Pousser** un commit vers gitrust et vérifier le résultat dans l'interface web

Pré-requis

- **Technique** : Git installé (`git --version` répond), un terminal, ton instance gitrust accessible
- **Pédagogique** : tutoriel 01 — *Premiers pas* complété (compte actif, 2FA configurée, un dépôt créé)
- **Temps estimé** : ~25 minutes

Vue d'ensemble

Avant de taper la première commande, voici ce que tu vas mettre en place et pourquoi.

SSH (*Secure Shell*) est le protocole que Git utilise pour envoyer et recevoir du code de manière sécurisée. La différence avec HTTPS : au lieu de saisir un mot de passe à chaque fois, tu déposes une **clé publique** sur le serveur gitrust, et ta machine prouve son identité grâce à la **clé privée** correspondante que tu gardes chez toi. C'est à la fois plus sûr et plus pratique.

Voici les trois acteurs de ce tutoriel :

```
sequenceDiagram
participant M as Ta machine (terminal)
participant A as Agent SSH (~/.ssh/)
participant G as gitrust (:2222 SSH)
participant W as gitrust (navigateur)

M->>M: ssh-keygen → génère id_ed25519 + id_ed25519.pub
M->>W: dépôt de id_ed25519.pub dans /settings/keys
M->>A: ssh-add id_ed25519 (optionnel)
M->>G: git clone git@gitrust.example.com:owner/repo.git
G-->>M: copie locale du dépôt
M->>M: édite un fichier, git add, git commit
M->>G: git push origin main
G-->>W: commit visible dans l'interface
```


© Gitrust

Étape 1 : Génère une clé SSH ed25519

> ***Note*** *: si tu as déjà généré une clé ed25519 pendant le tutoriel 01, passe directement à l'étape 2.*

Dans ton terminal, lance la commande suivante (remplace l'adresse e-mail par la tienne) :

```
ssh-keygen -t ed25519 -C "ton@email.com"
```

Sortie attendue :

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/toi/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/toi/.ssh/id_ed25519
Your public key has been saved in /home/toi/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:xXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXx ton@email.com
The key's randomart image is:
+--[ED25519 256]--+
|        .o+      |
|        ...      |
+----[SHA256]-----+
```

Appuie sur Entrée pour accepter le chemin par défaut. Une passphrase est recommandée — elle chiffre la clé privée sur disque.

Affiche ta clé **publique** (le fichier `.pub` — celui-là est fait pour être partagé) :

```
cat ~/.ssh/id_ed25519.pub
```

Sortie attendue :

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx t
```

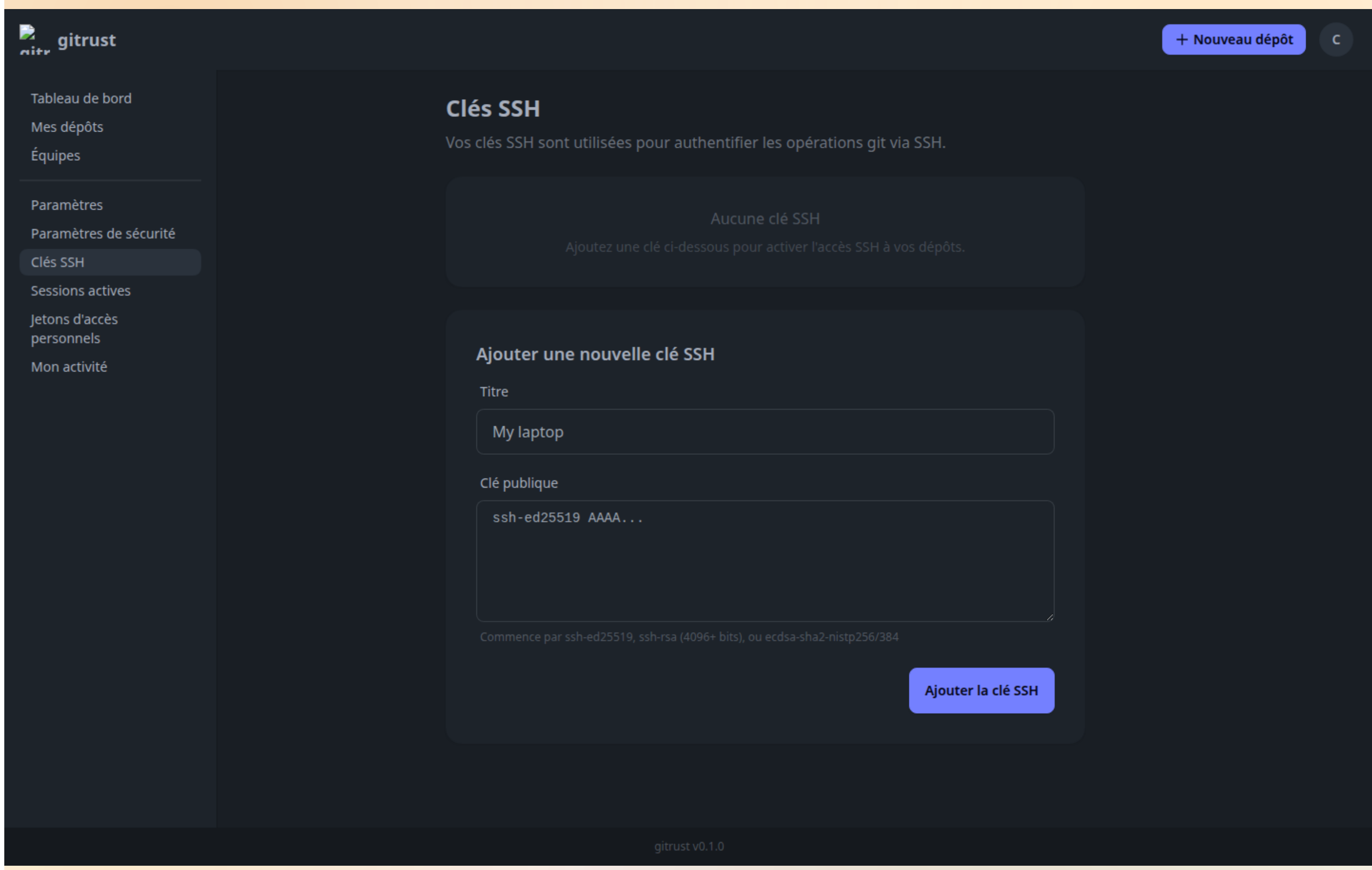
Copie cette ligne complète dans le presse-papier.

Checkpoint : la ligne commence obligatoirement par `ssh-ed25519`. Si elle commence par `-----BEGIN` tu affiches la clé **privée** par erreur — arrête-toi et relis la commande.

© Gitrust

Étape 2 : Déclare la clé dans ton profil gitrust

Dans ton navigateur, va sur **Paramètres → Clés SSH** (`/settings/keys`).



Clique **Ajouter une clé SSH**. Remplis le formulaire :

- **Titre** : un nom qui identifie la machine (ex. `laptop-maison-2026`)
- **Clé** : colle la ligne `ssh-ed25519 …` copiée à l'étape 1

Clique **Enregistrer**.

Sortie attendue (bannière verte dans l'interface) :

```
Clé SSH ajoutée avec succès.
```

Vérifie la connexion depuis le terminal :

```
ssh -T git@gitrust.example.com -p 2222
```


© Gitrust

Sortie attendue :

```
Bonjour tonpseudo ! Vous êtes authentifié, mais gitrust ne fournit pas d'accès shell.
```

Checkpoint : tu dois obtenir ce message de bienvenue. Si tu obtiens `Permission denied (publickey)`, relis l'encart « Et si ça ne marche pas » en bas de ce tutoriel.

Étape 3 : Récupère l'URL SSH de ton dépôt

Dans ton navigateur, ouvre la page de ton dépôt (`/<tonpseudo>/mon-premier-depot`).

Sur la page d'accueil du dépôt, clique sur le bouton **Cloner** (ou **Code**). Sélectionne l'onglet **SSH** et copie l'URL affichée.

Elle a la forme :

```
git@gitrust.example.com:tonpseudo/mon-premier-depot.git
```

Capture a venir

101 clone url ssh

voir scripts/screenshot-runner/

Checkpoint : l'URL SSH commence par `git@` (pas par `https://`). Si tu ne vois pas d'onglet SSH, l'instance n'a pas de serveur SSH configuré — demande à ton administrateur.

Étape 4 : Clone le dépôt en local

Dans ton terminal, place-toi dans le dossier où tu veux stocker tes projets, puis clone :

```
cd ~/projets
git clone git@gitrust.example.com:tonpseudo/mon-premier-depot.git
```


© Gitrust

Sortie attendue :

```
Cloning into 'mon-premier-depot'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
Receiving objects: 100% (3/3), done.
```

Entre dans le dossier cloné :

```
cd mon-premier-depot
ls
```

Sortie attendue :

```
README.md
```

Checkpoint : le fichier `README.md` est présent — ton dépôt est bien cloné. Si Git affiche `fatal: repository not found`, vérifie l'URL SSH copiée à l'étape précédente.

Étape 5 : Crée un fichier, commite et pousse

Crée un nouveau fichier dans le dépôt :

```
echo "# Notes de développement" > NOTES.md
```

Ajoute-le à l'index Git et crée un commit :

```
git add NOTES.md
git commit -m "ajoute NOTES.md"
```

Sortie attendue :

© Gitrust

```
[main 3a8c21f] ajoute NOTES.md
 1 file changed, 1 insertion(+)
 create mode 100644 NOTES.md
```

Pousse vers gitrust :

```
git push origin main
```

Sortie attendue :

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 298 bytes | 298.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To git@gitrust.example.com:tonpseudo/mon-premier-depot.git
   a1b2c3d..3a8c21f  main -> main
```

Checkpoint : la dernière ligne montre `main -> main` avec deux SHA différents, ce qui confirme que les objets ont bien été envoyés.

Étape 6 : Vérifie dans l'interface web

Retourne dans ton navigateur sur la page du dépôt (`/<tonpseudo>/mon-premier-depot`). Rafraîchis la page.

Tu dois voir :

- Le fichier `NOTES.md` apparaît dans la liste des fichiers
- La barre de commits affiche **2 commits** (le README initial + ton nouveau commit)
- Le message de commit `ajoute NOTES.md` est visible à côté du fichier

© Gitrust

Capture a venir

102 repo after push

voir scripts/screenshot-runner/

Checkpoint : si le fichier `NOTES.md` n'est pas visible après rafraîchissement, assure-toi que le `git push` a bien terminé sans erreur (relis la sortie de l'étape 5).

Récapitulatif

- √ **O1 accompli** : tu as généré une clé SSH ed25519 et vérifié son empreinte SHA256 — elle est enregistrée dans `/settings/keys`
- √ **O2 accompli** : tu as cloné le dépôt en local avec `git clone git@…` et obtenu une copie fonctionnelle
- √ **O3 accompli** : tu as poussé un commit avec `git push origin main` et vérifié sa présence dans l'interface web

---

© Gitrust

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`Permission denied (publickey)` au `git clone` ou `git push`	La clé privée locale ne correspond pas à la clé publique enregistrée sur gitrust, ou l'agent SSH ne la connaît pas	Exécute `ssh-add ~/.ssh/id_ed25519` pour ajouter la clé à l'agent, puis réessaie. Vérifie aussi que la bonne clé `.pub` a été copiée dans gitrust
`Host key verification failed`	L'empreinte SSH du serveur gitrust a changé ou est inconnue	Exécute `ssh-keyscan -p 2222 gitrust.example.com >> ~/.ssh/known_hosts`, puis réessaie. En environnement de test, tu peux vérifier l'empreinte avec l'administrateur
`fatal: repository 'git@…' not found`	L'URL SSH est incorrecte, ou le dépôt n'existe pas (typo dans le nom), ou tu n'as pas les droits	Vérifie l'URL sur la page web du dépôt. Assure-toi que le dépôt est bien accessible avec ton compte
`error: failed to push some refs` / `rejected`	La branche distante contient des commits que tu n'as pas en local	Exécute `git pull origin main --rebase` d'abord, résous les conflits éventuels, puis `git push` à nouveau

Prochaine étape

→ 03 — Collaborer : équipes, issues et pull requests : crée une équipe, invite un coéquipier, ouvre une issue et ta première pull request (~25 min)

© Gitrust

1.2.3 Collaborer : équipes, issues et pull requests

Objectifs

À la fin de ce tutoriel, tu sauras :

- **O1. Créer** une équipe et y inviter un coéquipier depuis `/teams/new`
- **O2. Ouvrir** une issue pour décrire un problème ou une tâche à réaliser
- **O3. Ouvrir** une pull request depuis une branche et la fusionner après review

Pré-requis

- **Technique** : dépôt cloné en local (tutoriel 02 complété), accès à un second compte gitrust (ou un coéquipier disponible)
- **Pédagogique** : tutoriel 02 — *Cloner et pousser* complété
- **Temps estimé** : ~25 minutes

Vue d'ensemble

gitrust organise la collaboration autour de trois concepts qui s'enchaînent naturellement :

- Une **équipe** (*team*) regroupe des personnes et définit leur niveau d'accès à un ou plusieurs dépôts. C'est le point d'entrée pour inviter quelqu'un sur un projet.
- Une **issue** est une unité de travail : un bug à corriger, une feature à ajouter, une question à trancher. Elle porte un numéro (`#N`) qui sert de référence partout.
- Une **pull request** (PR) est une demande de fusion d'une branche vers une autre. Elle lie le code proposé à l'issue qu'il résout et déclenche la review de l'équipe.

```
flowchart LR
T[Équipe créée
/teams/new] --> R[Dépôt assigné
à l'équipe]
R --> I[Issue ouverte
#1]
I --> B[Branche créée
feat/fix-readme]
B --> PR[Pull Request
ouverte]
PR --> Rev[Review approuvée]
Rev --> M[Fusion dans main]
```


© Gitrust

```
M --> C[Issue fermée
automatiquement]
```

> ***Scaffolding niveau 2** : dans ce tutoriel, certaines commandes comportent un trou `___` à compléter. L'aide est donnée juste après le bloc de code — essaie d'abord sans regarder.*

Étape 1 : Crée une équipe

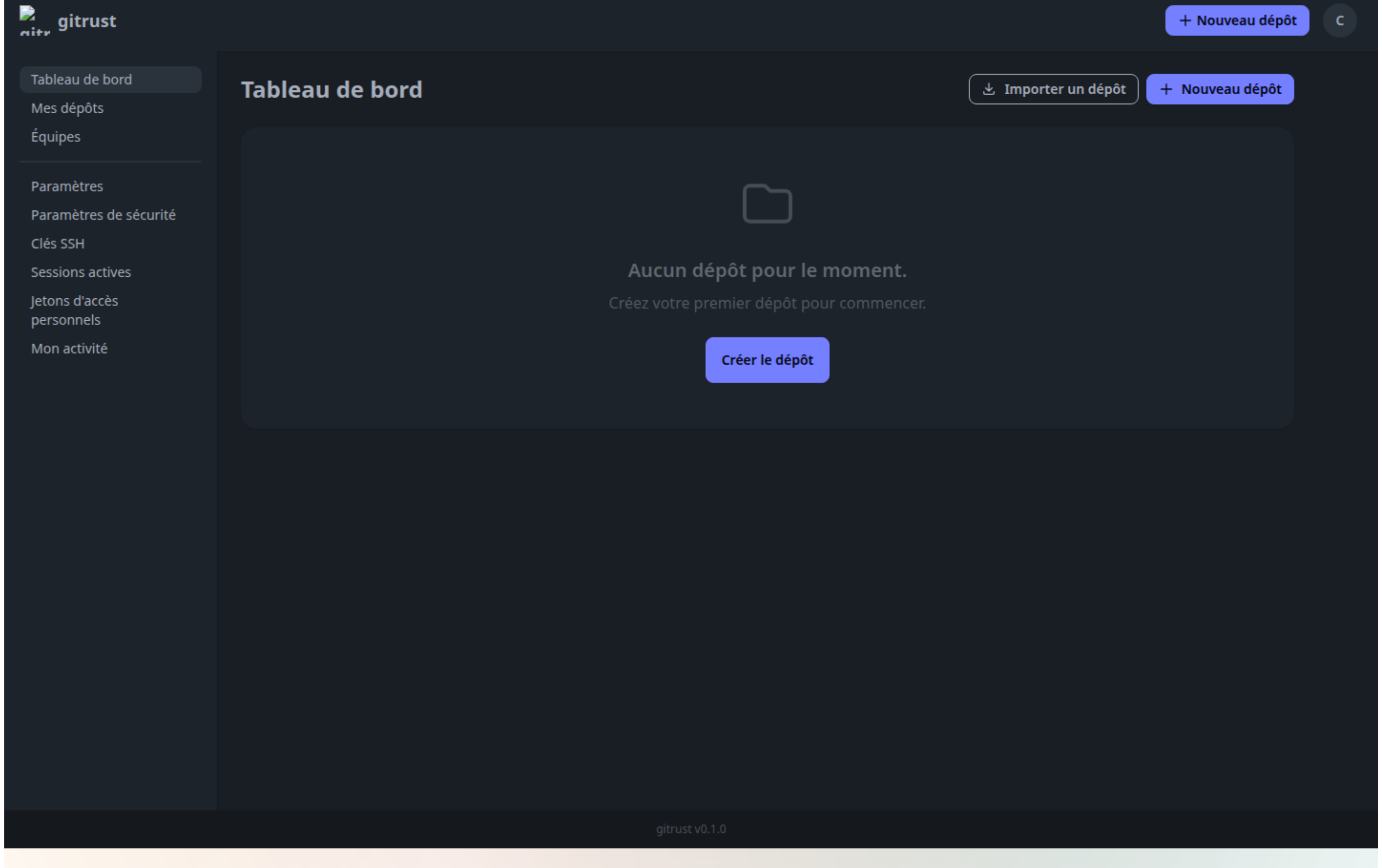
Dans ton navigateur, navigue vers `/teams/new`.

Remplis le formulaire :

- **Nom de l'équipe** : `equipe-alpha` (lettres minuscules, chiffres, tirets)
- **Description** : `Équipe de développement principale` (optionnel)

Clique **Créer l'équipe**.

Sortie attendue : gitrust te redirige vers la page de l'équipe `/teams/equipe-alpha`. La page affiche l'équipe avec toi comme premier membre avec le rôle **Owner**.



© Gitrust

Checkpoint : tu vois la page `/teams/equipe-alpha` avec ton pseudo listé en tant que membre. Si tu obtiens une erreur 422, le nom contient probablement des caractères interdits — utilise uniquement des minuscules et des tirets.

Étape 2 : Invite un coéquipier et assigne le dépôt

Sur la page de l'équipe, dans la section **Membres**, saisis le pseudo de ton coéquipier et clique **Inviter**.

Sortie attendue :

```
Membre ajouté à l'équipe.
```

Dans la section **Dépôts**, clique **Ajouter un dépôt**. Sélectionne `mon-premier-depot` dans la liste et choisis le niveau d'accès **Developer**. Clique **Confirmer**.

Sortie attendue :

```
Accès au dépôt accordé.
```

Checkpoint : ton coéquipier doit maintenant voir `mon-premier-depot` dans son tableau de bord. Demande-lui de vérifier. Le niveau Developer lui permet de cloner, pousser sur les branches non-protégées et ouvrir des PRs.

Étape 3 : Ouvre une issue

Dans ton navigateur, navigue vers `/<tonpseudo>/mon-premier-depot/issues/new`.

Remplis le formulaire :

- **Titre** : `Améliorer le README avec une description du projet`
- **Description** :

  ```
  Le fichier README.md est vide. Il faudrait ajouter :
  - Une description du projet
  - Les instructions d'installation
  ```

Clique **Ouvrir l'issue**.

Sortie attendue : gitrust crée l'issue et te redirige vers `/<tonpseudo>/mon-premier-depot/issues/1`. La page affiche l'issue avec le statut **Ouverte** et le numéro **#1**.

© Gitrust

Capture a venir

104 issue open

voir scripts/screenshot-runner/

Checkpoint : l'URL dans ton navigateur doit se terminer par `/issues/1`. Note ce numéro — tu l'utiliseras dans le message de commit pour fermer l'issue automatiquement.

Étape 4 : Crée une branche pour travailler

Dans ton terminal, depuis le dossier `mon-premier-depot`, crée une nouvelle branche :

```
git checkout -b ___
```

> *Complète `___` avec le nom de branche `feat/ameliorer-readme`. La convention `type/description-courte` rend les branches lisibles dans la liste.*

Réponse :

```
git checkout -b feat/ameliorer-readme
```

Sortie attendue :

```
Switched to a new branch 'feat/ameliorer-readme'
```

Modifie le fichier `README.md` :

```
cat > README.md << 'EOF'
# mon-premier-depot

Un dépôt de démonstration gitrust.
```


© Gitrust

````
## Installation

Cloner le dépôt :

```bash
git clone git@gitrust.example.com:tonpseudo/mon-premier-depot.git
````

EOF

````
Commite et pousse la branche :

```bash
git add README.md
git commit -m "améliore le README - closes #1"
git push origin feat/ameliorer-readme
````

Sortie attendue :

```
[feat/ameliorer-readme 7c3d4e5] améliore le README - closes #1
 1 file changed, 10 insertions(+), 1 deletion(-)
...
To git@gitrust.example.com:tonpseudo/mon-premier-depot.git
 * [new branch]      feat/ameliorer-readme -> feat/ameliorer-readme
```

Checkpoint : la sortie de `git push` montre `[new branch] feat/ameliorer-readme`. Si elle montre `rejected`, exécute d'abord `git pull origin main --rebase`.

Étape 5 : Ouvre une pull request

Dans ton navigateur, navigue vers `/<tonpseudo>/mon-premier-depot/pulls/new`.

gitrust peut avoir détecté ta branche et proposer une bannière « Ouvrir une PR pour feat/ameliorer-readme » — clique dessus si elle apparaît.

© Gitrust

Remplis le formulaire :

- **Titre** : `Améliore le README avec description et installation`
- **Branche source** : `feat/ameliorer-readme`
- **Branche cible** : `main`
- **Description** : `Closes #1` (gitrust fermera l'issue automatiquement à la fusion)
- **Reviewers** : sélectionne ton coéquipier (s'il est disponible)

Clique **Ouvrir la pull request**.

Sortie attendue : gitrust crée la PR et te redirige vers `/<tonpseudo>/mon-premier-depot/pulls/1`.

Capture a venir

105 pr open

voir scripts/screenshot-runner/

Checkpoint : la PR est bien à l'état **Ouverte** et l'onglet **Fichiers modifiés** montre les changements dans `README.md`. Si la PR est vide (pas de diff), vérifie que tu as bien sélectionné `feat/ameliorer-readme` comme branche source.

Étape 6 : Fusionne la pull request

> *Si ton coéquipier peut review, demande-lui d'approuver la PR avant de fusionner. Dans ce tutoriel, tu peux fusionner directement si tu es seul.*

Sur la page de la PR, fais défiler jusqu'à la section **Fusion**. Sélectionne la stratégie **Merge commit** (par défaut). Clique **Fusionner la pull request**.

Sortie attendue (message dans l'interface) :

```
Pull request fusionnée avec succès.
```

La PR passe au statut **Fusionnée** (violet). Navigue vers `/<tonpseudo>/mon-premier-depot/issues/1` — l'issue doit être **Fermée** automatiquement grâce au `closes #1` dans le message de commit.

© Gitrust

Capture a venir

106 issue closed

voir scripts/screenshot-runner/

Checkpoint : l'issue #1 est au statut **Fermée**. La page principale du dépôt affiche le nouveau contenu du `README.md`. Si l'issue est encore ouverte, vérifie que le message de commit contenait exactement `closes #1` (respecter la casse n'est pas requis, mais le numéro doit être correct).

Récapitulatif

- √ **O1 accompli** : tu as créé l'équipe `equipe-alpha` via `/teams/new`, invité un membre et assigné le dépôt avec le niveau Developer
- √ **O2 accompli** : tu as ouvert l'issue #1 pour décrire la tâche, obtenant une unité de travail traçable et numérotée
- √ **O3 accompli** : tu as ouvert une PR depuis `feat/ameliorer-readme`, la fusionner a fermé l'issue automatiquement grâce à `closes #1`

© Gitrust

Et si ça ne marche pas

Symptôme	Cause probable	Correction
Le coéquipier ne voit pas le dépôt dans son tableau de bord	L'accès au dépôt n'a pas été correctement accordé à l'équipe	Sur la page de l'équipe, vérifie que le dépôt apparaît dans la section **Dépôts** avec un niveau d'accès. Reconfirme si nécessaire
La PR n'a pas de diff (onglet **Fichiers modifiés** vide)	La branche source est identique à la branche cible	Vérifie que tu as bien poussé depuis `feat/ameliorer-readme` et non depuis `main`. Exécute `git log --oneline main..feat/ameliorer-readme` pour voir les commits en attente
L'issue #1 reste ouverte après fusion	Le message de commit ne contient pas le mot-clé reconnu, ou le numéro est incorrect	Les mots-clés acceptés sont `closes`, `fixes`, `resolves` (insensibles à la casse) suivi de `#N`. Vérifie le message exact avec `git log --oneline`
`git push` refusé avec `remote: push blocked`	La branche `main` est protégée et le push direct y est interdit	C'est normal : travaille toujours sur une branche séparée et utilise une PR pour fusionner dans `main`

Prochaine étape

→ 04 — Automatiser avec la CI : écris ton premier `.gitrust-ci.yml`, déclenche un pipeline et lis le résultat (~20 min)

© Gitrust

1.2.4 Automatiser les tests avec la CI intégrée

Objectifs

À la fin de ce tutoriel, tu sauras :

- **O1. Écrire** un fichier `.gitrust-ci.yml` minimal en mode Easy pour un projet Rust ou Node
- **O2. Déclencher** un pipeline CI en poussant un commit
- **O3. Lire** le résultat d'un pipeline (statut, logs) sur la page `/{owner}/{repo}/ci`

Pré-requis

- **Technique** : dépôt cloné en local avec au moins un fichier source (tutoriel 02 complété), Git installé
- **Pédagogique** : tutoriel 03 — *Collaborer* complété
- **Temps estimé** : ~20 minutes

Vue d'ensemble

La CI (*Continuous Integration*, intégration continue) automatise l'exécution de tes tests à chaque push. Au lieu de vérifier manuellement que le code compile et que les tests passent, gitrust le fait pour toi — et signale immédiatement si quelque chose est cassé.

Le mode Easy de gitrust CI utilise un fichier `.gitrust-ci.yml` que tu places à la racine de ton dépôt. Ce fichier décrit en quelques lignes ce que gitrust doit exécuter : compiler, linter, tester. Le serveur de build reçoit le code, l'exécute dans un container Docker isolé, et renvoie les logs en temps réel.

```
flowchart LR
subgraph "Ta machine"
C[Commit +
git push]
end
subgraph "gitrust (:4000)"
W[CiWorker
Tokio task]
DB[(Pipeline
en base)]
end
subgraph "Serveur de build"
D[Docker
+ Dagger]
CI[ci-engine]
end
```


© Gitrust

```
C -->|push SSH/HTTP| W
W --> DB
W -->|rsync + SSH| D
D --> CI
CI -->|logs ligne par ligne| W
W --> DB
DB -->|SSE stream| C
```

> ***Scaffolding niveau 2** : la configuration CI comporte une section incomplète à compléter. Essaie d'abord sans regarder l'indice.*

Étape 1 : Vérifie que la CI est activée sur le dépôt

Dans ton navigateur, ouvre ton dépôt et navigue vers **Settings** (`/{owner}/{repo}/settings`).

Fais défiler jusqu'à la section **CI**. Si tu ne vois pas cette section, l'administrateur de l'instance n'a pas encore activé le CI — contacte-le.

Coche **CI activé** et **Déclencher sur push**. Clique **Enregistrer**.

Capture a venir

107 repo settings ci

voir scripts/screenshot-runner/

Checkpoint : la case **CI activé** est cochée et sauvegardée. Si la section CI n'apparaît pas dans les paramètres, la fonctionnalité n'est pas activée sur cette instance — consulte la section « Et si ça ne marche pas ».

Étape 2 : Crée le fichier `.gitrust-ci.yml`

Dans ton terminal, depuis la racine du dépôt cloné, crée le fichier de configuration CI.

© Gitrust

Pour un projet Rust :

```
# .gitrust-ci.yml
language: rust

build:
  command: "cargo build"

checks:
  lint: "cargo clippy -- -D warnings"
  format: "cargo fmt -- --check"

tests:
  command: "___"
```

> *Complète `___` avec la commande de test Rust standard. Réponse : `cargo test`.*

Pour un projet Node.js :

```
# .gitrust-ci.yml
language: node

build:
  command: "npm install"

checks:
  lint: "npm run lint"

tests:
  command: "npm test"
```

Pour ce tutoriel, utilise le fichier le plus adapté à ton projet, ou crée un exemple minimaliste avec Node si tu n'as pas de projet existant :

```
# Exemple minimal Node — crée les fichiers nécessaires
echo '{"name":"demo","scripts":{"test":"node -e \"console.log(42===42)\"","lint":"echo
```

Puis crée `.gitrust-ci.yml` avec le contenu Node ci-dessus.

© Gitrust

Checkpoint : exécute `ls -la` dans le dépôt — tu dois voir `.gitrust-ci.yml` dans la liste. La taille du fichier doit être supérieure à 0 octet.

Étape 3 : Commite et pousse pour déclencher la CI

Ajoute le fichier et crée un commit :

```
git add .gitrust-ci.yml
git commit -m "ajoute configuration CI gitrust"
git push origin main
```

Sortie attendue :

```
[main a4f7b8c] ajoute configuration CI gitrust
 1 file changed, 10 insertions(+)
 create mode 100644 .gitrust-ci.yml
...
To git@gitrust.example.com:tonpseudo/mon-premier-depot.git
   3a8c21f..a4f7b8c  main -> main
```

Immédiatement après le push, dans ton navigateur, navigue vers `/{owner}/{repo}/ci`.

Sortie attendue : une entrée de pipeline apparaît avec le statut **En attente** ou **En cours**, associée au commit `a4f7b8c` et au message `ajoute configuration CI gitrust`.

Capture a venir

108 ci pipeline running

voir scripts/screenshot-runner/

Checkpoint : le pipeline apparaît dans la liste. S'il n'apparaît pas après 10 secondes, rafraîchis la page. S'il n'apparaît toujours pas, vérifie que la CI est bien activée (étape 1) et que le fichier `.gitrust-ci.yml` est bien à la racine du dépôt (pas dans un sous-dossier).

© Gitrust

Étape 4 : Lis les logs du pipeline

Clique sur le pipeline dans la liste pour ouvrir la page de détail.

La page affiche :

- Le **statut** en cours (Pending → Running → Success ou Failure)
- Le **commit** et la **branche** associés
- Les **logs** en temps réel, mis à jour ligne par ligne

Attends la fin de l'exécution. Pour un projet minimal, cela prend 1 à 3 minutes selon la rapidité du serveur de build.

Sortie attendue pour un pipeline réussi (dernières lignes des logs) :

```
[build] cargo build: ok
[checks] cargo clippy: ok
[tests] cargo test: ok
Pipeline terminé avec succès.
```

Pour un pipeline en échec, les logs montrent la commande qui a échoué et le code de retour :

```
[tests] cargo test: FAILED (exit code 1)
error[E0277]: the trait bound `...` is not satisfied
Pipeline terminé en échec.
```

Capture a venir

109 ci pipeline success

voir scripts/screenshot-runner/

Checkpoint : le pipeline passe au statut **Réussi** (vert) ou **Échoué** (rouge). Dans les deux cas, tu as atteint l'objectif O3 — tu sais lire le résultat. Si le statut reste bloqué sur **En cours** plus de 10 minutes, consulte la section « Et si ça ne marche pas ».

© Gitrust

Récapitulatif

- √ **O1 accompli** : tu as écrit un fichier `.gitrust-ci.yml` avec les sections `language`, `build`, `checks` et `tests`, adapté à ton langage
- √ **O2 accompli** : tu as déclenché le pipeline en poussant un commit — gitrust l'a automatiquement détecté et lancé l'exécution
- √ **O3 accompli** : tu as lu le statut et les logs sur `/{owner}/{repo}/ci`, identifiant si les étapes build/checks/tests ont réussi ou échoué

Et si ça ne marche pas

Symptôme	Cause probable	Correction
Aucun pipeline n'apparaît après le push	La CI n'est pas activée sur le dépôt, ou le fichier `.gitrust-ci.yml` est absent/mal placé	Vérifie Settings → CI (case **CI activé** cochée). Vérifie que `.gitrust-ci.yml` est à la racine avec `git ls-files .gitrust-ci.yml`
Le pipeline reste en statut **En attente** indéfiniment	Le serveur de build n'est pas accessible depuis gitrust, ou `CI_MAX_CONCURRENT` est atteint	Demande à l'administrateur de vérifier la connectivité SSH vers le builder et les logs `journalctl -u gitrust`
Le pipeline échoue à l'étape `build` avec `command not found`	Le langage sélectionné n'a pas les outils installés sur le serveur de build	Vérifie avec l'administrateur que le profil du langage (`rust`, `node`) est disponible. Consulte la référence des schémas CI
Les logs s'affichent puis le pipeline reste en statut **En cours**	Timeout atteint (`CI_DEFAULT_TIMEOUT`) ou processus bloqué	Vérifie si le test boucle indéfiniment. Contacte l'administrateur pour ajuster `CI_DEFAULT_TIMEOUT`

Prochaine étape

Tu as maintenant complété le parcours utilisateur de base : compte sécurisé → code versionné → collaboration en équipe → CI automatisée.

Pour aller plus loin :

- Comment gérer les clés SSH — plusieurs machines, rotation des clés
- Référence du schéma `.gitrust-ci.yml` — toutes les options disponibles
- Comprendre le cycle de vie d'une pull request — les états et transitions d'une PR

© Gitrust

1.3 Guides pratiques

1.3.1 Comment gérer ses clés SSH

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Ajouter une clé SSH depuis une nouvelle machine
- Générer une clé plus sécurisée (ed25519 au lieu de RSA)
- Vérifier l'empreinte d'une clé avant de l'ajouter
- Supprimer une clé d'une machine perdue ou compromise
- Diagnostiquer une erreur `Permission denied (publickey)`

Pré-requis

- Un compte gitrust actif
- Un terminal avec `ssh-keygen` disponible (`ssh-keygen --version` doit répondre)

Étapes

1. Générer une clé ed25519 (recommandé)

ed25519 est l'algorithme recommandé : clés courtes, très rapides, résistantes aux attaques modernes.

```
ssh-keygen -t ed25519 -C "description-de-la-machine"
```

Exemple de description : `laptop-maison-2026` , `serveur-ci` , `poste-bureau` .

Sortie attendue :

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/toi/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Your identification has been saved in /home/toi/.ssh/id_ed25519
Your public key has been saved in /home/toi/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:xXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXx description-de-la-machine
```


© Gitrust

Une **passphrase** est fortement recommandée : elle chiffre la clé privée sur le disque. Si ta machine est volée, la clé reste inutilisable sans la passphrase.

2. Générer une clé RSA-4096 (alternative)

Si un outil ou un service impose RSA :

```
ssh-keygen -t rsa -b 4096 -C "description-de-la-machine"
```

La clé sera dans `~/.ssh/id_rsa` et `~/.ssh/id_rsa.pub`. RSA-4096 est sûr mais plus lent qu'ed25519.

3. Afficher la clé publique à copier

La **clé publique** (fichier `.pub`) est faite pour être partagée :

```
cat ~/.ssh/id_ed25519.pub
```

Sortie (une ligne) :

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx d
```

Copie cette ligne entière.

4. Ajouter la clé dans gitrust (`/settings/keys`)

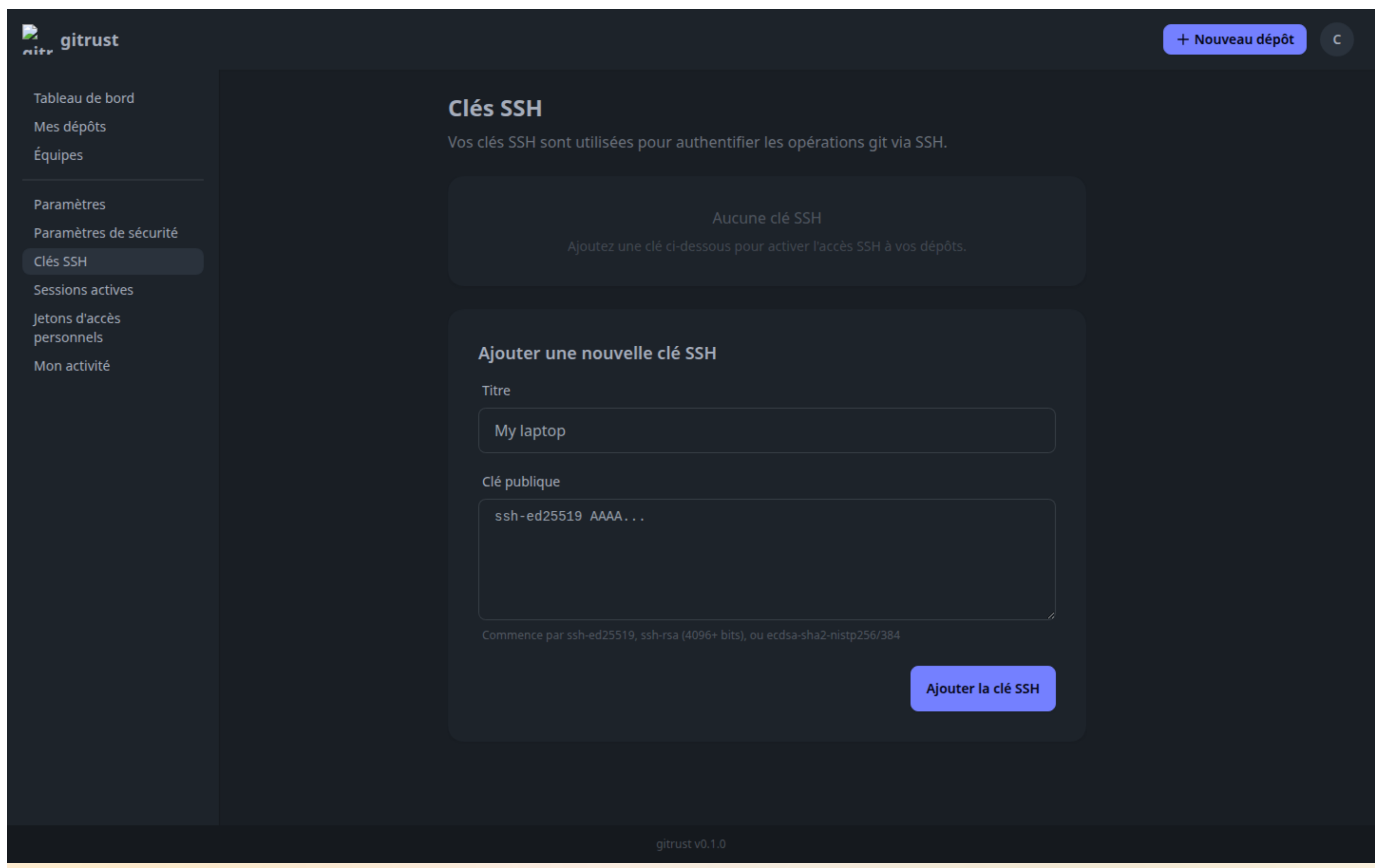
Dans ton navigateur, navigue vers **Paramètres → Clés SSH** (`/settings/keys`).

Clique **Ajouter une clé SSH** :

- **Titre** : le même que la description (ex. `laptop-maison-2026`)
- **Clé** : colle la ligne `ssh-ed25519 …` copiée

Clique **Enregistrer**.

© Gitrust



5. Lister les clés enregistrées

La page `/settings/keys` affiche toutes tes clés avec :

- Leur titre
- Leur empreinte SHA256 (pour vérification)
- La date d'ajout
- Un bouton **Supprimer**

6. Vérifier l'empreinte SHA256 d'une clé locale

Pour confirmer qu'une clé locale correspond bien à une clé enregistrée dans gitrust :

```
ssh-keygen -lf ~/.ssh/id_ed25519.pub
```

Sortie :

```
256 SHA256:xXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXxXx description-de-la-machine (ED25519
```

Compare ce SHA256 avec celui affiché dans `/settings/keys`. Ils doivent être identiques.

© Gitrust

7. Tester la connexion SSH

```
ssh -T git@gitrust.example.com -p 2222
```

Sortie attendue :

```
Bonjour tonpseudo ! Vous êtes authentifié, mais gitrust ne fournit pas d'accès shell.
```

Si tu as plusieurs clés, spécifie laquelle utiliser :

```
ssh -T -i ~/.ssh/id_ed25519 git@gitrust.example.com -p 2222
```

8. Supprimer une clé

Dans `/settings/keys`, clique **Supprimer** à côté de la clé concernée. La suppression est immédiate — toute connexion utilisant cette clé sera refusée dès la prochaine tentative.

Variantes

Plusieurs machines, plusieurs clés

Tu peux enregistrer autant de clés que nécessaire dans gitrust. Chaque machine a sa propre paire de clés — tu ne copies jamais une clé privée d'une machine à l'autre.

Pour avoir plusieurs clés sur la même machine et choisir laquelle utiliser selon le serveur, crée un fichier `~/.ssh/config` :

```
Host gitrust.example.com
    HostName gitrust.example.com
    User git
    Port 2222
    IdentityFile ~/.ssh/id_ed25519_gitrust
```

Ensuite `git clone git@gitrust.example.com:owner/repo.git` utilisera automatiquement cette clé.

Ajouter la clé à l'agent SSH (évite de saisir la passphrase à chaque push)

```
ssh-add ~/.ssh/id_ed25519
```

Sur macOS, ajoute `--apple-use-keychain` pour que la passphrase soit mémorisée dans le trousseau système.

© Gitrust

Dépannage `ssh -T`

Symptôme	Cause probable	Correction
`Permission denied (publickey)`	La clé privée locale ne correspond pas à une clé publique enregistrée, ou l'agent ne la connaît pas	Exécute `ssh-add ~/.ssh/id_ed25519`, puis réessaie. Vérifie que la bonne clé `.pub` est dans `/settings/keys`
`Connection refused`	Le port 2222 est bloqué ou le service SSH de gitrust est arrêté	Essaie le port 22 : `ssh -T git@gitrust.example.com -p 22`. Contacte l'administrateur
`Host key verification failed`	L'empreinte du serveur a changé ou est inconnue	Exécute `ssh-keyscan -p 2222 gitrust.example.com >> ~/.ssh/known_hosts`. Vérifie l'empreinte avec l'administrateur avant d'accepter
`Too many authentication failures`	L'agent SSH propose trop de clés	Spécifie explicitement : `ssh -i ~/.ssh/id_ed25519 -o IdentitiesOnly=yes -T git@… -p 2222`

Voir aussi

- Tutoriel 02 — Cloner et pousser : mise en place complète SSH + premier clone
- Comprendre la sécurité côté utilisateur : pourquoi les clés SSH sont plus sûres que les mots de passe
- Créer un Personal Access Token : alternative SSH pour les accès HTTPS et les scripts

© Gitrust

1.3.2 Comment configurer la double authentification (2FA)

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Activer la 2FA TOTP sur ton compte gitrust
- Scanner le QR code avec une nouvelle application
- Sauvegarder tes codes de secours
- Désactiver la 2FA (par exemple pour changer d'application)
- Régénérer tes codes de secours après les avoir utilisés

Pré-requis

- Un compte gitrust actif
- Une application d'authentification TOTP installée sur ton téléphone : Google Authenticator, Authy, Bitwarden Authenticator, ou tout équivalent compatible RFC 6238

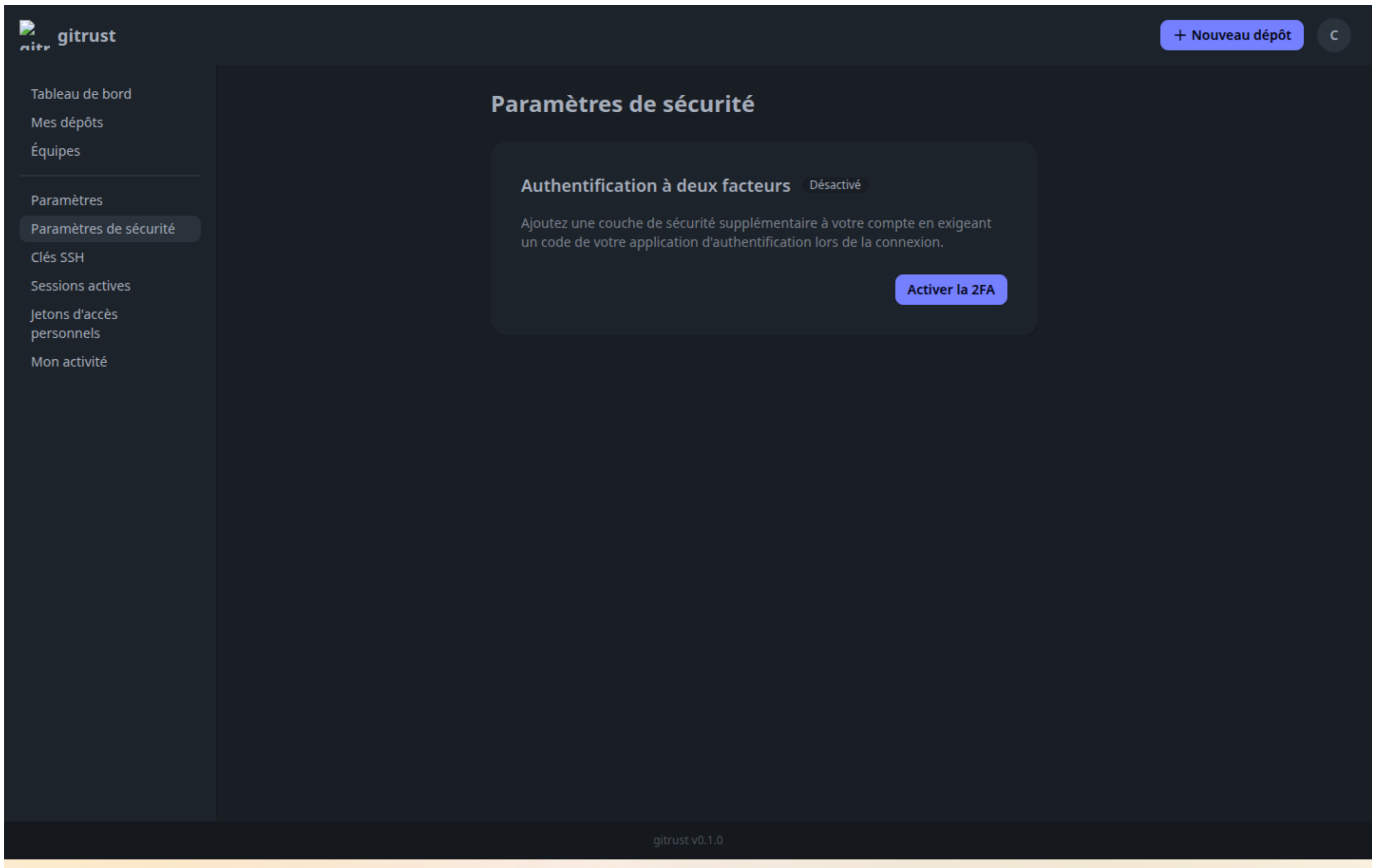
Étapes

1. Accéder à la page de sécurité

Dans ton navigateur, navigue vers **Paramètres → Sécurité** (`/settings/security`).

La section **Authentification à deux facteurs** affiche l'état actuel : « Désactivée » ou « Activée ».

© Gitrust



2. Lancer la configuration TOTP

Clique **Activer la 2FA**. gitrust génère un secret TOTP et affiche un **QR code**.

3. Scanner le QR code

Dans ton application d'authentification :

1. Ouvre l'application
2. Ajoute un nouveau compte (icône `+` ou « Ajouter un compte »)
3. Choisis **Scanner un QR code**
4. Pointe l'appareil photo vers le QR code affiché dans gitrust

L'application ajoute un compte nommé `gitrust (tonpseudo)` et affiche un code à 6 chiffres qui se renouvelle toutes les 30 secondes.

> ***Si tu ne peux pas scanner le QR code*** *(appareil photo indisponible) : clique sur « Afficher la clé secrète ». Copie la chaîne de 32 caractères et saisis-la manuellement dans ton application (option « Saisir une clé »).*

© Gitrust

4. Vérifier le code et valider

Dans gitrust, saisis le code à 6 chiffres affiché par ton application dans le champ **Code de vérification**. Clique **Confirmer**.

> *Le code est valide pendant 30 secondes. Si l'horloge de ton téléphone n'est pas synchronisée, le code peut être rejeté — active la synchronisation automatique de l'heure dans les paramètres de ton téléphone.*

Sortie attendue :

```
La 2FA a été activée avec succès.
Conservez vos codes de secours dans un endroit sûr.
```

5. Sauvegarder les codes de secours

gitrust affiche **10 codes de secours** au format `XXXX-XXXX`. Ces codes permettent de te connecter si tu perds accès à ton application d'authentification.

Actions à effectuer immédiatement :

- Copie les 10 codes
- Stocke-les dans un gestionnaire de mots de passe (Bitwarden, KeePass...) ou sur papier dans un endroit sécurisé
- Ne les stocke pas dans le même appareil que ton application 2FA

Chaque code ne peut être utilisé **qu'une seule fois**.

Variantes

Désactiver la 2FA

Sur `/settings/security`, dans la section 2FA, clique **Désactiver la 2FA**. gitrust te demande de confirmer avec ton mot de passe.

> *Après désactivation, ton compte n'est protégé que par ton mot de passe. Si ton instance force la 2FA globalement (décision administrateur), tu ne pourras pas la désactiver.*

Régénérer les codes de secours

Si tu as utilisé tous tes codes ou crains qu'ils soient compromis :

Sur `/settings/security`, clique **Régénérer les codes de secours**. Les anciens codes sont immédiatement invalidés. gitrust affiche 10 nouveaux codes — sauvegarde-les comme à l'étape 5.

© Gitrust

Changer d'application d'authentification

1. Désactive la 2FA (voir ci-dessus)
2. Installe la nouvelle application
3. Réactive la 2FA depuis le début (étape 1)

Si tu as encore accès à l'ancienne application, tu peux aussi simplement ajouter le même compte dans la nouvelle application en rescannant le QR code pendant l'activation.

Voir aussi

- Tutoriel 01 — Premiers pas : activation de la 2FA dans le contexte du parcours complet
- Comprendre la sécurité côté utilisateur : pourquoi la 2FA protège même si ton mot de passe est volé
- Créer un Personal Access Token : accès API sans code 2FA interactif

© Gitrust

1.3.3 Comment créer un Personal Access Token (PAT)

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Accéder à l'API gitrust depuis un script ou un outil CI sans exposer ton mot de passe
- Cloner un dépôt via HTTPS depuis un environnement où SSH n'est pas disponible
- Révoquer un accès accordé à un outil tiers sans changer ton mot de passe
- Comprendre quels scopes attribuer à un token selon l'usage

Pré-requis

- Un compte gitrust actif

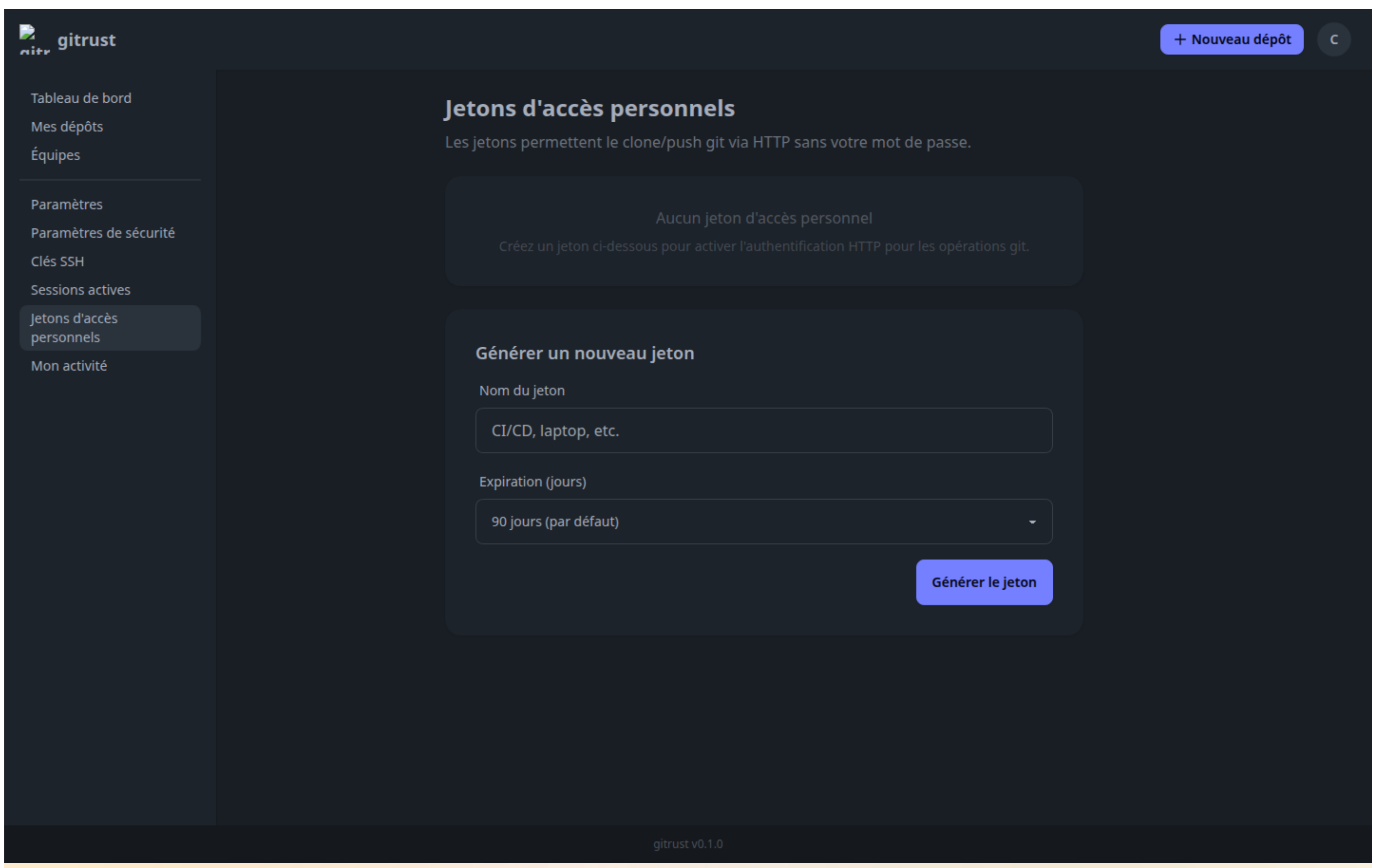
Étapes

1. Accéder à la gestion des tokens

Dans ton navigateur, navigue vers **Paramètres → Tokens d'accès** (`/settings/tokens`).

La page liste tes tokens existants avec leur nom, leur date de création et leur date d'expiration.

© Gitrust



2. Créer un nouveau token

Clique **Créer un token**. Remplis le formulaire :

- **Nom** : un nom descriptif qui rappelle l'usage (ex. `script-backup-2026`, `ci-github-actions`, `cli-perso`)
- **Expiration** : choisis une durée (30 jours, 90 jours, 1 an, ou sans expiration). Pour les scripts de CI, préfère une expiration explicite.
- **Scopes** : sélectionne uniquement les permissions nécessaires (principe du moindre privilège)

Scope	Permet
`repo:read`	Lire les dépôts, les issues, les PRs, les commits
`repo:write`	Créer des issues, commenter, pousser du code, ouvrir des PRs
`user:read`	Lire les informations de ton profil

Clique **Générer le token**.

3. Copier le token immédiatement

gitrust affiche le token **une seule fois** :

© Gitrust

```
gr_pat_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Copie-le immédiatement dans ton gestionnaire de mots de passe. Cette valeur ne sera plus jamais affichée — si tu la perds, tu devras révoquer le token et en créer un nouveau.

Utiliser le token

Avec `git clone` via HTTPS

```
git clone https://tonpseudo:gr_pat_xxxx@gitrust.example.com/owner/repo.git
```

> *Pour éviter que le token apparaisse dans l'historique du shell, stocke-le dans une variable d'environnement :*
>
> ```
> export GITRUST_TOKEN="gr_pat_xxxx"
> git clone https://tonpseudo:${GITRUST_TOKEN}@gitrust.example.com/owner/repo.git
> ```

Avec l'API REST (header `Authorization`)

```
curl -H "Authorization: Bearer gr_pat_xxxx" \
     https://gitrust.example.com/api/v1/user
```

Réponse attendue :

```
{
  "id": 42,
  "username": "tonpseudo",
  "email": "ton@email.com"
}
```

Dans un script Python

```
import urllib.request
import json

token = "gr_pat_xxxx"
req = urllib.request.Request(
```


© Gitrust

```
        "https://gitrust.example.com/api/v1/user",
        headers={"Authorization": f"Bearer {token}"}
    )
    with urllib.request.urlopen(req) as resp:
        user = json.load(resp)
        print(user["username"])
```

Révoquer un token

Dans `/settings/tokens`, clique **Révoquer** à côté du token concerné. La révocation est immédiate — toute requête utilisant ce token recevra une réponse `401 Unauthorized`.

Cas où révoquer immédiatement :

- Token exposé accidentellement dans un dépôt public ou des logs
- Machine ou outil auquel le token était destiné est compromise
- Token inutilisé depuis longtemps

Variantes

Token sans expiration

Acceptable pour les outils personnels sur une machine de confiance. À déconseiller pour les environnements partagés (CI d'entreprise, scripts déployés sur un serveur). Préfère dans ces cas un token avec expiration de 90 jours renouvelé via un secret manager.

Plusieurs tokens pour un même outil

Crée un token par environnement (local, staging, prod) avec des scopes distincts. Ainsi, compromettre le token de staging n'expose pas la production.

Voir aussi

- Référence API REST v1 : liste des endpoints et authentification JWT vs PAT
- Comprendre la sécurité côté utilisateur : comment gitrust stocke les tokens (hash, jamais en clair)
- Gérer ses clés SSH : alternative SSH pour les opérations Git

© Gitrust

1.3.4 Comment gérer les équipes et les permissions

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Créer une équipe et y regrouper des collaborateurs
- Inviter ou retirer un membre d'une équipe
- Assigner une équipe à un dépôt avec un niveau d'accès précis
- Choisir le bon rôle (Owner / Maintainer / Member) pour chaque membre
- Comprendre la différence entre les rôles d'équipe et les niveaux d'accès aux dépôts

Pré-requis

- Un compte gitrust actif
- Être propriétaire (Owner) de l'équipe ou de l'organisation pour gérer les membres et accès

---

Étapes

1. Créer une équipe

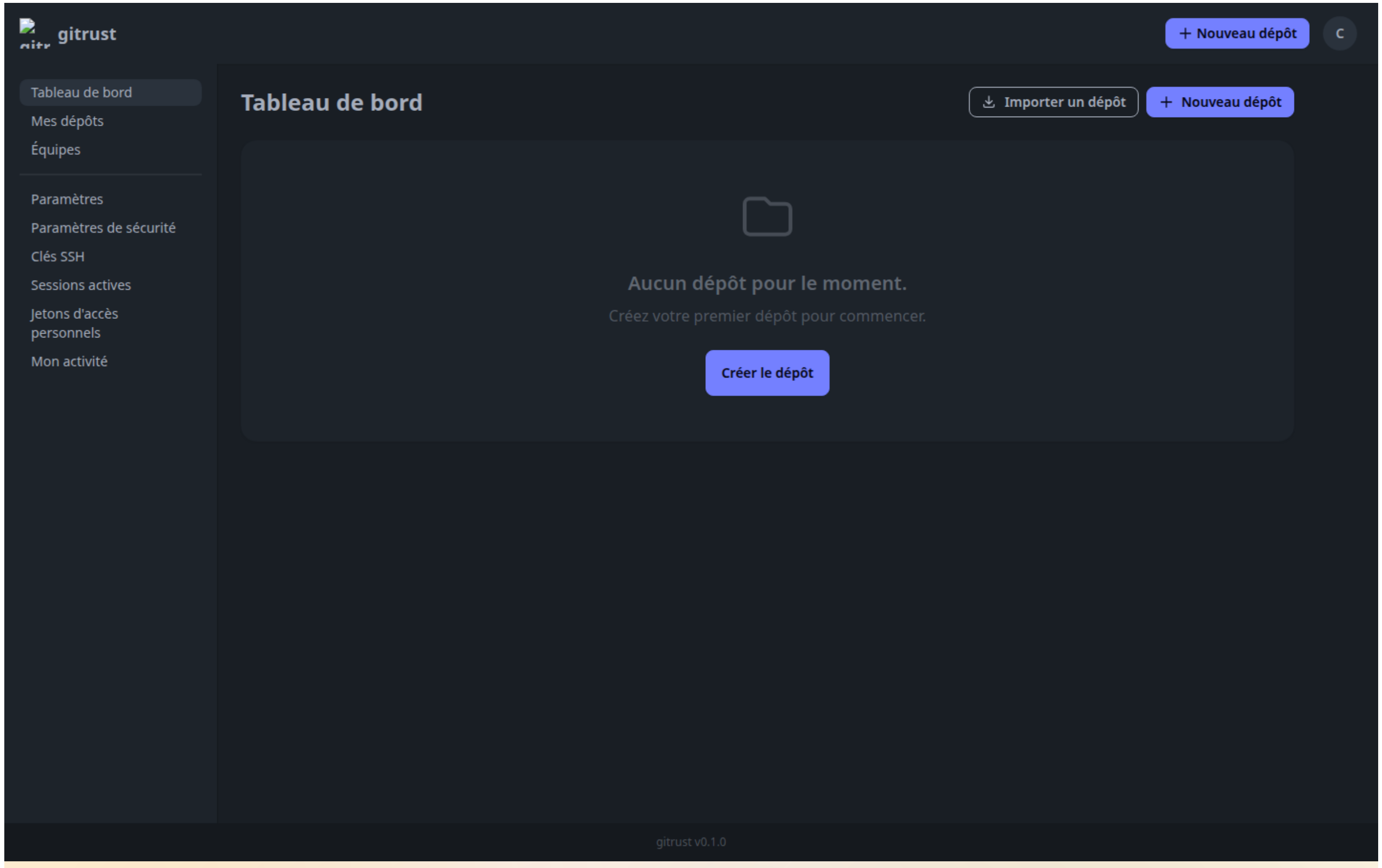
Navigue vers **Équipes → Nouvelle équipe** (`/teams/new`).

Remplis le formulaire :

- **Nom** : identifiant de l'équipe (lettres minuscules, chiffres, tirets — ex. `backend-core` , `devops-ops`)
- **Description** : optionnel, visible par les membres

Clique **Créer l'équipe**. gitrust te redirige vers `/teams/{slug}` .

© Gitrust



2. Ajouter des membres

Sur la page de l'équipe (`/teams/{slug}`), dans la section **Membres** :

1. Saisis le pseudo du collaborateur dans le champ de recherche
2. Sélectionne son compte dans la liste déroulante
3. Choisis son **rôle dans l'équipe** (voir tableau ci-dessous)
4. Clique **Inviter**

Rôle dans l'équipe	Ce que le membre peut faire
Member	Utiliser les dépôts assignés selon le niveau d'accès
Maintainer	Gérer les dépôts assignés à l'équipe
Owner	Gérer les membres, les dépôts et les paramètres de l'équipe

> *Un seul Owner par équipe est requis. Désigne un Owner de secours pour éviter le blocage en cas d'absence.*

© Gitrust

3. Assigner un dépôt à l'équipe

Sur la page de l'équipe, dans la section **Dépôts** :

1. Clique **Ajouter un dépôt**
2. Sélectionne le dépôt dans la liste
3. Choisis le **niveau d'accès au dépôt** (voir tableau ci-dessous)
4. Clique **Confirmer**

Niveau d'accès dépôt	Lire	Pousser	Ouvrir/merger PR	Gérer le dépôt
Reader	√	—	—	—
Developer	√	√ (branches)	√	—
Maintainer	√	√	√	partiel
Owner	√	√	√	√

Pour la référence complète des permissions par action, consulte le modèle de permissions.

4. Retirer un membre

Sur la page de l'équipe, dans la section **Membres**, clique **Retirer** à côté du membre concerné. Le retrait est immédiat — le membre perd l'accès aux dépôts de l'équipe.

5. Modifier le niveau d'accès d'un dépôt

Pour changer le niveau d'accès d'un dépôt déjà assigné à une équipe : retire le dépôt (bouton **Retirer** dans la section **Dépôts**), puis réassigne-le avec le nouveau niveau.

Variantes

Accès direct à un dépôt (sans équipe)

Il est possible d'ajouter un collaborateur directement sur un dépôt via `/{owner}/{repo}/collaborators`. Cette approche est adaptée aux cas ponctuels (ex. un prestataire externe sur un seul dépôt). Pour des équipes permanentes, l'approche par équipe est préférable : un seul point de gestion pour plusieurs dépôts.

Un utilisateur appartient à plusieurs équipes

Si un utilisateur appartient à plusieurs équipes qui ont des niveaux d'accès différents sur un même dépôt, c'est le **niveau le plus élevé** qui s'applique. Par exemple : Team A avec Reader + Team B avec Developer → l'utilisateur a Developer sur ce dépôt.

© Gitrust

Voir aussi

- Tutoriel 03 — Collaborer : mise en pratique complète avec une équipe réelle
- Modèle de permissions (référence) : matrice complète des 4 rôles × actions
- Ouvrir une pull request : prérequis d'accès pour ouvrir et fusionner une PR

© Gitrust

1.3.5 Comment ouvrir une pull request

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Proposer des modifications de code pour review avant fusion
- Lier une PR à une issue pour la fermer automatiquement à la fusion
- Assigner des reviewers à une PR
- Vérifier que la CI est verte avant de fusionner

Pré-requis

- Un dépôt avec au moins deux branches (ta branche de travail + `main` ou autre branche cible)
- Le niveau d'accès **Developer** minimum sur le dépôt

Étapes

1. Pousser ta branche de travail

Avant d'ouvrir une PR, ta branche doit être poussée sur gitrust :

```
git push origin ma-branche
```

Sortie attendue :

```
To git@gitrust.example.com:owner/repo.git
 * [new branch]      ma-branche -> ma-branche
```

2. Ouvrir le formulaire de nouvelle PR

Navigue vers `/{owner}/{repo}/pulls/new`.

gitrust peut aussi afficher une bannière de suggestion au sommet de la page du dépôt juste après un push récent : « Ouvrir une pull request pour `ma-branche` ». Clique dessus si elle apparaît.

© Gitrust

Capture a venir

114 new pr form

voir scripts/screenshot-runner/

3. Remplir le formulaire

Champ	Valeur recommandée
Titre	Phrase courte à l'impératif : `Ajoute la validation email`, `Corrige le bug #42`
Branche source	Ta branche de travail (`ma-branche`)
Branche cible	`main` (ou la branche de destination souhaitée)
Description	Contexte, lien vers l'issue (`Closes #N`), captures d'écran si pertinent
Reviewers	Pseudo(s) des coéquipiers à notifier
Labels	Optionnel — pour catégoriser (voir Utiliser les labels)

Mot-clé `Closes #N` : si tu écris `Closes #3` dans la description ou dans un message de commit sur cette branche, gitrust fermera automatiquement l'issue #3 lors de la fusion.

Clique **Ouvrir la pull request**.

4. Vérifier les onglets de la PR

Une fois créée, la PR s'affiche sur `/{owner}/{repo}/pulls/{num}` avec plusieurs onglets :

- **Conversation** : fil de discussion, commentaires généraux
- **Commits** : liste des commits inclus dans la PR
- **Fichiers modifiés** : diff complet, ligne par ligne — c'est ici que les reviewers commentent
- **CI** : statut du pipeline CI (si configuré)

© Gitrust

Capture a venir

115 pr tabs

voir scripts/screenshot-runner/

5. Suivre le statut CI

Si le dépôt a une configuration CI (`.gitrust-ci.yml`), gitrust exécute automatiquement le pipeline sur la branche de la PR. Le statut apparaît en bas de la PR :

- **En attente** / **En cours** : le pipeline tourne
- **Réussi** (vert) : tous les checks passent
- **Échoué** (rouge) : au moins un check a échoué — clique sur « Voir les détails » pour lire les logs

6. Répondre à une review

Quand un reviewer laisse des commentaires :

1. Pousse de nouveaux commits sur la même branche pour adresser les remarques
2. Réponds aux commentaires dans l'interface pour indiquer ce qui a été fait
3. Le reviewer peut alors **approuver** la PR

Variantes

PR en mode brouillon

Si ton code n'est pas encore prêt pour review, ouvre la PR en cochant **Brouillon** lors de la création. Une PR en brouillon ne peut pas être fusionnée. Elle est visible par l'équipe pour discussion préalable. Passe-la en « Prête pour review » quand tu es prêt.

Mettre à jour la branche avant fusion

Si des commits ont été ajoutés sur `main` depuis que tu as créé ta branche, rebaser avant de fusionner :

```
git fetch origin
git rebase origin/main
git push origin ma-branche --force-with-lease
```


© Gitrust

Voir aussi

- Tutoriel 03 — Collaborer : workflow complet issue → branche → PR → fusion
- Stratégies de fusion : choisir entre fast-forward, squash et merge commit
- Cycle de vie d'une pull request : comprendre les états et transitions

© Gitrust

1.3.6 Comment choisir une stratégie de fusion

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Comprendre la différence entre fast-forward, squash et merge commit
- Choisir la stratégie adaptée à ton projet ou à ta PR
- Savoir quelles permissions sont requises pour chaque stratégie

Pré-requis

- Une pull request ouverte sur le dépôt
- Le niveau d'accès **Developer** minimum (pour fusionner une PR dont tu es l'auteur) ou **Maintainer** (pour fusionner les PRs des autres)

Les trois stratégies

Fast-forward (`--ff-only`)

Ce que ça fait : déplace simplement le pointeur de la branche cible vers le dernier commit de la branche source. Aucun commit de merge n'est créé. L'historique reste parfaitement linéaire.

```
Avant :   main: A → B
          feat:       B → C → D

Après :   main: A → B → C → D
```

Quand l'utiliser : - Branches courtes avec peu de commits - Tu veux un historique linéaire lisible avec `git log` - La branche n'a pas divergé de `main` (pas de commits sur `main` pendant que tu travaillais)

Limitation : impossible si la branche cible a avancé depuis la création de la branche source — il faut d'abord rebaser.

Squash

Ce que ça fait : regroupe tous les commits de la branche en **un seul commit** sur la branche cible. L'historique de la branche est condensé.

```
Avant :   main: A → B
          feat:       B → C → D → E (3 commits)
```


© Gitrust

```
Après :   main: A → B → F
          (F = C+D+E fusionnés en un seul commit)
```

Quand l'utiliser : - Branche avec de nombreux micro-commits de type `fix typo`, `wip`, `encore un fix` - Tu veux que `main` ne contienne que des commits propres et signifiants - La PR correspond à une seule unité logique de travail

Limitation : les commits individuels sont perdus pour `main` — difficile de retrouver quel commit précis a introduit un changement avec `git bisect`.

Merge commit (`--no-ff`)

Ce que ça fait : crée un **commit de merge** explicite qui unit les deux branches. L'historique conserve la trace de la branche et de quand elle a été intégrée.

```
Avant :   main: A → B
          feat:       B → C → D

Après :   main: A → B → C → D → M
                            ↑___↑
                    (M = commit de merge)
```

Quand l'utiliser : - Branches longues avec historique riche qu'on veut conserver - Tu veux pouvoir `git revert` l'ensemble de la feature d'un seul coup (revert du commit M) - Les conventions de l'équipe exigent un historique non-rebasé

Limitation : l'historique du projet devient un graphe orienté (DAG) moins lisible avec `git log --oneline`.

© Gitrust

Tableau de comparaison

Critère	Fast-forward	Squash	Merge commit
Historique linéaire	√	√	—
Conserve les commits individuels	√	—	√
Commit de merge visible	—	—	√
`git revert` d'une feature entière	difficile	par commit	√ (revert M)
`git bisect` précis	√	difficile	√
Idéal pour	petites branches propres	branches bruyantes	features longues

Étapes pour fusionner dans gitrust

Sur la page de la PR (`/{owner}/{repo}/pulls/{num}`), fais défiler jusqu'à la section **Fusion** :

1. Sélectionne la stratégie dans le menu déroulant (**Merge commit**, **Squash**, ou **Fast-forward**)
2. Si fast-forward est choisi mais impossible (branche divergente), gitrust indique l'erreur — rebaser d'abord
3. Clique **Fusionner la pull request**
4. Confirme si demandé

Sortie attendue :

```
Pull request fusionnée avec succès.
```

Variantes

Fusionner via l'API

```
curl -X POST \
  -H "Authorization: Bearer gr_pat_xxxx" \
  -H "Content-Type: application/json" \
```


© Gitrust

```
    -d '{"merge_method": "squash"}' \
    https://gitrust.example.com/api/v1/repos/owner/repo/pulls/1/merge
```

Valeurs valides pour `merge_method` : `merge`, `squash`, `rebase` (fast-forward).

Stratégie par défaut de l'équipe

Conventionner une stratégie par dépôt réduit les décisions ad hoc. Les équipes optant pour trunk-based development préfèrent le squash. Les équipes avec des branches longues (GitFlow) préfèrent le merge commit.

Voir aussi

- Ouvrir une pull request : créer la PR avant de la fusionner
- Cycle de vie d'une pull request : comprendre les transitions d'états
- Modèle de collaboration : philosophie gitrust sur les petites PRs

© Gitrust

1.3.7 Comment utiliser les labels et les issues

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Créer et organiser des labels sur un dépôt
- Assigner des labels à une issue pour la catégoriser
- Filtrer les issues par label
- Fermer automatiquement une issue depuis un message de commit ou une PR
- Comprendre le système de labels à deux niveaux de gitrust

Pré-requis

- Un dépôt gitrust avec le niveau d'accès **Developer** minimum

---

Comprendre les labels à deux niveaux

gitrust organise les labels en **deux niveaux** : classification et sujet.

- **Niveau 1 — Classification** : la nature du ticket (`bug` , `feature` , `docs` , `chore`)
- **Niveau 2 — Sujet** : la zone fonctionnelle concernée (`auth` , `ci` , `ui` , `api`)

Combiner les deux niveaux donne une issue catégorisée précisément : une issue avec `bug` + `auth` est un bug dans le module d'authentification. Le filtrage par combinaison de labels permet de retrouver rapidement les issues pertinentes.

Étapes

1. Accéder à la gestion des labels

Navigue vers `/{owner}/{repo}/labels` .

La page liste les labels existants avec leur couleur et leur description.

© Gitrust

Capture a venir

116 labels page

voir scripts/screenshot-runner/

2. Créer un label

Sur la page des labels, remplis le formulaire en haut :

- **Nom** : court, en minuscules, sans espaces (ex. `bug`, `feature`, `auth`, `ci`)
- **Couleur** : choisis une couleur hexadécimale ou utilise le sélecteur
- **Description** : optionnel — explique en une phrase quand appliquer ce label

Clique **Créer le label**.

Labels recommandés pour démarrer :

Label	Niveau	Couleur suggérée	Usage
`bug`	Classification	`#d73a4a` (rouge)	Comportement inattendu
`feature`	Classification	`#0075ca` (bleu)	Nouvelle fonctionnalité
`docs`	Classification	`#0052cc` (bleu foncé)	Documentation uniquement
`chore`	Classification	`#e4e669` (jaune)	Maintenance, dépendances
`auth`	Sujet	`#7057ff` (violet)	Module authentification
`ci`	Sujet	`#008672` (vert)	Pipeline CI
`api`	Sujet	`#e99695` (rose)	API REST

3. Assigner des labels à une issue

Sur la page d'une issue (`/{owner}/{repo}/issues/{num}`), dans la barre latérale droite, clique sur **Labels**.

Sélectionne un ou plusieurs labels dans la liste déroulante. La sélection est appliquée immédiatement.

© Gitrust

Ou depuis l'API :

```
curl -X POST \
  -H "Authorization: Bearer gr_pat_xxxx" \
  -H "Content-Type: application/json" \
  -d '{"label_ids": [1, 3]}' \
  https://gitrust.example.com/api/v1/repos/owner/repo/issues/5/labels
```

4. Filtrer les issues par label

Sur la page des issues (`/{owner}/{repo}/issues`), clique sur le menu **Labels** dans la barre de filtres. Sélectionne un ou plusieurs labels — seules les issues portant ces labels s'affichent.

Pour combiner des filtres (ex. toutes les issues `bug` dans la zone `auth`) : sélectionne les deux labels dans le même filtre.

5. Fermer une issue automatiquement

Pour fermer automatiquement une issue lors de la fusion d'une PR ou d'un push sur `main` , utilise un **mot-clé de fermeture** dans :

- Le **message de commit** (si poussé directement sur `main`)
- La **description de la PR** (fermeture effective à la fusion)
- Un **commentaire de commit** dans la branche

Mots-clés acceptés (insensibles à la casse) :

Mot-clé	Exemple
`closes`	`closes #5`
`fixes`	`fixes #5`
`resolves`	`resolves #5`

Exemple de message de commit :

```
git commit -m "corrige la validation email - fixes #5"
```

Après fusion de la PR contenant ce commit, l'issue #5 passe automatiquement au statut **Fermée**.

© Gitrust

Variantes

Modifier ou supprimer un label

Sur `/{owner}/{repo}/labels`, chaque label a un bouton **Modifier** (pour changer le nom, la couleur ou la description) et un bouton **Supprimer**. La suppression retire le label de toutes les issues auxquelles il était assigné.

Labels suggérés automatiquement

gitrust peut suggérer des labels en fonction du contenu d'une issue. Cette suggestion s'affiche lors de la création d'une issue. Accepte ou ignore selon le contexte.

Voir aussi

- Ouvrir une pull request : utiliser `closes #N` dans la description de PR
- Tutoriel 03 — Collaborer : ouverture d'issue dans le contexte d'un workflow complet
- Syntaxe Markdown dans les issues : formatage des descriptions d'issues

© Gitrust

1.3.8 Comment importer un dépôt externe

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Migrer un dépôt depuis GitHub, GitLab ou tout autre forge Git vers gitrust
- Copier un dépôt public accessible en HTTPS sans avoir à le cloner/pousser manuellement
- Suivre la progression d'un import en cours
- Diagnostiquer un import qui échoue (timeout, dépôt trop volumineux)

Pré-requis

- Un compte gitrust actif
- L'URL HTTPS publique du dépôt source (ex. `https://github.com/owner/repo.git`)
- Le dépôt source accessible sans authentification (ou les credentials si privé)

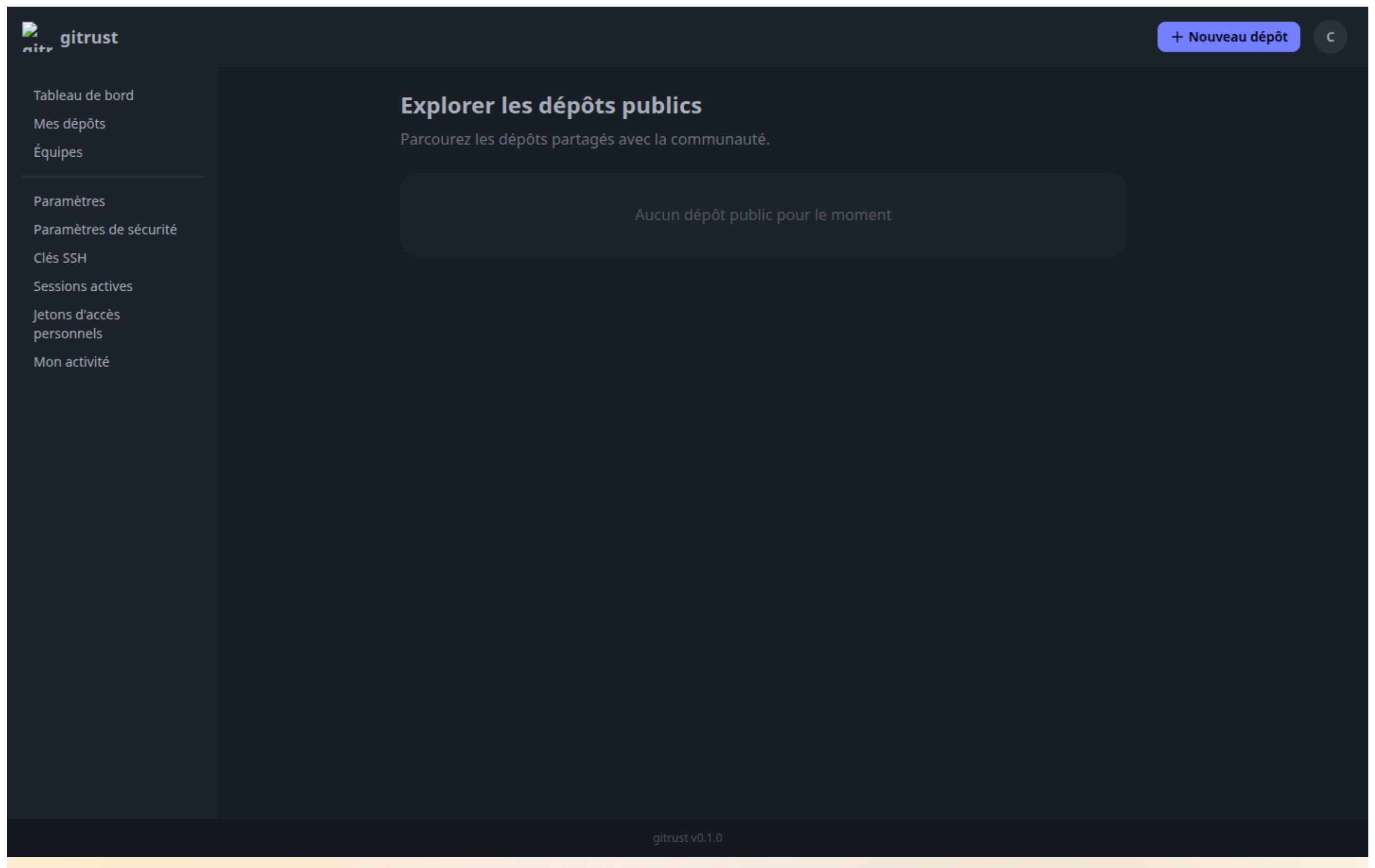
Étapes

1. Ouvrir le formulaire d'import

Navigue vers `/import` dans ton navigateur.

Le formulaire d'import apparaît.

© Gitrust



2. Remplir le formulaire

- **URL du dépôt source** : l'URL HTTPS du dépôt à importer (ex. `https://github.com/torvalds/linux.git`)
- **Nom du dépôt cible** : le nom que le dépôt aura dans gitrust (rempli automatiquement depuis l'URL, modifiable)
- **Visibilité** : Public ou Privé
- **Credentials** (si dépôt privé) : nom d'utilisateur + mot de passe ou token du service source

Clique **Lancer l'import**.

3. Suivre la progression

gitrust crée un **job d'import asynchrone** et te redirige vers `/imports/{id}`.

La page affiche la progression en temps réel via SSE (*Server-Sent Events*) :

```
Connexion au dépôt source…
Récupération des objets : 100% (1234/1234)
Résolution des deltas : 100% (456/456)
Import terminé avec succès.
```


© Gitrust

Les statuts possibles :

Statut	Signification
En attente	Job en file d'attente
En cours	Clonage en cours, logs en direct
Réussi	Dépôt disponible sur gitrust
Échoué	Voir les logs pour la cause
Annulé	Import annulé manuellement

4. Accéder au dépôt importé

Une fois le statut **Réussi**, clique sur le lien vers le dépôt dans l'interface. Le dépôt est disponible à `/{tonpseudo}/{nom-depot}` avec tout l'historique Git préservé.

5. Annuler un import en cours

Si l'import prend trop longtemps ou a été lancé par erreur, clique **Annuler** sur la page `/imports/{id}`.

Variantes

Dépôt privé avec token

Pour importer un dépôt privé depuis GitHub, utilise un token GitHub dans le champ **Credentials** :

- **Nom d'utilisateur** : ton pseudo GitHub
- **Mot de passe** : ton Personal Access Token GitHub (pas ton mot de passe GitHub)

Le token n'est jamais stocké par gitrust après la fin de l'import.

Importer un dépôt volumineux

gitrust impose un timeout par défaut sur les imports. Pour les dépôts très volumineux (plusieurs Go), l'administrateur peut ajuster `IMPORT_TIMEOUT` dans la configuration de l'instance. Si l'import échoue systématiquement par timeout, cloner manuellement et pousser directement sur gitrust est une alternative :

```
git clone --mirror https://github.com/owner/repo.git repo.git
cd repo.git
git remote add gitrust git@gitrust.example.com:tonpseudo/repo.git
git push gitrust --mirror
```


© Gitrust

Erreurs courantes

Symptôme	Cause probable	Correction
`Connexion refusée` ou `timeout`	URL inaccessible depuis le serveur gitrust (firewall, dépôt supprimé)	Vérifie que l'URL est accessible depuis un navigateur. Signale à l'administrateur si le serveur gitrust est derrière un proxy
`Authentification échouée`	Credentials incorrects ou token expiré	Vérifie le token source. Pour GitHub, génère un nouveau PAT avec le scope `repo`
`Dépôt trop volumineux`	Dépassement de la taille limite configurée	Contacte l'administrateur pour ajuster la limite, ou utilise l'import miroir manuel décrit ci-dessus
L'import reste en statut **En attente** longtemps	File d'attente saturée ou worker d'import non démarré	Rafraîchis la page. Si le problème persiste, contacte l'administrateur

Voir aussi

- Tutoriel 02 — Cloner et pousser : pousser manuellement du code sur un dépôt gitrust
- Modèle de permissions : droits nécessaires pour créer un dépôt

© Gitrust

1.3.9 Comment consulter le SBOM d'un dépôt

Quand utiliser ce guide

Utilise ce guide quand tu veux :

- Voir les composants logiciels détectés dans un dépôt (SBOM)
- Consulter les vulnérabilités identifiées par Dependency-Track
- Télécharger le fichier SBOM au format SPDX ou CycloneDX
- Comprendre ce qu'affiche l'onglet **Security** d'un dépôt

Pré-requis

- Un dépôt gitrust avec au moins un push récent
- La fonctionnalité SBOM activée par l'administrateur de l'instance (`CI_SBOM_ENABLED=true`)
- Le niveau d'accès **Reader** minimum sur le dépôt

Étapes

1. Accéder à l'onglet Security du dépôt

Navigue vers `/{owner}/{repo}/security` .

La page **Security** s'affiche avec l'encart SBOM.

Capture a venir

118 repo security

voir scripts/screenshot-runner/

© Gitrust

2. Lire les informations du SBOM

L'encart SBOM affiche pour le dernier push analysé :

Information	Signification
Statut	`success` , `pending` , `processing` , `failed`
Composants détectés	Nombre de dépendances identifiées par Syft
Commit analysé	SHA du dernier commit scanné
Hash SHA256 du BOM	Empreinte du fichier SBOM généré (traçabilité)
Dernière analyse	Date et heure du scan

3. Consulter les vulnérabilités (si Dependency-Track est configuré)

Si l'administrateur a configuré Dependency-Track, l'encart affiche en plus les compteurs par sévérité :

Sévérité	Signification
Critical	Vulnérabilité critique, exploitation active connue
High	Vulnérabilité haute, correctif disponible
Medium	Impact modéré
Low	Impact faible

Un lien **Voir dans Dependency-Track** permet d'accéder au détail de chaque vulnérabilité (CVE, description, composant affecté, version corrigée).

4. Télécharger le fichier SBOM

Si le SBOM a été généré avec succès, un bouton **Télécharger le SBOM** est disponible dans l'encart. Le fichier téléchargé est au format **CycloneDX JSON** — le format standard utilisé par gitrust pour générer et transmettre les SBOM.

Variantes

Statut `processing` qui ne se résout pas

Si le statut reste `processing` longtemps, cela signifie que Dependency-Track n'a pas terminé son analyse dans le délai imparti (30 secondes). L'analyse continue en arrière-plan sur Dependency-Track. Pour voir les résultats définitifs, consulte directement l'interface Dependency-Track en cliquant sur le lien de l'encart.

© Gitrust

Pas de données SBOM sur un dépôt

Deux raisons possibles :

1. **Aucun push depuis l'activation du SBOM** : le scan ne se déclenche qu'à chaque push. Pousse un commit pour déclencher le premier scan.
2. **Fonctionnalité non activée** : l'administrateur n'a pas activé `CI_SBOM_ENABLED`. Contacte-le.

Voir aussi

- Tutoriel 04 — Automatiser avec la CI : la CI et le SBOM sont deux fonctionnalités distinctes déclenchées par le même push
- Comprendre la sécurité côté utilisateur : pourquoi surveiller les dépendances

© Gitrust

1.4 Référence

1.4.1 Modèle de permissions — 4 rôles par dépôt

Cette page est une référence factuelle. Pour comprendre comment assigner ces rôles, consulte Gérer les équipes et les permissions.

Les 4 rôles de dépôt

Rôle	Description
Reader	Lecture seule — consulte le code, les issues et les PRs
Developer	Contribue — pousse sur les branches non protégées, ouvre et commente des PRs
Maintainer	Administre partiellement — fusionne les PRs, gère les branches protégées, les labels
Owner	Contrôle total — paramètres du dépôt, visibilité, suppression, gestion des accès

© Gitrust

Matrice des actions × rôles

© Gitrust

Action	Reader	Developer	Maintainer	Owner
Lire le code (branches, commits, blobs)	√	√	√	√
Lire les issues et les PRs	√	√	√	√
Cloner via SSH ou HTTPS	√	√	√	√
Ouvrir une issue	—	√	√	√
Commenter une issue / une PR	—	√	√	√
Fermer / rouvrir une issue	—	√	√	√
Pousser sur une branche non protégée	—	√	√	√
Ouvrir une pull request	—	√	√	√
Fusionner une pull request (propre)	—	√	√	√
Fusionner la PR d'un autre contributeur	—	—	√	√
Pousser sur une branche protégée (`main`)	—	—	√	√
Créer / supprimer des labels	—	—	√	√
Créer / supprimer des branches	—	√	√	√
Forcer le push (`--force`)	—	—	—	√
Modifier les paramètres du dépôt	—	—	—	√
Changer la visibilité (public/privé)	—	—	—	√
Gérer les collaborateurs et les équipes	—	—	—	√
Supprimer le dépôt	—	—	—	√
Activer / désactiver la CI	—	—	—	√
Lire les pipelines CI	√	√	√	√
Déclencher un pipeline manuellement	—	√	√	√
Lire le SBOM (onglet Security)	√	√	√	√

© Gitrust

Héritage utilisateur × équipe

Un utilisateur peut avoir un accès à un dépôt par deux voies :

1. **Accès direct** : ajouté comme collaborateur sur `/{owner}/{repo}/collaborators` avec un niveau explicite
2. **Via une équipe** : membre d'une équipe à laquelle le dépôt a été assigné avec un niveau

Règle de résolution : le niveau d'accès effectif est le **plus élevé** parmi tous les accès (directs + équipes). Il n'y a pas de rôle négatif (interdit explicite) dans gitrust.

Exemple :

Source	Niveau
Équipe `frontend`	Developer
Équipe `leads`	Maintainer
Accès direct	Reader
Effectif	**Maintainer**

Rôles d'équipe vs niveaux d'accès dépôt

Ne pas confondre les deux systèmes :

Système	Valeurs	Portée
Rôle dans l'équipe	Member / Maintainer / Owner	Qui peut gérer l'équipe elle-même
Niveau d'accès dépôt	Reader / Developer / Maintainer / Owner	Ce que les membres de l'équipe peuvent faire sur un dépôt

Un membre avec le rôle `Member` dans une équipe peut avoir le niveau `Owner` sur un dépôt — ces deux dimensions sont indépendantes.

Voir aussi

- [Gérer les équipes et les permissions](#) : appliquer ces rôles dans l'interface
- [Comprendre le modèle de collaboration](#) : philosophie derrière les revues obligatoires

© Gitrust

1.4.2 API REST v1

Référence des endpoints publics de l'API gitrust v1. Pour tester interactivement, accède à l'interface Swagger de ton instance sur `/api/docs`.

Authentification

L'API accepte deux méthodes d'authentification :

JWT Bearer (session web)

Obtenu via `POST /api/v1/auth/login`. Durée de vie courte, renouvelable via `/api/v1/auth/refresh`.

```
Authorization: Bearer <jwt_token>
```

Personal Access Token (PAT)

Créé dans `/settings/tokens`. Durée de vie configurable, révocable à tout moment.

```
Authorization: Bearer gr_pat_xxxx...
```

Les deux méthodes utilisent le même header `Authorization: Bearer`. gitrust détecte automatiquement le type par le préfixe du token.

Pagination

Tous les endpoints de liste supportent la pagination par curseur de page :

Paramètre	Type	Défaut	Description
`page`	int	`1`	Numéro de page (commence à 1)
`limit`	int	`20`	Résultats par page (max `100`)

Exemple :

```
GET /api/v1/repos/owner/repo/issues?page=2&limit=50
```


© Gitrust

Les réponses paginées incluent les headers :

```
X-Total-Count: 142
X-Page: 2
X-Per-Page: 50
```

Codes de réponse

Code	Signification
`200 OK`	Succès
`201 Created`	Ressource créée
`204 No Content`	Succès sans corps de réponse
`400 Bad Request`	Paramètres invalides
`401 Unauthorized`	Token absent ou expiré
`403 Forbidden`	Authentifié mais permissions insuffisantes
`404 Not Found`	Ressource introuvable ou non accessible
`422 Unprocessable Entity`	Validation échouée (corps JSON avec détails)
`429 Too Many Requests`	Rate limit atteint

Les erreurs retournent un corps JSON :

```
{
  "error": "repository not found",
  "code": "REPO_NOT_FOUND"
}
```


© Gitrust

Endpoints — Authentification

Méthode	Endpoint	Description
`POST`	`/api/v1/auth/login`	Connexion email + password (+ code 2FA si activé)
`POST`	`/api/v1/auth/register`	Créer un compte
`POST`	`/api/v1/auth/refresh`	Renouveler le JWT
`POST`	`/api/v1/auth/logout`	Invalider la session
`GET`	`/api/v1/auth/me`	Infos de l'utilisateur connecté (alias de `/api/v1/user`)
`POST`	`/api/v1/auth/forgot-password`	Envoyer le lien de réinitialisation
`POST`	`/api/v1/auth/reset-password`	Changer le mot de passe via token email
`POST`	`/api/v1/auth/2fa/verify`	Vérifier le code TOTP lors de la connexion

Endpoints — Utilisateur

Méthode	Endpoint	Description
`GET`	`/api/v1/user`	Profil de l'utilisateur authentifié
`GET`	`/api/v1/user/repos`	Dépôts accessibles par l'utilisateur authentifié

Exemple de réponse `GET /api/v1/user` :

```
{
  "id": 42,
  "username": "tonpseudo",
  "email": "ton@email.com",
  "avatar_url": "https://gitrust.example.com/avatars/42.png",
  "created_at": "2026-01-15T10:30:00Z"
}
```


© Gitrust

Endpoints — Dépôts

Méthode	Endpoint	Description
`GET`	`/api/v1/repos/{owner}/{repo}`	Détail d'un dépôt
`GET`	`/api/v1/repos/{owner}/{repo}/branches`	Liste des branches
`GET`	`/api/v1/repos/{owner}/{repo}/tags`	Liste des tags
`GET`	`/api/v1/repos/{owner}/{repo}/commits`	Liste des commits (paginée)

Exemple de réponse `GET /api/v1/repos/owner/repo` :

```
{
  "id": 7,
  "owner": "owner",
  "name": "repo",
  "full_name": "owner/repo",
  "description": "Mon dépôt",
  "private": false,
  "default_branch": "main",
  "stars_count": 3,
  "forks_count": 0,
  "created_at": "2026-02-01T09:00:00Z",
  "updated_at": "2026-04-15T14:22:00Z"
}
```


© Gitrust

Endpoints — Issues

Méthode	Endpoint	Description
`GET`	`/api/v1/repos/{owner}/{repo}/issues`	Lister les issues (filtres : `state`, `labels`, `page`, `limit`)
`POST`	`/api/v1/repos/{owner}/{repo}/issues`	Créer une issue
`GET`	`/api/v1/repos/{owner}/{repo}/issues/{num}`	Détail d'une issue
`PATCH`	`/api/v1/repos/{owner}/{repo}/issues/{num}`	Modifier une issue
`POST`	`/api/v1/repos/{owner}/{repo}/issues/{num}/comments`	Ajouter un commentaire

Exemple `POST /api/v1/repos/owner/repo/issues` :

```
{
  "title": "Bug: validation email incorrecte",
  "body": "La validation rejette les adresses avec un + dans le local-part."
}
```

Endpoints — Pull Requests

Méthode	Endpoint	Description
`GET`	`/api/v1/repos/{owner}/{repo}/pulls`	Lister les PRs
`POST`	`/api/v1/repos/{owner}/{repo}/pulls`	Créer une PR
`GET`	`/api/v1/repos/{owner}/{repo}/pulls/{num}`	Détail d'une PR
`POST`	`/api/v1/repos/{owner}/{repo}/pulls/{num}/merge`	Fusionner une PR
`POST`	`/api/v1/repos/{owner}/{repo}/pulls/{num}/comments`	Commenter une PR

Exemple `POST /api/v1/repos/owner/repo/pulls/{num}/merge` :

© Gitrust

```
{
  "merge_method": "squash",
  "commit_title": "feat: améliore la validation email (#5)"
}
```

Valeurs `merge_method` : `merge` (merge commit), `squash`, `rebase` (fast-forward).

Endpoints — CI

Méthode	Endpoint	Description
`GET`	`/api/v1/repos/{owner}/{repo}/ci/pipelines`	Lister les pipelines
`POST`	`/api/v1/repos/{owner}/{repo}/ci/pipelines/trigger`	Déclencher un pipeline manuellement
`GET`	`/api/v1/repos/{owner}/{repo}/ci/pipelines/{id}`	Détail d'un pipeline
`GET`	`/api/v1/repos/{owner}/{repo}/ci/pipelines/{id}/logs`	Logs d'un pipeline
`GET`	`/api/v1/repos/{owner}/{repo}/ci/config`	Lire la config CI
`PUT`	`/api/v1/repos/{owner}/{repo}/ci/config`	Mettre à jour la config CI
`GET`	`/api/v1/repos/{owner}/{repo}/ci/variables`	Lister les variables CI
`POST`	`/api/v1/repos/{owner}/{repo}/ci/variables`	Créer une variable CI
`DELETE`	`/api/v1/repos/{owner}/{repo}/ci/variables/{id}`	Supprimer une variable CI

Documentation interactive

L'interface Swagger complète est disponible sur ton instance à `/api/docs`. Elle permet de tester chaque endpoint directement depuis le navigateur avec ton token d'authentification.

La spécification OpenAPI brute (YAML) est disponible sur `/api/docs/openapi.yaml`.

© Gitrust

Voir aussi

- Créer un Personal Access Token : obtenir les credentials pour l'API
- Référence des notifications : événements SSE disponibles

© Gitrust

1.4.3 Schéma du fichier `.gitrust-ci.yml` (mode Easy)

Référence complète du fichier de configuration CI pour le **mode Easy** de gitrust. Ce fichier doit se trouver à la **racine** du dépôt.

Pour une introduction pratique, consulte le tutoriel 04 — Automatiser avec la CI.

Structure générale

```
language: <string>        # Profil de langage (requis)

build:
  command: <string>       # Commande de build

checks:
  <nom>: <string>         # Une ou plusieurs commandes de vérification
  <nom>: <string>

tests:
  command: <string>       # Commande de test principale

deploy:                   # Optionnel
  command: <string>
  only_on: <string>       # Branche déclenchant le déploiement (ex. "main")

artifacts:                # Optionnel
  paths:
    - <chemin>

env:                      # Variables d'environnement pour toutes les étapes
  <NOM>: <valeur>
```

Champ `language`

Type : string — **Requis**

© Gitrust

Charge un profil préconfigué avec les outils et l'image Docker adaptés au langage.

Valeur	Image Docker de base	Outils pré-installés
`rust`	`rust:latest`	`cargo` , `rustfmt` , `clippy`
`node`	`node:lts`	`npm` , `npx`
`python`	`python:3-slim`	`pip` , `pytest`
`go`	`golang:latest`	`go` , `gofmt` , `golangci-lint`

Si ton langage n'est pas listé, utilise `language: node` ou `language: python` comme base générique et surcharge les commandes manuellement. Les profils sont définis sur le serveur de build dans `deployment/ci-engine/profiles/` .

Section `build`

Optionnel

Exécutée en premier. Arrête le pipeline en cas d'échec (code de sortie non nul).

```
build:
  command: "cargo build --release"
```

Section `checks`

Optionnel

Dictionnaire de vérifications statiques (lint, format, sécurité). Chaque clé est un nom arbitraire affiché dans les logs. Les checks sont exécutés **en parallèle** si le serveur de build le supporte.

```
checks:
  lint: "cargo clippy -- -D warnings"
  format: "cargo fmt -- --check"
  audit: "cargo audit"
```

© Gitrust

Section `tests`

Optionnel

Commande de test principale. Exécutée après `build` et `checks`.

```
tests:
  command: "cargo test --release"
```

Section `deploy`

Optionnel

Déploiement conditionnel. Exécuté uniquement sur la branche spécifiée dans `only_on`.

```
deploy:
  command: "./scripts/deploy.sh production"
  only_on: "main"
```

Si `only_on` est omis, le déploiement s'exécute sur toutes les branches — généralement non souhaitable.

Section `artifacts`

Optionnel

Chemins des fichiers ou dossiers à conserver après le pipeline. Les artefacts sont accessibles depuis la page de détail du pipeline dans l'UI.

```
artifacts:
  paths:
    - target/release/mon-binaire
    - dist/
    - coverage/
```

Section `env`

Optionnel

© Gitrust

Variables d'environnement disponibles dans toutes les commandes du pipeline. Ne jamais placer de secrets ici — utilise les variables CI chiffrées via l'UI (`/{owner}/{repo}/ci/variables`).

```
env:
  RUST_BACKTRACE: "1"
  NODE_ENV: "test"
```

Les variables définies dans l'UI ont priorité sur celles du fichier YAML en cas de conflit de nom.

Exemples complets par langage

Rust

```
language: rust

build:
  command: "cargo build --release"

checks:
  lint: "cargo clippy -- -D warnings"
  format: "cargo fmt -- --check"

tests:
  command: "cargo test --release"

artifacts:
  paths:
    - target/release/mon-binaire
```

Node.js

```
language: node

build:
  command: "npm ci"

checks:
  lint: "npm run lint"
  typecheck: "npm run typecheck"
```


© Gitrust

```yaml
tests:
  command: "npm test"

artifacts:
  paths:
    - dist/
```

Python

```yaml
language: python

build:
  command: "pip install -r requirements.txt"

checks:
  lint: "flake8 src/"
  format: "black --check src/"

tests:
  command: "pytest --tb=short"
```

Go

```yaml
language: go

build:
  command: "go build ./..."

checks:
  lint: "golangci-lint run ./..."
  vet: "go vet ./..."

tests:
  command: "go test ./..."
```

© Gitrust

Détection automatique du mode

gitrust détecte le mode à exécuter lors d'un push selon la règle suivante :

Contenu de l'arbre du commit	Mode
`.gitrust-ci.yml` présent	**Easy** — ce fichier
`.dagger/` présent	**Power** — module Dagger complet
Aucun des deux	Pas de pipeline

Si les deux sont présents, le mode **Power** a la priorité.

Voir aussi

- Tutoriel 04 — Automatiser avec la CI : mise en pratique pas à pas
- Référence API REST — CI : déclencher et interroger les pipelines via l'API

© Gitrust

1.4.4 Markdown et syntaxe supportée

Référence de la syntaxe Markdown acceptée dans les **issues**, les **pull requests**, les **commentaires** et les fichiers **README** des dépôts gitrust.

Titres

```
# Titre niveau 1
## Titre niveau 2
### Titre niveau 3
#### Titre niveau 4
```

Les titres génèrent des ancres de navigation (ex. `#titre-niveau-2`).

Mise en forme du texte

Syntaxe	Rendu
`**gras**`	**gras**
`*italique*`	*italique*
`~~barré~~`	~~barré~~
`` `code inline` ``	`code inline`

Blocs de code

Délimiter par trois backticks. Spécifier le langage pour la coloration syntaxique :

````
```rust
fn main() {
    println!("Hello, gitrust!");
}
```
````


© Gitrust

Langages supportés pour la coloration : `rust`, `go`, `python`, `javascript`, `typescript`, `json`, `yaml`, `toml`, `bash`, `sql`, `html`, `css`, `markdown`, et la plupart des langages courants.

Listes

Liste non ordonnée

```
- Premier élément
- Deuxième élément
  - Élément imbriqué
```

Liste ordonnée

```
1. Première étape
2. Deuxième étape
3. Troisième étape
```

Liste de tâches (cases à cocher)

```
- [x] Tâche accomplie
- [ ] Tâche en attente
- [ ] Autre tâche
```

Rendu : cases cliquables dans les issues et les PRs. Cocher une case met à jour le texte en temps réel.

Tableaux

```
| Colonne A | Colonne B | Colonne C |
|-----------|-----------|-----------|
| Valeur 1  | Valeur 2  | Valeur 3  |
| Valeur 4  | Valeur 5  | Valeur 6  |
```

Alignement des colonnes :

© Gitrust

```
| Gauche | Centre | Droite |
|:-------|:------:|-------:|
| texte  | texte  |  texte |
```

Liens et images

```
[Texte du lien](https://example.com)
[Lien relatif](../how-to/gerer-cles-ssh.md)
![Texte alternatif](https://example.com/image.png)
```

Citations

```
> Ceci est une citation.
> Elle peut s'étendre sur plusieurs lignes.
```

Ligne de séparation horizontale

```
---
```

Références aux issues et aux PRs

gitrust transforme automatiquement les références en liens cliquables :

Syntaxe	Rendu
`#5`	Lien vers l'issue ou PR numéro 5 du dépôt courant
`owner/repo#5`	Lien vers l'issue 5 d'un autre dépôt

Mentions d'utilisateurs

© Gitrust

```
@tonpseudo
```

gitrust envoie une notification à l'utilisateur mentionné et crée un lien vers son profil.

Mots-clés de fermeture d'issue

Utilisés dans les messages de commit ou les descriptions de PR pour fermer automatiquement une issue lors de la fusion :

```
Closes #5
Fixes #12
Resolves #8
```

Insensibles à la casse. Plusieurs issues peuvent être référencées :

```
Closes #5, fixes #7
```

Emojis

La syntaxe `:nom_emoji:` est supportée :

Syntaxe	Rendu
`:rocket:`	🚀
`:bug:`	🐛
`:white_check_mark:`	✅
`:warning:`	⚠️
`:information_source:`	ℹ️

Labels dans les descriptions

Les labels assignés à une issue sont affichés dans la barre latérale et non directement dans le corps Markdown. Pour référencer un label dans le texte, utilise son nom entre backticks : `` `bug` ``.

© Gitrust

Diagrammes Mermaid

gitrust supporte les blocs Mermaid dans les README et les wikis (pas dans les commentaires d'issues) :

````
```mermaid
graph LR
    A[Début] --> B[Étape 1]
    B --> C[Fin]
```
````

Types supportés : `graph`, `sequenceDiagram`, `flowchart`, `stateDiagram-v2`, `classDiagram`, `gantt`, `pie`.

Voir aussi

- Utiliser les labels et les issues : appliquer les labels et références d'issues
- Ouvrir une pull request : rédiger une description de PR efficace

© Gitrust

1.4.5 Notifications

Référence des événements déclenchant des notifications, des canaux disponibles et des préférences configurables via `/settings/notifications`.

Canaux de notification

gitrust envoie les notifications par deux canaux :

Canal	Description	Configuration
SSE in-app	Notifications en temps réel dans l'interface web, via Server-Sent Events. Icône cloche en haut à droite.	Toujours actif quand tu es connecté
Email	Emails envoyés à l'adresse de ton compte	Configurable par type d'événement dans `/settings/notifications`

© Gitrust

Matrice des événements × canaux

Événement	SSE in-app	Email	Déclencheur
PR assignée (tu es reviewer)	√	√	Quelqu'un t'assigne comme reviewer d'une PR
Issue mentionnée (tu es mentionné par `@pseudo`)	√	√	Ton pseudo apparaît dans une issue ou un commentaire
Review demandée	√	√	L'auteur d'une PR te demande explicitement une review
CI échouée (sur ton push)	√	√	Un pipeline CI lancé par ton push se termine en échec
CI réussie (sur ton push)	√	—	Un pipeline CI lancé par ton push se termine en succès
Commentaire sur ta PR	√	√	Quelqu'un commente une PR dont tu es l'auteur
Commentaire sur ton issue	√	√	Quelqu'un commente une issue que tu as ouverte
PR fusionnée (tu es auteur)	√	—	Ta PR est fusionnée
PR fermée (tu es auteur)	√	—	Ta PR est fermée sans fusion
Issue fermée (tu es assigné)	√	—	Une issue qui t'est assignée est fermée
Membre ajouté à une équipe	√	√	Tu es ajouté à une équipe
Accès dépôt accordé	√	—	Un dépôt t'est rendu accessible (équipe ou direct)

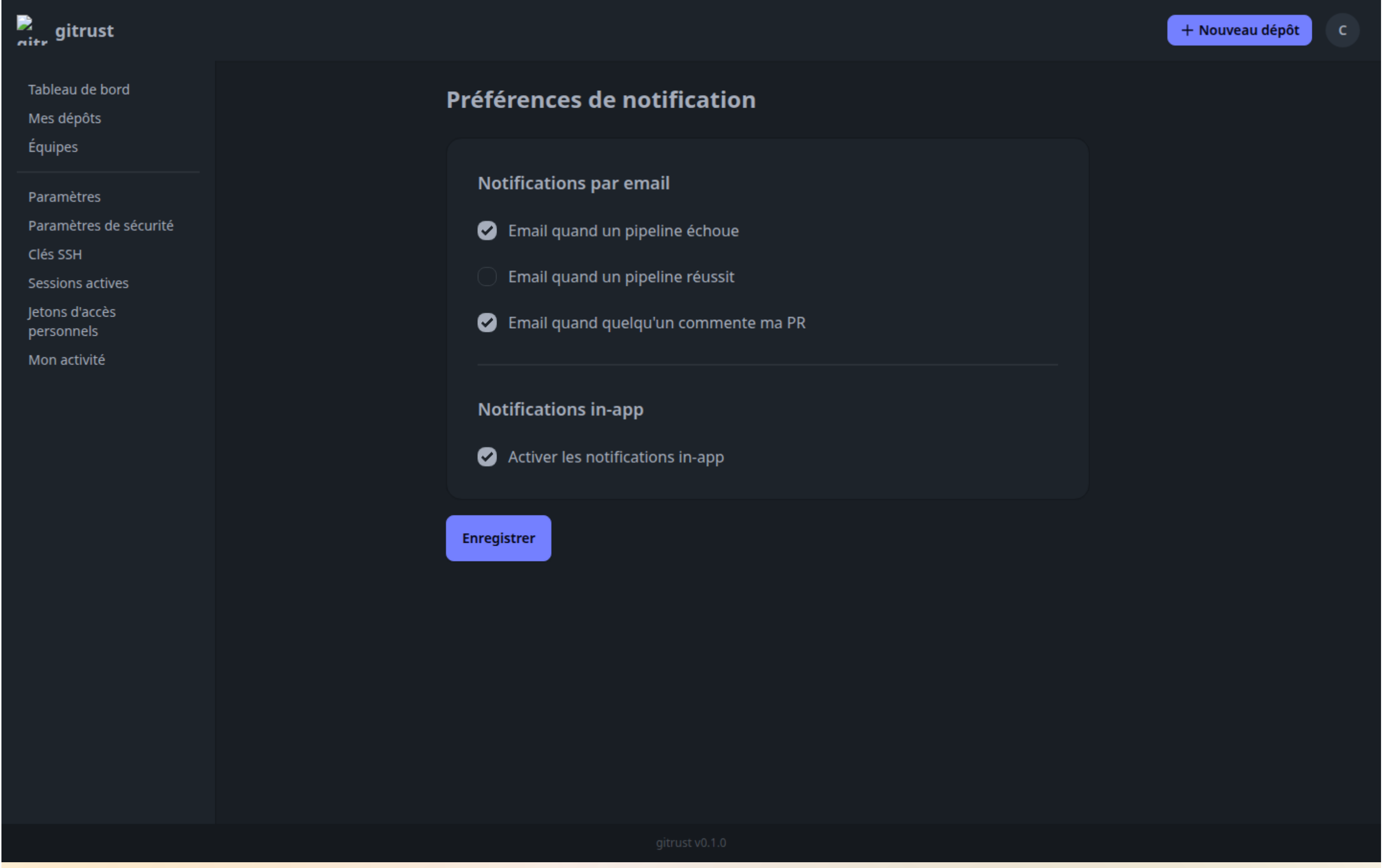
`✓` = activé par défaut — `–` = non envoyé par défaut (peut être activé dans les préférences)

Configurer les préférences

Navigue vers **Paramètres → Notifications** (`/settings/notifications`).

© Gitrust

La page affiche un tableau de cases à cocher pour chaque type d'événement × canal. Décoche les emails que tu ne veux pas recevoir et sauvegarde.



Stream SSE en temps réel

Les notifications SSE sont disponibles sur le endpoint :

```
GET /notifications/stream
```

Connexion en `text/event-stream`. Chaque notification est un événement JSON :

```
data: {"id": 42, "type": "pr_assigned", "repo": "owner/repo", "pr_num": 7, "created_at"
```

Ce stream alimente l'icône cloche de l'interface. Il peut aussi être consommé par des outils externes (scripts, bots) via un PAT.

© Gitrust

Marquer les notifications comme lues

- **Une notification** : `POST /notifications/{id}/read`
- **Toutes les notifications** : `POST /notifications/read-all`
- **Via l'interface** : clique sur une notification dans la liste `/notifications` , ou utilise le bouton **Tout marquer comme lu**

Voir aussi

- Créer un Personal Access Token : accéder au stream SSE par script
- API REST v1 : les endpoints de notifications ne sont pas listés dans l'API v1 publique — le stream SSE est le seul canal programmatique exposé

© Gitrust

1.5 Explications

1.5.1 Comprendre le modèle de collaboration sur gitrust

Ce que tu vas comprendre

- Expliquer pourquoi gitrust impose une review avant toute fusion sur `main` (Bloom 2)
- Analyser les compromis entre petites PRs fréquentes et grosses PRs rares (Bloom 4)
- Évaluer si le modèle de review obligatoire convient à un contexte donné (Bloom 5)

Le problème concret

Tu travailles seul sur une feature pendant deux semaines. La branche diverge, s'accumule. Quand tu ouvres enfin une PR, elle contient 800 lignes de diff. Personne dans l'équipe ne veut (ni ne peut) la reviewer sérieusement — le review se réduit à « ça a l'air bien, LGTM ». Le code est fusionné avec des bugs qui auraient été détectés si quelqu'un avait lu attentivement 50 lignes.

gitrust est conçu pour rendre ce scénario difficile et le modèle opposé — petites PRs fréquentes, review courte et sincère — naturel et peu coûteux.

L'analogie

Imagine un journal scientifique avec comité de lecture. Chaque article soumis est lu par deux relecteurs indépendants avant publication. Les articles courts et bien ciblés obtiennent des retours rapides et précis. Les articles de 80 pages avec 200 références prennent six mois et reviennent avec des commentaires superficiels.

La **code review** dans gitrust fonctionne exactement comme le comité de lecture : plus le changement proposé est petit et focalisé, plus la review est rapide, sincère et utile. L'auto-merge (fusionner sa propre PR sans review externe) est interdit pour la même raison qu'un auteur ne peut pas valider lui-même son propre article : le biais de l'auteur rend les erreurs invisibles.

Le modèle

```
flowchart TD
subgraph "Philosophie gitrust"
    A[Issue tracée
avant tout travail]
```


© Gitrust

```
        B[Branche courte
    1 issue = 1 branche]
          C[PR petite
  ≤200 lignes idéalement]
        D[Review sincère
    par au moins 1 pair]
          E[CI verte
      avant fusion]
       F[Fusion dans main
     historique propre]
            end

  A --> B --> C --> D --> E --> F
      F -->|Nouvelle issue| A
```

Trois principes structurants :

1. Toute modification passe par une PR

gitrust n'interdit pas le push direct sur `main` au niveau du protocole Git, mais le modèle de collaboration encourage systématiquement la PR : c'est le seul endroit où la CI est déclenchée, où la review est documentée, où l'issue est liée et fermée automatiquement.

Les équipes qui veulent appliquer cela strictement protègent la branche `main` dans les paramètres du dépôt (niveau Maintainer requis pour le push direct).

2. L'auto-merge est refusé par convention

gitrust ne force pas techniquement la review externe — un Owner peut fusionner sa propre PR. Mais la convention du projet, renforcée par l'outillage (assignation de reviewers, statut « En attente de review »), crée une pression sociale pour attendre une approbation externe.

L'exception acceptable : un dépôt solo où l'on est seul contributeur. Dans ce cas, la CI joue le rôle du reviewer.

3. La taille d'une PR est un signal de qualité

Taille du diff	Signal	Action recommandée
< 100 lignes	Excellent	Review en 5-10 min, commentaires précis
100-300 lignes	Acceptable	Review en 30 min, décomposer si possible
300-600 lignes	Attention	Décomposer en sous-tâches
> 600 lignes	Problème	Presque impossible à reviewer sérieusement

© Gitrust

Alternatives et compromis

Trunk-based development sans PR

Certaines équipes poussent directement sur `main` (trunk) avec feature flags pour isoler les fonctionnalités en développement. Plus rapide, moins de friction, mais nécessite une discipline de test très forte et une culture de confiance élevée. Adapté aux petites équipes seniors avec une couverture de tests élevée.

GitFlow avec longues branches de release

GitFlow maintient des branches `develop` , `release/*` , `hotfix/*` avec des cycles de merge longs. Adapté aux logiciels avec releases versionnées (ex. bibliothèques), inadapté aux services web déployés en continu.

Le choix de gitrust

gitrust adopte une variante de **trunk-based development avec PR courtes** : pas de `develop` séparée, pas de longues branches de feature, mais une PR obligatoire même pour les petits changements. C'est le compromis entre la vitesse du trunk-based et la traçabilité des PRs.

Vérifier ta compréhension

1. Un développeur propose une PR de 900 lignes modifiant le système d'authentification. Quelles questions lui poser pour l'aider à la découper ? (Bloom 2 — expliquer)

2. Ton équipe travaille sur un projet open-source avec des contributeurs externes que tu ne connais pas. Quels mécanismes de gitrust utiliserais-tu pour maintenir la qualité du code sans bloquer les contributions ? (Bloom 4 — analyser)

Pour aller plus loin

- Ouvrir une pull request : mettre en pratique ce modèle
- Stratégies de fusion : choisir la stratégie cohérente avec l'historique souhaité
- Cycle de vie d'une pull request : les états et transitions d'une PR de l'ouverture à la fusion

© Gitrust

1.5.2 Comprendre la sécurité côté utilisateur

Ce que tu vas comprendre

- Expliquer pourquoi la 2FA protège même si ton mot de passe est volé (Bloom 2)
- Analyser les différences entre clé SSH, mot de passe et PAT en termes de surface d'attaque (Bloom 4)
- Évaluer quelle combinaison de protections adopter selon ton niveau d'exposition (Bloom 5)

Le problème concret

Ton mot de passe a été compromis. Peut-être lors d'une fuite de données d'un autre service où tu utilisais le même mot de passe. L'attaquant se connecte à ton compte gitrust, clone tous tes dépôts privés, y dépose un backdoor et efface l'historique. Tu ne t'en aperçois que trois jours plus tard.

gitrust fournit quatre mécanismes de protection indépendants pour rendre ce scénario soit impossible, soit détectable immédiatement. Ce document explique comment ils s'articulent.

L'analogie

Imagine un coffre-fort bancaire avec plusieurs couches de sécurité :

- La **porte principale** (mot de passe) : nécessaire mais insuffisante seule
- Le **code secret temporaire** (2FA TOTP) : change toutes les 30 secondes, connu uniquement de toi
- La **clé physique nominative** (clé SSH) : associée à une machine précise, inopérante sans la clé privée
- Le **badge d'accès limité** (PAT) : n'ouvre que certaines salles, révocable instantanément

Chaque couche est indépendante. Compromettre l'une n'ouvre pas les autres.

Le modèle

```
graph TD
subgraph "Vecteurs d'attaque"
    V1[Mot de passe volé
phishing / fuite DB]
    V2[Machine compromise
malware / vol]
    V3[Token exposé
commit accidentel / log]
```


© Gitrust

```
end

subgraph "Protections gitrust"
P1[2FA TOTP
code valide 30s]
P2[Clé SSH + passphrase
liée à la machine]
P3[PAT à durée limitée
scope minimal]
P4[Audit log
connexions tracées]
end

V1 -->|bloqué par| P1
V2 -->|atténué par| P2
V3 -->|atténué par| P3
V1 & V2 & V3 -->|détectable via| P4
```

2FA — pourquoi le mot de passe seul ne suffit plus

Le mot de passe est un **secret statique** : une fois volé, il est valable indéfiniment jusqu'à ce que tu le changes. Un code TOTP (Time-based One-Time Password) est un **secret dynamique** : il expire en 30 secondes et ne peut être utilisé qu'une seule fois. Même si un attaquant intercepte ton code TOTP, il est déjà expiré au moment où il essaie de l'utiliser.

gitrust stocke uniquement le **secret TOTP** (une chaîne de 32 caractères) qui permet de générer des codes — jamais les codes eux-mêmes. Ce secret est chiffré en base.

Clés SSH — une identité par machine

Une clé SSH ed25519 est une paire mathématique (clé privée + clé publique) liée à une machine. gitrust ne stocke que la clé **publique** — celle-ci ne permet pas de reconstruire la clé privée. Pour s'authentifier, ta machine doit prouver qu'elle possède la clé privée en signant un challenge.

Ce que gitrust stocke : clé publique + empreinte SHA256 (pour identification) + date d'ajout.

Ce que gitrust ne stocke jamais : la clé privée (elle ne quitte jamais ta machine).

Si ta machine est volée, supprimer la clé dans `/settings/keys` révoque immédiatement l'accès — même si le voleur a copié les fichiers `~/.ssh/`.

PAT — accès minimal et révocable

Un Personal Access Token est haché (SHA-256) dès sa création. gitrust ne stocke que le hash — si un attaquant accède à la base de données, les tokens en clair ne sont pas exposés. La valeur réelle du token (`gr_pat_xxxx`) n'est affichée qu'une seule fois à la création.

© Gitrust

Le scope minimal (`repo:read` uniquement pour un script de lecture) limite les dégâts si un token est exposé. La date d'expiration garantit qu'un token oublié dans un vieux script ne reste pas valide indéfiniment.

Audit log — tout est tracé

Gitrust enregistre les connexions réussies et les tentatives échouées dans un journal d'audit accessible via `/settings/activity`. Si tu constates des connexions depuis des IP inconnues ou à des heures inhabituelles, c'est le signal d'une compromission.

Alternatives et compromis

Mot de passe fort sans 2FA

Un mot de passe long et unique (généré par un gestionnaire de mots de passe) réduit considérablement le risque de compromission par brute-force ou réutilisation. Mais il ne protège pas contre le phishing (tu es trompé pour le saisir sur un faux site) ni contre les fuites de base de données. La 2FA couvre ces deux vecteurs.

Clé SSH sans passphrase

Plus pratique (pas de saisie à chaque push), mais si ta machine est volée ou compromise par un malware, la clé privée est directement utilisable. Une passphrase chiffre la clé sur le disque — seule une combinaison vol physique + connaissance de la passphrase permet l'accès.

PAT sans expiration

Acceptable pour les outils personnels sur une machine de confiance. Inacceptable pour un token partagé dans un environnement CI, un script de serveur, ou un dépôt accessible à plusieurs personnes.

Vérifier ta compréhension

1. Un collègue dit « je n'ai pas besoin de la 2FA, mon mot de passe fait 30 caractères aléatoires ». Quels scénarios d'attaque son mot de passe fort ne couvre-t-il pas ? (Bloom 2 — expliquer)

2. Tu découvres que ton token PAT a été accidentellement commité dans un dépôt public il y a 6 heures. Quelles actions effectues-tu dans quel ordre, et pourquoi ? (Bloom 4 — analyser)

Pour aller plus loin

- Configurer la double authentification (2FA) : activer le TOTP pas à pas
- Gérer ses clés SSH : générer, déclarer et révoquer des clés
- Créer un Personal Access Token : scopes, expiration, révocation

© Gitrust

1.5.3 Comprendre le cycle de vie d'une pull request

Ce que tu vas comprendre

- Identifier les états possibles d'une PR et les transitions entre eux (Bloom 2)
- Analyser pourquoi certaines transitions sont automatiques et d'autres manuelles (Bloom 4)
- Évaluer à quel moment une PR est prête à être fusionnée (Bloom 5)

Le problème concret

Tu as ouvert une PR il y a trois jours. Un reviewer a laissé des commentaires. Tu as poussé des corrections. La CI a repassé au vert. Mais tu ne sais pas si tu dois re-demander une review, si tu peux fusionner maintenant, ou si quelque chose bloque encore. L'état de la PR n'est pas clair.

Comprendre le cycle de vie d'une PR te permet de lire son état actuel d'un coup d'œil et de savoir exactement quelle action effectuer ensuite.

L'analogie

Une pull request ressemble à un dossier de demande de permis de construire :

- **Ouvert** : le dossier est déposé, les voisins (reviewers) peuvent faire des remarques
- **En cours de review** : l'architecte (reviewer) examine les plans et demande des modifications
- **Approuvé** : l'architecte signe les plans modifiés
- **CI verte** : les normes techniques sont vérifiées automatiquement (sécurité incendie, etc.)
- **Fusionné** : le permis est accordé, les travaux peuvent commencer sur `main`
- **Refusé / fermé** : le dossier est archivé sans suite

Comme pour un permis, une PR peut être rouverte si elle a été fermée par erreur ou prématurément.

Le modèle — diagramme de séquence complet

```
stateDiagram-v2
[*] --> Draft : créer en brouillon
[*] --> Open : créer directement

Draft --> Open : marquer "Prête pour review"
Open --> Draft : repasser en brouillon
```


© Gitrust

```
Open --> ReviewRequested : assigner un reviewer
ReviewRequested --> ChangesRequested : reviewer demande modifications
ReviewRequested --> Approved : reviewer approuve

ChangesRequested --> ReviewRequested : auteur pousse corrections
+ re-demande review
Approved --> Open : nouveau push (invalidation auto)

Open --> Closed : fermer sans fusionner
Closed --> Open : rouvrir

Approved --> Merged : fusionner (CI verte requise si configurée)
Open --> Merged : fusionner (si permissions suffisantes)

Merged --> [*]
Closed --> [*]
```

Les états en détail

Draft (Brouillon)

La PR est visible par l'équipe mais ne peut pas être fusionnée. Utilise cet état pour : - Partager du code en cours de travail pour des commentaires précoces - Réserver un numéro de PR pour une issue - Travailler en collaboration sur la même branche

Transition manuelle : l'auteur clique **Marquer comme prête pour review**.

Open (Ouverte)

État par défaut à la création. La PR peut être commentée, reviewée et fusionnée (selon les permissions). C'est l'état de travail normal.

ReviewRequested (Review demandée)

Un reviewer a été assigné. gitrust lui envoie une notification. La PR attend son verdict. L'auteur peut continuer à pousser des commits pendant ce temps.

ChangesRequested (Modifications demandées)

Le reviewer a explicitement demandé des changements. La PR est bloquée — elle ne peut pas être fusionnée tant que le reviewer n'a pas approuvé ou n'a pas retiré son refus.

Transition de sortie : l'auteur pousse les corrections et re-demande une review (bouton **Re-request review**).

© Gitrust

Approved (Approuvée)

Au moins un reviewer a approuvé. La PR peut être fusionnée si la CI est verte.

Invalidation automatique : si l'auteur pousse un nouveau commit après l'approbation, le statut repasse à **Open** — le reviewer doit approuver à nouveau. Ce comportement garantit que ce qui est fusionné correspond exactement à ce qui a été approuvé.

Merged (Fusionnée)

État terminal. Le code est dans la branche cible. La PR est archivée en lecture seule. Si la description contenait `closes #N`, l'issue N est fermée automatiquement à ce moment.

Closed (Fermée)

Fermée sans fusion — le travail a été abandonné ou la PR était incorrecte. Peut être rouverte si nécessaire. Les commits de la branche ne sont pas supprimés.

Transitions automatiques

Déclencheur	Transition
Nouveau commit poussé après approbation	Approved → Open (invalidation de l'approbation)
Fusion dans la branche cible	Open/Approved → Merged
Fusion + `closes #N` dans le commit ou la description	Issue N → Closed
CI configurée et pipeline échoué	Bouton **Fusionner** désactivé (blocage soft)

Alternatives et compromis

Review obligatoire vs optionnelle

gitrust permet de fusionner une PR sans review externe si l'on a les permissions Owner ou Maintainer. Certaines équipes configurent une **branche protégée** exigeant au moins une approbation — ce qui rend la review techniquement obligatoire. D'autres s'appuient sur la convention sans protection forcée. La protection forcée convient aux projets critiques ou réglementés ; la convention suffit pour les petites équipes de confiance.

© Gitrust

Invalidation automatique après push

Certaines forges (GitHub) ont rendu ce comportement optionnel. gitrust invalide systématiquement l'approbation après un nouveau push — c'est plus strict mais garantit que le reviewer a vu exactement le code fusionné. Inconvénient : un correctif de typo force une re-review complète.

Vérifier ta compréhension

1. Une PR est à l'état **ChangesRequested**. L'auteur pousse ses corrections mais ne re-demande pas de review. Que se passe-t-il ? Peut-on fusionner ? (Bloom 2 — identifier l'état)

2. Tu es Maintainer d'un dépôt. Une PR approuvée contient un conflit de merge introduit par un commit récent sur `main`. Quelles options as-tu, et quels sont les avantages et inconvénients de chacune ? (Bloom 4 — analyser les compromis)

Pour aller plus loin

- Ouvrir une pull request : créer et suivre une PR
- Stratégies de fusion : choisir entre fast-forward, squash et merge commit
- Modèle de collaboration : pourquoi la review est centrale dans gitrust

© Gitrust