Manuel développeur

Gitrust — forge Git self-hosted écrite en Rust

Équipe Gitrust

© Gitrust

Sommaire

1. Manuel développeur 3
1.1 Manuel développeur 3
1.2 Tutoriels 6
1.3 Guides pratiques 39
1.4 Référence 98
1.5 Explications 170

© Gitrust

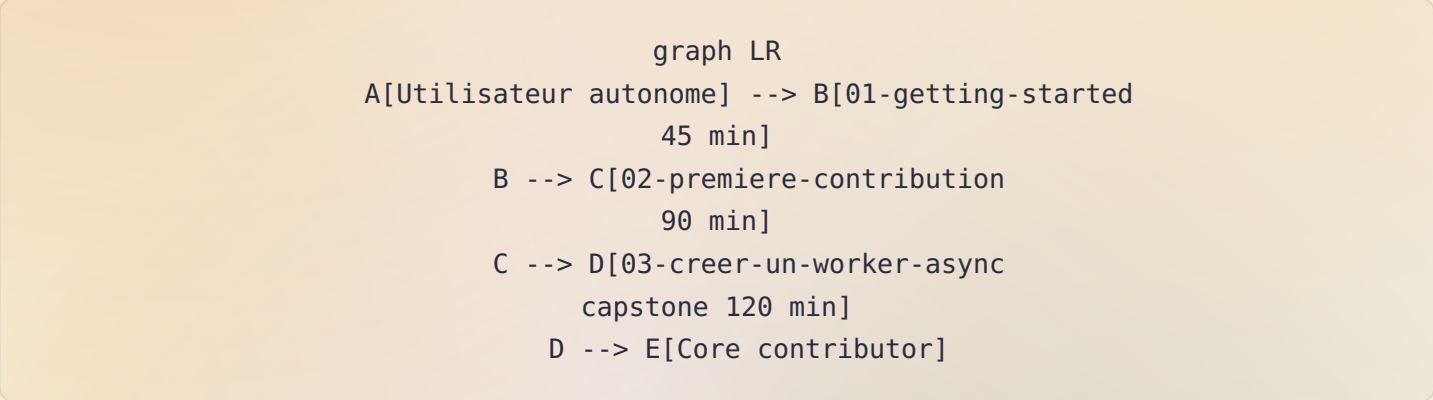
1. Manuel développeur

1.1 Manuel développeur

Ce manuel s'adresse à deux publics : les contributeurs au code gitrust (core ou via fork rustwarden) et les développeurs externes qui construisent des intégrations sur l'API publique. Il fournit tout le nécessaire pour comprendre l'architecture des 6 crates, contribuer du code conforme aux standards du projet, ou bâtir un client qui parle à l'API gitrust. Il ne couvre pas l'installation d'une instance de production ni l'usage quotidien de la forge.

1.1.1 Parcours d'apprentissage — contributeur core

Le parcours contributeur représente environ 4 heures au total, avec un étayage dégressif : le premier tutoriel est entièrement guidé, le troisième (capstone) demande de résoudre sans guide pas-à-pas.



> ***Checkpoint de parcours** : avant de passer au tutoriel 02, vous devez avoir compilé gitrust localement, lancé les tests unitaires et obtenu une instance de développement fonctionnelle. Si ce n'est pas le cas, relisez le tutoriel 01.*
>
> ***Checkpoint de parcours** : avant le tutoriel 03 (capstone), vous devez avoir soumis une PR complète avec tests, passé la gate QA et reçu une review. Si ce n'est pas le cas, relisez le tutoriel 02.*

1.1.2 Tutoriels — apprendre en faisant

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

- 01 — Getting started : build local, tests, instance de dev — ~45 min
- 02 — Première contribution : clone, build, test, PR — ~90 min
- 03 — Créer un worker async (capstone) — ~120 min

© Gitrust

1.1.3 How-to — recettes pour les tâches courantes

Les how-to répondent à « comment faire X » sans explication de fond. Consultez-les dans n'importe quel ordre selon votre besoin.

- Workflow de contribution : Git + review + QA
- Lancer les tests unitaires
- Lancer les tests E2E (Playwright)
- Ajouter une route web (patron rustwarden → axum)
- Ajouter un service métier
- Ajouter une migration de base de données
- Implémenter des webhooks
- Implémenter un endpoint API
- Importer un dépôt externe (code)
- Contribuer à rustwarden upstream
- Passer la QA avant merge

1.1.4 Référence — informations techniques exactes

La référence documente de façon exhaustive l'architecture, les API internes, le schéma de données et les règles QA. Destinée à la consultation ponctuelle.

- Architecture des 6 crates
- Services et API interne
- Schéma de base de données (ERD)
- Règles QA et gates ANSSI
- API REST v1 — auth, pagination, codes d'erreur
- Git hooks post-receive
- rustwarden-core API — helpers, extracteurs, middleware

© Gitrust

1.1.5 Explication — comprendre le pourquoi

Les explanations construisent les modèles mentaux nécessaires pour contribuer de façon éclairée. Lisez-les quand vous voulez comprendre les décisions d'architecture, pas seulement les appliquer.

- Vue d'ensemble de l'architecture gitrust
- Flux de requêtes (diagrammes de séquence)
- Modèle de permissions RBAC 4 niveaux
- Fonctionnement de la 2FA
- Hiérarchie des labels
- CI Dagger : Easy Mode vs Power Mode
- Décisions UI : HTMX + DaisyUI
- Patron worker async

© Gitrust

1.2 Tutoriels

1.2.1 Premiers pas avec gitrust : compilez, configurez et poussez votre premier commit

> ***Source** : ce tutoriel intègre le contenu de `gitrust/docs/GETTING_STARTED.md` avec la structure pédagogique complète (objectifs Bloom, modèle mental, checkpoints observables, section « Et si ça ne marche pas »).*

Objectifs

À la fin de ce tutoriel, vous saurez :

- **O1. Compiler** gitrust depuis les sources et lancer l'instance de développement localement
- **O2. Configurer** l'environnement de développement (base de données, variables d'environnement, assets CSS)
- **O3. Exécuter** un premier push SSH vers votre instance locale et vérifier le résultat dans l'interface web

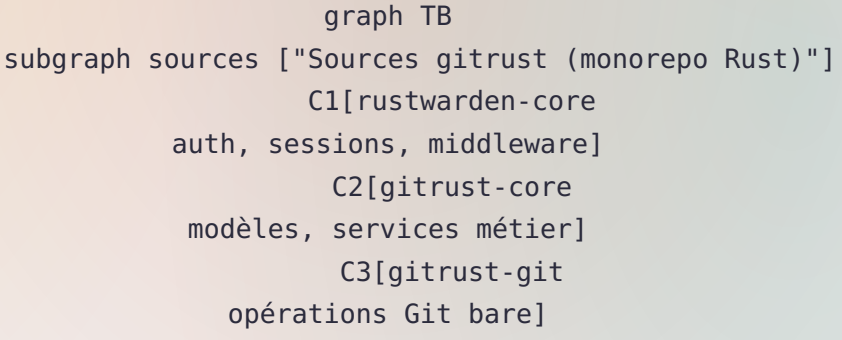
Pré-requis

- **Technique** : Rust stable ≥ 1.80 (`rustc --version`), PostgreSQL ≥ 15 (`psql --version`), Node.js ≥ 18 (`node --version`), Git 2.x (`git --version`), Docker (optionnel, recommandé pour la base de données)
- **Pédagogique** : être un utilisateur autonome de gitrust (parcours `user` complété ou équivalent) ; familiarité avec Rust (au moins 3 mois de pratique) et avec HTTP/PostgreSQL
- **Temps estimé** : ~45 minutes

Vue d'ensemble

Avant d'exécuter la première commande, prenons 3 minutes pour comprendre ce que vous allez monter et comment les pièces s'articulent.

gitrust est un **monorepo Rust à 6 crates** qui compile en un seul binaire. Quand vous lancez `cargo run`, ce binaire démarre deux serveurs simultanément : un serveur HTTP (`:4000`) pour l'interface web et l'API REST, et un serveur SSH (`:2222`) pour les opérations `git push/pull` . Ils partagent la même base de données PostgreSQL et le même stockage de dépôts bare sur disque.



© Gitrust

```
                C4[gitrust-web
            handlers HTTP + templates]
                C5[gitrust-ssh
           serveur SSH Git]
               C6[gitrust-hooks
          post-receive, pre-receive]
                    end

      subgraph runtime ["Processus unique (cargo run)"]
                    HTTP["HTTP :4000
           UI + API REST"]
                    SSH["SSH :2222
           git push/pull"]
                    end

       subgraph storage ["Stockage"]
                    PG[(PostgreSQL
          comptes, dépôts,
            issues, PRs)]
              REPOS[Dépôts bare
            data/repos/]
                    end

C1 & C2 & C3 & C4 & C5 & C6 -->|cargo build| HTTP
C1 & C2 & C3 & C4 & C5 & C6 -->|cargo build| SSH
          HTTP & SSH --> PG
         HTTP & SSH --> REPOS
```

Ce que vous allez faire : cloner les sources → préparer PostgreSQL → copier et remplir `.env` → compiler les assets CSS → lancer `cargo run` → vérifier l'instance → pousser un premier commit via SSH.

Étape 1 : Vérifiez les pré-requis système

Avant de cloner quoi que ce soit, vérifiez que tous les outils nécessaires sont installés :

```
rustc --version && cargo --version && psql --version && node --version && git --version
```

Sortie attendue (versions exactes peuvent varier) :

© Gitrust

```
rustc 1.82.0 (f6e511eec 2024-10-15)
cargo 1.82.0 (8f40fc59f 2024-10-15)
psql (PostgreSQL) 16.3
v20.11.0
git version 2.43.0
```

Si des dépendances système manquent, installez-les :

```
# Debian / Ubuntu
sudo apt install build-essential pkg-config libssl-dev libpq-dev cmake git

# macOS
brew install postgresql openssl cmake
```

Checkpoint : toutes les commandes de vérification doivent retourner un numéro de version sans erreur. Si `rustc` n'est pas trouvé, installez Rust via `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh` puis rechargez votre shell.

Étape 2 : Clonez le projet

```
git clone https://demo.gitrust.eu/gitrust/gitrust.git
cd gitrust
```

Sortie attendue :

```
Cloning into 'gitrust'...
remote: Enumerating objects: 12847, done.
remote: Counting objects: 100% (12847/12847), done.
Receiving objects: 100% (12847/12847), 18.42 MiB | 4.20 MiB/s, done.
```

Checkpoint : vérifiez que la structure de crates est présente :

```
ls crates/
```

Sortie attendue :

© Gitrust

```
gitrust-core  gitrust-git  gitrust-hooks  gitrust-ssh  gitrust-web  rustwarden-core
```

Étape 3 : Configurez l'environnement (.env)

Copiez le fichier d'exemple et éditez les 5 valeurs essentielles :

```
cp .env.example .env
```

Ouvrez `.env` dans votre éditeur et renseignez ces valeurs :

```
# Connexion à la base de données
DATABASE_URL=postgres://gitrust:motdepasse@localhost:5432/gitrust

# Clé secrète JWT (générez-la avec : openssl rand -hex 32)
JWT_SECRET=votre_cle_secrete_64_caracteres

# Compte administrateur créé au premier lancement
ADMIN_USERNAME=admin
ADMIN_EMAIL=admin@example.com
ADMIN_PASSWORD=MotDePasseFort123!
```

> *Les autres variables ont des valeurs par défaut raisonnables pour le développement. Consultez `.env.example` pour la liste complète.*

Générez une vraie valeur pour `JWT_SECRET` :

```
openssl rand -hex 32
```

Sortie attendue (valeur différente à chaque exécution) :

```
4a7f2c9e1b8d3f6a0e5c2b9d7f4a1e8c3b6d9f2a5e8c1b4d7f0a3e6c9b2d5f8a
```

Copiez cette valeur dans `.env` pour `JWT_SECRET`.

Checkpoint : vérifiez qu'aucune valeur d'exemple reste dans `.env` :

© Gitrust

```
grep -E "votre_cle|example\.com|MotDePasse" .env
```

La commande doit retourner vide (aucune ligne). Si elle retourne des lignes, ces valeurs placeholder sont encore présentes — l'application démarrera mais avec une configuration non sécurisée.

Étape 4 : Préparez la base de données PostgreSQL

Vous avez deux options :

Option A — Docker (recommandé pour débuter) :

```
cd database && docker compose up -d && cd ..
```

Sortie attendue :

```
[+] Running 2/2
 ✔ Container gitrust_postgres_dev  Started    0.8s
 ✔ Container gitrust_pgadmin       Started    1.1s
```

Option B — PostgreSQL existant sur la machine :

```
psql -U postgres << 'SQL'
CREATE USER gitrust WITH PASSWORD 'motdepasse';
CREATE DATABASE gitrust OWNER gitrust;
SQL
```

Sortie attendue :

```
CREATE ROLE
CREATE DATABASE
```

Adaptez le mot de passe dans `DATABASE_URL` dans `.env` si vous utilisez l'option B.

Checkpoint : vérifiez que la base est accessible depuis l'application :

```
psql "$DATABASE_URL" -c "SELECT version();"
```


© Gitrust

Sortie attendue :

```
                                                 version
-----------------------------------------------------------------------------------------
 PostgreSQL 16.3 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 13.2.0-23ubuntu4) 13.2
(1 row)
```

Si vous obtenez `psql: error: connection to server failed`, vérifiez que PostgreSQL tourne et que `DATABASE_URL` correspond.

Étape 5 : Compilez les assets CSS

gitrust utilise Tailwind CSS + DaisyUI. Les assets doivent être compilés localement — aucun CDN, tous les fichiers sont servis depuis `static/` :

```
npm install
npx tailwindcss -i static/css/input.css -o static/css/style.css --minify
```

Sortie attendue :

```
added 98 packages in 3s

Rebuilding...
Done in 1842ms.
```

Checkpoint : vérifiez que le fichier CSS est généré :

```
ls -lh static/css/style.css
```

Sortie attendue :

```
-rw-r--r-- 1 vous vous 45K avr 17 10:15 static/css/style.css
```

Si le fichier fait 0 octet ou n'existe pas, la compilation a échoué — relancez `npm install` puis la commande `npx tailwindcss`.

© Gitrust

Étape 6 : Lancez l'application

```
cargo run
```

La première compilation prend 2 à 5 minutes (toutes les dépendances sont compilées). Les compilations suivantes sont incrémentales et beaucoup plus rapides.

Au premier lancement, gitrust exécute automatiquement : 1. Les migrations de base de données (création de toutes les tables) 2. La création du compte administrateur (depuis `.env`) 3. La génération de la clé SSH host (`data/ssh_host_ed25519_key`) 4. La création du répertoire de stockage des dépôts (`data/repos/`)

Sortie attendue (après compilation) :

```
   Compiling gitrust-web v0.9.0 (/home/vous/gitrust/crates/gitrust-web)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 187.42s
     Running `target/debug/gitrust`
2026-04-17T10:20:00Z INFO  gitrust > Starting gitrust v0.9.0
2026-04-17T10:20:00Z INFO  gitrust > Running database migrations... applied 42
2026-04-17T10:20:01Z INFO  gitrust > Admin account created: admin
2026-04-17T10:20:01Z INFO  gitrust > SSH host key: SHA256:xxxxxxxxxxxxxxxxxxxx
Server listening on http://0.0.0.0:4000
```

Checkpoint : ouvrez `http://localhost:4000` dans votre navigateur. Vous devez voir la page d'accueil gitrust. Connectez-vous avec les identifiants `ADMIN_USERNAME` / `ADMIN_PASSWORD` définis dans `.env`. Si la page ne charge pas, vérifiez que le message `Server listening on http://0.0.0.0:4000` est bien apparu dans les logs.

Étape 7 : Créez un dépôt et poussez votre premier commit

Dans l'interface web (`http://localhost:4000`), cliquez sur **« New repository »**, donnez un nom (ex. `mon-projet`) et cliquez **Créer**.

Ajoutez votre clé SSH publique dans **Settings** > **SSH Keys** en collant le contenu de `~/.ssh/id_ed25519.pub`.

Puis dans votre terminal :

```
git clone ssh://git@localhost:2222/admin/mon-projet.git
cd mon-projet

echo "# Mon projet" > README.md
git add README.md
```


© Gitrust

```
git commit -m "Initial commit"
git push origin main
```

Sortie attendue :

```
Cloning into 'mon-projet'...
warning: You appear to have cloned an empty repository.
[main (root-commit) a1b2c3d] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 226 bytes | 226.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To ssh://localhost:2222/admin/mon-projet.git
 * [new branch]      main -> main
```

Checkpoint : retournez dans le navigateur sur `http://localhost:4000/admin/mon-projet`. La page doit afficher le fichier `README.md` avec le contenu `# Mon projet` et indiquer 1 commit sur la branche `main`.

Fonctionnalités supplémentaires disponibles

CI intégrée (optionnelle)

La CI s'exécute sur un serveur de build via SSH + rsync. Configurez dans `.env` :

```
# Même machine (défaut pour le développement)
CI_REMOTE_HOST=localhost

# Ou serveur dédié
CI_REMOTE_HOST=192.168.13.110
CI_REMOTE_USER=ci-runner
```

Easy Mode — ajoutez `.gitrust-ci.yml` à la racine de votre dépôt :

```
language: rust

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


© Gitrust

```
tests:
  command: "cargo test"
```

Power Mode — placez un module Dagger dans `.dagger/` à la racine. gitrust exécutera `dagger call -m .dagger/ ci`.

Notifications

Configurez SMTP dans `.env` pour les notifications e-mail :

```
SMTP_HOST=smtp.example.com
SMTP_PORT=587
SMTP_USER=user
SMTP_PASSWORD=password
SMTP_FROM=noreply@gitrust.eu
```

Langue de l'interface

gitrust est disponible en 6 langues (fr, en, de, es, pt, it). Définissez la langue par défaut de l'instance :

```
DEFAULT_LOCALE=fr
```

Récapitulatif

- √ **O1 accompli** : gitrust est compilé depuis les sources et tourne localement — `cargo run` affiche « Server listening on http://0.0.0.0:4000 »
- √ **O2 accompli** : l'environnement est configuré — `.env` rempli, PostgreSQL connecté, assets CSS générés, migrations appliquées
- √ **O3 accompli** : un premier push SSH a réussi — le commit est visible dans l'interface web à `http://localhost:4000/admin/mon-projet`

© Gitrust

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`error[E0463]: can't find crate for 'std'` à la compilation	Toolchain Rust manquante ou mauvaise version	Exécutez `rustup update stable && rustup default stable`. Vérifiez avec `rustc --version` ≥ 1.80
`connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed`	PostgreSQL non démarré ou `DATABASE_URL` incorrect	Démarrez PostgreSQL (`sudo systemctl start postgresql` ou `docker compose up -d` dans `database/`). Vérifiez que `DATABASE_URL` dans `.env` correspond
`Permission denied (publickey)` lors du `git push`	Clé SSH non enregistrée dans gitrust ou mauvais port	Vérifiez dans *Settings* > *SSH Keys* que la clé est présente. Testez avec `ssh -T git@localhost -p 2222` — vous devez voir « Vous êtes authentifié »

Prochaine étape

→ 02 — Première contribution : clone, build, test, PR : apprenez à créer une feature branch, écrire des tests, et soumettre votre première pull request en suivant le workflow gitrust (~90 min)

© Gitrust

1.2.2 Votre première contribution à gitrust

Objectifs

À la fin de ce tutoriel, vous saurez :

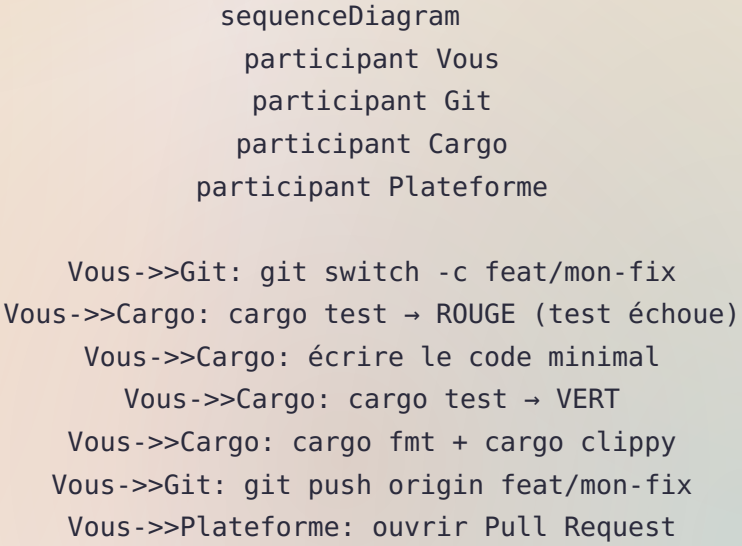
- O1. Créer une branche de fonctionnalité depuis `main` et identifier un *good first issue* dans le backlog.
- O2. Écrire un test unitaire qui échoue avant d'écrire le code de production (cycle TDD rouge-vert).
- O3. Appliquer la chaîne QA complète (`cargo fmt`, `cargo clippy`, `cargo test`) et ouvrir une pull request sur la plateforme gitrust.

Pré-requis

- **Technique** : Rust ≥ 1.77 installé, `git` ≥ 2.40, accès en lecture/écriture à une instance gitrust (locale ou `demo.gitrust.eu`), tutoriel `01-getting-started` complété (environnement local qui compile).
- **Pédagogique** : tutoriel `01-getting-started` complété.
- **Temps estimé** : ~90 minutes.

Vue d'ensemble

Contribuer à gitrust suit un cycle court et répétable : **fork ou branche → test rouge → code minimal → tests verts → QA → PR**.



Le cycle TDD (Test-Driven Development) est essentiel ici : en écrivant d'abord le test, vous définissez précisément ce que votre code doit faire avant de l'écrire. C'est ce que gitrust appelle **rouge → vert → refactor**.

© Gitrust

Étape 1 : Identifier un *good first issue*

Rendez-vous sur la page issues du dépôt gitrust (ou sur votre instance locale). Filtrez par le label de classification `good-first-issue`.

Lisez l'issue attentivement. Pour ce tutoriel, nous allons simuler le cas d'un issue fictif :

> ***Issue #42*** — `RepoSlug` *devrait rejeter les slugs commençant par un tiret.*

Assignez-vous l'issue (bouton « Assignee » dans la sidebar) pour signaler que vous travaillez dessus.

Checkpoint : vous devriez voir votre nom dans la liste des assignés de l'issue.

Étape 2 : Créer une branche de travail

Depuis la racine du dépôt gitrust, créez une branche nommée selon la convention `fix/<numéro>-description-courte` ou `feat/<numéro>-description-courte` :

```
git switch main
git pull --rebase origin main
git switch -c fix/42-repo-slug-rejects-leading-dash
```

Sortie attendue :

```
Switched to a new branch 'fix/42-repo-slug-rejects-leading-dash'
```

Checkpoint : `git branch` doit afficher votre branche avec un `*` devant.

Étape 3 : Écrire un test qui échoue (rouge)

Avant d'écrire le moindre code de production, écrivez le test qui décrit le comportement attendu. Ouvrez le fichier contenant la validation des slugs :

```
crates/gitrust-core/src/models/repository.rs
```

Repérez le bloc `#[cfg(test)]` existant en bas du fichier (ou créez-le s'il est absent). Ajoutez ce test dans la section `tests` :

```
#[cfg(test)]
mod tests {
    use super::*;
```


© Gitrust

```
    // Tests existants...

    #[test]
    fn repo_slug_rejects_leading_dash() {
        // Un slug qui commence par '-' doit être invalide.
        // Ce test DOIT échouer avant notre correctif.
        let result = RepoSlug::new("-invalid-slug");
        assert!(
            result.is_err(),
            "Un slug commençant par '-' devrait être rejeté"
        );
    }
}
```

Exécutez maintenant les tests pour confirmer que ce test échoue :

```
cargo test --package gitrust-core repo_slug_rejects_leading_dash
```

Sortie attendue (rouge) :

```
test models::repository::tests::repo_slug_rejects_leading_dash ... FAILED

failures:
    models::repository::tests::repo_slug_rejects_leading_dash

test result: FAILED. 0 passed; 1 failed
```

Checkpoint : le test échoue. C'est normal — c'est exactement l'état « rouge ».

> ***Jargon*** *:* `RepoSlug` *est un newtype — un type Rust qui enveloppe* `String` *et applique une validation à la construction via* `RepoSlug::new()`*. Cette technique garantit qu'un* `RepoSlug` *valide ne peut jamais exister sans passer par la validation.*

Étape 4 : Écrire le code minimal (vert)

Localisez la fonction `RepoSlug::new` dans `crates/gitrust-core/src/models/repository.rs`. Elle ressemble à ceci (simplifié) :

```
impl RepoSlug {
    pub fn new(s: &str) -> Result<Self, ValidationError> {
```


© Gitrust

```
        if s.is_empty() || s.len() > 64 {
            return Err(ValidationError::InvalidSlug("longueur invalide".into()));
        }
        if s.contains("..") || s.contains('/') {
            return Err(ValidationError::InvalidSlug("caractères interdits".into()));
        }
        // ... autres validations
        Ok(Self(s.to_lowercase()))
    }
}
```

Ajoutez **uniquement** la vérification manquante — ne modifiez rien d'autre :

```
impl RepoSlug {
    pub fn new(s: &str) -> Result<Self, ValidationError> {
        if s.is_empty() || s.len() > 64 {
            return Err(ValidationError::InvalidSlug("longueur invalide".into()));
        }
        // Nouveau : rejeter les slugs commençant par un tiret
        if s.starts_with('-') {
            return Err(ValidationError::InvalidSlug(
                "un slug ne peut pas commencer par un tiret".into(),
            ));
        }
        if s.contains("..") || s.contains('/') {
            return Err(ValidationError::InvalidSlug("caractères interdits".into()));
        }
        Ok(Self(s.to_lowercase()))
    }
}
```

Relancez le test :

```
cargo test --package gitrust-core repo_slug_rejects_leading_dash
```

Sortie attendue (vert) :

© Gitrust

```
test models::repository::tests::repo_slug_rejects_leading_dash ... ok

test result: ok. 1 passed; 0 failed
```

Checkpoint : le test passe. Vous êtes en état « vert ».

Étape 5 : Vérifier que rien d'autre ne régresse

Un seul test qui passe ne suffit pas. Vérifiez l'ensemble de la suite du crate :

```
cargo test --package gitrust-core
```

Sortie attendue :

```
running XX tests
...
test result: ok. XX passed; 0 failed; 0 ignored
```

Checkpoint : aucun test ne régresse.

Étape 6 : Passer la chaîne QA

Gitrust impose trois gates obligatoires avant toute PR. Exécutez-les dans cet ordre :

Gate 1 — Formatage :

```
cargo fmt --all -- --check
```

Si des différences apparaissent, appliquez le format automatiquement :

```
cargo fmt --all
```

Gate 2 — Linting (zéro warning) :

```
cargo clippy --workspace -- -D warnings
```

Sortie attendue :

© Gitrust

```
    Checking gitrust-core v0.1.0
    ...
    Finished `dev` profile
```

Gate 3 — Tests complets :

```
cargo test --workspace
```

Sortie attendue :

```
test result: ok. XXX passed; 0 failed; 0 ignored
```

Checkpoint : les trois commandes se terminent sans erreur ni warning.

Étape 7 : Committer selon la convention

Gitrust utilise des messages de commit conventionnels (`feat:` , `fix:` , `test:` , `refactor:` , `docs:`). Commitez votre travail :

```
git add crates/gitrust-core/src/models/repository.rs
git commit -m "fix: rejeter les slugs commençant par un tiret (issue #42)"
```

> *Ne committez **jamais** avec `git add -A` sans avoir inspecté `git diff --staged` au préalable. Vérifiez qu'aucun fichier `.env` , `target/` , ou fichier généré n'est inclus.*

Checkpoint : `git log --oneline -3` montre votre commit en tête.

Étape 8 : Pousser et ouvrir une PR

Poussez votre branche vers l'origine :

```
git push origin fix/42-repo-slug-rejects-leading-dash
```

Rendez-vous sur la plateforme gitrust. Un bandeau « Vous venez de pousser une branche — ouvrir une PR ? » devrait apparaître. Sinon, naviguez vers `/{votre-username}/gitrust/pulls/new` .

© Gitrust

Remplissez le formulaire :

- **Titre** : `fix: rejeter les slugs commençant par un tiret`
- **Corps** : Renseignez le contexte, la cause du bug, la solution choisie, et ajoutez `Closes #42` pour lier automatiquement l'issue.
- **Branche source** : `fix/42-repo-slug-rejects-leading-dash`
- **Branche cible** : `main`

Cliquez sur « Ouvrir la pull request ».

Checkpoint : la PR apparaît dans la liste `/{owner}/gitrust/pulls` avec le statut `open`.

Récapitulatif

- **O1 accompli** en identifiant l'issue #42 et en créant la branche `fix/42-repo-slug-rejects-leading-dash`.
- **O2 accompli** en écrivant `repo_slug_rejects_leading_dash` qui échouait avant l'ajout de `s.starts_with('-')`.
- **O3 accompli** en passant `cargo fmt`, `cargo clippy -- -D warnings` et `cargo test --workspace` sans erreur, puis en ouvrant la PR liée à l'issue.

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`cargo clippy` signale `needless_pass_by_value`	Votre nouvelle fonction prend une `String` alors qu'un `&str` suffit	Changez le paramètre en `&str`
`cargo test` échoue sur un test non lié	Votre modification a cassé un invariant adjacent	Relisez les tests voisins ; revenez à un diff minimal
`git push` refusé avec `remote: pre-receive hook declined`	La branche est protégée ou votre token SSH n'est pas enregistré	Vérifiez `/settings/keys` sur la plateforme

Prochaine étape

→ Tutoriel 03 — Créer un worker async (capstone)

© Gitrust

1.2.3 Créer un worker asynchrone de bout en bout (capstone)

Objectifs

À la fin de ce tutoriel, vous saurez :

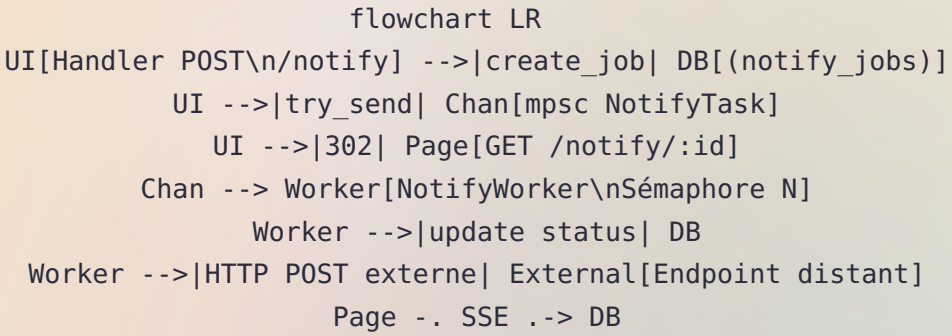
- O1. Concevoir le modèle de données SeaORM pour une table de jobs avec une state machine à 5 états.
- O2. Implémenter la boucle d'un worker Tokio avec sémaphore de concurrence et timeout.
- O3. Tester l'ensemble du cycle (enqueue → exécution → SSE) avec des tests d'intégration hermétiques.

Pré-requis

- **Technique** : tutoriels 01 et 02 complétés ; notions de Tokio (async/await, `mpsc`, `spawn`) ; PostgreSQL local ou testcontainer disponible.
- **Pédagogique** : tutoriel `02-premiere-contribution` complété.
- **Temps estimé** : ~120 minutes.

Vue d'ensemble

Ce tutoriel est un **capstone** : vous combinez des recettes connues (migration, service, handler) sans guide pas-à-pas détaillé. L'objectif est de reproduire le patron du worker d'import (`import_service` + `import_worker`) pour un cas générique appelé `notify_worker` — un worker qui envoie des notifications externes de manière asynchrone.



Le patron est identique à celui utilisé pour l'import de dépôts externe (voir `crates/gitrust-core/src/services/import_service.rs`) et pour le worker CI. Vous n'inventez rien : vous **combinez** des pièces existantes.

> ***Modèle mental*** *: pensez à une file de caissiers de supermarché. La file d'attente (`mpsc channel`) accumule les clients (tâches). Le nombre de caisses ouvertes est le sémaphore. Chaque caissier (worker goroutine) traite un client et met à jour le tableau d'affichage (DB + SSE).*

© Gitrust

Étape 1 : Concevoir la state machine

Avant d'écrire une ligne de code, formalisez la state machine de votre job.

```
stateDiagram-v2
    [*] --> pending : create_job()
    pending --> running : worker démarre
    pending --> cancelled : POST /cancel
    running --> success : HTTP 200 reçu
    running --> failed : erreur ou timeout
    running --> cancelled : flag DB checked
```

Les 5 états sont : `pending`, `running`, `success`, `failed`, `cancelled`.

Règles de transition : - Seul `pending` peut être annulé avant démarrage. - `running` peut être annulé via un flag en DB vérifié par le worker. - `success` et `failed` sont terminaux.

Checkpoint : dessinez la state machine sur papier avant de continuer. Répondez à : « que se passe-t-il si le serveur redémarre pendant un job `running` ? » (réponse : au redémarrage, les jobs `running` doivent être marqués `failed` avec le message « server restarted »).

Étape 2 : Créer la migration SeaORM

Créez le fichier de migration :

```
crates/gitrust-core/src/migrations/m20260501_000023_create_notify_jobs.rs
```

```
use sea_orm_migration::prelude::*;

#[derive(DeriveMigrationName)]
pub struct Migration;

#[async_trait::async_trait]
impl MigrationTrait for Migration {
    async fn up(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .create_table(
                Table::create()
                    .table(NotifyJobs::Table)
                    .if_not_exists()
                    .col(ColumnDef::new(NotifyJobs::Id).uuid().not_null().primary_key()
                    .col(ColumnDef::new(NotifyJobs::OwnerId).uuid().not_null())
```


© Gitrust

```
                    .col(ColumnDef::new(NotifyJobs::TargetUrl).text().not_null())
                    .col(ColumnDef::new(NotifyJobs::Payload).text().not_null())
                    .col(
                        ColumnDef::new(NotifyJobs::Status)
                            .string()
                            .not_null()
                            .default("pending"),
                    )
                    .col(ColumnDef::new(NotifyJobs::ErrorMessage).text().null())
                    .col(ColumnDef::new(NotifyJobs::StartedAt).timestamp_with_time_zone
                    .col(ColumnDef::new(NotifyJobs::FinishedAt).timestamp_with_time_zon
                    .col(
                        ColumnDef::new(NotifyJobs::CreatedAt)
                            .timestamp_with_time_zone()
                            .not_null(),
                    )
                    .col(
                        ColumnDef::new(NotifyJobs::UpdatedAt)
                            .timestamp_with_time_zone()
                            .not_null(),
                    )
                    .to_owned(),
            )
            .await?;

        manager
            .create_index(
                Index::create()
                    .table(NotifyJobs::Table)
                    .name("notify_jobs_owner_status_idx")
                    .col(NotifyJobs::OwnerId)
                    .col(NotifyJobs::Status)
                    .to_owned(),
            )
            .await
    }

    async fn down(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .drop_table(Table::drop().table(NotifyJobs::Table).to_owned())
            .await
    }
}
```


© Gitrust

```
#[derive(DeriveIden)]
enum NotifyJobs {
    Table,
    Id,
    OwnerId,
    TargetUrl,
    Payload,
    Status,
    ErrorMessage,
    StartedAt,
    FinishedAt,
    CreatedAt,
    UpdatedAt,
}
```

Enregistrez la migration dans `crates/gitrust-core/src/migrations/mod.rs` en ajoutant `m20260501_000023_create_notify_jobs::Migration` à la liste du `Migrator`.

Test rouge à faire passer :

```
#[tokio::test]
async fn migration_creates_notify_jobs_table() {
    let db = setup_test_db().await;
    // Si la migration s'est appliquée, la requête ne renvoie pas d'erreur.
    let count: i64 = db
        .query_one(Statement::from_string(
            DatabaseBackend::Postgres,
            "SELECT COUNT(*) FROM notify_jobs".to_owned(),
        ))
        .await
        .expect("La table notify_jobs doit exister")
        .unwrap()
        .try_get("", "count")
        .unwrap();
    assert_eq!(count, 0);
}
```

Checkpoint : `cargo test --package gitrust-core migration_creates_notify_jobs_table` passe au vert.

© Gitrust

Étape 3 : Définir les DTOs et l'enum de statut

Créez `crates/gitrust-core/src/dto/notify_dto.rs` :

```
use serde::{Deserialize, Serialize};

/// Statut d'un job de notification — state machine à 5 états.
#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
#[serde(rename_all = "snake_case")]
pub enum NotifyStatus {
    Pending,
    Running,
    Success,
    Failed,
    Cancelled,
}

impl NotifyStatus {
    pub fn as_str(&self) -> &'static str {
        match self {
            Self::Pending => "pending",
            Self::Running => "running",
            Self::Success => "success",
            Self::Failed => "failed",
            Self::Cancelled => "cancelled",
        }
    }

    pub fn from_str(s: &str) -> Option<Self> {
        match s {
            "pending" => Some(Self::Pending),
            "running" => Some(Self::Running),
            "success" => Some(Self::Success),
            "failed" => Some(Self::Failed),
            "cancelled" => Some(Self::Cancelled),
            _ => None,
        }
    }
}

/// Payload envoyé via SSE au client.
#[derive(Debug, Serialize)]
pub struct NotifyJobSse {
    pub status: String,
```


© Gitrust

```
    pub error_message: Option<String>,
}

/// Input pour créer un job.
#[derive(Debug, Deserialize)]
pub struct CreateNotifyJobInput {
    pub target_url: String,
    pub payload: String,
}
```

Tests round-trip obligatoires :

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn status_round_trip() {
        for (s, expected) in [
            ("pending", NotifyStatus::Pending),
            ("running", NotifyStatus::Running),
            ("success", NotifyStatus::Success),
            ("failed", NotifyStatus::Failed),
            ("cancelled", NotifyStatus::Cancelled),
        ] {
            let status = NotifyStatus::from_str(s).expect("doit parser");
            assert_eq!(status, expected);
            assert_eq!(status.as_str(), s);
        }
    }

    #[test]
    fn unknown_status_returns_none() {
        assert!(NotifyStatus::from_str("unknown").is_none());
    }
}
```

Checkpoint : `cargo test --package gitrust-core status_round_trip` passe.

© Gitrust

Étape 4 : Implémenter le service

Créez `crates/gitrust-core/src/services/notify_service.rs`. Le service ne fait que orchestrer la DB — aucune logique métier IO ici :

```
use chrono::Utc;
use sea_orm::{ActiveModelTrait, ActiveValue::Set, DatabaseConnection, EntityTrait};
use uuid::Uuid;

use crate::{
    dto::notify_dto::{CreateNotifyJobInput, NotifyStatus},
    error::GitrustError,
    models::notify_job,
};

pub struct NotifyService;

impl NotifyService {
    /// Crée un job en statut `pending` et retourne son ID.
    pub async fn create_job(
        db: &DatabaseConnection,
        owner_id: Uuid,
        input: CreateNotifyJobInput,
    ) -> Result<Uuid, GitrustError> {
        let id = Uuid::new_v4();
        let now = Utc::now();
        let model = notify_job::ActiveModel {
            id: Set(id),
            owner_id: Set(owner_id),
            target_url: Set(input.target_url),
            payload: Set(input.payload),
            status: Set(NotifyStatus::Pending.as_str().to_owned()),
            error_message: Set(None),
            started_at: Set(None),
            finished_at: Set(None),
            created_at: Set(now),
            updated_at: Set(now),
        };
        model.insert(db).await.map_err(GitrustError::Database)?;
        Ok(id)
    }

    /// Passe le job en `running`.
    pub async fn mark_running(db: &DatabaseConnection, id: Uuid) -> Result<(), GitrustE
```


© Gitrust

```
        Self::update_status(db, id, NotifyStatus::Running, None).await
    }

    /// Passe le job en `success`.
    pub async fn mark_success(db: &DatabaseConnection, id: Uuid) -> Result<(), GitrustE
        Self::update_status(db, id, NotifyStatus::Success, None).await
    }

    /// Passe le job en `failed` avec un message d'erreur.
    pub async fn mark_failed(
        db: &DatabaseConnection,
        id: Uuid,
        error: &str,
    ) -> Result<(), GitrustError> {
        Self::update_status(db, id, NotifyStatus::Failed, Some(error)).await
    }

    /// Passe le job en `cancelled`.
    pub async fn mark_cancelled(db: &DatabaseConnection, id: Uuid) -> Result<(), Gitrus
        Self::update_status(db, id, NotifyStatus::Cancelled, None).await
    }

    /// Vérifie si le job a été annulé (pour le poll du worker).
    pub async fn is_cancelled(
        db: &DatabaseConnection,
        id: Uuid,
    ) -> Result<bool, GitrustError> {
        let job = notify_job::Entity::find_by_id(id)
            .one(db)
            .await
            .map_err(GitrustError::Database)?
            .ok_or_else(|| GitrustError::NotFound("job introuvable".into()))?;
        Ok(job.status == NotifyStatus::Cancelled.as_str())
    }

    async fn update_status(
        db: &DatabaseConnection,
        id: Uuid,
        status: NotifyStatus,
        error: Option<&str>,
    ) -> Result<(), GitrustError> {
        use sea_orm::ActiveModelTrait;
        let now = Utc::now();
```


© Gitrust

```
        let mut model: notify_job::ActiveModel = notify_job::Entity::find_by_id(id)
            .one(db)
            .await
            .map_err(GitrustError::Database)?
            .ok_or_else(|| GitrustError::NotFound("job introuvable".into()))?
            .into();

        model.status = Set(status.as_str().to_owned());
        model.updated_at = Set(now);
        model.error_message = Set(error.map(str::to_owned));

        if status == NotifyStatus::Running {
            model.started_at = Set(Some(now));
        }
        if matches!(status, NotifyStatus::Success | NotifyStatus::Failed | NotifyStatus
            model.finished_at = Set(Some(now));
        }

        model.update(db).await.map_err(GitrustError::Database)?;
        Ok(())
    }
}
```

Étape 5 : Implémenter le worker

Le worker tourne dans un `tokio::spawn` séparé. Il lit les tâches depuis un canal `mpsc` et les exécute avec un sémaphore de concurrence :

```
// crates/gitrust-core/src/services/notify_worker.rs

use std::sync::Arc;

use sea_orm::DatabaseConnection;
use tokio::sync::{mpsc, Semaphore};
use uuid::Uuid;

use super::notify_service::NotifyService;

/// Tâche envoyée via le canal mpsc au worker.
pub struct NotifyTask {
    pub job_id: Uuid,
    pub target_url: String,
```


© Gitrust

```
    pub payload: String,
}

pub struct NotifyWorkerConfig {
    /// Nombre maximal de jobs concurrents.
    pub max_concurrent: usize,
    /// Timeout en secondes par job.
    pub timeout_secs: u64,
}

/// Boucle principale du worker. À appeler avec `tokio::spawn`.
pub async fn run(
    db: DatabaseConnection,
    mut rx: mpsc::Receiver<NotifyTask>,
    config: NotifyWorkerConfig,
) {
    let semaphore = Arc::new(Semaphore::new(config.max_concurrent));

    while let Some(task) = rx.recv().await {
        let permit = semaphore.clone().acquire_owned().await.expect("semaphore fermé");
        let db = db.clone();
        let timeout = config.timeout_secs;

        tokio::spawn(async move {
            let _permit = permit; // libéré à la fin du bloc
            execute_task(&db, task, timeout).await;
        });
    }
}

async fn execute_task(db: &DatabaseConnection, task: NotifyTask, timeout_secs: u64) {
    // Vérifier si annulé avant de démarrer
    match NotifyService::is_cancelled(db, task.job_id).await {
        Ok(true) => return,
        Err(e) => {
            tracing::error!(job_id = %task.job_id, error = %e, "Erreur lecture job");
            return;
        }
        Ok(false) => {}
    }

    if let Err(e) = NotifyService::mark_running(db, task.job_id).await {
        tracing::error!(job_id = %task.job_id, error = %e, "Impossible de marquer runni
```


© Gitrust

```
            return;
        }

        let result = tokio::time::timeout(
            std::time::Duration::from_secs(timeout_secs),
            send_notification(&task.target_url, &task.payload),
        )
        .await;

        match result {
            Ok(Ok(())) => {
                if let Err(e) = NotifyService::mark_success(db, task.job_id).await {
                    tracing::error!(job_id = %task.job_id, error = %e, "Impossible de marqu
                }
            }
            Ok(Err(e)) => {
                let msg = format!("Erreur HTTP: {e}");
                if let Err(db_err) = NotifyService::mark_failed(db, task.job_id, &msg).awai
                    tracing::error!(job_id = %task.job_id, error = %db_err, "Impossible de
                }
            }
            Err(_elapsed) => {
                let msg = format!("Timeout après {timeout_secs}s");
                if let Err(e) = NotifyService::mark_failed(db, task.job_id, &msg).await {
                    tracing::error!(job_id = %task.job_id, error = %e, "Impossible de marqu
                }
            }
        }
    }

    async fn send_notification(url: &str, payload: &str) -> Result<(), reqwest::Error> {
        let client = reqwest::Client::new();
        client
            .post(url)
            .header("Content-Type", "application/json")
            .body(payload.to_owned())
            .send()
            .await?
            .error_for_status()?;
        Ok(())
    }
```

Checkpoint : `cargo build --package gitrust-core` compile sans erreur.

© Gitrust

Étape 6 : Ajouter le handler et la route SSE

Dans `crates/gitrust-web/src/handlers/notify.rs`, créez le handler POST qui enqueues le job et redirige, ainsi que le handler SSE :

```
use axum::{
    extract::{Path, State},
    response::{sse::{Event, KeepAlive, Sse}, Redirect},
    Extension, Form,
};
use sea_orm::DatabaseConnection;
use tokio::time::{interval, Duration};
use tokio_stream::wrappers::IntervalStream;
use tokio_stream::StreamExt;
use uuid::Uuid;

use gitrust_core::{
    dto::notify_dto::CreateNotifyJobInput,
    services::{notify_service::NotifyService, notify_worker::NotifyTask},
};

use crate::auth::AuthUser;

pub type NotifySender = tokio::sync::mpsc::Sender<NotifyTask>;

/// POST /notify — crée un job et redirige vers la page de suivi.
pub async fn create_notify_job(
    State(db): State<DatabaseConnection>,
    Extension(tx): Extension<NotifySender>,
    user: AuthUser,
    Form(input): Form<CreateNotifyJobInput>,
) -> Result<Redirect, crate::error::AppError> {
    let job_id = NotifyService::create_job(&db, user.user_id, input.clone()).await?;

    // Essai d'envoi dans le canal — si plein, le job reste `pending` jusqu'au
    // prochain redémarrage (les jobs `pending` peuvent être re-envoyés au démarrage).
    let _ = tx.try_send(NotifyTask {
        job_id,
        target_url: input.target_url,
        payload: input.payload,
    });

    Ok(Redirect::to(&format!("/notify/{job_id}")))
}
```


© Gitrust

```
/// GET /notify/:id/stream — SSE : pousse les mises à jour de statut au client.
pub async fn notify_job_stream(
    State(db): State<DatabaseConnection>,
    Path(id): Path<Uuid>,
) -> Sse<impl tokio_stream::Stream<Item = Result<Event, std::convert::Infallible>>> {
    let stream = IntervalStream::new(interval(Duration::from_secs(1)))
        .map(move |_| {
            let db = db.clone();
            async move {
                // Poll la DB pour l'état courant
                use gitrust_core::models::notify_job;
                use sea_orm::EntityTrait;
                let job = notify_job::Entity::find_by_id(id)
                    .one(&db)
                    .await
                    .ok()
                    .flatten();

                let data = match job {
                    Some(j) => format!(
                        r#"{{"status":"{}","error":{}}}"#,
                        j.status,
                        j.error_message
                            .as_deref()
                            .map(|e| format!(r#""{}""#, e.replace('"', "\\\"")))
                            .unwrap_or("null".to_owned())
                    ),
                    None => r#"{"status":"not_found"}"#.to_owned(),
                };

                Ok(Event::default().data(data))
            }
        })
        .then(|fut| fut);

    Sse::new(stream).keep_alive(KeepAlive::default())
}
```

Enregistrez les routes dans `routes.rs` :

© Gitrust

```
.route("/notify", post(handlers::notify::create_notify_job))
.route("/notify/:id", get(handlers::notify::notify_job_page))
.route("/notify/:id/stream", get(handlers::notify::notify_job_stream))
.route("/notify/:id/cancel", post(handlers::notify::cancel_notify_job))
```

Étape 7 : Intégrer le worker dans `main.rs`

Dans `crates/gitrust-web/src/main.rs`, initialisez le canal et démarrez le worker :

```
// Dans la fonction main(), après la construction de l'app :
let (notify_tx, notify_rx) = tokio::sync::mpsc::channel::<NotifyTask>(100);

let notify_db = db.clone();
tokio::spawn(gitrust_core::services::notify_worker::run(
    notify_db,
    notify_rx,
    NotifyWorkerConfig {
        max_concurrent: std::env::var("NOTIFY_MAX_CONCURRENT")
            .unwrap_or("4".into())
            .parse()
            .unwrap_or(4),
        timeout_secs: std::env::var("NOTIFY_TIMEOUT_SECS")
            .unwrap_or("30".into())
            .parse()
            .unwrap_or(30),
    },
));

// Injecter l'émetteur dans le router via Extension
let app = routes::routes(&static_path)
    .layer(Extension(notify_tx))
    .with_state(db);
```

Étape 8 : Tests d'intégration

Écrivez au minimum ces trois tests dans `crates/gitrust-core/src/services/notify_service_test.rs` :

```
#[tokio::test]
async fn create_job_inserts_with_pending_status() {
    let db = setup_test_db().await;
    let id = NotifyService::create_job(
        &db,
```


© Gitrust

```
        Uuid::new_v4(),
        CreateNotifyJobInput {
            target_url: "https://example.com/hook".into(),
            payload: r#"{"event":"test"}"#.into(),
        },
    )
    .await
    .expect("create_job doit réussir");

    let job = notify_job::Entity::find_by_id(id)
        .one(&db)
        .await
        .unwrap()
        .unwrap();
    assert_eq!(job.status, "pending");
    assert!(job.started_at.is_none());
}

#[tokio::test]
async fn mark_running_sets_started_at() {
    let db = setup_test_db().await;
    let id = create_test_job(&db).await;

    NotifyService::mark_running(&db, id).await.unwrap();

    let job = notify_job::Entity::find_by_id(id).one(&db).await.unwrap().unwrap();
    assert_eq!(job.status, "running");
    assert!(job.started_at.is_some());
}

#[tokio::test]
async fn cancel_before_running_is_terminal() {
    let db = setup_test_db().await;
    let id = create_test_job(&db).await;

    NotifyService::mark_cancelled(&db, id).await.unwrap();

    let is_cancelled = NotifyService::is_cancelled(&db, id).await.unwrap();
    assert!(is_cancelled);
}
```

Checkpoint : `cargo test --package gitrust-core notify` — tous les tests passent.

© Gitrust

Récapitulatif

- **O1 accompli** en concevant la table `notify_jobs` avec 5 états (pending/running/success/failed/cancelled) et la migration SeaORM correspondante.
- **O2 accompli** en implémentant `notify_worker::run` avec `Semaphore`, `tokio::time::timeout`, et lecture du flag d'annulation via la DB.
- **O3 accompli** en écrivant les tests `create_job_inserts_with_pending_status`, `mark_running_sets_started_at` et `cancel_before_running_is_terminal`, tous hermétiques (pas de réseau réel).

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`channel closed` au démarrage	Le `Receiver` est droppé avant le spawn du worker	Vérifiez que `tokio::spawn(run(..., notify_rx, ...))` est appelé **avant** que `notify_rx` ne soit droppé
Le job reste `pending` indéfiniment	Le canal est plein ou le worker n'est pas démarré	Ajoutez un log dans `execute_task` ; vérifiez que `tokio::spawn` est bien appelé dans `main`
SSE ne reçoit aucun événement	Le `Content-Type: text/event-stream` n'est pas reconnu	Vérifiez que le handler retourne bien `Sse<...>` et non `String`

Prochaine étape

Vous êtes désormais **core contributor**. Consultez les how-to pour les recettes quotidiennes :

- Ajouter une route web
- Passer la QA avant merge
- Comprendre le patron worker async

© Gitrust

1.3 Guides pratiques

1.3.1 Contribuer via le workflow Git et pull request

Ce guide décrit le workflow Git utilisé par le projet gitrust pour toute contribution : nouvelle fonctionnalité, correctif, ou documentation.

Pré-requis

- Accès en lecture/écriture au dépôt (ou à votre fork).
- Environnement de développement opérationnel (voir `tutorials/01-getting-started.md`).

Créer une branche depuis main

Toujours partir d'un `main` à jour :

```
git switch main
git pull --rebase origin main
git switch -c <type>/<numéro>-<description-courte>
```

Convention de nommage des branches :

Préfixe	Usage
`feat/`	Nouvelle fonctionnalité
`fix/`	Correctif de bug
`refactor/`	Refactorisation sans changement de comportement
`docs/`	Documentation uniquement
`test/`	Tests uniquement

Exemples : `feat/42-webhook-dispatch`, `fix/88-slug-leading-dash`, `docs/15-api-rest`.

Conventions de commit

Gitrust suit les **Conventional Commits**. Le format est :

© Gitrust

```
<type>(<scope optionnel>): <description en impératif>

[corps optionnel]

[footer : Closes #N, BREAKING CHANGE, etc.]
```

Types acceptés : `feat`, `fix`, `refactor`, `test`, `docs`, `chore`, `perf`.

Exemples valides :

```
feat(webhooks): ajouter la signature HMAC-SHA256 sur les livraisons
fix(slug): rejeter les slugs commençant par un tiret (Closes #42)
test(import): ajouter test hermétique pour le timeout worker
```

Règle des petits commits atomiques : un commit = un changement logique cohérent. Évitez les commits « WIP » ou « fix fix fix » — utilisez `git commit --amend` ou `git rebase -i` pour réécrire l'historique avant de pousser.

Rebase avant push

Avant de pousser votre branche, rebasez-la sur `main` pour obtenir un historique linéaire :

```
git fetch origin
git rebase origin/main
```

En cas de conflit, résolvez-les fichier par fichier, puis :

```
git add <fichier-résolu>
git rebase --continue
```

Ne fusionnez **jamais** `main` dans votre branche de feature (`git merge main`) — cela crée des commits de merge parasites.

© Gitrust

Checklist pre-merge (QA obligatoire)

Avant d'ouvrir la PR, vérifiez chaque point :

- [ ] `cargo fmt --all -- --check` — zéro diff de formatage
- [ ] `cargo clippy --workspace -- -D warnings` — zéro warning
- [ ] `cargo test --workspace` — 100 % de tests passants
- [ ] `npm run test:e2e` — tests Playwright passants (si templates ou routes modifiés)
- [ ] CSS rebuild : `npx tailwindcss -i static/css/input.css -o static/css/style.css --minify` (si templates modifiés)
- [ ] Aucun secret, token ou clé en clair dans le code ou les logs
- [ ] Aucune modification dans `crates/rustwarden-core/` (framework read-only)

Voir la checklist complète dans `passer-la-qa-avant-merge.md`.

Ouvrir la pull request

Poussez votre branche :

```
git push origin <votre-branche>
```

Sur la plateforme gitrust, naviguez vers `/{owner}/gitrust/pulls/new` et remplissez :

- **Titre** : message de commit principal (convention Conventional Commits).
- **Corps** : contexte du problème, solution choisie, alternatives considérées, lien vers l'issue (`Closes #N`).
- **Reviewers** : assignez au moins un reviewer.
- **Labels** : appliquez les labels de classification appropriés.

Critères d'approbation

Un reviewer approuve quand :

1. Tous les gates QA passent (CI verte).
2. La logique métier est correcte et les cas limites sont couverts par des tests.
3. Aucun `.unwrap()` , `.expect()` , ou `panic!()` non justifié.
4. Le diff ne contient pas de changements de périmètre non demandés.

Stratégie de merge par défaut

Gitrust utilise le **merge commit** pour conserver la traçabilité des branches dans l'historique. Le squash est réservé aux PRs de correction mineure (1-2 commits). Le rebase-merge est proscrit car il réécrit les SHA des commits existants.

© Gitrust

Après le merge

Supprimez votre branche locale et distante :

```
git switch main
git pull --rebase origin main
git branch -d <votre-branche>
git push origin --delete <votre-branche>
```

Vérifiez que l'issue liée est bien fermée automatiquement par le `Closes #N` dans le message de commit ou le corps de la PR.

© Gitrust

1.3.2 Lancer les tests unitaires

Référence des commandes et de la stratégie de tests pour gitrust.

Infrastructure existante

Tests unitaires Rust

```
cargo test --workspace          # Tous les tests
cargo test -p gitrust-core      # Un crate spécifique
cargo test -p gitrust-git       # Tests Git (tempdir)
```

Les tests unitaires sont intégrés dans chaque crate via `#[cfg(test)]`.

Tests e2e Playwright

17 specs couvrant : auth, admin, repos, issues, labels, teams, tokens, settings, navigation.

```
# Pré-requis (première fois)
npm install
./scripts/e2e-setup-db.sh       # Crée la DB gitrust_test

# Lancer les tests
npm run test:e2e                 # Mode headless
npm run test:e2e:ui              # Mode interactif (navigateur visible)
npm run test:e2e:debug           # Mode debug
npm run test:e2e:report          # Voir le rapport HTML
```

Configuration : `playwright.config.ts` (port 4001, DB `gitrust_test`, locale `fr-FR`).

Structure des répertoires

```
tests/
  e2e/                    # Specs Playwright (TypeScript)
    global-setup.ts       # Setup : lancer l'app, créer admin
    global-teardown.ts    # Teardown : arrêter l'app
    fixtures.ts           # Fixtures Playwright partagées
    auth.spec.ts          # Tests authentification
    admin.spec.ts         # Tests administration
    issues.spec.ts        # Tests issues
    ...
  integration/            # Tests d'intégration Rust (DB réelle) [à créer]
```


© Gitrust

```
  fixtures/              # Fichiers de test (repos, configs) [à créer]
  seeds/                 # Seeds Rust pour initialiser la DB de test
    mod.rs
    seed.rs
```

Tests à ajouter

1. Tests fonctionnels API REST (intégration Rust)

Tests avec une DB PostgreSQL réelle pour vérifier les endpoints API.

Emplacement : `tests/integration/` ou `crates/gitrust-web/tests/`

Approche recommandée : utiliser `reqwest` + un serveur de test :

```
// tests/integration/api_test.rs
use reqwest::Client;

#[tokio::test]
#[ignore] // Nécessite une DB + serveur running
async fn test_api_login() {
    let client = Client::new();
    let res = client.post("http://localhost:4001/api/v1/auth/login")
        .json(&serde_json::json!({
            "login": "admin",
            "password": "test_password"
        }))
        .send()
        .await
        .unwrap();
    assert_eq!(res.status(), 200);
}
```

Alternative : `axum::test` pour tester les handlers sans serveur HTTP :

```
use axum::body::Body;
use axum::http::Request;
use tower::ServiceExt;

#[tokio::test]
async fn test_health_check() {
```


© Gitrust

```
    let app = create_test_app().await; // Build router with test DB
    let response = app
        .oneshot(Request::builder().uri("/api/hello").body(Body::empty()).unwrap())
        .await
        .unwrap();
    assert_eq!(response.status(), 200);
}
```

2. Tests fonctionnels CI

Test	Description	Priorité
CI detection	Repo avec `.gitrust-ci.yml` -> Easy, `.dagger/` -> Power, rien -> None	Haute
Pipeline CRUD	Créer, lister, mettre à jour, annuler un pipeline	Haute
Config CI	Activer/désactiver CI, modifier triggers, vérifier effet	Haute
Variables héritage	Team var + repo var -> merge correct	Moyenne
Auto-cancel	Nouveau push annule les pipelines en cours	Moyenne
Logs streaming	Append logs + lecture paginée	Moyenne
Notifications CI	Pipeline échoue -> notification créée	Moyenne

Pour les tests CI : mocker `dagger` avec un script shell qui simule success/failure :

```
#!/bin/bash
# tests/fixtures/mock-dagger.sh
echo "Step 1: Building..."
echo "Step 2: Testing..."
if [ "$MOCK_FAIL" = "true" ]; then
    echo "FAILED" >&2
    exit 1
fi
echo "All checks passed"
exit 0
```

Configurer `CI_DAGGER_BIN=tests/fixtures/mock-dagger.sh` dans les tests.

© Gitrust

3. Tests e2e Playwright à ajouter

Nouvelles specs pour les fonctionnalités CI et notifications :

Spec	Couverture
`ci-pipelines.spec.ts`	Liste pipelines, détail, status badges
`ci-config.spec.ts`	Config CI (enable/disable, triggers, timeout)
`ci-variables.spec.ts`	CRUD variables CI, masquage secrets
`notifications.spec.ts`	Liste notifications, marquer lu, préférences
`api-docs.spec.ts`	Swagger UI accessible, spec chargée
`i18n.spec.ts`	Changement de langue, textes traduits
`docker-smoke.spec.ts`	`docker compose up` + smoke test

Exemple de spec CI :

```
// tests/e2e/ci-pipelines.spec.ts
import { test, expect } from './fixtures';

test.describe('CI Pipelines', () => {
  test('affiche la page CI vide', async ({ authenticatedPage }) => {
    await authenticatedPage.goto('/admin/mon-repo/ci');
    await expect(authenticatedPage.locator('text=Aucun pipeline')).toBeVisible();
  });

  test('bouton trigger CI visible pour le maintainer', async ({ authenticatedPage }) =>
    await authenticatedPage.goto('/admin/mon-repo/ci');
    await expect(authenticatedPage.locator('text=Lancer CI')).toBeVisible();
  });
});
```


© Gitrust

4. Tests settings ↔ comportement

Les tests les plus critiques vérifient que **changer un setting modifie le comportement observable** :

Setting	Action	Vérification
PAT révoqué	Appel API	401 immédiatement
`ci_enabled = false`	Push	Pas de pipeline créé
`trigger_on_push = false`	Push	Pas de pipeline
`auto_cancel = true`	2 pushes rapides	1er pipeline annulé
`email_on_pipeline_failure = false`	Pipeline échoue	Pas d'email, notif in-app ok
`DEFAULT_LOCALE = en`	Charger page	Texte en anglais

Commandes de test

```
# Tests unitaires Rust
cargo test --workspace

# Tests avec filtrage
cargo test -p gitrust-core -- ci_service
cargo test -p gitrust-core -- notification

# Tests e2e Playwright
npm run test:e2e

# Tests e2e spécifiques
npx playwright test tests/e2e/ci-pipelines.spec.ts
npx playwright test --grep "notifications"

# Lint
cargo clippy --workspace -- -D warnings

# Format check
cargo fmt --all -- --check
```


© Gitrust

CI locale (exécuter avant un push)

```
# Script rapide de vérification pré-push
cargo fmt --all -- --check && \
cargo clippy --workspace -- -D warnings && \
cargo test --workspace && \
echo "All checks passed"
```


© Gitrust

1.3.3 Lancer les tests end-to-end (E2E)

Les tests E2E de gitrust utilisent **Playwright** pour piloter un navigateur Chromium contre une instance de l'application configurée avec une base de données et un répertoire de dépôts dédiés.

Pré-requis

- Node.js (déjà présent pour Tailwind/DaisyUI)
- PostgreSQL accessible en local (Docker via `database/docker-compose.yml`)
- `psql`, `createdb`, `dropdb` disponibles dans le PATH
- Chromium installé via Playwright (`npx playwright install chromium`)

Commandes

```
# Créer (ou remettre à zéro) la base de données de test
npm run test:e2e:setup

# Lancer tous les tests E2E
npm run test:e2e

# Mode debug interactif (inspecteur Playwright, pas-à-pas)
npm run test:e2e:debug

# Mode UI (interface graphique Playwright)
npm run test:e2e:ui

# Consulter le rapport HTML après exécution
npm run test:e2e:report
```

Architecture

```
.env.test                          # Variables d'environnement de test
playwright.config.ts               # Configuration Playwright

scripts/
  e2e-setup-db.sh                  # Crée la DB gitrust_test + nettoie depot_test
  e2e-server.sh                    # Démarre l'app avec .env.test

tests/e2e/
  fixtures.ts                      # Fixtures partagées (loginAs, credentials)
  global-setup.ts                  # Seed : crée les utilisateurs via /register
```


© Gitrust

```
  global-teardown.ts          # Nettoyage post-tests
  auth.spec.ts                # Tests d'authentification
  repository.spec.ts          # Tests de gestion des dépôts
  admin.spec.ts               # Tests du panneau d'administration
  navigation.spec.ts          # Tests de navigation et UI
```

Isolation de l'environnement de test

Ressource	Développement	Tests E2E
Base de données	`gitrust`	`gitrust_test`
Port HTTP	`4000`	`4001`
Port SSH	`2222`	`2223`
Dépôts Git	`./data/repos/`	`./depot_test/`
Rate limiting	Standard	Relâché (×20)
Validation email	Activée	Désactivée

Fonctionnement

1. `npm run test:e2e:setup` supprime puis recrée la base `gitrust_test` et vide `depot_test/`.
2. **Playwright** démarre automatiquement l'application via `scripts/e2e-server.sh`, qui charge les variables de `.env.test` avant d'exécuter `cargo run`.
3. Le **global-setup** crée les utilisateurs de test (alice, bob, charlie) en passant par le formulaire `/register` — approche 100% E2E, aucune insertion directe en base.
4. Les **tests** s'exécutent séquentiellement (un seul worker) pour éviter les conflits sur la DB partagée.
5. Le **global-teardown** est exécuté à la fin de la session.

Fixtures

Le fichier `fixtures.ts` fournit une fixture `loginAs` réutilisable dans tous les tests :

```
import { test, expect } from './fixtures';

test('mon test', async ({ page, loginAs }) => {
  await loginAs('alice');
  // alice est connectée, on peut naviguer
});
```


© Gitrust

Utilisateurs disponibles : `admin`, `alice`, `bob`, `charlie`.

Liste des tests

`auth.spec.ts` — Authentification

Test	Description
login avec des identifiants valides	Vérifie le login standard et la redirection vers `/dashboard`
login avec un mauvais mot de passe	Vérifie l'affichage d'un message d'erreur
login avec un utilisateur inexistant	Vérifie le rejet d'un username inconnu
logout déconnecte l'utilisateur	Vérifie la déconnexion et l'impossibilité d'accéder au dashboard
accès au dashboard sans login redirige vers login	Vérifie la protection des routes authentifiées
option remember me est présente	Vérifie la présence de la checkbox remember_me
formulaire d'inscription est accessible	Vérifie la présence de tous les champs du formulaire
inscription avec username déjà pris échoue	Vérifie l'unicité des usernames
inscription avec mots de passe différents échoue	Vérifie la validation de confirmation du mot de passe

`repository.spec.ts` — Gestion des dépôts

Test	Description
créer un dépôt avec nom et description	Crée un dépôt complet et vérifie la redirection
créer un dépôt sans description	Vérifie que la description est optionnelle
créer un dépôt avec un nom déjà pris échoue	Vérifie l'unicité des slugs par utilisateur
le formulaire de création nécessite un login	Vérifie la protection de la route `/new`
le dashboard liste les dépôts de l'utilisateur	Vérifie l'affichage des dépôts sur le dashboard
un dépôt public est accessible sans login	Vérifie l'accès anonyme aux dépôts publics

© Gitrust

`admin.spec.ts` — Administration

Test	Description
l'admin peut accéder au panneau d'administration	Vérifie l'accès admin à `/admin`
un utilisateur normal ne peut pas accéder à l'admin	Vérifie le contrôle d'accès
lister les utilisateurs	Vérifie l'affichage de la liste des utilisateurs
créer un nouvel utilisateur via l'admin	Teste le formulaire de création d'utilisateur admin
créer un utilisateur avec un username déjà pris échoue	Vérifie la gestion des doublons côté admin
les liens du menu admin sont présents	Vérifie la navigation du panneau admin

`navigation.spec.ts` — Navigation et UI

Test	Description
la page de login est accessible	Vérifie le rendu de la page de login
la navbar est présente après login	Vérifie la présence de la barre de navigation
le sidebar est présent après login	Vérifie la présence du menu latéral
le lien vers le dashboard fonctionne	Vérifie la navigation vers le dashboard
le lien vers la création de dépôt fonctionne	Vérifie le lien `/new`
la page utilise le thème DaisyUI	Vérifie la présence de l'attribut `data-theme`

Ajouter un nouveau test

1. Créer un fichier `tests/e2e/mon-module.spec.ts`
2. Importer les fixtures : `import { test, expect } from './fixtures';`
3. Utiliser `loginAs` pour les tests nécessitant une authentification
4. Lancer avec `npm run test:e2e` ou cibler le fichier : `npx playwright test mon-module`

© Gitrust

1.3.4 Ajouter une route web (handler axum + template Askama)

Ce guide explique comment ajouter un endpoint HTTP SSR complet dans gitrust : route, handler, template Askama, et test E2E Playwright.

Pré-requis

- Familiarité avec le tutoriel `01-getting-started.md`.
- Notions de base d'Axum (extracteurs, `State`, `IntoResponse`).

Vue d'ensemble du flux

```
routes.rs → handler (handlers/*.rs) → service (gitrust-core) → template Askama
```

Gitrust suit le patron **vertical slice** : chaque route traverse toutes les couches. Ne créez jamais un handler sans template, ni un service sans route.

Étape 1 : Déclarer la route dans `routes.rs`

Ouvrez `crates/gitrust-web/src/routes.rs`. Ajoutez votre route dans la section qui correspond à son contexte (routes authentifiées, routes repo, routes admin…) :

```
// Exemple : GET /settings/notifications
.route(
    "/settings/notifications",
    get(handlers::notifications::preferences_form)
        .post(handlers::notifications::preferences_submit),
)
```

La signature de `routes.rs` utilise `Router<DatabaseConnection>` — l'état partagé est toujours la connexion DB injectée via `State`.

Étape 2 : Créer le handler

Créez ou complétez `crates/gitrust-web/src/handlers/mon_handler.rs` :

```
use axum::{extract::State, response::IntoResponse};
use sea_orm::DatabaseConnection;

use crate::{
    auth::AuthUser,
```


© Gitrust

```
    error::AppError,
    templates::MonPageTemplate,
};
use gitrust_core::services::mon_service::MonService;

/// GET /settings/mon-endpoint
pub async fn mon_form(
    State(db): State<DatabaseConnection>,
    user: AuthUser,
) -> Result<impl IntoResponse, AppError> {
    let data = MonService::get_data(&db, user.user_id).await?;

    Ok(MonPageTemplate {
        username: user.username.clone(),
        is_admin: user.has_role("admin"),
        current_path: "/settings/mon-endpoint".to_owned(),
        data,
    })
}

/// POST /settings/mon-endpoint
pub async fn mon_submit(
    State(db): State<DatabaseConnection>,
    user: AuthUser,
    axum::Form(input): axum::Form<MonInput>,
) -> Result<impl IntoResponse, AppError> {
    MonService::update_data(&db, user.user_id, input).await?;
    Ok(axum::response::Redirect::to("/settings/mon-endpoint"))
}
```

Règles obligatoires :

- Pas de `.unwrap()` ni `.expect()` — toutes les erreurs remontent via `?` vers `AppError`.
- `AppError` implémente `IntoResponse` et mappe les variantes vers les codes HTTP appropriés.
- `AuthUser` (extracteur rustwarden-core) rejette automatiquement les requêtes non authentifiées avec `401`.

Étape 3 : Déclarer le module handler

Dans `crates/gitrust-web/src/handlers/mod.rs`, ajoutez :

```
pub mod mon_handler;
```


© Gitrust

Étape 4 : Créer la struct de template

Dans `crates/gitrust-web/src/templates.rs`, ajoutez la struct Askama :

```
#[derive(Template)]
#[template(path = "settings/mon_endpoint.html")]
pub struct MonPageTemplate {
    pub username: String,
    pub is_admin: bool,
    pub current_path: String,
    pub data: MonData, // votre DTO métier
}
```

Les champs `username`, `is_admin`, et `current_path` sont présents sur **toutes** les structs de template — ils alimentent la sidebar contextuelle.

Étape 5 : Créer le template Askama

Créez `crates/gitrust-web/templates/settings/mon_endpoint.html` :

```
{% extends "base.html" %}

{% block title %}Mon endpoint — gitrust{% endblock %}

{% block content %}
<div class="container mx-auto px-4 py-8">
  <h1 class="text-2xl font-bold mb-6">Mon endpoint</h1>

  <form method="POST" action="/settings/mon-endpoint">
    <input type="hidden" name="_csrf" value="{{ csrf_token }}">

    <div class="form-control mb-4">
      <label class="label">
        <span class="label-text">Valeur</span>
      </label>
      <input
        type="text"
        name="valeur"
        value="{{ data.valeur }}"
        class="input input-bordered"
        required
      >
    </div>
```


© Gitrust

```
    <button type="submit" class="btn btn-primary">Enregistrer</button>
  </form>
</div>
{% endblock %}
```

Règle CSRF : tout formulaire `POST` doit inclure `<input type="hidden" name="_csrf" value="{{ csrf_token }}">`. Le middleware rustwarden-core rejette les requêtes sans token CSRF valide avec `403`.

Étape 6 : Gestion des erreurs dans les templates

Pour afficher les erreurs de validation à l'utilisateur, utilisez le pattern flash :

```
// Dans le handler POST, en cas d'erreur de validation :
return Ok(axum::response::Redirect::to(
    "/settings/mon-endpoint?error=valeur+invalide"
));
```

Dans le template :

```
{% if let Some(err) = query_params.error %}
  <div class="alert alert-error">{{ err }}</div>
{% endif %}
```

Étape 7 : Écrire les tests E2E Playwright

Créez `tests/e2e/mon_endpoint.spec.ts` :

```
import { test, expect } from "@playwright/test";

test("la page mon-endpoint est accessible après login", async ({ page }) => {
  // Authentification (helper existant)
  await page.goto("/login");
  await page.fill("[name=username]", "alice");
  await page.fill("[name=password]", "SecurePass123!");
  await page.click("[type=submit]");

  await page.goto("/settings/mon-endpoint");
  await expect(page).toHaveTitle(/Mon endpoint/);
});
```


© Gitrust

```
  test("soumettre le formulaire redirige vers la même page", async ({ page }) => {
    // ... login ...
    await page.goto("/settings/mon-endpoint");
    await page.fill("[name=valeur]", "nouvelle-valeur");
    await page.click("[type=submit]");
    await expect(page).toHaveURL("/settings/mon-endpoint");
  });
```

Lancez les tests E2E :

```
npm run test:e2e
```

Exemple complet : la page de gestion des labels

Pour un exemple réel de ce patron appliqué à une feature complète, consultez :

- Route : `crates/gitrust-web/src/routes.rs` — section Issues / Labels
- Handler : `crates/gitrust-web/src/handlers/labels.rs`
- Template : `crates/gitrust-web/templates/repository/labels.html`
- Service : `crates/gitrust-core/src/services/label_service.rs`
- Tests E2E : `tests/e2e/labels.spec.ts`

Voir aussi

- Ajouter un service métier
- Référence rustwarden-core API

© Gitrust

1.3.5 Ajouter un service métier dans gitrust-core

Ce guide couvre la création d'un nouveau service dans `crates/gitrust-core/src/services/`, en respectant les conventions du projet.

Pré-requis

- Lecture de `reference/services-api-interne.md` pour comprendre les patterns existants.
- Migration de base de données créée si nécessaire (voir `ajouter-migration-db.md`).

Convention générale

Tous les services gitrust-core suivent ce patron uniforme :

```
Handler (axum) → Service (logique métier) → SeaORM (base de données)
```

Règles structurelles :

- Les services sont des **structs sans état** (pas de champs, uniquement des méthodes `async fn` associées).
- Le premier paramètre de chaque méthode est toujours `db: &DatabaseConnection`.
- Le type de retour est toujours `Result<T, GitrustError>`.
- Aucun `unwrap()` / `expect()` / `panic!()` en production.
- Aucun SQL brut — tout passe par SeaORM.

Créer le fichier du service

Créez `crates/gitrust-core/src/services/mon_service.rs` :

```
use sea_orm::{
    ActiveModelTrait, ActiveValue::Set, DatabaseConnection, EntityTrait,
};
use uuid::Uuid;

use crate::{
    dto::mon_dto::{CreateMonInput, MonOutput},
    error::GitrustError,
    models::mon_model,
};

pub struct MonService;

impl MonService {
```


© Gitrust

```
    /// Crée un nouvel élément.
    ///
    /// # Erreurs
    /// - `GitrustError::Validation` si le nom est vide ou trop long.
    /// - `GitrustError::Conflict` si un élément avec le même nom existe déjà.
    /// - `GitrustError::Database` pour toute erreur SeaORM.
    pub async fn create(
        db: &DatabaseConnection,
        owner_id: Uuid,
        input: CreateMonInput,
    ) -> Result<mon_model::Model, GitrustError> {
        // 1. Validation aux frontières
        if input.name.is_empty() || input.name.len() > 64 {
            return Err(GitrustError::Validation(
                "Le nom doit comporter entre 1 et 64 caractères".into(),
            ));
        }

        // 2. Vérification unicité
        let existing = mon_model::Entity::find()
            .filter(mon_model::Column::OwnerId.eq(owner_id))
            .filter(mon_model::Column::Name.eq(&input.name))
            .one(db)
            .await
            .map_err(GitrustError::Database)?;

        if existing.is_some() {
            return Err(GitrustError::Conflict(format!(
                "Un élément nommé '{}' existe déjà",
                input.name
            )));
        }

        // 3. Insertion
        let now = chrono::Utc::now();
        let model = mon_model::ActiveModel {
            id: Set(Uuid::new_v4()),
            owner_id: Set(owner_id),
            name: Set(input.name),
            description: Set(input.description),
            created_at: Set(now),
            updated_at: Set(now),
        };
```


© Gitrust

```
    model.insert(db).await.map_err(GitrustError::Database)
}

/// Liste les éléments d'un propriétaire.
pub async fn list_by_owner(
    db: &DatabaseConnection,
    owner_id: Uuid,
) -> Result<Vec<mon_model::Model>, GitrustError> {
    mon_model::Entity::find()
        .filter(mon_model::Column::OwnerId.eq(owner_id))
        .order_by_asc(mon_model::Column::Name)
        .all(db)
        .await
        .map_err(GitrustError::Database)
}

/// Trouve un élément par ID et vérifie l'appartenance.
///
/// # Erreurs
/// - `GitrustError::NotFound` si l'ID est inconnu.
/// - `GitrustError::Forbidden` si `owner_id` ne correspond pas (anti-IDOR).
pub async fn find_by_id(
    db: &DatabaseConnection,
    id: Uuid,
    owner_id: Uuid,
) -> Result<mon_model::Model, GitrustError> {
    let model = mon_model::Entity::find_by_id(id)
        .one(db)
        .await
        .map_err(GitrustError::Database)?
        .ok_or_else(|| GitrustError::NotFound("Élément introuvable".into()))?;

    // Anti-IDOR : vérification d'ownership systématique
    if model.owner_id != owner_id {
        return Err(GitrustError::Forbidden);
    }

    Ok(model)
}

/// Supprime un élément (vérifie l'ownership).
pub async fn delete(
```


© Gitrust

```
        db: &DatabaseConnection,
        id: Uuid,
        owner_id: Uuid,
    ) -> Result<(), GitrustError> {
        let model = Self::find_by_id(db, id, owner_id).await?;
        mon_model::Entity::delete_by_id(model.id)
            .exec(db)
            .await
            .map_err(GitrustError::Database)?;
        Ok(())
    }
}
```

Déclarer le module dans `services/mod.rs`

Ouvrez `crates/gitrust-core/src/services/mod.rs` et ajoutez :

```
pub mod mon_service;
```

Injecter le service dans un handler via `State`

Les services gitrust ne sont pas des structs instanciées — leurs méthodes sont associées (pas de `self`). Vous n'avez pas besoin de les injecter dans le `State` d'Axum. Il suffit d'appeler la méthode statique depuis le handler :

```
use gitrust_core::services::mon_service::MonService;

pub async fn list_handler(
    State(db): State<DatabaseConnection>,
    user: AuthUser,
) -> Result<impl IntoResponse, AppError> {
    let items = MonService::list_by_owner(&db, user.user_id).await?;
    Ok(MonListTemplate { items, username: user.username, is_admin: user.has_role("admin
}
```

Écrire les tests unitaires

Gitrust impose des **tests sur vraie DB** (pas de mocks) pour la couche persistance. Utilisez une base SQLite in-memory pour les tests unitaires rapides, ou testcontainers PostgreSQL pour les tests qui dépendent de comportements PG spécifiques (contraintes, index partiels, etc.).

© Gitrust

```
#[cfg(test)]
mod tests {
    use super::*;
    use sea_orm::{Database, DatabaseConnection};

    async fn setup_db() -> DatabaseConnection {
        let db = Database::connect("sqlite::memory:").await.unwrap();
        // Appliquer les migrations nécessaires
        crate::migrations::run_migrations(&db).await.unwrap();
        db
    }

    #[tokio::test]
    async fn create_inserts_model() {
        let db = setup_db().await;
        let owner = Uuid::new_v4();

        let result = MonService::create(
            &db,
            owner,
            CreateMonInput {
                name: "mon-element".into(),
                description: None,
            },
        )
        .await;

        assert!(result.is_ok());
        let model = result.unwrap();
        assert_eq!(model.name, "mon-element");
        assert_eq!(model.owner_id, owner);
    }

    #[tokio::test]
    async fn create_rejects_empty_name() {
        let db = setup_db().await;
        let err = MonService::create(
            &db,
            Uuid::new_v4(),
            CreateMonInput { name: "".into(), description: None },
        )
        .await
        .unwrap_err();
```


© Gitrust

```
        assert!(matches!(err, GitrustError::Validation(_)));
    }

    #[tokio::test]
    async fn find_by_id_rejects_wrong_owner() {
        let db = setup_db().await;
        let owner = Uuid::new_v4();
        let other = Uuid::new_v4();

        let model = MonService::create(
            &db,
            owner,
            CreateMonInput { name: "priv".into(), description: None },
        )
        .await
        .unwrap();

        // Un autre utilisateur ne doit pas accéder à cet élément
        let err = MonService::find_by_id(&db, model.id, other).await.unwrap_err();
        assert!(matches!(err, GitrustError::Forbidden));
    }
}
```

Gestion des erreurs — variantes `GitrustError`

Variante	Code HTTP	Quand l'utiliser
`GitrustError::NotFound(msg)`	404	Ressource introuvable par ID
`GitrustError::Validation(msg)`	400	Input utilisateur invalide
`GitrustError::Forbidden`	403	Ownership non vérifiée (anti-IDOR)
`GitrustError::Conflict(msg)`	409	Contrainte d'unicité violée
`GitrustError::Database(err)`	500	Erreur SeaORM remontée telle quelle

N'utilisez jamais `GitrustError::Database` pour une erreur de validation — la distinction est importante pour les messages d'erreur utilisateur.

© Gitrust

Voir aussi

- Ajouter une migration DB
- Référence services API interne
- Ajouter une route web

© Gitrust

1.3.6 Ajouter une migration de base de données (SeaORM)

Ce guide couvre la création, l'application et le rollback d'une migration SeaORM dans gitrust.

Pré-requis

- Environnement local fonctionnel (voir `tutorials/01-getting-started.md`).
- PostgreSQL local démarré.

Convention de nommage

Chaque fichier de migration suit le format :

```
m{AAAAMMJJ}_{NNNNNN}_{description_snake_case}.rs
```

- `AAAAMMJJ` : date du jour en UTC.
- `NNNNNN` : numéro séquentiel sur 6 chiffres, incrémenté par rapport au dernier fichier existant.
- `description_snake_case` : description courte en minuscules.

Exemple : `m20260501_000023_create_notify_jobs.rs`

Consultez le tableau des migrations existantes dans `reference/schema-base-donnees.md` avant d'attribuer un numéro pour éviter les collisions.

Structure d'un fichier de migration

Créez `crates/gitrust-core/src/migrations/m20260501_000023_create_notify_jobs.rs` :

```
use sea_orm_migration::prelude::*;

/// Nom de migration dérivé automatiquement du nom du module.
#[derive(DeriveMigrationName)]
pub struct Migration;

#[async_trait::async_trait]
impl MigrationTrait for Migration {
    async fn up(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .create_table(
                Table::create()
                    .table(NotifyJobs::Table)
                    .if_not_exists()
```


© Gitrust

```
                    .col(ColumnDef::new(NotifyJobs::Id).uuid().not_null().primary_key()
                    .col(ColumnDef::new(NotifyJobs::OwnerId).uuid().not_null())
                    .col(
                        ColumnDef::new(NotifyJobs::Status)
                            .string()
                            .not_null()
                            .default("pending"),
                    )
                    .col(ColumnDef::new(NotifyJobs::ErrorMessage).text().null())
                    .col(
                        ColumnDef::new(NotifyJobs::CreatedAt)
                            .timestamp_with_time_zone()
                            .not_null(),
                    )
                    .col(
                        ColumnDef::new(NotifyJobs::UpdatedAt)
                            .timestamp_with_time_zone()
                            .not_null(),
                    )
                    .to_owned(),
            )
            .await?;

        // Index composite pour les requêtes fréquentes
        manager
            .create_index(
                Index::create()
                    .table(NotifyJobs::Table)
                    .name("notify_jobs_owner_status_idx")
                    .col(NotifyJobs::OwnerId)
                    .col(NotifyJobs::Status)
                    .to_owned(),
            )
            .await
    }

    async fn down(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .drop_table(Table::drop().table(NotifyJobs::Table).to_owned())
            .await
    }
}
```


© Gitrust

```
/// Identifiant Iden pour chaque colonne et la table.
#[derive(DeriveIden)]
enum NotifyJobs {
    Table,
    Id,
    OwnerId,
    Status,
    ErrorMessage,
    CreatedAt,
    UpdatedAt,
}
```

Règles obligatoires :

- `up()` doit toujours utiliser `.if_not_exists()` pour être idempotent.
- `down()` doit annuler exactement ce que `up()` a créé. Toute migration sans `down()` fonctionnel sera refusée en review.
- Les colonnes `created_at` et `updated_at` sont `TIMESTAMPTZ NOT NULL` (pas `TIMESTAMP` — gitrust est UTC strict).
- Les clés primaires sont des `UUID` (jamais des entiers auto-incrémentés).

Enregistrer la migration dans le Migrator

Ouvrez `crates/gitrust-core/src/migrations/mod.rs` et ajoutez la migration dans la liste du `Migrator` **en respectant l'ordre chronologique** :

```
pub mod m20260501_000023_create_notify_jobs;

pub struct Migrator;

#[async_trait::async_trait]
impl MigratorTrait for Migrator {
    fn migrations() -> Vec<Box<dyn MigrationTrait>> {
        vec![
            // ... migrations existantes dans l'ordre ...
            Box::new(m20260501_000023_create_notify_jobs::Migration),
        ]
    }
}
```

L'ordre est strict : SeaORM exécute les migrations dans l'ordre de la liste. Ne réorganisez jamais les migrations existantes.

© Gitrust

Appliquer la migration en local

```
cargo run --bin gitrust -- migrate
```

Sortie attendue :

```
Applying migration 'm20260501_000023_create_notify_jobs'
Migration applied successfully
```

Vérifiez la table dans PostgreSQL :

```
psql $DATABASE_URL -c "\d notify_jobs"
```

Tester le rollback

```
cargo run --bin gitrust -- migrate down
```

Sortie attendue :

```
Rolling back migration 'm20260501_000023_create_notify_jobs'
Rollback applied successfully
```

Vérifiez que la table a bien disparu :

```
psql $DATABASE_URL -c "\dt notify_jobs"
# → doit retourner "no relations found"
```

Cas d'usage courants

Ajouter une colonne à une table existante

```
async fn up(&self, manager: &SchemaManager) -> Result<(), DbErr> {
    manager
        .alter_table(
            Table::alter()
                .table(Repositories::Table)
                .add_column(
```


© Gitrust

```
                    ColumnDef::new(Repositories::IsArchived)
                        .boolean()
                        .not_null()
                        .default(false),
                )
                .to_owned(),
        )
        .await
}

async fn down(&self, manager: &SchemaManager) -> Result<(), DbErr> {
    manager
        .alter_table(
            Table::alter()
                .table(Repositories::Table)
                .drop_column(Repositories::IsArchived)
                .to_owned(),
        )
        .await
}
```

Créer un index unique composite

```
manager
    .create_index(
        Index::create()
            .table(Labels::Table)
            .name("labels_owner_repo_name_type_idx")
            .col(Labels::OwnerId)
            .col(Labels::RepositoryId)
            .col(Labels::Name)
            .col(Labels::LabelType)
            .unique()
            .to_owned(),
    )
    .await
```


© Gitrust

Gotchas PostgreSQL

Situation	Comportement	Solution
Ajouter une colonne `NOT NULL` sans `DEFAULT` sur une table peuplée	Erreur PG : `column cannot be added without a default value`	Toujours fournir un `DEFAULT` lors de l'ajout d'une colonne NOT NULL
`ALTER TABLE ... DROP COLUMN` avec des FK dépendantes	Erreur PG : contrainte FK bloquante	Supprimer d'abord les FK avec `ForeignKey::drop()` dans `down()`
`TIMESTAMPTZ` vs `TIMESTAMP`	`TIMESTAMP` ignore le fuseau horaire, crée des bugs sur les serveurs non-UTC	Utiliser toujours `timestamp_with_time_zone()`
Migration dans une transaction	Toutes les migrations gitrust s'exécutent dans une transaction implicite	Ne pas appeler `BEGIN` / `COMMIT` manuellement dans `up()`

Voir aussi

- Ajouter un service métier
- Schéma base de données

© Gitrust

1.3.7 Implémenter des webhooks dans gitrust

Ce guide couvre l'ajout d'un nouveau déclencheur webhook : modèle de données, service de dispatch, signature HMAC, configuration UI par dépôt, et rotation de secret.

Pré-requis

- Lecture de `ajouter-service-metier.md` et `ajouter-migration-db.md`.
- Compréhension du patron vertical slice (voir `tutorials/01-getting-started.md`).

Vue d'ensemble

```
sequenceDiagram
    participant Push as git push
    participant Hook as receive-pack handler
    participant Svc as WebhookService
    participant Disp as WebhookDispatcher
    participant Ext as Endpoint HTTP distant

    Push->>Hook: réception des objets
    Hook->>Svc: list_by_repo_and_event(repo_id, "push")
    Svc-->>Hook: Vec
    Hook->>Disp: deliver(webhook, "push", payload)
    Disp->>Ext: POST url (payload signé HMAC-SHA256)
    Ext-->>Disp: 200 OK
    Disp->>Svc: record_delivery(webhook_id, status, duration)
```

Modèle de données

La table `webhooks` stocke la configuration par dépôt :

```
CREATE TABLE webhooks (
    id           UUID PRIMARY KEY,
    repository_id UUID NOT NULL REFERENCES repositories(id) ON DELETE CASCADE,
    url          TEXT NOT NULL,
    secret_hash  TEXT NOT NULL,        -- HMAC secret haché SHA-256 (jamais en clair)
    events       TEXT NOT NULL,        -- JSON array : ["push","pull_request","issues"]
    is_active    BOOLEAN NOT NULL DEFAULT true,
    created_at   TIMESTAMPTZ NOT NULL,
    updated_at   TIMESTAMPTZ NOT NULL
);
```


© Gitrust

```
CREATE INDEX webhooks_repo_active_idx ON webhooks(repository_id, is_active);
```

La table `webhook_deliveries` trace chaque envoi :

```
CREATE TABLE webhook_deliveries (
    id           UUID PRIMARY KEY,
    webhook_id   UUID NOT NULL REFERENCES webhooks(id) ON DELETE CASCADE,
    event        TEXT NOT NULL,
    payload      TEXT NOT NULL,
    response_status INTEGER NULL,
    response_body   TEXT NULL,
    duration_ms  INTEGER NULL,
    success      BOOLEAN NOT NULL,
    created_at   TIMESTAMPTZ NOT NULL
);
```

Créez les deux migrations correspondantes (voir `ajouter-migration-db.md`).

WebhookService — méthodes publiques

```
// crates/gitrust-core/src/services/webhook_service.rs
pub struct WebhookService;

impl WebhookService {
    /// Crée un webhook pour un dépôt.
    pub async fn create(
        db: &DatabaseConnection,
        repo_id: Uuid,
        url: String,
        secret: &str,          // secret en clair — haché avant stockage
        events: Vec<String>,
    ) -> Result<webhook::Model, GitrustError>;

    /// Liste les webhooks actifs pour un dépôt et un événement donné.
    pub async fn list_by_repo_and_event(
        db: &DatabaseConnection,
        repo_id: Uuid,
        event: &str,
    ) -> Result<Vec<webhook::Model>, GitrustError>;
```


© Gitrust

```
    /// Met à jour l'URL, le secret ou les événements.
    pub async fn update(
        db: &DatabaseConnection,
        webhook_id: Uuid,
        repo_id: Uuid,           // anti-IDOR : vérifie l'appartenance
        url: Option<String>,
        new_secret: Option<&str>,
        events: Option<Vec<String>>,
    ) -> Result<webhook::Model, GitrustError>;

    /// Désactive / active un webhook.
    pub async fn set_active(
        db: &DatabaseConnection,
        webhook_id: Uuid,
        repo_id: Uuid,
        active: bool,
    ) -> Result<(), GitrustError>;

    /// Supprime un webhook et ses deliveries en cascade.
    pub async fn delete(
        db: &DatabaseConnection,
        webhook_id: Uuid,
        repo_id: Uuid,
    ) -> Result<(), GitrustError>;

    /// Enregistre le résultat d'une livraison.
    pub async fn record_delivery(
        db: &DatabaseConnection,
        webhook_id: Uuid,
        event: &str,
        payload: &str,
        response_status: Option<i32>,
        response_body: Option<&str>,
        duration_ms: i32,
        success: bool,
    ) -> Result<(), GitrustError>;

    /// Liste les dernières deliveries d'un webhook (pagination).
    pub async fn list_deliveries(
        db: &DatabaseConnection,
        webhook_id: Uuid,
        repo_id: Uuid,
        page: u64,
```


© Gitrust

```
        per_page: u64,
    ) -> Result<Vec<webhook_delivery::Model>, GitrustError>;
}
```

Sécurité — stockage du secret : le secret ne doit **jamais** être stocké en clair. Lors de la création ou mise à jour, hachez-le avec SHA-256 avant insertion. La signature HMAC utilise le secret en clair fourni par l'utilisateur au moment de la création (retourné une seule fois) ou récupéré depuis un secret manager.

```
fn hash_secret(secret: &str) -> String {
    use sha2::{Digest, Sha256};
    let mut hasher = Sha256::new();
    hasher.update(secret.as_bytes());
    format!("{:x}", hasher.finalize())
}
```

WebhookDispatcher — envoi avec retry et signature HMAC

```
// crates/gitrust-core/src/services/webhook_dispatcher.rs
pub struct WebhookDispatcher;

impl WebhookDispatcher {
    /// Envoie le payload à l'URL du webhook avec signature HMAC-SHA256.
    /// Retry exponentiel : 3 tentatives (0s, 5s, 25s).
    pub async fn deliver(
        webhook: &webhook::Model,
        event: &str,
        payload: serde_json::Value,
        secret_plain: &str,     // secret en clair pour la signature
    ) -> DeliveryResult;
}

pub struct DeliveryResult {
    pub success: bool,
    pub response_status: Option<i32>,
    pub response_body: Option<String>,
    pub duration_ms: i32,
    pub attempts: u32,
}
```


© Gitrust

Calcul de la signature HMAC

```
use hmac::{Hmac, Mac};
use sha2::Sha256;

fn compute_hmac_signature(secret: &str, body: &str) -> String {
    let mut mac = Hmac::<Sha256>::new_from_slice(secret.as_bytes())
        .expect("HMAC accepte toutes les longueurs de clé");
    mac.update(body.as_bytes());
    let result = mac.finalize();
    format!("sha256={}", hex::encode(result.into_bytes()))
}
```

Le header HTTP envoyé avec chaque livraison :

```
X-Gitrust-Signature-256: sha256=<hmac_hex>
X-Gitrust-Event: push
X-Gitrust-Delivery: <uuid de la delivery>
Content-Type: application/json
```

Retry exponentiel

```
const RETRY_DELAYS: [u64; 3] = [0, 5, 25]; // secondes

for (attempt, delay) in RETRY_DELAYS.iter().enumerate() {
    if *delay > 0 {
        tokio::time::sleep(Duration::from_secs(*delay)).await;
    }
    match client.post(&webhook.url).send().await {
        Ok(resp) if resp.status().is_success() => return Ok(DeliveryResult { success: t
        Ok(resp) => last_error = format!("HTTP {}", resp.status()),
        Err(e) => last_error = e.to_string(),
    }
}
```

Règle SSRF : l'URL du webhook est validée à la création. Les adresses RFC1918 (`10.0.0.0/8` , `172.16.0.0/12` , `192.168.0.0/16`) et `localhost` sont interdites en production.

© Gitrust

Injecter le dispatch dans le hook `post-receive`

Dans le handler `receive_pack` (`crates/gitrust-web/src/handlers/git_http.rs`), après la réception des objets Git :

```
// Déclencher les webhooks "push"
let payload = serde_json::json!({
    "repository": { "id": repo.id, "slug": repo.slug, "owner": owner.username },
    "pusher": { "username": user.username },
    "ref": pushed_ref,
    "commits": commit_summaries,
});

let webhooks = WebhookService::list_by_repo_and_event(&db, repo.id, "push").await?;
for webhook in webhooks {
    // Le dispatch est non-bloquant : on ne veut pas bloquer le push si le
    // serveur distant est lent.
    let db_clone = db.clone();
    let payload_clone = payload.clone();
    tokio::spawn(async move {
        let result = WebhookDispatcher::deliver(&webhook, "push", payload_clone, &secre
        let _ = WebhookService::record_delivery(
            &db_clone,
            webhook.id,
            "push",
            &payload_clone.to_string(),
            result.response_status,
            result.response_body.as_deref(),
            result.duration_ms,
            result.success,
        ).await;
    });
}
```

UI de configuration par dépôt

Ajoutez les routes dans `routes.rs` :

```
.route(
    "/{owner}/{repo}/settings/webhooks",
    get(handlers::webhooks::list_webhooks).post(handlers::webhooks::create_webhook),
)
.route(
    "/{owner}/{repo}/settings/webhooks/{id}",
```


© Gitrust

```
        get(handlers::webhooks::webhook_detail),
    )
    .route(
        "/{owner}/{repo}/settings/webhooks/{id}/edit",
        post(handlers::webhooks::edit_webhook),
    )
    .route(
        "/{owner}/{repo}/settings/webhooks/{id}/delete",
        post(handlers::webhooks::delete_webhook),
    )
    .route(
        "/{owner}/{repo}/settings/webhooks/{id}/test",
        post(handlers::webhooks::test_webhook),
    )
```

Rotation du secret

Quand un utilisateur régénère le secret via l'UI, le nouveau secret est :

1. Affiché **une seule fois** à l'utilisateur (jamais re-lisible depuis l'UI).
2. Haché SHA-256 et stocké dans `webhooks.secret_hash`.
3. Utilisé immédiatement pour les livraisons suivantes.

Les livraisons en cours avec l'ancien secret ne sont pas réessayées — informez l'utilisateur dans l'UI.

Checklist de test

- [ ] Créer un webhook avec URL + secret + événements → visible dans la liste.
- [ ] Push sur le dépôt → delivery créée avec statut `success` ou `failed`.
- [ ] Désactiver le webhook → push ne déclenche aucune livraison.
- [ ] URL RFC1918 → rejet à la création avec message d'erreur.
- [ ] Bouton « Test » → delivery avec événement `ping` créée.
- [ ] Rotation du secret → livraisons suivantes signées avec le nouveau secret.

Voir aussi

- Implémenter un endpoint API
- Référence git hooks post-receive

© Gitrust

1.3.8 Implémenter un endpoint REST `/api/v1/...`

Ce guide couvre l'ajout d'un nouvel endpoint JSON à l'API REST v1 de gitrust : modèle de requête/réponse, authentification PAT/JWT, codes d'erreur standardisés, et tests.

Pré-requis

- Lecture de `ajouter-service-metier.md`.
- Référence complète de l'API : `reference/api-rest-v1.md`.

Vue d'ensemble

Les handlers API et les handlers SSR **partagent les mêmes services**. La seule différence est le format de réponse :

```
Handler SSR → service → réponse HTML (template Askama)
Handler API → service → réponse JSON (serde_json)
```

Ne dupliquez jamais la logique métier entre un handler SSR et un handler API.

Structure des handlers API

Les handlers API se trouvent dans `crates/gitrust-web/src/handlers/api/`. Chaque domaine a son fichier :

```
handlers/api/
├── repos.rs      — dépôts, branches, commits
├── issues.rs     — issues et commentaires
├── pulls.rs      — pull requests
├── ci.rs         — pipelines CI
├── user.rs       — profil utilisateur courant
└── docs.rs       — Swagger UI
```

Étape 1 : Définir les types de requête et réponse

Créez ou complétez les DTOs dans `crates/gitrust-core/src/dto/` :

```
// crates/gitrust-core/src/dto/mon_api_dto.rs
use serde::{Deserialize, Serialize};
use uuid::Uuid;

/// Corps de la requête POST /api/v1/repos/{owner}/{repo}/mon-resource
#[derive(Debug, Deserialize)]
```


© Gitrust

```
pub struct CreateMonResourceRequest {
    pub name: String,
    pub description: Option<String>,
}

/// Réponse JSON pour un élément unique
#[derive(Debug, Serialize)]
pub struct MonResourceResponse {
    pub id: Uuid,
    pub name: String,
    pub description: Option<String>,
    pub created_at: chrono::DateTime<chrono::Utc>,
}

/// Réponse JSON pour une liste paginée
#[derive(Debug, Serialize)]
pub struct PaginatedResponse<T> {
    pub items: Vec<T>,
    pub total: u64,
    pub page: u64,
    pub per_page: u64,
}
```

Étape 2 : Déclarer la route dans `routes.rs`

Dans la fonction `api_v1_routes()` de `crates/gitrust-web/src/routes.rs` :

```
.route(
    "/api/v1/repos/{owner}/{repo}/mon-resource",
    get(handlers::api::mon_resource::list).post(handlers::api::mon_resource::create),
)
.route(
    "/api/v1/repos/{owner}/{repo}/mon-resource/{id}",
    get(handlers::api::mon_resource::detail)
        .patch(handlers::api::mon_resource::update)
        .delete(handlers::api::mon_resource::delete_resource),
)
```

Étape 3 : Implémenter le handler

© Gitrust

```
// crates/gitrust-web/src/handlers/api/mon_resource.rs
use axum::{
    extract::{Path, Query, State},
    http::StatusCode,
    response::IntoResponse,
    Json,
};
use sea_orm::DatabaseConnection;
use serde::Deserialize;

use crate::auth::AuthUser;
use gitrust_core::{
    dto::mon_api_dto::{CreateMonResourceRequest, MonResourceResponse, PaginatedResponse
    services::mon_service::MonService,
};

#[derive(Deserialize)]
pub struct PaginationParams {
    #[serde(default = "default_page")]
    pub page: u64,
    #[serde(default = "default_per_page")]
    pub per_page: u64,
}

fn default_page() -> u64 { 1 }
fn default_per_page() -> u64 { 30 }

/// GET /api/v1/repos/{owner}/{repo}/mon-resource
pub async fn list(
    State(db): State<DatabaseConnection>,
    Path((owner, repo)): Path<(String, String)>,
    Query(pagination): Query<PaginationParams>,
    user: AuthUser,
) -> impl IntoResponse {
    let per_page = pagination.per_page.min(100); // cap à 100

    match MonService::list_paginated(
        &db, &owner, &repo, user.user_id, pagination.page, per_page,
    )
    .await
    {
        Ok((items, total)) => {
            let response = PaginatedResponse {
```


© Gitrust

```
                items: items.into_iter().map(MonResourceResponse::from).collect(),
                total,
                page: pagination.page,
                per_page,
            };
            let mut headers = axum::http::HeaderMap::new();
            headers.insert("X-Total-Count", total.to_string().parse().unwrap());
            headers.insert("X-Page", pagination.page.to_string().parse().unwrap());
            headers.insert("X-Per-Page", per_page.to_string().parse().unwrap());
            (StatusCode::OK, headers, Json(response)).into_response()
        }
        Err(e) => api_error(e).into_response(),
    }
}

/// POST /api/v1/repos/{owner}/{repo}/mon-resource
pub async fn create(
    State(db): State<DatabaseConnection>,
    Path((owner, repo)): Path<(String, String)>,
    user: AuthUser,
    Json(body): Json<CreateMonResourceRequest>,
) -> impl IntoResponse {
    match MonService::create_for_repo(&db, &owner, &repo, user.user_id, body.into()).aw
        Ok(model) => (StatusCode::CREATED, Json(MonResourceResponse::from(model))).into
        Err(e) => api_error(e).into_response(),
    }
}
```

Authentification : PAT et JWT Bearer

L'extracteur `AuthUser` supporte deux modes d'authentification :

1. **Cookie JWT** (`jwt_token`) — utilisé par le navigateur web.
2. **Header** `Authorization: Bearer <token>` — utilisé par les clients API avec un PAT ou un JWT.

Les Personal Access Tokens sont vérifiés via `PatService::validate_token()` dans le middleware. Aucune modification n'est nécessaire côté handler — `AuthUser` gère les deux méthodes transparentement.

Codes de retour en cas d'échec d'auth : - `401 Unauthorized` — token absent, expiré, ou invalide. Réponse JSON, jamais de redirect HTML. - `403 Forbidden` — token valide mais permissions insuffisantes.

© Gitrust

Format des erreurs JSON standardisé

Toutes les erreurs API retournent le même format :

```
{
    "error": "not_found",
    "message": "Le dépôt 'alice/myrepo' est introuvable",
    "status": 404
}
```

La fonction utilitaire `api_error` à placer dans `handlers/api/mod.rs` :

```
use axum::{http::StatusCode, response::IntoResponse, Json};
use serde_json::json;
use gitrust_core::error::GitrustError;

pub fn api_error(err: GitrustError) -> impl IntoResponse {
    let (status, code) = match &err {
        GitrustError::NotFound(_) => (StatusCode::NOT_FOUND, "not_found"),
        GitrustError::Validation(_) => (StatusCode::BAD_REQUEST, "validation_error"),
        GitrustError::Forbidden => (StatusCode::FORBIDDEN, "forbidden"),
        GitrustError::Conflict(_) => (StatusCode::CONFLICT, "conflict"),
        _ => (StatusCode::INTERNAL_SERVER_ERROR, "internal_error"),
    };
    (
        status,
        Json(json!({
            "error": code,
            "message": err.to_string(),
            "status": status.as_u16()
        })),
    )
}
```

Pagination

Tous les endpoints de liste supportent `?page=N&per_page=N` (défaut : `page=1`, `per_page=30`, max `per_page=100`).

Headers de réponse :

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


© Gitrust

```
X-Per-Page: 30
Link: <https://demo.gitrust.eu/api/v1/repos/alice/myrepo/issues?page=1&per_page=30>; re
      <https://demo.gitrust.eu/api/v1/repos/alice/myrepo/issues?page=3&per_page=30>; re
```

Rate limiting

Les endpoints API sont couverts par le middleware de rate limiting de rustwarden-core :

Catégorie	Limite	Fenêtre
Endpoints lecture	1000 req	1 heure
Endpoints écriture	200 req	1 heure
Trigger CI manuel	10 req	1 heure

En cas de dépassement : `429 Too Many Requests` avec header `Retry-After: <secondes>`.

Documenter l'endpoint (OpenAPI)

Ajoutez les annotations utoipa sur le handler :

```
#[utoipa::path(
    get,
    path = "/api/v1/repos/{owner}/{repo}/mon-resource",
    params(
        ("owner" = String, Path, description = "Propriétaire du dépôt"),
        ("repo" = String, Path, description = "Slug du dépôt"),
        ("page" = Option<u64>, Query, description = "Numéro de page (défaut: 1)"),
        ("per_page" = Option<u64>, Query, description = "Éléments par page (défaut: 30,
    ),
    responses(
        (status = 200, description = "Liste paginée", body = PaginatedResponse<MonResou
        (status = 401, description = "Non authentifié"),
        (status = 403, description = "Accès refusé"),
        (status = 404, description = "Dépôt introuvable"),
    ),
    security(("bearer_token" = []), ("cookie_auth" = []))
)]
pub async fn list(...) { ... }
```


© Gitrust

Tester l'endpoint

Test curl rapide :

```
# Avec un PAT
curl -H "Authorization: Bearer <votre-pat>" \
     https://demo.gitrust.eu/api/v1/repos/alice/myrepo/mon-resource

# Vérifier le format d'erreur sur auth invalide
curl -H "Authorization: Bearer token-invalide" \
     https://demo.gitrust.eu/api/v1/repos/alice/myrepo/mon-resource
# → {"error":"unauthorized","message":"Token invalide","status":401}
```

Test d'intégration Rust :

```
#[tokio::test]
async fn list_returns_401_without_auth() {
    let app = setup_test_app().await;
    let response = app
        .oneshot(
            Request::builder()
                .uri("/api/v1/repos/alice/myrepo/mon-resource")
                .body(Body::empty())
                .unwrap(),
        )
        .await
        .unwrap();

    assert_eq!(response.status(), StatusCode::UNAUTHORIZED);
    let body: serde_json::Value = parse_json_body(response).await;
    assert_eq!(body["error"], "unauthorized");
}

#[tokio::test]
async fn create_returns_201_with_valid_pat() {
    let app = setup_test_app().await;
    let pat = create_test_pat(&app).await;

    let response = app
        .oneshot(
            Request::builder()
                .method("POST")
                .uri("/api/v1/repos/alice/myrepo/mon-resource")
```


© Gitrust

```
                .header("Authorization", format!("Bearer {pat}"))
                .header("Content-Type", "application/json")
                .body(Body::from(r#"{"name":"test"}"#))
                .unwrap(),
        )
        .await
        .unwrap();

    assert_eq!(response.status(), StatusCode::CREATED);
}
```

Voir aussi

- Référence API REST v1
- Implémenter des webhooks
- Passer la QA avant merge

© Gitrust

1.3.9 Importer un dépôt externe

Référence du flux d'import de dépôts externes dans gitrust : rôle de la base de données, schéma de séquence complet et pistes d'optimisation.

1. Rôle de la base de données

Pendant un import, la DB sert **quatre fonctions distinctes** :

Rôle	Table	Fréquence
File d'attente persistante	`import_jobs`	1 INSERT à la création
Suivi d'état pour l'UI	`import_jobs`	1 UPDATE toutes les ~1.5 s
Enregistrement final du dépôt	`repositories` + `resources`	2 INSERT à la fin
Journal d'audit	`audit_log`	1 INSERT à la fin

Le clone lui-même ne passe **jamais** par la DB : seuls les métadonnées et compteurs de progression y transitent. Les objets Git sont écrits directement sur disque dans `{GIT_REPOS_BASE_PATH}/{owner}/{slug}.git/`.

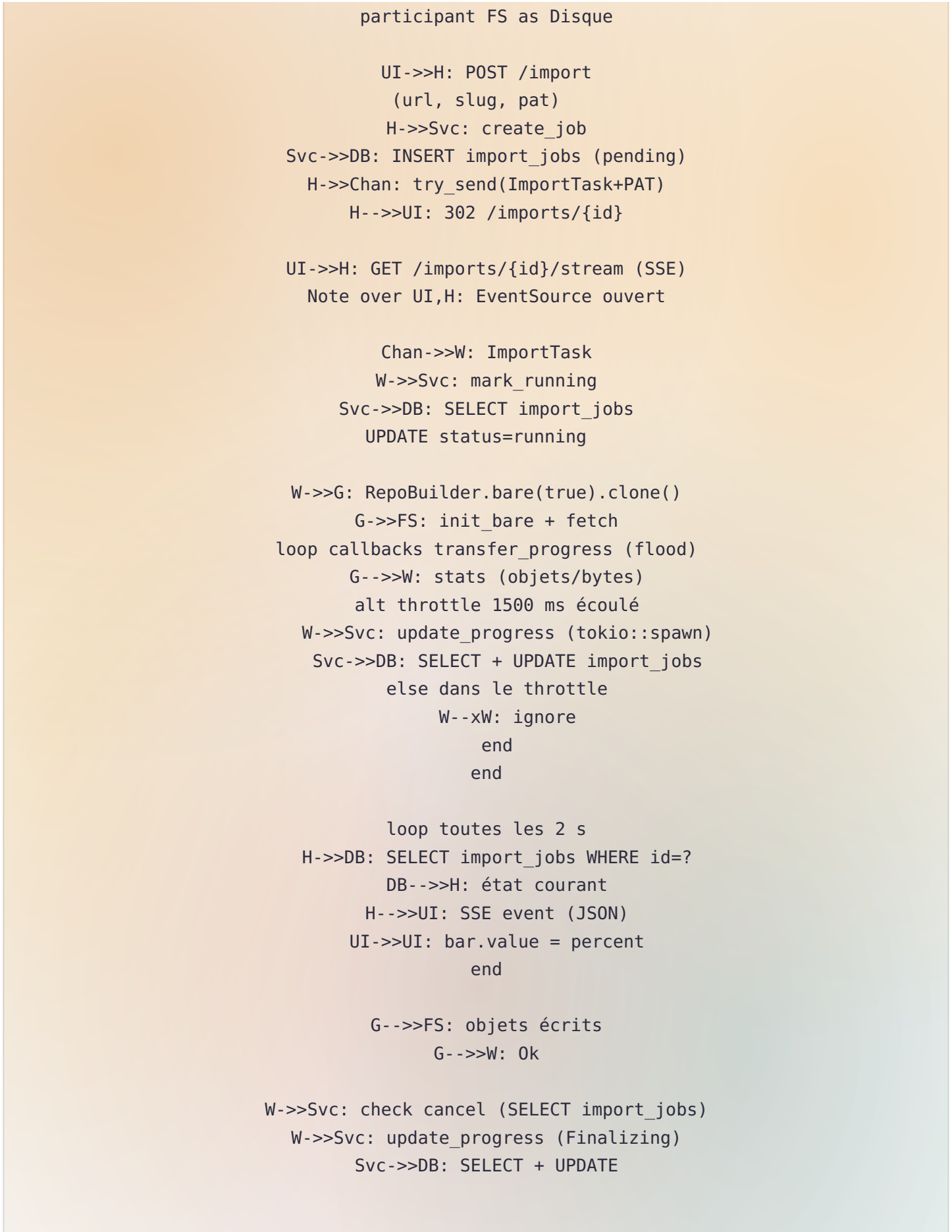
Pourquoi persister l'état ?

- **Reprise après redémarrage** : le serveur peut crasher pendant un clone de plusieurs minutes. La table permet de marquer ces jobs comme `failed` au redémarrage.
- **SSE** : l'UI lit la progression via un endpoint SSE qui interroge la DB toutes les 2 s. Sans DB, il faudrait un canal in-memory plus un mécanisme de routage vers le bon client.
- **Multi-navigateurs** : si l'utilisateur ferme l'onglet puis le rouvre, il retrouve l'état exact du job.
- **Audit / historique** : conservation des imports passés avec leurs statistiques.

2. Schéma du flux actuel

```
sequenceDiagram
    autonumber
    participant UI as Navigateur
    participant H as Handler HTTP
    participant Svc as ImportService
    participant Chan as mpsc Channel
    participant W as ImportWorker
    participant G as git2 (libgit2)
    participant DB as PostgreSQL
```


© Gitrust


participant FS as Disque
UI->>H: POST /import
(url, slug, pat)
H->>Svc: create_job
Svc->>DB: INSERT import_jobs (pending)
H->>Chan: try_send(ImportTask+PAT)
H-->>UI: 302 /imports/{id}
UI->>H: GET /imports/{id}/stream (SSE)
Note over UI,H: EventSource ouvert
Chan->>W: ImportTask
W->>Svc: mark_running
Svc->>DB: SELECT import_jobs
UPDATE status=running
W->>G: RepoBuilder.bare(true).clone()
G->>FS: init_bare + fetch
loop callbacks transfer_progress (flood)
G-->>W: stats (objets/bytes)
alt throttle 1500 ms écoulé
W->>Svc: update_progress (tokio::spawn)
Svc->>DB: SELECT + UPDATE import_jobs
else dans le throttle
W--xW: ignore
end
end
loop toutes les 2 s
H->>DB: SELECT import_jobs WHERE id=?
DB-->>H: état courant
H-->>UI: SSE event (JSON)
UI->>UI: bar.value = percent
end
G-->>FS: objets écrits
G-->>W: Ok
W->>Svc: check cancel (SELECT import_jobs)
W->>Svc: update_progress (Finalizing)
Svc->>DB: SELECT + UPDATE

© Gitrust

```
        W->>RepositoryService: create
RepositoryService->>DB: INSERT resources
          INSERT repositories
       W->>DB: UPDATE repositories
(import_source_url, is_empty=false)
         W->>Svc: mark_success
Svc->>DB: SELECT + UPDATE import_jobs
     (status=success, duration)
W->>AuditService: log REPO_IMPORTED
AuditService->>DB: INSERT audit_log

H->>DB: SELECT (prochain tick SSE)
       H-->>UI: terminal=true
UI->>UI: window.location.reload()
```

3. Coût DB par import

Pour un clone de **90 secondes** avec throttle 1500 ms :

Étape	Opérations DB	Cumul
create_job	1 INSERT	1
mark_running	1 SELECT + 1 UPDATE	3
update_progress pendant clone	~60 x (1 SELECT + 1 UPDATE)	123
check cancel	1 SELECT	124
update_progress finalizing	1 SELECT + 1 UPDATE	126
RepositoryService::create	2 INSERT (repo + resource)	128
update repositories	1 UPDATE	129
mark_success	1 SELECT + 1 UPDATE	131
audit log	1 INSERT	132
SSE stream (45 ticks à 2 s)	45 SELECT	177

~177 requêtes pour un import, dont **120+ sont des progress updates**.

© Gitrust

4. Pistes d'optimisation (sans changer libgit2)

4.1 Éliminer le SELECT avant UPDATE dans `update_progress`

`update_progress` fait actuellement `load_model` (SELECT) puis `active.update(db)` (UPDATE + RETURNING). Le SELECT est redondant : on connaît l'ID, on veut juste patcher 4 colonnes.

Gain : divise par 2 les requêtes de progression (~60 requêtes en moins pour un clone de 90 s).

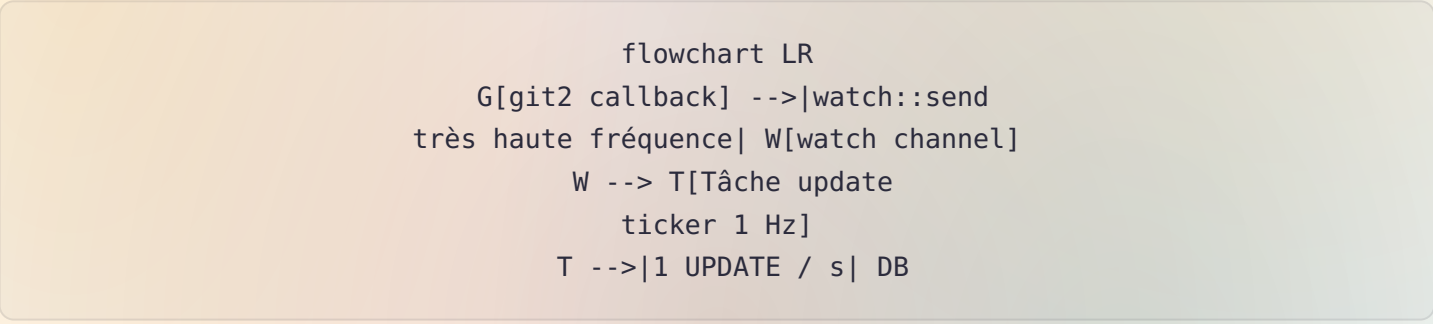
Refonte suggérée avec `UpdateMany` (pattern déjà utilisé dans `ci_service.rs::cancel_running_pipelines`) :

```
import_job::Entity::update_many()
    .col_expr(Column::ReceivedObjects, Expr::value(received as i32))
    .col_expr(Column::TotalObjects, Expr::value(total as i32))
    .col_expr(Column::ReceivedBytes, Expr::value(bytes as i64))
    .col_expr(Column::Phase, Expr::value(phase.as_str()))
    .col_expr(Column::UpdatedAt, Expr::value(Utc::now()))
    .filter(Column::Id.eq(job_id))
    .exec(db)
    .await?;
```

Même principe pour `mark_running`, `mark_success`, `mark_failed`, `mark_cancelled` → toutes divisent leurs aller-retours DB par 2.

4.2 Downsampler la progression dans le worker (dédié)

Plan alternatif : le callback écrit simplement dans un `tokio::sync::watch<TransferStats>` (en mémoire, zéro DB). Une **tâche dédiée** consomme ce channel et UPDATE la DB à 1 Hz seulement.



Gains : - **1 seul émetteur DB** au lieu de N tokio::spawn concurrents → zéro contention sur le pool - **Fréquence déterministe** (1 Hz) indépendante de la vitesse réseau - Code plus clair (la tâche est async, on peut `await` sans détour)

4.3 Supprimer le `RETURNING` implicite de SeaORM

`ActiveModel::update(db)` renvoie le modèle complet via `... RETURNING *`. Pour les progress updates, on n'utilise pas le retour. `UpdateMany::exec` n'émet pas de `RETURNING` → moins de bytes sur le wire, moins de désérialisation.

© Gitrust

4.4 Pool DB dédié pour le worker

Aujourd'hui le worker partage le pool principal avec les handlers HTTP. Un pool de 4-8 connexions dédié au worker éviterait qu'un import sature le pool et fasse timeouter les requêtes utilisateur.

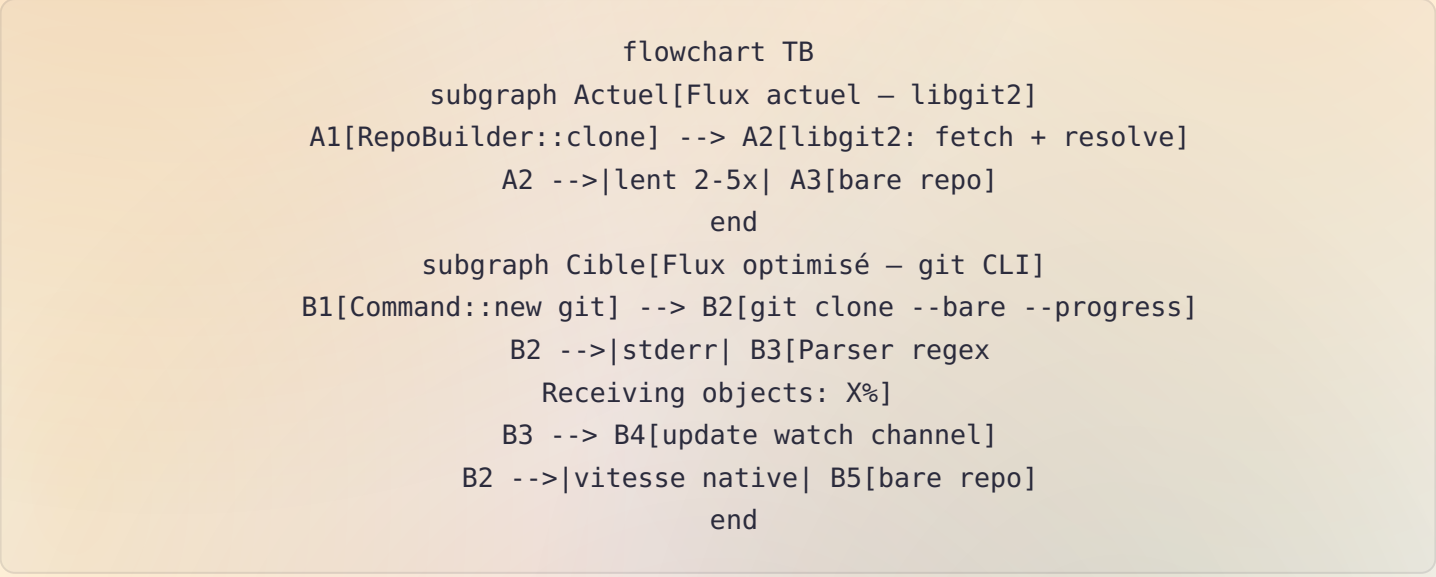
4.5 Debouncer le SSE côté serveur

Le SSE SELECT toutes les 2 s même quand rien ne change. Alternative : **PostgreSQL LISTEN/NOTIFY** sur une chaîne `import_job_{uuid}`. Chaque UPDATE émet un NOTIFY, la connexion SSE fait LISTEN → push direct, zéro polling.

Coût : nécessite une connexion DB dédiée par SSE (LISTEN est stateful). Compromis à évaluer.

5. Optimisation majeure : shell-out vers `git clone --bare`

Indépendant des optimisations DB mais de loin le plus gros gain. libgit2 est structurellement 2-5x plus lent que `git` CLI sur gros clones HTTPS (pas de multiplexage, résolution de deltas mono-threadée, etc.).



Avantages : - **Vitesse native** (parité avec `git` CLI) - **Progression native** via stderr `Receiving objects: 42% (368/876), 1.10 MiB | 2.05 MiB/s` - **HEAD correctement positionné** (plus besoin du workaround `RepoBuilder`) - **Moins de dépendance sur libgit2** pour le cas du clone initial

Contrepartie : binaire `git` requis sur le serveur (déjà le cas — utilisé par `SbomService::git_archive`).

6. Priorités recommandées

Si on voulait optimiser maintenant, ordre proposé :

1. **Shell-out `git clone --bare`** — gain réel pour l'utilisateur (clone 3-5x plus rapide).
2. **`update_many` pour `update_progress`** — divise la pression DB par 2, changement mécanique de ~20 lignes.
3. **`watch` channel + tâche 1 Hz** — architecture plus propre, élimine les problèmes de pool.
4. **Pool DB dédié worker** — filet de sécurité opérationnel.
5. LISTEN/NOTIFY pour SSE — seulement si beaucoup de clients simultanés.

© Gitrust

Les gains 2-4 sont utiles même avec libgit2 ; le gain 1 est le plus visible pour l'utilisateur final.

© Gitrust

1.3.10 Contribuer à rustwarden

Ce document décrit les interactions entre gitrust et le framework rustwarden-core, et explique comment étendre le framework côté gitrust sans modifier ses sources.

Services framework utilisés

Service	Usage dans gitrust
`UserService`	CRUD users, roles, `delete_user`, `admin_reset_password`, `change_password_direct`, `unassign_role_from_user`
`RoleService`	Liste/création/suppression roles, `get_role_by_name`
`PasswordService`	`hash_password` (async), `verify_password` (async), `generate_secure_password`
`ResourceService`	`register` / `unregister` repos, `set_public`, `effective_permission`, `find_by_type_and_id`
`AuthService`	`authenticate_user` (login)
`JwtService`	`generate_token` (login)
`EmailValidationService`	Vérification email
`PasswordResetService`	Flux "mot de passe oublié"
`AppSettingsService`	`get_bool` pour `registration_enabled`

Hook `on_user_deleted` implémenté

Le hook `on_user_deleted` dans `gitrust-hooks` supprime le répertoire utilisateur sur le filesystem (`repos_base_path/username/`) avant la cascade DB. Appelé automatiquement par `UserService::delete_user`.

Règle fondamentale

Ne jamais modifier `crates/rustwarden-core/`. Ce crate est le framework partagé. Toute extension passe par :

- Des **wrappers** côté `gitrust-core` (services qui délèguent au framework)
- Des **trait impls** (`RustwardenHooks` dans `gitrust-hooks`)
- Des **extensions** via les mécanismes d'injection Axum (`Extension` , `State`)

© Gitrust

Aucun contournement en cours

Tous les anciens contournements (entités directes, hash manuels) ont été remplacés par les services du framework suite à la mise à jour du 2026-03-26.

Ajouter une fonctionnalité nécessitant un service framework absent

Si un service nécessaire n'existe pas dans rustwarden-core :

1. Implémenter le service côté `gitrust-core` en réutilisant les primitives du framework (ex: `PasswordService::hash_password` pour le hashage)
2. Si la fonctionnalité est générique et bénéficierait au framework lui-même, ouvrir une issue ou une PR upstream sur le dépôt rustwarden-core
3. Ne pas dupliquer la logique d'auth, de session, ou de JWT — ces domaines appartiennent exclusivement au framework

Pattern d'extension via hooks

Les hooks (`gitrust-hooks/src/lib.rs`) implémentent `RustwardenHooks` et sont injectés dans le routeur Axum au démarrage :

```
// Dans main.rs
let hooks = Arc::new(GitrustHooks::new(config.clone()));
// Passé via .merge_routes(app_routes(hooks))
// Puis disponible via axum::Extension dans les handlers
```

Pour réagir à un nouvel événement du framework, ajouter la méthode correspondante dans l'impl `RustwardenHooks` de `gitrust-hooks` , sans toucher au code du framework qui émet l'événement.

© Gitrust

1.3.11 Passer la QA avant un merge

Ce guide liste dans l'ordre exact toutes les vérifications à effectuer avant d'ouvrir ou de merger une pull request dans gitrust.

Pourquoi cette checklist existe

Gitrust vise la conformité ANSSI PA-074. Chaque gate protège contre une catégorie spécifique de régressions : qualité de code, sécurité mémoire, vulnérabilités de dépendances, ou comportement utilisateur. Aucun gate ne peut être ignoré sauf exception documentée dans la PR.

Ordre d'exécution local

Exécutez les gates dans cet ordre — un gate qui échoue bloque les suivants.

Gate 1 — Formatage

```
cargo fmt --all -- --check
```

En cas de diff : appliquez le format, puis relancez `--check`.

```
cargo fmt --all
cargo fmt --all -- --check   # doit maintenant retourner 0
```

Sortie attendue (succès) :

```
(aucune sortie — code de retour 0)
```

Gate 2 — Linting (zéro warning)

```
cargo clippy --workspace -- -D warnings
```

Sortie attendue :

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in Xs
```

Aucun `warning:` ne doit apparaître. Les warnings sont traités comme des erreurs (`-D warnings`).

© Gitrust

Lints Rust obligatoires (déjà configurés dans `lib.rs` de chaque crate) :

```
#![forbid(unsafe_code)]          // crates core, web, hooks
#![deny(clippy::unwrap_used)]
#![deny(clippy::expect_used)]
#![deny(clippy::panic)]
#![deny(clippy::indexing_slicing)]
#![deny(clippy::mem_forget)]
```

Gate 3 — Tests unitaires et d'intégration

```
cargo test --workspace
```

Sortie attendue :

```
test result: ok. XXX passed; 0 failed; 0 ignored; 0 measured
```

Règles de test : - Tests sur vraie DB (pas de mocks pour la couche persistance). - Tests négatifs obligatoires : pour chaque chemin heureux, un test couvre l'accès non authentifié, l'input invalide, et la ressource inexistante. - Pas de `sleep()` dans les tests — utilisez `retry` / `poll` avec timeout. - Chaque test crée ses propres données (pas de dépendance sur l'ordre d'exécution).

Gate 4 — Tests E2E Playwright (si UI modifiée)

À exécuter si vous avez modifié des templates, routes SSR, ou le comportement des pages :

```
npm run test:e2e
```

Les tests E2E se trouvent dans `tests/e2e/`. Sortie attendue :

```
  XX passed (Xs)
```

En cas d'échec : consultez `tests/e2e/reports/` pour les captures d'écran et traces.

Gate 5 — Rebuild CSS (si templates modifiés)

À exécuter si vous avez modifié un fichier dans `templates/` ou ajouté des classes Tailwind :

© Gitrust

```
npx tailwindcss -i static/css/input.css -o static/css/style.css --minify
```

Commitez le fichier `static/css/style.css` généré dans votre PR.

Gate 6 — Audit des dépendances

```
cargo audit
```

Sortie attendue :

```
    Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
    Scanning Cargo.lock for vulnerabilities (X crates)
0 vulnerabilities found
```

En cas de vulnérabilité : mettez à jour la dépendance concernée ou ajoutez une exemption documentée dans `deny.toml`.

Gate 7 — Licences et dépendances bannies

```
cargo deny check
```

Sortie attendue :

```
    Checking advisories
    Checking bans
    Checking licenses
    Checking sources
```

Aucune violation ne doit apparaître. Gitrust interdit les licences GPL et AGPL dans les dépendances de production.

© Gitrust

Checklist sécurité (features touchant auth/secrets/permissions)

Pour tout changement touchant l'authentification, les tokens, les permissions ou les secrets :

- [ ] Secrets jamais en clair en base — hash SHA-256 ou bcrypt systématique.
- [ ] Secrets jamais loggés — vérifiez `tracing::info/debug`, `Display` et `Debug` impls.
- [ ] Comparaison en temps constant — `subtle::ConstantTimeEq` pour la validation de secrets (pas de `==`).
- [ ] Rate limiting présent sur les endpoints d'authentification.
- [ ] Token CSRF sur tous les formulaires `POST` / `PUT` / `DELETE`.
- [ ] Vérification d'ownership côté serveur (anti-IDOR) — `user_id` vérifié sur chaque ressource accédée.
- [ ] Expiration obligatoire sur tout token ou session.
- [ ] Zeroize — les types contenant des secrets implémentent `Zeroize` / `ZeroizeOnDrop`.
- [ ] Path traversal — validation des chemins disque (pas de `..`, `/`, `\` dans les slugs).

Checklist pre-merge finale

- [ ] Gate 1 : `cargo fmt --all -- --check` → 0 diff
- [ ] Gate 2 : `cargo clippy --workspace -- -D warnings` → 0 warning
- [ ] Gate 3 : `cargo test --workspace` → 0 échec
- [ ] Gate 4 : `npm run test:e2e` → 0 échec (si UI modifiée)
- [ ] Gate 5 : CSS rebuild commité (si templates modifiés)
- [ ] Gate 6 : `cargo audit` → 0 vulnérabilité
- [ ] Gate 7 : `cargo deny check` → 0 violation
- [ ] Aucun secret en clair dans le diff (`git diff --staged | grep -i "password\|secret\|token\|key"`)
- [ ] Aucune modification dans `crates/rustwarden-core/` (framework read-only)
- [ ] Checklist sécurité §ci-dessus validée (si applicable)

Vérification des secrets dans le diff

Avant tout commit, scannez votre diff :

```
git diff --staged | grep -iE "(password|secret|token|api_key|private_key)\s*=" | grep -
```

Si la commande retourne des lignes, examinez-les une par une pour confirmer qu'il ne s'agit pas de vraies valeurs sensibles.

Voir aussi

- Workflow de contribution PR
- Référence règles QA et ANSSI

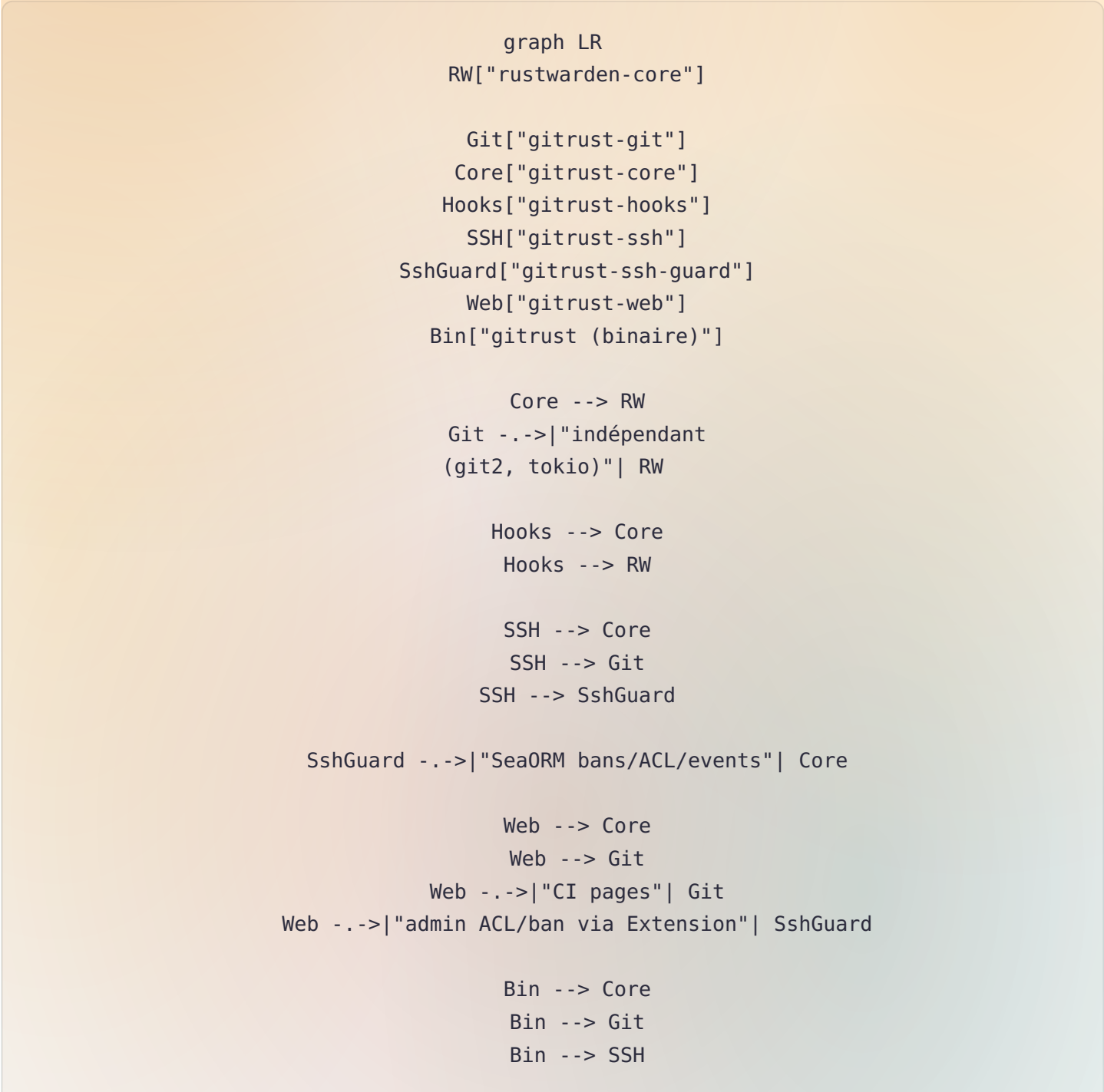
© Gitrust

1.4 Référence

1.4.1 Architecture par crates

Référence structurelle de gitrust : crates, modules, routes, dépendances et composants CI.

Dépendances entre crates



© Gitrust

```
    Bin --> SshGuard
    Bin --> Web
    Bin --> Hooks
    Bin --> RW

  subgraph "Externes CI (Phase 5b)"
    Dagger["Dagger CLI"]
    CiEng["ci-engine (Python)"]
    Syft["Syft"]
    DTrack["Dependency-Track"]
    end

    Core -.->|"CiWorker
  sous-processus"| Dagger
    Dagger -.-> CiEng
    Core -.-> Syft
    Syft -.-> DTrack
```


© Gitrust

Crates — Responsabilités

gitrust-core

Cœur métier. Ne dépend que de rustwarden-core et des crates Rust standards.

Module	Rôle
`config`	`GitrustConfig::from_env()` — ports SSH, chemin repos, limites
`error`	`GitrustError` — erreurs domaine Git + `IntoResponse` HTTP
`roles`	`enum Role { Reader, Developer, Maintainer, Owner }` avec `Ord`
`types`	Newtypes validés : `RepoSlug`, `TeamSlug`, `Fingerprint`, `TokenHash`
`models/`	Entités SeaORM : repository, ssh_key, team, team_member, team_repository_access, ci_pipeline, ci_job, ci_log, ci_config, ci_variable
`migrations/`	Migrations SeaORM (tables gitrust) combinées avec les migrations core
`services/`	`SshKeyService`, `RepositoryService`, `TeamService`, `AccessService`, `CiService`, `CiVariableService`, `CiDetectionService`, `CiWorker`, `NotificationService`
`dto/`	Structures d'entrée/sortie pour les services

© Gitrust

gitrust-git

Opérations Git sur le système de fichiers. Utilise `git2` (libgit2). Pas de dépendance vers la base de données.

Module	Rôle
`errors`	`enum GitError` (Git, RepoNotFound, RefNotFound, PathTraversal, ...)
`bare_repo`	Init, open, delete, exists + validation chemin canonique sous base
`branch` / `tag`	Listing branches (BranchInfo) et tags (TagInfo, annotated/lightweight)
`reference`	resolve_ref (branch -> tag -> SHA -> revparse)
`tree_browser`	list_tree (dirs first, sorted), TreeEntry, EntryKind
`blob_reader`	read_blob -> Text/Binary, détection binaire (null bytes)
`commit_log`	list_commits (paginé), find_commit, CommitInfo
`readme`	find_readme (README.md > README > readme.md > ...)
`pack_protocol`	advertise_refs, serve_pack (async, git-upload-pack/git-receive-pack)

gitrust-ssh

Serveur SSH basé sur `russh`. Authentification par clé publique.

Module (prévu)	Rôle
`server`	Démarrage, génération clé hôte Ed25519, wrapper du `TcpListener` par `SecureListener` (ssh-guard)
`auth`	Authentification par fingerprint -> SshKeyService, appel à `AuthTracker.record_auth_attempt` après chaque tentative
`session`	Handler SSH (exec, shell), porte `ClientIdentity` du listener jusqu'aux décisions auth
`command_handler`	Parsing `git-upload-pack` / `git-receive-pack`

© Gitrust

gitrust-ssh-guard

Couche de durcissement SSH intercalée entre le `TcpListener` et `russh`. Voir la page dédiée Crate gitrust-ssh-guard pour le détail des modules et l'API publique. Synthèse :

© Gitrust

Module	Rôle
`config`	`GuardConfig::from_env()` — lecture `SSH_GUARD_*`, sélection `DeploymentProfile`, validation
`runtime`	`GuardHandles::build(db)` — assemblage partageable entre serveur SSH et routeur admin
`listener`	`SecureListener` wrappe `TcpListener` : extraction IP réelle (PROXY v1/v2), ACL, flood, retour `AcceptOutcome`
`proxy`	Parseur PROXY protocol v1/v2 avec timeout
`identity`	`ClientIdentity` — IP + session_id + user/fingerprint enrichis pendant l'auth
`tracker`	`AuthTracker` — persiste, émet, dispatche aux détecteurs
`ban`	`BanManager` + `EffectiveStatus` — priorité `deny > auto_ban > allow > default`
`events`	`GuardEvent` — schéma JSON stable (tag = "event") pour fail2ban / SIEM
`sinks`	`TracingSink`, `FileSink`, `MultiSink` — destination des événements
`detector/{brute_force,user_enumeration,key_scanning,connection_flood}`	4 motifs d'attaque, push-based
`store/{memory,postgres,hybrid}`	Backend bans/ACL/events — `hybrid` (RAM + write-through) par défaut

© Gitrust

gitrust-web

Interface web SSR (Server-Side Rendering) avec Askama (templates compilés) + HTMX (interactions dynamiques). Fonctionne en mode `headless()` : les pages UI du framework (`rustwarden-ui`) ne sont pas montées, gitrust-web réimplémente toutes les pages avec un design spécifique (sidebar contextuelle, navigation Git, DaisyUI).

Architecture des routes :

```
gitrust (port 4000)
├── /api/v1/auth/*          ← Routes API du framework (JSON)
│   ├── POST /login            AuthService::authenticate_user
│   ├── POST /register         UserService::create_user
│   ├── POST /refresh          RefreshTokenService (rotation JWT)
│   ├── POST /logout           JwtBlacklistService
│   ├── GET  /me               Claims extraction
│   ├── POST /forgot-password  PasswordResetService::request_reset
│   ├── POST /reset-password   PasswordResetService::reset_password
│   ├── GET  /verify-email/{t} EmailValidationService::verify_email
│   └── POST /resend-verif     EmailValidationService::resend
│
├── /api/v1/*               ← API REST Gitrust (JSON, auth JWT/PAT)
│   ├── /user, /user/repos     Profil utilisateur
│   ├── /repos/{o}/{r}         Détail dépôt, branches, tags, commits
│   ├── /repos/{o}/{r}/issues  CRUD issues + commentaires
│   ├── /repos/{o}/{r}/pulls   CRUD PRs + merge
│   └── /repos/{o}/{r}/ci/*   Pipelines CI (list, trigger, cancel, logs)
│
├── /*                      ← Pages SSR gitrust (HTML)
│   ├── /login, /register       Pages formulaires (design gitrust)
│   ├── /dashboard, /teams      Pages utilisateur
│   ├── /settings/*             Profile, SSH keys, sessions, notifications
│   ├── /notifications          Liste notifications + SSE stream
│   ├── /admin/*                Administration (+ admin CI)
│   ├── /{owner}/{repo}/*       Navigation dépôt Git
│   └── /{owner}/{repo}/ci/*    Pages CI (pipelines, logs, config, variables)
│
└── /static/*               ← Assets locaux (CSS, JS, HTMX)
```

Les routes API du framework sont montées via `framework_api_routes()` dans `routes.rs`. Elles sont utilisées par : - Les liens dans les emails (vérification, reset password) - Le refresh token (rotation JWT, session longue durée) - L'accès programmatique (GET /me, POST /logout)

© Gitrust

Les pages SSR et les routes API coexistent : les formulaires HTML soumettent aux handlers gitrust (SSR), tandis que les services framework génèrent des liens vers les routes API.

Module	Rôle
`routes.rs`	Routeur principal : framework API + API REST Gitrust + pages SSR + fichiers statiques
`handlers/`	Handlers Axum SSR (un fichier par domaine) + handlers API JSON
`templates.rs`	Structs Askama + SidebarContext + RepoNav
`helpers.rs`	require_auth, require_admin, sidebar_*, resolve_repo_access
`templates/`	Templates HTML (Askama, extends base.html)
`static_files.rs`	ServeDir pour /static

gitrust-hooks

Implémentation de `RustwardenHooks` pour réagir aux événements du framework. Les hooks sont injectés dans le routeur Axum via `Arc<dyn RustwardenHooks>` + `axum::Extension` (dans `main.rs`).

Hook	Action
`on_user_registered`	Crée le répertoire utilisateur sur le FS (anti path-traversal)
`on_user_deleted`	Supprime le répertoire utilisateur + bare repos sur le FS
`on_resource_shared`	Audit log si resource_type = "repository"
`on_resource_unshared`	Audit log si resource_type = "repository"

Notifications

Système de notification à deux canaux : **in-app** (SSE temps réel + page) et **email** (worker SMTP du framework).

```
Événement (pipeline échoue, PR commentée, etc.)
    |
    ▼
NotificationService::notify(user_id, event_type, title, body, link)
    |
```


© Gitrust

```
      ├── INSERT INTO notifications (in-app)
      │     → broadcast::Sender → SSE /notifications/stream
      │     → Badge header mis à jour en temps réel (HTMX)
      │
      └── if user.email_on_{event} = true:
            EmailQueueService::enqueue() (framework SMTP worker)
```

Les préférences de notification sont par utilisateur (email on/off par type d'événement).

CI/CD — Modes et composants

Système CI hybride à deux niveaux, activé via `CI_ENABLED=true`.

Modes

Mode	Détection	Exécution
Easy	`.gitrust-ci.yml` à la racine du repo	Module Dagger Python générique (`ci-engine`) interprète le YAML
Power	`.dagger/` à la racine du repo	`dagger call -m .dagger/` exécute le module utilisateur directement
Aucun	Ni l'un ni l'autre	Pas de CI pour ce repo

Architecture CI

```
git push (HTTP ou SSH)
    │
    ▼
receive-pack handler
    │
    ├── CiDetectionService::detect(bare_repo, commit_sha)
    │     → Easy / Power / None
    │
    ├── CiService::create_pipeline(repo_id, sha, mode)
    │
    └── mpsc::Sender.send(job) ──► CiWorker (tokio::spawn)
                                       │
                                       ├── 1. Validation statique
                                       │     Easy: schéma YAML, whitelist images/packag
                                       │     Power: analyse code .dagger/ (si activé)
```


© Gitrust

```
                                    │
                                    ├── 2. SBOM Gate (si CI_SBOM_ENABLED)
                                    │     syft scan → sbom.cdx.json
                                    │     → Dependency-Track (si CI_DTRACK_ENABLED)
                                    │     → PASS/FAIL selon politique
                                    │
                                    ├── 3. Résolution variables (héritage team → rep
                                    │     CiVariableService::resolve_for_pipeline()
                                    │
                                    ├── 4. Exécution Dagger (isolé, timeout, cgroups
                                    │     Easy: dagger call -m ci-engine test --prof
                                    │     Power: dagger call -m .dagger/ <function>
                                    │
                                    └── 5. Rapport + audit
                                          Status, logs (broadcast SSE), ci_audit_log
```

Composants externes (tous optionnels sauf Dagger)

Composant	Rôle	Variable de contrôle
Dagger Engine	Exécution des pipelines dans des containers isolés	`CI_ENABLED`
ci-engine (Python)	Moteur générique pour Easy Mode	`CI_DAGGER_MODULE_PATH`
Syft	Génération SBOM CycloneDX	`CI_SBOM_ENABLED`
Dependency-Track	Analyse de vulnérabilités sur les SBOM	`CI_DTRACK_ENABLED`

Variables et secrets CI

Héritage **team → repo** : les variables définies au niveau team sont partagées entre tous les repos de l'équipe. Le repo peut surcharger une clé team avec sa propre valeur. Les secrets sont chiffrés en base (AES-256-GCM) et masqués dans les logs (`***`).

© Gitrust

Sécurité (ANSSI PA-074)

Directive	Scope
`#![forbid(unsafe_code)]`	gitrust-core, gitrust-web, gitrust-hooks, gitrust-ssh-guard
`#![deny(unsafe_code)]`	gitrust-git, gitrust-ssh (FFI libgit2/russh)
`deny(clippy::unwrap_used, expect_used, panic)`	Tous les crates
`deny(clippy::indexing_slicing)`	Tous les crates
`deny(clippy::mem_forget)`	Tous les crates
`Zeroize` on drop	`TokenHash` (ANSSI R23)
RSA minimum 4096 bits	`SshKeyService`
Path traversal validation	`RepositoryService`, `gitrust-git`
Slug validation + noms réservés	`RepoSlug`, `TeamSlug`

© Gitrust

1.4.2 Crate `gitrust-ssh-guard`

Référence structurelle de la crate de durcissement SSH. Pour la motivation et le rationale de design, voir Conception de ssh-guard. Pour l'exploitation côté admin, voir Configurer ssh-guard.

1. Position dans l'arborescence

```
crates/
├── gitrust-ssh-guard/
│   ├── Cargo.toml
│   └── src/
│       ├── lib.rs                (re-exports publics)
│       ├── config.rs             (SSH_GUARD_*, profils, validation)
│       ├── runtime.rs            (GuardHandles : assemblage)
│       ├── listener.rs           (SecureListener, AcceptOutcome)
│       ├── proxy.rs              (parseur PROXY protocol v1/v2)
│       ├── identity.rs           (ClientIdentity)
│       ├── tracker.rs            (AuthTracker, dispatch détecteurs)
│       ├── ban.rs                (BanManager, EffectiveStatus)
│       ├── events.rs             (GuardEvent, GuardEventSink)
│       ├── sinks.rs              (TracingSink, FileSink, MultiSink)
│       ├── detector/
│       │   ├── mod.rs
│       │   ├── brute_force.rs
│       │   ├── user_enumeration.rs
│       │   ├── key_scanning.rs
│       │   └── connection_flood.rs
│       └── store/
│           ├── mod.rs            (trait GuardStore)
│           ├── entities.rs       (modèles SeaORM)
│           ├── memory.rs         (MemoryStore — DashMap)
│           ├── postgres.rs       (PostgresStore — write-through)
│           └── hybrid.rs         (HybridStore — RAM chaude + DB)
```


© Gitrust

2. Tableau des modules

© Gitrust

Module	Rôle	Types principaux
`config`	Lecture des env vars `SSH_GUARD_*`, sélection du profil de déploiement, validation de cohérence (ex. : un profil proxifié sans `trusted_proxies` est rejeté au démarrage).	`GuardConfig`, `DeploymentProfile`, `ProxyProtocol`, `StoreBackend`, `LogFormat`, `LogTarget`, `DetectorThreshold`, `ConfigError`
`runtime`	Assemble tous les composants à partir de `GuardConfig` + connexion `DatabaseConnection`. Le binaire en construit **une seule** instance et la partage entre le serveur SSH et le routeur Axum admin.	`GuardHandles`, `BuildError`
`listener`	Wrappe un `tokio::net::TcpListener` : extraction d'IP réelle (PROXY ou `peer_addr`), consultation de l'ACL/ban, rate-limit flood, puis remise du `TcpStream` au handler `russh`.	`SecureListener`, `AcceptOutcome`, `AcceptError`, `ProxyListenerConfig`
`proxy`	Parseur PROXY protocol v1 (texte HAProxy legacy) et v2 (binaire nginx stream / HAProxy moderne) avec timeout dédié.	`parse_header`, `ParsedHeader`, `ProxyError`
`identity`	Identité d'un client SSH au fil de la session : IP, `session_id` (UUID), nom d'utilisateur et fingerprint de clé renseignés au moment de l'auth.	`ClientIdentity`
`tracker`	Reçoit les événements d'auth émis par `gitrust-ssh`, les persiste dans le store, les diffuse au sink, puis appelle `on_event` sur chaque détecteur actif.	`AuthTracker`, `AuthOutcome`
`ban`		

© Gitrust

Module	Rôle	Types principaux
	Compose ACL admin (allow/deny) et bans actifs en un statut effectif. Pose les bans auto (TTL) et manuels (CIDR), gère le `unban`.	`BanManager`, `EffectiveStatus`
`events`	Définition stable du schéma JSON émis. Toute évolution incompatible doit créer un nouveau variant plutôt que renommer un champ existant.	`GuardEvent`, `GuardEventSink`, `AuthMethod`, `BanReason`, `DropReason`
`sinks`	Implémentations de `GuardEventSink` : `tracing` (stderr / journald), fichier en append (ligne JSON), fan-out vers plusieurs sinks.	`TracingSink`, `FileSink`, `MultiSink`, `build_sink`
`detector::brute_force`	Compte les `AuthFailed` par IP dans la fenêtre, déclenche `auto_ban(BruteForce)` au seuil.	`BruteForceDetector`
`detector::user_enumeration`	Compte les usernames distincts essayés par IP dans la fenêtre.	`UserEnumerationDetector`
`detector::key_scanning`	Compte les fingerprints de clé distincts essayés par IP dans la fenêtre.	`KeyScanningDetector`
`detector::connection_flood`	Token bucket GCRA (`governor`) keyé par IP. Pas de ban persistant : drop immédiat au listener.	`ConnectionFloodDetector`
`store`	Trait abstrait pour bans, ACL et événements d'auth. Trois implémentations interchangeables.	`GuardStore`, `MemoryStore`, `PostgresStore`, `HybridStore`, `BanEntry`, `AclEntry`, `AclKind`, `RehydrateStats`, `StoreError`

© Gitrust

3. Surface API publique (`lib.rs`)

```
// Assemblage et runtime
pub use runtime::{BuildError, GuardHandles};

// Configuration
pub use config::{
    ConfigError, DeploymentProfile, DetectorThreshold, GuardConfig,
    LogFormat, LogTarget, ProxyProtocol, StoreBackend,
};

// Listener
pub use listener::{AcceptError, AcceptOutcome, ProxyListenerConfig, SecureListener};

// Identité et tracking
pub use identity::ClientIdentity;
pub use tracker::{AuthOutcome, AuthTracker};

// Décision de ban
pub use ban::{BanManager, EffectiveStatus};

// Détecteurs
pub use detector::{
    BruteForceDetector, ConnectionFloodDetector,
    KeyScanningDetector, UserEnumerationDetector,
};

// Événements et sinks
pub use events::{AuthMethod, BanReason, DropReason, GuardEvent, GuardEventSink};
pub use sinks::{build_sink, FileSink, MultiSink, TracingSink};

// Stockage
pub use store::{
    AclEntry, AclKind, BanEntry, GuardStore, HybridStore,
    MemoryStore, PostgresStore, RehydrateStats, StoreError,
};
```

Lints actifs sur la crate (alignés ANSSI PA-074) :

```
#![forbid(unsafe_code)]
#![deny(
    clippy::unwrap_used,
```


© Gitrust

```
    clippy::expect_used,
    clippy::panic,
    clippy::indexing_slicing,
    clippy::mem_forget
)]
```

4. `GuardHandles` — point d'entrée du runtime

`GuardHandles` est l'objet construit une fois au démarrage du binaire, puis cloné (tous les champs sont `Arc<…>`) là où il est nécessaire :

- vers `gitrust_ssh::server::start_server` pour que `SecureListener` et `AuthTracker` partagent les mêmes `BanManager` / `GuardStore` ;
- vers le routeur Axum via `axum::Extension`, pour que les actions admin (ajouter une IP en denylist, lever un ban) soient immédiatement visibles par le listener — sans attendre un `rehydrate` au prochain redémarrage.

```
#[derive(Clone)]
pub struct GuardHandles {
    pub config: GuardConfig,
    pub store: Arc<dyn GuardStore>,
    pub sink: Arc<dyn GuardEventSink>,
    pub ban_manager: Arc<BanManager>,
    pub tracker: Arc<AuthTracker>,
    pub flood: Option<Arc<ConnectionFloodDetector>>,
    pub proxy_config: Option<ProxyListenerConfig>,
}

impl GuardHandles {
    pub async fn build(db: DatabaseConnection) -> Result<Self, BuildError>;
}
```


© Gitrust

`build` enchaîne :

1. `GuardConfig::from_env()` (lecture + validation).
2. `build_sink(&config)` selon `LogTarget`.
3. Construction du store selon `SSH_GUARD_STORE_BACKEND` :
4. `memory` → `MemoryStore` (rien en DB) ;
5. `postgres` → `PostgresStore` (chaque écriture en DB) ;
6. `hybrid` → `HybridStore` (RAM chaude + flush write-through périodique + `rehydrate` au boot pour repeupler la mémoire depuis la DB).
7. Instanciation du `BanManager`.
8. Instanciation des détecteurs (chaque détecteur dont le seuil est désactivé — `u32::MAX` — n'est pas instancié).
9. Construction de l'`AuthTracker` avec les détecteurs actifs.
10. Construction de `ProxyListenerConfig` si `proxy_protocol != Disabled`.

---

5. `SecureListener` — séquence d'`accept`

```
pub enum AcceptOutcome {
    Accepted { stream: TcpStream, identity: ClientIdentity },
    Dropped  { ip: IpAddr,        reason:   DropReason     },
}

impl SecureListener {
    pub async fn accept(&self) -> Result<AcceptOutcome, AcceptError>;
}
```


© Gitrust

Étapes appliquées à chaque connexion entrante :

1. `tcp.accept()` → `(stream, peer)`.
2. `resolve_real_ip(stream, peer.ip())` :
3. profil direct (pas de `proxy_config`) → IP = `peer.ip()` ;
4. profil proxifié → vérifie que `peer.ip()` ∈ `trusted_proxies`, puis parse l'en-tête PROXY pour extraire l'IP cliente réelle ;
5. en mode `strict`, un parsing en échec retourne `Err(DropReason::ProxyHeaderInvalid|Missing|UntrustedProxy)` ; en mode souple, fallback sur `peer.ip()` avec warn.
6. `ban_manager.effective_status(ip)` :
7. `DeniedByAcl` → `Dropped { reason: Banned }` ;
8. `BannedAuto(_)` → `Dropped { reason: Banned }` ;
9. `AllowListed` → `Accepted` immédiat (bypass flood) ;
10. `Normal` → étape 4.
11. Si `flood` actif : `flood.check(ip)`. Sur false → `Dropped { reason: FloodLimit }` (les events sont déjà émis par `flood.check`).
12. Sinon : construction du `ClientIdentity`, émission de `ConnectionAccepted`, retour `Accepted`.

Toute erreur du store est traitée en **fail-open** (warn + on laisse passer), pour ne pas couper le service quand PostgreSQL est indisponible.

---

6. `BanManager` — priorités d'ACL

```
pub enum EffectiveStatus {
    Normal,
    AllowListed,
    DeniedByAcl,
    BannedAuto(Box<BanEntry>),
}
```

Priorité de résolution dans `effective_status(ip)` :

1. `acl_match(ip) == Some(Deny)` → `DeniedByAcl`.
2. `bans_covering(ip)` non vide → `BannedAuto(ban)`.
3. `acl_match(ip) == Some(Allow)` → `AllowListed`.
4. Sinon → `Normal`.

© Gitrust

Un appel `auto_ban(ip, reason)` est **no-op** dans trois cas :

- `dry_run` actif (un événement `IpBanned` est tout de même émis pour fail2ban / observabilité) ;
- l'IP est allowlistée ;
- un ban auto actif couvre déjà l'IP (évite le spam d'événements `IpBanned`).

Un `manual_ban(cidr, reason, ttl)` accepte un CIDR (pas seulement un /32) et est permanent si `ttl=None`.
`unban(ban_id, by)` retire le ban et émet `IpUnbanned`.

---

7. `AuthTracker` — pivot de la chaîne d'événements

```
pub enum AuthOutcome { Success, Failure }

impl AuthTracker {
    pub async fn record_auth_attempt(
        &self,
        identity: &ClientIdentity,
        method:   AuthMethod,
        outcome:  AuthOutcome,
    );
}
```

Pour chaque appel :

1. Construit `GuardEvent::AuthSucceeded` ou `AuthFailed` selon `outcome`.
2. `store.record_event(&event)` — alimente la table consultée par les détecteurs.
3. `sink.emit(&event)` — alimente fail2ban / observabilité.
4. Pour chaque détecteur configuré : `detector.on_event(&event).await`.

Côté `gitrust-ssh`, le handler `russh` appelle `record_auth_attempt` après chaque tentative d'authentification (clé publique, password, keyboard-interactive).

---

8. Schéma JSON des événements

Tous les `GuardEvent` se sérialisent avec `#[serde(tag = "event", rename_all = "snake_case")]`. La forme stable est :

```
{"event":"<nom_variant>","ts":"...","ip":"...", ...}
```

Pour le détail des champs par variant et leur usage côté admin (fail2ban, SIEM), voir Événements ssh-guard (JSON).

© Gitrust

Convention de stabilité :

- ajouter un nouveau variant n'est **pas** un breaking change pour les consommateurs ;
- ajouter un champ optionnel à un variant existant ne l'est pas non plus ;
- renommer un champ ou un variant **est** un breaking change et nécessite un nouveau nom.

9. Backends de stockage (`GuardStore`)

Backend	Vie des bans/ACL	Latence lecture	Cas d'usage
`MemoryStore`	Perdues au restart	DashMap, lock-free	Tests, profil `private`, dev
`PostgresStore`	Persistantes	Round-trip DB par lecture	Audits stricts, instances multi-noeuds (sans cache RAM)
`HybridStore`	RAM chaude + write-through DB, `rehydrate` au boot	DashMap	**Défaut recommandé** : performance + persistance

Le trait `GuardStore` expose : `acl_match`, `bans_covering`, `is_banned`, `insert_ban`, `unban`, `list_active_bans`, `insert_acl`, `record_event`, `count_auth_failures`, `count_distinct_users`, `count_distinct_keys`.

10. Intégration côté `gitrust-ssh`

Pseudo-code condensé du wiring effectif (`crates/gitrust-ssh/src/server.rs`) :

```
let secure = SecureListener::new_with_proxy(
    tcp_listener,
    guard.ban_manager.clone(),
    guard.flood.clone(),
    guard.sink.clone(),
    guard.proxy_config.clone(),
);

loop {
    match secure.accept().await? {
        AcceptOutcome::Accepted { stream, identity } => {
            let session = GitSession::new_with_guard(
                db.clone(), ssh_config.clone(), ci_tx.clone(),
```


© Gitrust

```
                identity, guard.tracker.clone(),
            );
            tokio::spawn(async move { russh::server::run_stream(/* ... */).await });
        }
        AcceptOutcome::Dropped { .. } => {
            // L'événement ConnectionDropped a déjà été émis par le listener.
        }
    }
}
```

Côté `GitSession` (handler `russh`) : à chaque tentative d'auth, l'identité est enrichie (`set_username`, `set_fingerprint`) puis `tracker.record_auth_attempt(&identity, method, outcome)` est appelé.

---

11. Tests

Toute la crate utilise des tests unitaires `#[tokio::test]` avec un `MemoryStore` et un `RecorderSink` qui capture les `GuardEvent`. Ordre de grandeur : **80+ tests** couvrant chaque détecteur, la priorité d'ACL, le mode `dry_run`, l'idempotence du ban, l'expiration TTL, les cas d'erreur du parseur PROXY, la séparation des budgets de flood par IP.

Les tests d'intégration de bout en bout (PROXY v2 binaire émis par un client de test, déclenchement réel du seuil brute-force depuis `gitrust-ssh`) vivent dans `crates/gitrust-ssh/tests/`.

---

12. Voir aussi

- Conception de ssh-guard — pourquoi cette crate, modèle de menace, design async
- Architecture des crates — position de ssh-guard dans le graphe de dépendances
- Configurer ssh-guard — recettes par profil de déploiement
- Variables d'environnement — section SSH_GUARD_*
- Événements ssh-guard (JSON)

© Gitrust

1.4.3 Services et API interne — Référence

Ce document liste tous les services publics de gitrust (rustwarden-core + gitrust-core), leurs signatures Rust, les erreurs possibles, et quand les utiliser.

Architecture générale

```
Handler Axum → Service (logique métier) → SeaORM (base de données)
```

Conventions communes : - Les services sont des structs sans état avec des méthodes associées `async fn`. - Premier paramètre toujours `db: &DatabaseConnection`. - Type de retour toujours `Result<T, AppError>` (rustwarden-core) ou `Result<T, GitrustError>` (gitrust-core). - Aucun `unwrap()` / `expect()` en production. - Aucun SQL brut — tout passe par SeaORM.

Services rustwarden-core (framework)

Importés via `use rustwarden_core::prelude::*` pour les exports du prélude, ou via leur chemin complet.

`AuthService`

Module : `rustwarden_core::services::auth_service`

Méthode	Signature	Description
`verify_credentials`	`(db, identifier: &str, password: &str) → Result<user::Model>`	Vérifie les identifiants. `identifier` accepte un username ou un email. Utilise un dummy hash si l'utilisateur n'existe pas (protection timing attack SEC-C1).
`authenticate_user`	`(db, identifier: &str, password: &str) → Result<user::Model>`	Authentification de haut niveau. Appelle `verify_credentials` et logue `must_change_password`.

© Gitrust

`UserService`

Module : `rustwarden_core::services::user_service` **— Export : prélude**

Méthode	Signature	Description
`create_user`	`(db, username, email, password_hash, must_change_password) → Result<user::Model>`	Crée un utilisateur. Le mot de passe doit être **pré-haché** via `PasswordService::hash_password`.
`get_user_by_id`	`(db, user_id: Uuid) → Result<Option<user::Model>>`	Recherche par ID.
`get_user_by_username`	`(db, username: &str) → Result<Option<user::Model>>`	Recherche par nom d'utilisateur.
`get_user_by_email`	`(db, email: &str) → Result<Option<user::Model>>`	Recherche par email.
`list_all_users`	`(db) → Result<Vec<user::Model>>`	Liste tous les utilisateurs.
`get_user_roles`	`(db, user_id: Uuid) → Result<Vec<String>>`	Retourne les noms des rôles.
`assign_role_to_user`	`(db, user_id, role_id) → Result<user_role::Model>`	Assigne un rôle.
`update_profile`	`(db, user_id, username: Option<String>, email: Option<String>) → Result<user::Model>`	Met à jour le profil. Valide l'unicité.
`change_password`	`(db, user_id, old_password, new_password, current_token_id) → Result<(Uuid, String)>`	Initie un changement de mot de passe (demande pendante). Retourne `(request_id, plain_token)`.

© Gitrust

`PasswordService`

Module : `rustwarden_core::services::password_service`

Méthode	Signature	Description
`hash_password`	`async (password: &str) → Result<String>`	Hash bcrypt via `spawn_blocking`. Coût configurable via `BCRYPT_COST` (défaut : 12).
`verify_password`	`async (password: &str, hash: &str) → Result<bool>`	Vérification bcrypt via `spawn_blocking`.
`generate_secure_password`	`() → String`	Génère un mot de passe de 20 caractères avec mix garanti.

`TotpService`

Module : `rustwarden_core::services::totp_service`

Méthode	Description
`setup_totp(db, user_id, username, issuer)`	Génère un secret TOTP + QR code URI.
`verify_and_enable(db, user_id, code)`	Vérifie un code et active le 2FA. Retourne les 10 codes de secours.
`verify_code(db, user_id, code)`	Vérifie un code TOTP (6 chiffres).
`verify_backup_code(db, user_id, code)`	Vérifie un code de secours (8 chars, usage unique).
`disable_totp(db, user_id)`	Désactive le 2FA.
`create_challenge(db, user_id)`	Crée un challenge 2FA (token temporaire, 5 min, max 5 tentatives).
`validate_challenge(db, challenge_token)`	Valide un challenge.
`consume_challenge(db, challenge_token)`	Marque un challenge comme consommé.

© Gitrust

`RoleService` / `PermissionService`

Module : `rustwarden_core::services::role_service` — **Export : prélude**

Méthode	Description
`RoleService::create_role(db, name, description)`	Crée un rôle.
`RoleService::get_role_by_name(db, name)`	Recherche par nom.
`RoleService::list_all_roles(db)`	Liste tous les rôles.
`RoleService::initialize_default_roles(db)`	Seed `admin`, `user`, `app_manager`. Idempotent.
`PermissionService::get_permissions_for_user(db, user_id)`	Union des permissions de tous les rôles.
`PermissionService::set_role_permissions(db, role_id, perm_ids)`	Remplace atomiquement les permissions d'un rôle.
`PermissionService::initialize_default_permissions(db)`	Seed les 12 permissions par défaut. Idempotent.

`ResourceService`

Module : `rustwarden_core::services::resource_service` — **Export : prélude**

Système générique de registre de ressources et de partage.

© Gitrust

Hiérarchie des permissions : `read` < `write` < `admin`

Méthode	Description
`register(db, resource_type, resource_id, owner_id)`	Enregistre une ressource dans le registre.
`set_public(db, resource_type, resource_id, is_public)`	Bascule la visibilité publique.
`unregister(db, resource_type, resource_id)`	Supprime du registre (cascade sur les partages).
`share(db, resource_type, resource_id, shared_with_user_id, permission_level, shared_by_user_id)`	Partage une ressource. Met à jour si déjà partagé.
`revoke_share(db, share_id)`	Révoque un partage par ID.
`user_can_access(db, user_id, resource_type, resource_id, required_level)`	Vérifie l'accès : owner → public+read → partagé.
`effective_permission(db, user_id, resource_type, resource_id)`	Retourne `"owner"`, le niveau de partage, `"read"` (public), ou `None`.
`accessible_by_user(db, user_id, resource_type)`	Toutes les ressources accessibles (possédées + partagées + publiques).

© Gitrust

`RefreshTokenService`

Module : `rustwarden_core::services::refresh_token_service`

Méthode	Description
`create_refresh_token(db, user_id, expiration_days)`	Crée un refresh token. Révoque les plus anciens si > 10 actifs (SEC-M4).
`validate_refresh_token(db, token)`	Valide un token (hash SHA-256, existence, non révoqué, non expiré).
`revoke_all_tokens_for_user(db, user_id)`	Révoque toutes les sessions (logout, changement de mot de passe).
`list_user_sessions(db, user_id)`	Liste les sessions actives.
`revoke_session(db, token_id)`	Révoque une session spécifique.

`AppSettingsService`

Module : `rustwarden_core::services::app_settings_service`

Méthode	Description
`get_setting(db, key)`	Lire un paramètre.
`set_setting(db, key, value, updated_by)`	Créer ou mettre à jour un paramètre.
`get_bool(db, key, default)`	Lire un booléen avec valeur par défaut.
`initialize_default_settings(db)`	Seed ~30 paramètres par défaut. Idempotent.

© Gitrust

Services gitrust-core

`RepositoryService`

Module : `gitrust_core::services::repository_service`

Méthode	Signature	Description
`create`	`(db, owner_id, slug, description, is_public) → Result<repository::Model>`	Crée un dépôt + répertoire bare sur disque. Enregistre dans `ResourceService`.
`find_by_owner_and_slug`	`(db, owner_username, slug) → Result<Option<repository::Model>>`	Recherche par owner/slug (chemin URL).
`list_by_owner`	`(db, owner_id) → Result<Vec<repository::Model>>`	Dépôts d'un propriétaire.
`update`	`(db, repo_id, owner_id, description, default_branch) → Result<repository::Model>`	Met à jour les métadonnées. Anti-IDOR.
`delete`	`(db, repo_id, owner_id) → Result<()>`	Supprime DB + bare repo sur disque. Anti-IDOR.
`mark_non_empty`	`(db, repo_id) → Result<()>`	Passe `is_empty = false` après le premier push.

© Gitrust

SshKeyService

Module : `gitrust_core::services::ssh_key_service`

Méthode	Signature	Description
`create`	`(db, user_id, title, key_data) → Result<ssh_key::Model>`	Parse la clé, calcule le fingerprint SHA256, valide le type (ed25519, rsa ≥ 4096 bits, ecdsa-p256/p384).
`list_by_user`	`(db, user_id) → Result<Vec<ssh_key::Model>>`	Liste les clés d'un utilisateur.
`find_by_fingerprint`	`(db, fingerprint) → Result<Option<ssh_key::Model>>`	Recherche par empreinte (utilisé par le serveur SSH).
`delete`	`(db, key_id, user_id) → Result<()>`	Suppression avec vérification d'ownership.
`update_last_used`	`(db, key_id) → Result<()>`	Met à jour `last_used_at` à chaque authentification SSH.

IssueService

Module : `gitrust_core::services::issue_service`

Méthode	Description
`create(db, repo_id, author_id, title, body, label_ids, subject_names)`	Numéro auto-incrémenté par dépôt.
`list(db, repo_id, state_filter, label_filter, page, per_page)`	Liste paginée avec filtres.
`find_by_number(db, repo_id, number)`	Par numéro d'issue.
`close(db, issue_id, closed_by)`	Ferme l'issue + audit.
`reopen(db, issue_id)`	Rouvre l'issue + audit.
`enrich_issue(db, issue)`	Enrichit avec username auteur, labels, nombre de commentaires.

© Gitrust

`LabelService`

Module : `gitrust_core::services::label_service`

Méthode	Description
`create_classification(db, owner_id, name, color, description)`	Label scope owner (tous ses dépôts).
`create_subject(db, repo_id, name, color)`	Label scope dépôt. Couleur défaut `#6b7280`.
`find_or_create_subject(db, repo_id, name)`	Trouve ou crée (pour création inline dans les issues).
`list_classification_by_owner(db, owner_id)`	Labels classification de l'owner.
`list_subject_by_repo(db, repo_id)`	Labels subject du dépôt.
`search_subject(db, repo_id, query, limit)`	Recherche préfixe (autocomplete HTMX).
`assign_to_issue(db, issue_id, label_id)`	Association issue ↔ label.

`TeamService`

Module : `gitrust_core::services::team_service`

Méthode	Description
`create(db, owner_id, slug, description)`	Crée une équipe.
`add_member(db, team_id, user_id, role)`	Ajoute un membre.
`remove_member(db, team_id, user_id)`	Retire un membre.
`grant_repo_access(db, team_id, repo_id, permission)`	Accorde l'accès à un dépôt (`read` / `write` / `admin`).
`revoke_repo_access(db, team_id, repo_id)`	Révoque l'accès.

© Gitrust

`ImportService`

Module : `gitrust_core::services::import_service`

Méthode	Description
`create_job(db, owner_id, source_url, target_slug)`	Crée un job `pending` dans `import_jobs`.
`mark_running(db, job_id)`	Passe en `running` + `started_at`.
`update_progress(db, job_id, received, total, bytes)`	Met à jour les compteurs de progression.
`mark_success(db, job_id, repo_id)`	Lie le job au dépôt créé.
`mark_failed(db, job_id, error_message)`	Enregistre l'échec.
`mark_cancelled(db, job_id)`	Annulation.

Gestion des erreurs

`AppError` (rustwarden-core)

Variante	HTTP	Usage
`AppError::NotFound(msg)`	404	Ressource introuvable
`AppError::Validation(msg)`	400	Données invalides
`AppError::Forbidden`	403	Accès refusé (RBAC)
`AppError::Unauthorized`	401	Non authentifié
`AppError::TokenExpired`	401	JWT expiré
`AppError::TokenReused`	401	Refresh token réutilisé (attaque détectée)
`AppError::Conflict(msg)`	409	Contrainte d'unicité violée
`AppError::Internal(msg)`	500	Erreur interne

`GitrustError` (gitrust-core)

Mêmes variantes que `AppError` avec `GitrustError::Database(sea_orm::DbErr)` en plus pour les erreurs SeaORM non mappées.

© Gitrust

Patterns communs

Accès à un service depuis un handler

```
use gitrust_core::services::issue_service::IssueService;

async fn list_issues_handler(
    State(db): State<DatabaseConnection>,
    user: AuthUser,
) -> Result<impl IntoResponse, AppError> {
    let issues = IssueService::list(&db, repo_id, "open", None, 1, 30).await?;
    // ...
}
```

Vérification d'ownership systématique (anti-IDOR)

```
// Toujours passer owner_id au service — le service vérifie en interne
let repo = RepositoryService::find_by_owner_and_slug(&db, &owner, &repo_slug).await?;
// Puis vérifier que l'utilisateur courant peut accéder
if !ResourceService::user_can_access(&db, user.user_id, "repository", repo.id, "read").
    return Err(AppError::Forbidden);
}
```

Voir aussi

- Ajouter un service métier
- Schéma base de données
- API rustwarden-core

© Gitrust

1.4.4 Schéma de la base de données — Référence développeur

Ce document décrit le schéma complet de la base de données gitrust, orienté développeur : types SeaORM côté Rust, enums, contraintes de modèle, et ordre des migrations.

Pour la vue admin (sauvegardes, supervision), consultez le manuel d'administration.

ERD — Diagramme entité-relation

```
erDiagram
%% ===== Tables rustwarden-core =====
users {
uuid id PK
string username UK
string email UK
string password_hash
boolean must_change_password
boolean email_verified
timestamptz last_login_at
timestamptz created_at
timestamptz updated_at
}

resources {
uuid id PK
string resource_type
uuid resource_id
uuid owner_id FK
boolean is_public
timestamptz created_at
}

resource_shares {
uuid id PK
uuid resource_id FK
uuid shared_with_user_id FK
string permission_level
uuid shared_by_user_id FK
timestamptz created_at
}

user_totp {
```

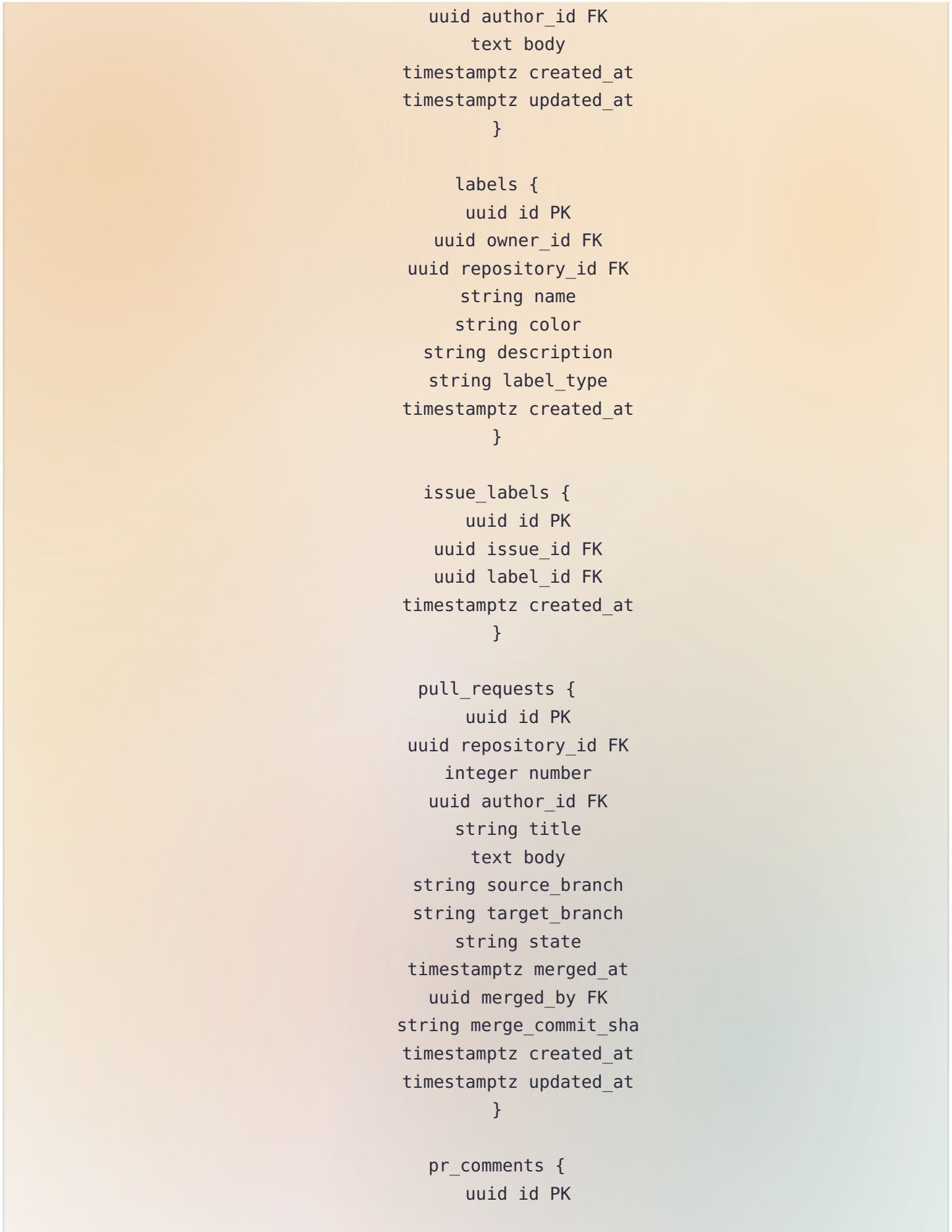

© Gitrust

```
        uuid id PK
    uuid user_id FK
  text encrypted_secret
    boolean enabled
  text backup_codes_json
  timestamptz enabled_at
  timestamptz created_at
  timestamptz updated_at
        }

%% ===== Tables gitrust-core =====
    repositories {
        uuid id PK
     uuid owner_id FK
       string slug
    string description
     string disk_path
  string default_branch
    boolean is_empty
  timestamptz created_at
  timestamptz updated_at
        }

    ssh_keys {
       uuid id PK
    uuid user_id FK
     string title
  string fingerprint UK
    string key_type
     text key_data
 timestamptz last_used_at
  timestamptz created_at
        }

     teams {
       uuid id PK
    uuid owner_id FK
       string slug
    string description
  timestamptz created_at
  timestamptz updated_at
        }
```


© Gitrust

```
    team_members {
        uuid id PK
      uuid team_id FK
      uuid user_id FK
        string role
   timestamptz created_at
        }

team_repository_access {
        uuid id PK
      uuid team_id FK
   uuid repository_id FK
     string permission
   timestamptz created_at
        }

personal_access_tokens {
        uuid id PK
      uuid user_id FK
        string name
    string token_hash UK
       string scopes
   timestamptz expires_at
  timestamptz last_used_at
   timestamptz created_at
        }

       issues {
        uuid id PK
   uuid repository_id FK
      integer number
     uuid author_id FK
        string title
         text body
       string state
     uuid closed_by FK
   timestamptz created_at
   timestamptz updated_at
        }

    issue_comments {
        uuid id PK
     uuid issue_id FK
```


© Gitrust


uuid author_id FK
text body
timestamptz created_at
timestamptz updated_at
}
labels {
uuid id PK
uuid owner_id FK
uuid repository_id FK
string name
string color
string description
string label_type
timestamptz created_at
}
issue_labels {
uuid id PK
uuid issue_id FK
uuid label_id FK
timestamptz created_at
}
pull_requests {
uuid id PK
uuid repository_id FK
integer number
uuid author_id FK
string title
text body
string source_branch
string target_branch
string state
timestamptz merged_at
uuid merged_by FK
string merge_commit_sha
timestamptz created_at
timestamptz updated_at
}
pr_comments {
uuid id PK

© Gitrust

```
        uuid pull_request_id FK
           uuid author_id FK
               text body
        timestamptz created_at
                }

           import_jobs {
              uuid id PK
        uuid repository_id FK
           uuid owner_id FK
           text source_url
           text target_slug
             string status
             string phase
         int received_objects
           int total_objects
        bigint received_bytes
          text error_message
        timestamptz started_at
        timestamptz finished_at
            int duration_ms
        timestamptz created_at
        timestamptz updated_at
                }

              %% Relations
        users ||--o{ repositories : "possède"
        users ||--o{ ssh_keys : "enregistre"
            users ||--o{ teams : "crée"
         users ||--o{ resources : "possède"
        users ||--o| user_totp : "configure"
    repositories ||--o| resources : "enregistre dans"
      resources ||--o{ resource_shares : "partage"
         teams ||--o{ team_members : "contient"
    teams ||--o{ team_repository_access : "accède à"
repositories ||--o{ team_repository_access : "accessible par"
       repositories ||--o{ issues : "contient"
         issues ||--o{ issue_comments : "a"
       issues ||--o{ issue_labels : "étiquetée"
       labels ||--o{ issue_labels : "appliquée"
     repositories ||--o{ pull_requests : "contient"
         pull_requests ||--o{ pr_comments : "a"
```


© Gitrust

```
    users ||--o{ personal_access_tokens : "crée"
    users ||--o{ import_jobs : "lance"
```

Tables rustwarden-core (framework — read-only pour gitrust)

Ces tables sont gérées exclusivement par le framework rustwarden-core. Ne les modifiez jamais directement depuis gitrust.

Table	Rôle
`users`	Comptes utilisateurs
`roles` / `permissions` / `role_permissions` / `user_roles`	RBAC global
`resources` / `resource_shares`	Registre générique de ressources et partages
`refresh_tokens` / `jwt_blacklist`	Gestion JWT
`app_settings`	Configuration applicative clé-valeur
`audit_logs`	Journal d'audit
`user_totp` / `totp_challenges`	Authentification 2FA TOTP
`email_queue` / `email_delivery_status`	File d'emails asynchrone
`password_reset_tokens` / `password_change_requests`	Flux de changement de mot de passe

© Gitrust

Tables gitrust-core

`repositories`

Colonne	Type Rust (SeaORM)	Contrainte PG	Notes
`id`	`Uuid`	PK	
`owner_id`	`Uuid`	FK → users ON DELETE CASCADE	
`slug`	`String`	VARCHAR(64) NOT NULL	Newtype `RepoSlug` à la frontière
`description`	`Option<String>`	VARCHAR(500) NULL	
`disk_path`	`String`	VARCHAR(512) NOT NULL	Chemin absolu du bare repo
`default_branch`	`String`	VARCHAR(255) NOT NULL DEFAULT 'main'	
`is_empty`	`bool`	BOOLEAN NOT NULL DEFAULT true	Passe à false après le premier push
`created_at`	`DateTimeUtc`	TIMESTAMPTZ NOT NULL	
`updated_at`	`DateTimeUtc`	TIMESTAMPTZ NOT NULL	

Index : `UNIQUE(owner_id, slug)`, `INDEX(owner_id)`

Lien ResourceService : chaque dépôt est enregistré dans `resources` avec `resource_type = "repository"` et `resource_id = repositories.id`.

© Gitrust

`ssh_keys`

Colonne	Type Rust	Contrainte PG	Notes
`id`	`Uuid`	PK	
`user_id`	`Uuid`	FK → users CASCADE	
`title`	`String`	VARCHAR(255) NOT NULL	
`fingerprint`	`String`	VARCHAR(128) UNIQUE NOT NULL	SHA256, format `SHA256:xxx`
`key_type`	`String`	VARCHAR(32) NOT NULL	`ssh-ed25519`, `ssh-rsa`, `ecdsa-sha2-nistp256/384`
`key_data`	`String`	TEXT NOT NULL	Format `authorized_keys` complet
`last_used_at`	`Option<DateTimeUtc>`	TIMESTAMPTZ NULL	Mis à jour par le serveur SSH
`created_at`	`DateTimeUtc`	TIMESTAMPTZ NOT NULL	

Validation : RSA ≥ 4096 bits obligatoire. Fingerprint calculé à l'insertion.

`teams` et `team_members`

Colonne	Type Rust	Notes
`teams.slug`	`String`	Newtype `TeamSlug` ; UNIQUE(owner_id, slug)
`team_members.role`	`String`	`"read"` (défaut), `"write"`, `"admin"`
`team_repository_access.permission`	`String`	`"read"`, `"write"`, `"admin"`

© Gitrust

`personal_access_tokens`

Colonne	Type Rust	Notes
`token_hash`	`String`	SHA-256 du token en clair. Le token en clair n'est retourné qu'à la création.
`scopes`	`String`	Espace-séparé : `"repo:read repo:write issues:write"`
`expires_at`	`Option<DateTimeUtc>`	NULL = pas d'expiration

`issues`

Colonne	Type Rust	Notes
`number`	`i32`	Auto-incrémenté par dépôt (pas global). UNIQUE(repository_id, number)
`state`	`String`	`"open"` (défaut) ou `"closed"`
`body`	`String`	Markdown brut

`labels`

Colonne	Type Rust	Notes
`label_type`	`String`	`"classification"` (scope owner) ou `"subject"` (scope repo)
`owner_id`	`Option<Uuid>`	Non-null pour classification, null pour subject
`repository_id`	`Option<Uuid>`	Non-null pour subject, null pour classification
`color`	`String`	Hex `#RRGGBB`, validé à l'insertion

Index : `UNIQUE(owner_id, repository_id, name, label_type)`

`pull_requests`

Colonne	Type Rust	Notes
`number`	`i32`	Auto-incrémenté par dépôt. UNIQUE(repository_id, number)
`state`	`String`	`"open"`, `"closed"`, `"merged"`
`merge_commit_sha`	`Option<String>`	SHA40, null avant merge

© Gitrust

`import_jobs`

Colonne	Type Rust	Notes
`status`	`String`	State machine : `pending` → `running` → `success` / `failed` / `cancelled`
`phase`	`Option<String>`	`"cloning"`, `"resolving"`, `"finalizing"`
`repository_id`	`Option<Uuid>`	Null pendant le clonage, rempli au succès uniquement

© Gitrust

Ordre des migrations

#	Nom	Tables créées
1	`m20260305_000001_initial_schema`	users, refresh_tokens, jwt_blacklist, audit_logs (core)
2	`m20260306_000002_create_app_settings_table`	app_settings (core)
3	`m20260309_000003_create_permissions_tables`	roles, permissions, role_permissions, user_roles (core)
4	`m20260309_000004_create_resources_tables`	resources, resource_shares (core)
5	`m20260310_000005_create_oauth_accounts_table`	oauth_accounts (core)
6	`m20260325_000001_create_repositories`	**repositories**
7	`m20260325_000002_create_ssh_keys`	**ssh_keys**
8	`m20260325_000003_create_teams`	**teams**
9	`m20260325_000004_create_team_members`	**team_members**
10	`m20260325_000005_create_team_repository_access`	**team_repository_access**
11	`m20260327_000006_create_personal_access_tokens`	**personal_access_tokens**
12	`m20260327_000007_create_issues`	**issues**
13	`m20260327_000008_create_issue_comments`	**issue_comments**
14	`m20260327_000009_create_labels`	**labels**, **issue_labels**
15	`m20260327_000010_add_label_type`	ALTER labels (label_type)
16	`m20260327_000011_labels_owner_scope`	ALTER labels (owner_id, repository_id nullable)
17	`m20260327_000012_create_pull_requests`	**pull_requests**
18	`m20260327_000013_create_pr_comments`	**pr_comments**
19	`m20260409_000006_create_totp_tables`	**user_totp**, **totp_challenges** (core)
20+	`m20260416_000022_create_import_jobs`	**import_jobs**

© Gitrust

Toutes les migrations sont exécutées par `AppMigrator` (combine core + gitrust) dans `gitrust_core::migrations::run_migrations()`.

Voir aussi

- Ajouter une migration DB
- Services API interne
- Modèle de permissions RBAC

© Gitrust

1.4.5 Règles QA & conformité ANSSI

Ce document définit les règles de qualité et de sécurité applicables à tout le projet gitrust. Chaque lot doit satisfaire ces règles avant merge.

1. Gates de compilation obligatoires

Toute modification doit passer ces 4 gates sans erreur :

Gate	Commande	Critère
Formatage	`cargo fmt --all -- --check`	Zéro diff
Linting	`cargo clippy --workspace -- -D warnings`	Zéro warning
Tests unitaires	`cargo test --workspace`	100% pass
Build CSS	`npx tailwindcss -i static/css/input.css -o static/css/style.css --minify`	Si templates modifiés

Gates supplémentaires (à mettre en place)

Gate	Outil	Rôle
Audit dépendances	`cargo audit`	Détection CVE dans les deps
Licences & bans	`cargo deny check`	Licences compatibles, pas de crate bannie
Secrets dans le code	Recherche de patterns sensibles	Pas de token/mot de passe en dur

2. Règles de sécurité (ANSSI PA-074)

2.1 Lints Rust obligatoires (déjà en place)

```
#![forbid(unsafe_code)]                    // core, web, hooks
#![deny(unsafe_code)]                      // git, ssh (FFI nécessaire)
#![deny(clippy::unwrap_used)]
#![deny(clippy::expect_used)]
#![deny(clippy::panic)]
```


© Gitrust

```
#![deny(clippy::indexing_slicing)]
#![deny(clippy::mem_forget)]
```

2.2 Checklist sécurité par feature

Avant toute implémentation touchant l'authentification, les secrets ou les permissions, vérifier :

- [ ] **Secrets jamais en clair en base** — hash SHA-256 ou bcrypt
- [ ] **Secrets jamais loggés** — vérifier `tracing::info/debug/warn`, `Display` et `Debug` impls ne révèlent pas le secret
- [ ] **Comparaison constant-time** — `subtle::ConstantTimeEq` pour toute validation de secret (pas de `==` sur des hash)
- [ ] **Rate-limiting** — endpoint d'authentification protégé
- [ ] **CSRF** — token valide sur toute action mutante (POST/PUT/DELETE)
- [ ] **Ownership vérifiée (anti-IDOR)** — le user ne peut agir que sur ses propres ressources (vérifier `user_id` côté serveur)
- [ ] **Expiration obligatoire** — tout token/session a une durée de vie max
- [ ] **Audit log** — création, révocation, utilisation suspecte tracées
- [ ] **Zeroize** — types sensibles implémentent `Zeroize` / `ZeroizeOnDrop`
- [ ] **Path traversal** — validation des chemins disque (pas de `..`, `/`, `\`)

2.3 Marqueurs SEC-

Le code utilise des commentaires `// SEC-XX` pour tracer les décisions de sécurité. Tout nouveau contrôle doit être marqué avec le prochain numéro disponible dans sa catégorie :

Préfixe	Catégorie	Exemples
SEC-C	Cryptographie	Timing attack, CSRF, PKCE
SEC-H	HTTP/Headers	X-Forwarded-For, cookies, nonce
SEC-L	Logique métier	Hashing, validation, defaults
SEC-M	Mémoire/sessions	Rate-limit DoS, refresh tokens

© Gitrust

3. Tests requis par catégorie

3.1 Matrice de couverture

Catégorie	Quand appliquer	Exemples
Unitaire	Logique pure (validation, parsing, conversion)	`RepoSlug::new("../evil")` → erreur
Intégration	Service avec DB (CRUD, contraintes, transactions)	`PatService::validate` token expiré → None
Handler	Endpoint HTTP (status, redirect, CSRF, auth)	POST sans CSRF → 403
E2E Playwright	Flow utilisateur complet	Créer token → copier → cloner un dépôt
Sécurité négatif	Tout bypass imaginable	Token user A sur ressource user B → 401

3.2 Règles de tests

1. **Tests d'intégration sur vraie DB** — pas de mocks pour la couche persistence (les mocks masquent les bugs de migration)
2. **Tests négatifs obligatoires** — pour chaque chemin heureux, tester au moins : input invalide, non-authentifié, non-autorisé, expiré, révoqué
3. **Tests E2E en français** — cohérent avec l'UI (locale `fr-FR`)
4. **Pas de `sleep()` dans les tests** — utiliser des retry/poll avec timeout
5. **Données de test isolées** — chaque test crée ses propres données (pas de dépendance sur l'ordre d'exécution)

4. Règles de code

4.1 Framework boundary

- **Ne jamais modifier** `crates/rustwarden-core/`
- Réutiliser les services framework avant d'implémenter (auth, users, sessions, ResourceService, i18n, middleware)
- Si un service manque, l'étendre côté gitrust (wrappers, trait impls)

4.2 Assets

- **Zéro CDN** — tous les CSS/JS servis depuis `static/`
- La CSP du framework bloque les domaines externes

© Gitrust

4.3 Gestion d'erreurs

- `GitrustError` avec `IntoResponse` pour le mapping HTTP
- Pas de `.unwrap()` / `.expect()` / `panic!()` / `[index]`
- Errors utilisateur : messages génériques (pas de leak d'info interne)

4.4 Validation aux frontières

- Valider les inputs utilisateur (formulaires, query params, headers)
- Ne pas re-valider entre services internes
- Newtypes avec validation à la construction (`RepoSlug` , `TeamSlug` , `Fingerprint` , `TokenHash`)

---

5. Checklist pré-merge

Avant chaque merge de lot :

- [ ] `cargo fmt --all -- --check` passe
- [ ] `cargo clippy --workspace -- -D warnings` passe
- [ ] `cargo test --workspace` — tous les tests passent
- [ ] Tests E2E Playwright passent (`npm run test:e2e`)
- [ ] CSS rebuild si templates modifiés
- [ ] Aucun secret en clair dans le code ou les logs
- [ ] Checklist sécurité §2.2 validée (si applicable)
- [ ] Pas de modification dans `crates/rustwarden-core/`

© Gitrust

1.4.6 API REST v1 — Référence

L'API REST gitrust est accessible sous le préfixe `/api/v1/`. Elle accepte et retourne du JSON. Elle partage les mêmes services que l'interface web SSR.

Documentation interactive : `GET /api/docs` sur toute instance gitrust.

Authentification

Deux méthodes sont acceptées sur tous les endpoints authentifiés :

Cookie JWT (navigateur) — défini automatiquement par le serveur après login :

```
Cookie: jwt_token=<JWT>
```

Bearer PAT (clients API, CI, scripts) — Personal Access Token créé depuis `/settings/tokens` :

```
Authorization: Bearer <PAT>
```

Les PAT supportent des scopes : `repo:read`, `repo:write`, `issues:read`, `issues:write`, `pulls:read`, `pulls:write`, `ci:read`, `ci:write`.

Un endpoint qui requiert `repo:write` rejette un PAT avec seulement `repo:read` avec `403 Forbidden`.

Format des erreurs

Toutes les erreurs retournent ce format JSON :

```
{
    "error": "not_found",
    "message": "Le dépôt 'alice/myrepo' est introuvable",
    "status": 404
}
```


© Gitrust

Codes d'erreur standards :

`error`	HTTP	Cause
`unauthorized`	401	Token absent, expiré ou invalide
`forbidden`	403	Permissions insuffisantes
`not_found`	404	Ressource inexistante
`validation_error`	400	Corps de requête invalide
`conflict`	409	Contrainte d'unicité violée
`rate_limited`	429	Trop de requêtes
`internal_error`	500	Erreur serveur

Pagination

Tous les endpoints de liste acceptent `?page=N&per_page=N` (défaut : `page=1`, `per_page=30`, max `per_page=100`).

Headers de réponse :

```
X-Total-Count: 142
X-Page: 2
X-Per-Page: 30
```

Rate limiting

En cas de dépassement : `429 Too Many Requests` avec `Retry-After: <secondes>`.

Endpoints

Authentification

`POST /api/v1/auth/login`

```
// Requête
{ "username": "alice", "password": "SecurePass123!" }

// Réponse 200 (sans 2FA)
{
```


© Gitrust

```
    "access_token": "<JWT>",
    "refresh_token": "<token>",
    "token_type": "Bearer",
    "user": { "id": "...", "username": "alice", "email": "alice@example.com" }
}

// Réponse 200 (avec 2FA activé)
{ "requires_2fa": true, "challenge_token": "abc123..." }
```

`POST /api/v1/auth/2fa/verify`

```
// Requête
{ "challenge_token": "abc123...", "code": "123456" }

// Réponse 200
{ "access_token": "...", "refresh_token": "...", "user": { ... } }
```

`POST /api/v1/auth/refresh`

```
// Requête
{ "refresh_token": "<token>" }

// Réponse 200
{ "access_token": "<nouveau JWT>", "refresh_token": "<nouveau token>" }
```

`GET /api/v1/auth/me`

```
// Réponse 200
{
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "username": "alice",
    "email": "alice@example.com",
    "email_verified": true,
    "roles": ["user"]
}
```


© Gitrust

Utilisateur courant

`GET /api/v1/user`

Profil de l'utilisateur authentifié. Identique à `GET /api/v1/auth/me`.

`GET /api/v1/user/repos`

Liste les dépôts accessibles par l'utilisateur courant (possédés + partagés).

```
// Réponse 200
{
    "items": [
        {
            "id": "...",
            "owner": "alice",
            "slug": "myrepo",
            "description": "Mon premier dépôt",
            "default_branch": "main",
            "is_empty": false,
            "is_public": false,
            "created_at": "2026-03-25T10:00:00Z"
        }
    ],
    "total": 1,
    "page": 1,
    "per_page": 30
}
```

Dépôts

`GET /api/v1/repos/{owner}/{repo}`

```
// Réponse 200
{
    "id": "...",
    "owner": "alice",
    "slug": "myrepo",
    "description": "Mon dépôt",
    "default_branch": "main",
    "is_empty": false,
    "is_public": true,
    "created_at": "2026-03-25T10:00:00Z",
```


© Gitrust

```
    "updated_at": "2026-04-01T12:00:00Z"
}
```

`GET /api/v1/repos/{owner}/{repo}/branches`

```
// Réponse 200
{
    "items": [
        { "name": "main", "is_default": true, "commit_sha": "abc123..." },
        { "name": "feat/42-fix", "is_default": false, "commit_sha": "def456..." }
    ],
    "total": 2, "page": 1, "per_page": 30
}
```

`GET /api/v1/repos/{owner}/{repo}/commits`

Paramètres optionnels : `?ref=main&page=1&per_page=30`

```
// Réponse 200
{
    "items": [
        {
            "sha": "abc123...",
            "message": "feat: ajouter la signature HMAC",
            "author": { "name": "Alice", "email": "alice@example.com" },
            "committed_at": "2026-04-01T09:00:00Z"
        }
    ],
    "total": 42, "page": 1, "per_page": 30
}
```

Issues

`GET /api/v1/repos/{owner}/{repo}/issues`

Paramètres : `?state=open&page=1&per_page=30`

```
// Réponse 200
{
    "items": [
        {
```


© Gitrust

```
            "id": "...",
            "number": 1,
            "title": "Bug dans le parser",
            "state": "open",
            "author": "alice",
            "labels": [
                { "name": "bug", "color": "#d73a4a", "type": "classification" }
            ],
            "comment_count": 3,
            "created_at": "2026-03-27T08:00:00Z"
        }
    ],
    "total": 5, "page": 1, "per_page": 30
}
```

`POST /api/v1/repos/{owner}/{repo}/issues`

```
// Requête
{ "title": "Nouveau bug", "body": "Description en Markdown..." }

// Réponse 201
{ "id": "...", "number": 6, "title": "Nouveau bug", "state": "open", ... }
```

`PATCH /api/v1/repos/{owner}/{repo}/issues/{num}`

```
// Requête (tous les champs optionnels)
{ "state": "closed" }

// Réponse 200
{ "id": "...", "number": 1, "state": "closed", ... }
```

Pull Requests

`GET /api/v1/repos/{owner}/{repo}/pulls`

Paramètres : `?state=open&page=1&per_page=30`

`POST /api/v1/repos/{owner}/{repo}/pulls`

```
// Requête
{
```


© Gitrust

```
    "title": "feat: ajouter la signature HMAC",
    "body": "Closes #42",
    "source_branch": "feat/42-hmac",
    "target_branch": "main"
}

// Réponse 201
{ "id": "...", "number": 3, "state": "open", ... }
```

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

```
// Requête
{ "merge_method": "merge" }
// merge_method: "merge" | "squash" | "rebase"

// Réponse 200
{ "merged": true, "merge_commit_sha": "abc123..." }
```

CI Pipelines

`GET /api/v1/repos/{owner}/{repo}/ci/pipelines`

Paramètres : `?page=1&per_page=30`

```
// Réponse 200
{
    "items": [
        {
            "id": "...",
            "status": "success",
            "trigger": "push",
            "ref": "main",
            "commit_sha": "abc123...",
            "duration_ms": 45000,
            "created_at": "2026-04-01T09:00:00Z"
        }
    ],
    "total": 12, "page": 1, "per_page": 30
}
```


© Gitrust

`POST /api/v1/repos/{owner}/{repo}/ci/pipelines/trigger`

Déclenche un pipeline manuellement.

```
// Requête
{ "ref": "main" }

// Réponse 201
{ "id": "...", "status": "pending", ... }
```

`GET /api/v1/repos/{owner}/{repo}/ci/pipelines/{id}/logs`

```
// Réponse 200
{
    "items": [
        { "line": 1, "content": "$ cargo build --release", "stream": "stdout", "timesta
        { "line": 2, "content": "   Compiling gitrust v0.1.0", "stream": "stdout", "tim
    ],
    "total": 248, "page": 1, "per_page": 100
}
```

Exemples clients

Lister les issues avec curl

```
curl -s \
  -H "Authorization: Bearer ${GITRUST_PAT}" \
  "https://demo.gitrust.eu/api/v1/repos/alice/myrepo/issues?state=open" \
  | jq '.items[].title'
```

Créer une issue avec Python

```
import httpx

client = httpx.Client(
    base_url="https://demo.gitrust.eu",
    headers={"Authorization": f"Bearer {PAT}"},
)

resp = client.post(
```


© Gitrust

```
    "/api/v1/repos/alice/myrepo/issues",
    json={"title": "Bug trouvé", "body": "Description..."},
)
resp.raise_for_status()
print(resp.json()["number"])
```

Créer une issue avec Rust

```
use reqwest::Client;
use serde_json::json;

let client = Client::new();
let resp = client
    .post("https://demo.gitrust.eu/api/v1/repos/alice/myrepo/issues")
    .bearer_auth(&pat)
    .json(&json!({"title": "Bug", "body": "..."}))
    .send()
    .await?;

println!("Issue #{}", resp.json::<serde_json::Value>().await?["number"]);
```

Voir aussi

- Implémenter un endpoint API
- Services API interne

© Gitrust

1.4.7 Hooks gitrust — Référence

Ce document décrit les hooks exposés par `crates/gitrust-hooks/`, quand ils sont appelés, leur payload, leurs effets de bord, et comment les tester localement.

Vue d'ensemble

`gitrust-hooks` implémente le trait `RustwardenHooks` de rustwarden-core. Ce trait est un mécanisme de lifecycle : le framework appelle les hooks aux moments clés du cycle de vie des ressources (inscription utilisateur, suppression, partage).

```
// crates/gitrust-hooks/src/lib.rs
#[async_trait]
impl RustwardenHooks for GitrustHooks {
    async fn on_user_registered(...) -> anyhow::Result<()>;
    async fn on_resource_shared(...) -> anyhow::Result<()>;
    async fn on_user_deleted(...) -> anyhow::Result<()>;
    async fn on_resource_unshared(...) -> anyhow::Result<()>;
}
```

Les hooks Git serveur (receive-pack, upload-pack) sont implémentés directement dans `gitrust-web/src/handlers/git_http.rs` et dans le serveur SSH (`gitrust-ssh`).

Hooks de lifecycle (`RustwardenHooks`)

`on_user_registered`

Déclencheur : après la création réussie d'un compte utilisateur (via `/register` ou l'API `/api/v1/auth/register`).

Payload :

```
async fn on_user_registered(
    db: &DatabaseConnection,
    user_id: Uuid,
    username: &str,
) -> anyhow::Result<()>
```

Effet de bord : crée le répertoire de dépôts de l'utilisateur sur le système de fichiers :

```
{REPOS_BASE_PATH}/{username}/
```


© Gitrust

Validation de sécurité : le username est validé contre la traversal de chemin (`..`, `/`, `\`) avant toute opération disque. Un username suspect déclenche un log `WARN` et retourne une erreur — le compte n'est **pas** créé.

Idempotence : si le répertoire existe déjà, l'appel réussit silencieusement.

`on_user_deleted`

Déclencheur : juste avant la suppression d'un utilisateur depuis le panel admin.

Payload :

```
async fn on_user_deleted(
    db: &DatabaseConnection,
    user_id: Uuid,
    username: &str,
) -> anyhow::Result<()>
```

Effets de bord : 1. Supprime le répertoire disque `{REPOS_BASE_PATH}/{username}/` et tous les bare repos qu'il contient. 2. Enregistre un événement `user_deleted` dans `audit_logs` (niveau WARN).

Note : la cascade DB (suppression des lignes `repositories`) est gérée par la contrainte FK `ON DELETE CASCADE`. Ce hook gère uniquement le système de fichiers.

En cas d'erreur disque (permissions, NFS indisponible), l'erreur est loggée en WARN mais **ne bloque pas** la suppression en base.

`on_resource_shared`

Déclencheur : après un appel réussi à `ResourceService::share()` — quand un dépôt est partagé avec un collaborateur.

Payload :

```
async fn on_resource_shared(
    db: &DatabaseConnection,
    resource_type: &str,
    resource_id: Uuid,
    shared_with_user_id: Uuid,
    permission_level: &str,
```


© Gitrust

```
    shared_by_user_id: Uuid,
) -> anyhow::Result<()>
```

Effet de bord : si `resource_type == "repository"`, enregistre l'événement `REPO_SHARED` dans `audit_logs` avec les détails du partage.

`on_resource_unshared`

Déclencheur : après un appel réussi à `ResourceService::revoke_share()` ou `revoke_user_access()`.

Payload :

```
async fn on_resource_unshared(
    db: &DatabaseConnection,
    resource_type: &str,
    resource_id: Uuid,
    user_id: Uuid,
) -> anyhow::Result<()>
```

Effet de bord : si `resource_type == "repository"`, enregistre l'événement `REPO_UNSHARED` dans `audit_logs`.

Hooks Git serveur (receive-pack / upload-pack)

Ces hooks ne sont pas dans `gitrust-hooks` mais dans les handlers HTTP et SSH.

Receive-pack (après un `git push`)

Déclencheur : `POST /{owner}/{repo}/git-receive-pack` (HTTP) ou connexion SSH suivie d'un `git-receive-pack`.

Séquence d'effets :

```
sequenceDiagram
    participant Client as git push
    participant Handler as receive_pack handler
    participant DB as Base de données
    participant CI as CI Worker
    participant WH as Webhook Dispatcher

    Client->>Handler: objets packés
    Handler->>DB: RepositoryService::mark_non_empty() si premier push
    Handler->>DB: CiService::create_pipeline() si .gitrust-ci.yml détecté
```


© Gitrust

```
                    Handler->>CI: mpsc::send(CiTask)
            Handler->>DB: WebhookService::list_by_repo_and_event("push")
          Handler->>WH: tokio::spawn → deliver(webhook, "push", payload)
                    Handler-->>Client: 200 OK (pack-protocol)
```

Payload webhook push :

```
{
    "repository": {
        "id": "...",
        "slug": "myrepo",
        "owner": "alice"
    },
    "pusher": { "username": "alice" },
    "ref": "refs/heads/main",
    "before": "0000000000000000000000000000000000000000",
    "after": "abc123...",
    "commits": [
        {
            "sha": "abc123...",
            "message": "feat: ajouter la signature HMAC",
            "author": { "name": "Alice", "email": "alice@example.com" }
        }
    ]
}
```

Upload-pack (après un `git fetch` / `git clone`)

Déclencheur : `GET /{owner}/{repo}/info/refs?service=git-upload-pack` + `POST /{owner}/{repo}/git-upload-pack`.

Effets de bord : aucun effet de bord métier. Vérification des permissions d'accès (public → lecture libre ; privé → authentification requise).

Hooks SBOM (post-push)

Après chaque push sur un dépôt avec `SBOM_ENABLED=true`, un job SBOM est déclenché via `SbomService` :

1. `SbomService::trigger_scan(db, repo_id, commit_sha)` crée un job dans `ci_jobs`.
2. Le worker CI exécute `syft` sur le répertoire de travail.
3. Le résultat est uploadé vers Dependency-Track via HTTP.
4. Les findings critiques peuvent bloquer la CI selon la configuration `sbom_block_on_violation`.

© Gitrust

Tester les hooks localement

Tester `on_user_registered`

```
// tests/hooks_test.rs
#[tokio::test]
async fn on_user_registered_creates_dir() {
    let tmp = tempfile::TempDir::new().unwrap();
    let hooks = GitrustHooks::new(tmp.path().to_path_buf());
    let db = sea_orm::Database::connect("sqlite::memory:").await.unwrap();

    hooks
        .on_user_registered(&db, Uuid::new_v4(), "alice")
        .await
        .unwrap();

    assert!(tmp.path().join("alice").exists());
}

#[tokio::test]
async fn on_user_registered_rejects_traversal() {
    let tmp = tempfile::TempDir::new().unwrap();
    let hooks = GitrustHooks::new(tmp.path().to_path_buf());
    let db = sea_orm::Database::connect("sqlite::memory:").await.unwrap();

    assert!(hooks
        .on_user_registered(&db, Uuid::new_v4(), "../etc/passwd")
        .await
        .is_err());
}
```

Simuler un push en local

Pour tester le cycle complet receive-pack → CI → webhook sans client réel :

```
# 1. Démarrer gitrust localement
cargo run

# 2. Créer un dépôt via l'UI, cloner en SSH ou HTTP
git clone http://localhost:4000/alice/myrepo.git

# 3. Ajouter un fichier et pusher
cd myrepo
```


© Gitrust

```
echo "hello" > README.md
git add README.md && git commit -m "test push"
git push origin main

# 4. Vérifier les effets de bord dans les logs
# → Log "Created pipeline" si .gitrust-ci.yml présent
# → Log "Delivered webhook" si webhook configuré
```

Ajouter un nouveau hook

Pour réagir à un nouvel événement de lifecycle :

1. Définissez la méthode dans le trait `RustwardenHooks` (dans `rustwarden-core/src/hooks.rs`).
2. Implémentez la méthode dans `GitrustHooks` (dans `gitrust-hooks/src/lib.rs`).
3. Appelez le hook depuis le service rustwarden-core approprié.
4. Écrivez un test unitaire avec `tempfile::TempDir` et `sea_orm::Database::connect("sqlite::memory:")`.

Règle : les hooks ne doivent jamais bloquer un flux utilisateur critique. En cas d'erreur non fatale (écriture disque, audit log), loggez en WARN et retournez `Ok(())`.

Voir aussi

- Implémenter des webhooks
- Patron worker async

© Gitrust

1.4.8 API rustwarden-core — Référence

Ce document décrit les helpers, extracteurs Axum, middleware, et APIs d'extension fournis par le framework rustwarden-core que gitrust consomme.

Extracteurs Axum

`AuthUser`

Module : `rustwarden_core::auth::extractors`
Export : prélude

Extracteur qui valide le JWT depuis le cookie `jwt_token` **ou** le header `Authorization: Bearer <token>`. Retourne `401 Unauthorized` si le token est absent, expiré, invalide, ou blacklisté.

```
use rustwarden_core::prelude::*;

async fn handler(user: AuthUser) -> impl IntoResponse {
    // Champs disponibles :
    let _id: Uuid = user.user_id;
    let _name: &str = &user.username;
    let _roles: &[String] = &user.roles;
    let _perms: &[String] = &user.permissions;
}
```

Méthodes de vérification :

Méthode	Description
`has_role(role: &str) → bool`	L'utilisateur a-t-il ce rôle ?
`has_any_role(roles: &[&str]) → bool`	A-t-il au moins un des rôles ?
`has_all_roles(roles: &[&str]) → bool`	A-t-il tous les rôles ?
`has_permission(permission: &str) → bool`	A-t-il cette permission ? Le rôle `admin` retourne toujours `true`.

`AuthenticatedSession`

Module : `rustwarden_core::auth::session`

© Gitrust

Extracteur similaire à `AuthUser` mais inclut les informations de session (device info, IP). Utilisé pour la page `/settings/sessions`.

```
async fn sessions_handler(
    State(db): State<DatabaseConnection>,
    session: AuthenticatedSession,
) -> impl IntoResponse {
    let current_token_id = session.token_id; // UUID du refresh token courant
    // Utilisé pour marquer "Current session" dans l'UI
}
```

`OptionalUser`

Extracteur qui ne rejette pas les requêtes non authentifiées — retourne `Option<AuthUser>`. Utilisé pour les pages publiques (dépôts publics, explore).

```
async fn public_page(user: OptionalUser) -> impl IntoResponse {
    match user.0 {
        Some(u) => format!("Bonjour {}", u.username),
        None => "Visiteur anonyme".to_owned(),
    }
}
```

Middleware

Middleware d'authentification JWT

Intégré automatiquement dans `RustwardenApp::build()`. Gère : - Lecture du JWT depuis `Cookie: jwt_token=<JWT>` ou `Authorization: Bearer <JWT>`. - Vérification de la signature HMAC-SHA256 avec la clé `JWT_SECRET`. - Vérification de l'expiration (`exp`). - Vérification de la blacklist (table `jwt_blacklist`).

Ce middleware **ne bloque pas** les requêtes anonymes — c'est l'extracteur `AuthUser` dans le handler qui décide si l'authentification est requise.

Middleware CSRF

Protège toutes les requêtes mutantes (POST, PUT, DELETE, PATCH) émises par des formulaires HTML. Le token CSRF est : - Généré à la connexion et stocké dans un cookie `csrf_token` (HttpOnly, SameSite=Strict). - Attendu dans le corps du formulaire sous le champ `_csrf`. - Absent ou invalide → `403 Forbidden`.

© Gitrust

Pour les endpoints API JSON (header `Content-Type: application/json`), la vérification CSRF est désactivée — l'authentification Bearer PAT/JWT suffit.

```
<!-- Dans tout formulaire POST -->
<input type="hidden" name="_csrf" value="{{ csrf_token }}">
```

Middleware de rate limiting

Basé sur une fenêtre glissante en mémoire (par IP ou par `user_id`). Configurable via les variables d'environnement :

```
RATE_LIMIT_LOGIN=10          # tentatives de login / 15 min / IP
RATE_LIMIT_API_READ=1000     # requêtes lecture / heure / user
RATE_LIMIT_API_WRITE=200     # requêtes écriture / heure / user
```

`Claims` — contenu du JWT

```
pub struct Claims {
    pub sub: Uuid,              // user_id
    pub username: String,
    pub roles: Vec<String>,
    pub permissions: Vec<String>,
    pub exp: i64,               // timestamp d'expiration
    pub iat: i64,               // timestamp d'émission
    pub iss: String,            // issuer (app_domain)
    pub jti: Uuid,              // JWT ID unique (pour blacklist)
}
```


© Gitrust

`JwtService`

Module : `rustwarden_core::auth::jwt`
Export : prélude

Méthode	Signature	Description
`generate_token`	`(config, user_id, username, roles, permissions) → Result<String>`	Génère un JWT signé HS256.
`validate_token`	`(config, token: &str) → Result<Claims>`	Valide et décode. Retourne `TokenExpired` si expiré.
`extract_token_from_header`	`(header: &str) → Result<&str>`	Extrait le token du header `Authorization: Bearer <token>`.

Trait `RustwardenHooks`

Interface à implémenter dans votre projet consommateur pour réagir aux événements du framework.

```
#[async_trait]
pub trait RustwardenHooks: Send + Sync {
    async fn on_user_registered(
        &self,
        db: &DatabaseConnection,
        user_id: Uuid,
        username: &str,
    ) -> anyhow::Result<()> {
        Ok(())  // défaut no-op
    }

    async fn on_user_deleted(
        &self,
        db: &DatabaseConnection,
        user_id: Uuid,
        username: &str,
    ) -> anyhow::Result<()> {
        Ok(())
    }
```


© Gitrust

```
    async fn on_resource_shared(
        &self,
        db: &DatabaseConnection,
        resource_type: &str,
        resource_id: Uuid,
        shared_with_user_id: Uuid,
        permission_level: &str,
        shared_by_user_id: Uuid,
    ) -> anyhow::Result<()> {
        Ok(())
    }

    async fn on_resource_unshared(
        &self,
        db: &DatabaseConnection,
        resource_type: &str,
        resource_id: Uuid,
        user_id: Uuid,
    ) -> anyhow::Result<()> {
        Ok(())
    }
}
```

Enregistrement dans `main.rs` :

```
let hooks = Arc::new(GitrustHooks::new(repos_base_path.clone()));
let app = RustwardenApp::builder()
    .from_env()
    .hooks(hooks)
    .build()
    .await?;
```

RBAC API

Vérification des permissions dans un handler

```
async fn admin_handler(user: AuthUser) -> Result<impl IntoResponse, AppError> {
    if !user.has_role("admin") {
        return Err(AppError::Forbidden);
    }
```


© Gitrust

```
    // ...
}
```

Vérification des permissions sur une ressource

```
use rustwarden_core::prelude::*;

async fn repo_handler(
    State(db): State<DatabaseConnection>,
    user: AuthUser,
    // ... autres extracteurs
) -> Result<impl IntoResponse, AppError> {
    // Vérifier que l'utilisateur peut accéder en lecture
    if !ResourceService::user_can_access(
        &db, user.user_id, "repository", repo.id, "read"
    ).await? {
        return Err(AppError::Forbidden);
    }

    // Déterminer le niveau d'accès effectif
    match ResourceService::effective_permission(
        &db, user.user_id, "repository", repo.id
    ).await? {
        Some(ref p) if p == "owner" => { /* accès total */ }
        Some(ref p) if p == "write" => { /* écriture permise */ }
        Some(ref p) if p == "read"  => { /* lecture seule */ }
        _ => return Err(AppError::Forbidden),
    }
}
```

---

i18n — Internationalisation

Module : `rustwarden_core::i18n`

Gitrust supporte 6 langues : `fr`, `en`, `de`, `es`, `it`, `pt`.

Les fichiers de traduction sont dans `crates/rustwarden-core/locales/{lang}.yml` et `crates/gitrust-web/locales/{lang}.yml`.

Dans un template Askama :

© Gitrust

```
<!-- Clé de traduction avec paramètre -->
<h1>{{ t!("repository.title", repo_name = repo.slug) }}</h1>

<!-- Clé simple -->
<button>{{ t!("actions.save") }}</button>
```

Dans un handler (pour les messages d'erreur i18n) :

```
use rustwarden_core::i18n::I18n;

let message = I18n::translate(&user_lang, "errors.forbidden", &[]);
```

La langue est déterminée dans l'ordre : préférences utilisateur (table `user_preferences`) → cookie `lang` → header `Accept-Language`.

Email templating

Module : `rustwarden_core::services::email_service`

Pour envoyer un email depuis gitrust, utilisez `EmailQueueService::enqueue` (envoi asynchrone) :

```
use rustwarden_core::services::email_queue_service::EmailQueueService;

EmailQueueService::enqueue(
    &db,
    &to_email,
    Some(&to_name),
    &from_email,
    Some(&from_name),
    "Votre dépôt a été importé",
    &html_body,
    &text_body,
    "import_success",
).await?;
```

Les templates d'email sont dans `crates/rustwarden-core/templates/emails/` (format HTML + texte brut).

© Gitrust

Étendre rustwarden-core

Règle absolue : ne modifiez jamais le code de `crates/rustwarden-core/`. C'est un framework partagé entre plusieurs projets (gitrust, PasterMan, etc.).

Pour étendre une fonctionnalité :

1. **Wrappers dans gitrust-core** : créez un service wrapper dans `crates/gitrust-core/src/services/` qui appelle le service rustwarden-core et ajoute la logique spécifique gitrust.
2. **Trait impls dans gitrust-hooks** : implémentez `RustwardenHooks` pour réagir aux événements du framework.
3. **Feature flags** : certaines fonctionnalités rustwarden-core sont derrière des features Cargo (`oauth`, `totp`, etc.) — activez-les dans `Cargo.toml`.

Voir aussi

- Services API interne
- Hooks git post-receive
- Modèle permissions RBAC

© Gitrust

1.5 Explications

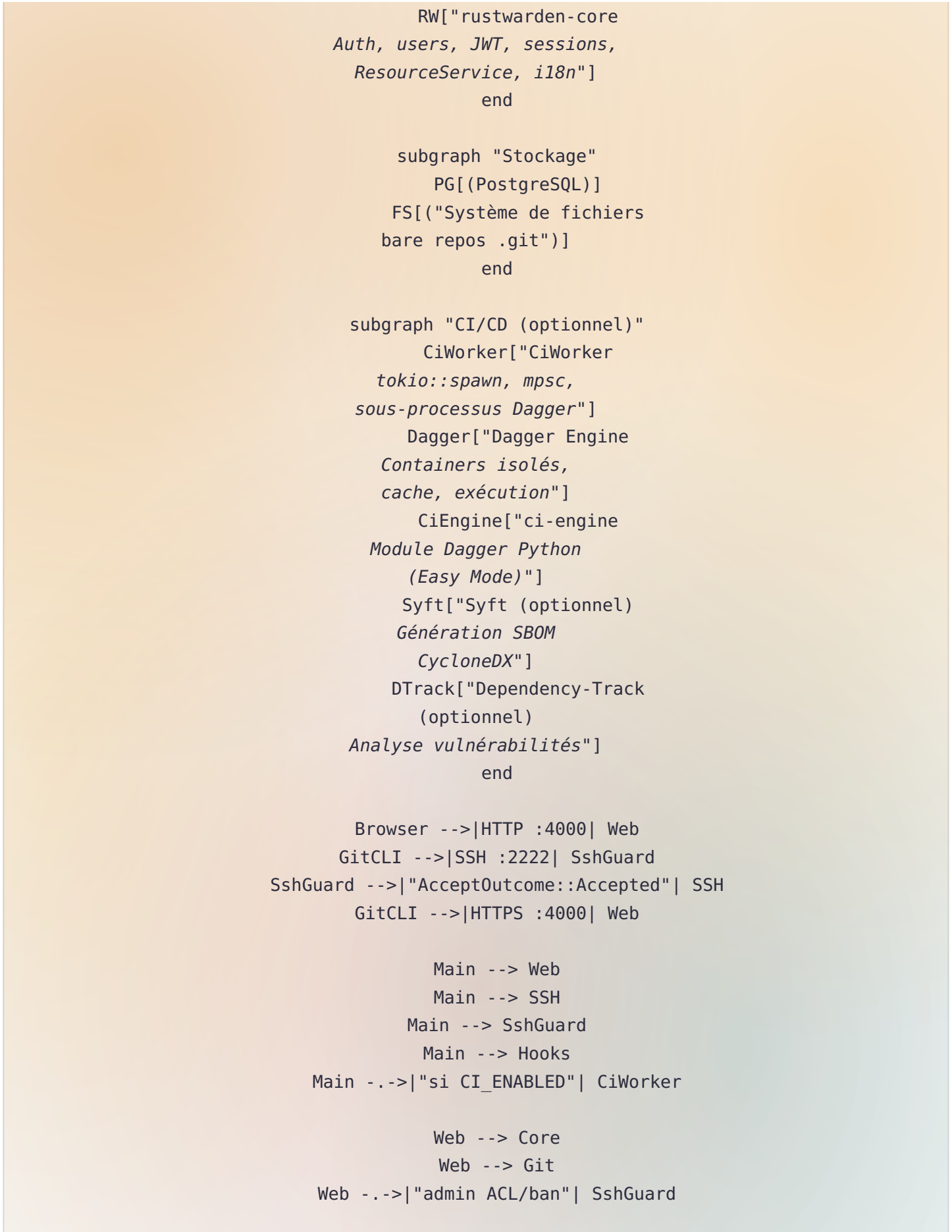
1.5.1 Vue d'ensemble de l'architecture

Gitrust est une plateforme d'hébergement Git auto-hébergée construite sur le framework **rustwarden-core**. L'application expose deux points d'entrée réseau (HTTP + SSH) et persiste ses données dans PostgreSQL.

Vue d'ensemble

```
graph TB
    subgraph Clients
        Browser["Navigateur Web"]
        GitCLI["Git CLI (ssh/https)"]
    end

    subgraph "Gitrust Binary (src/main.rs)"
        Main["main()"]
    end

    subgraph "Crates Gitrust"
        Web["gitrust-web
        Routes HTTP, templates,
        handlers Axum"]
        SSH["gitrust-ssh
        Serveur SSH (russh),
        auth par clé, sessions"]
        SshGuard["gitrust-ssh-guard
        SecureListener, détecteurs,
        BanManager, AuthTracker"]
        Hooks["gitrust-hooks
        impl RustwardenHooks
        (on_user_registered, ...)"]
        Core["gitrust-core
        Models, services, migrations,
        rôles, types, DTOs"]
        Git["gitrust-git
        Bare repos, tree browser,
        pack protocol (git2)"]
    end

    subgraph "Framework"
```


© Gitrust


RW["rustwarden-core
Auth, users, JWT, sessions,
ResourceService, i18n"]
end
subgraph "Stockage"
PG[(PostgreSQL)]
FS[("Système de fichiers
bare repos .git")]
end
subgraph "CI/CD (optionnel)"
CiWorker["CiWorker
tokio::spawn, mpsc,
sous-processus Dagger"]
Dagger["Dagger Engine
Containers isolés,
cache, exécution"]
CiEngine["ci-engine
Module Dagger Python
(Easy Mode)"]
Syft["Syft (optionnel)
Génération SBOM
CycloneDX"]
DTrack["Dependency-Track
(optionnel)
Analyse vulnérabilités"]
end
Browser -->|HTTP :4000| Web
GitCLI -->|SSH :2222| SshGuard
SshGuard -->|"AcceptOutcome::Accepted"| SSH
GitCLI -->|HTTPS :4000| Web
Main --> Web
Main --> SSH
Main --> SshGuard
Main --> Hooks
Main -.->|"si CI_ENABLED"| CiWorker
Web --> Core
Web --> Git
Web -.->|"admin ACL/ban"| SshGuard

© Gitrust

```
SSH --> Core
SSH --> Git
SSH --> SshGuard
SshGuard --> Core
Hooks --> Core

Core --> RW
Core --> PG
Git --> FS

CiWorker --> Core
CiWorker -->|"dagger call"| Dagger
Dagger -->|"Easy Mode"| CiEngine
Dagger -->|"Power Mode"| FS
CiWorker -.->|"si CI_SBOM_ENABLED"| Syft
Syft -.->|"si CI_DTRACK_ENABLED"| DTrack
```

Démarrage de l'application

Le diagramme suivant détaille la séquence d'initialisation depuis `main()` jusqu'au démarrage des serveurs HTTP et SSH.

```
sequenceDiagram
participant Main as main()
participant Builder as RustwardenBuilder
participant App as RustwardenApp
participant Mig as AppMigrator
participant DB as PostgreSQL
participant SMTP as Serveur SMTP

Main->>Main: GitrustConfig::from_env()
Main->>Main: Arc (hooks FS)

Main->>Builder: builder().from_env()
Note over Builder: Charge .env, init tracing
Main->>Builder: .headless().auto_migrate(false)
Main->>Builder: .merge_routes(app_routes(hooks))
Note over Builder: Routes = framework API
+ pages SSR gitrust
+ Extension(hooks)
Main->>Builder: .build().await
Builder->>DB: Connexion pool
```


© Gitrust

```
    Builder->>DB: Migrations rustwarden-core
    Builder->>SMTP: EmailQueueProcessor.start()
    Note over SMTP: Worker de fond : dépile
        email_queue toutes les 30s
        Builder-->>Main: RustwardenApp

    Main->>Mig: run_migrations(app.database())
Mig->>DB: core_migrations() + gitrust_migrations()
    Note over DB: Tables: users, resources,
          repositories, ssh_keys,
           teams, team_members,
          team_repository_access
               Mig-->>Main: Ok

    Main->>Main: tokio::spawn(SSH server :2222)

              alt CI_ENABLED=true
       Main->>Main: CiWorker::start(config.ci)
       Note over Main: Vérifie dagger dans PATH
       Si CI_SBOM_ENABLED: vérifie syft
     Si CI_DTRACK_ENABLED: vérifie API DT
       Main->>Main: tokio::spawn(CiWorker loop)
         Note over Main: Écoute mpsc channel
              pour les jobs CI
                      end

         Main->>App: app.run().await
      Note over App: HTTP :4000 démarre
```

Principes de conception

Séparation framework / métier gitrust

Gitrust est construit **au-dessus** de rustwarden-core, un framework Rust d'authentification et de gestion d'utilisateurs. Cette séparation est intentionnelle :

- **rustwarden-core** gère : auth JWT, sessions, users, roles génériques, i18n, SMTP, OAuth, audit log générique.
- **gitrust-core** gère : tout ce qui est spécifique à la forge Git — dépôts, clés SSH, équipes, permissions RBAC à 4 niveaux, CI, import de dépôts.

La règle fondamentale : **ne jamais modifier** `crates/rustwarden-core/`. Toute extension passe par des wrappers, des hooks (`RustwardenHooks`) ou des traits implémentés côté gitrust.

© Gitrust

Mode headless

gitrust-web opère en mode `headless()` : les pages UI standard de rustwarden (login, register, settings) sont **désactivées** et réimplémentées entièrement dans `gitrust-web/templates/`. Cela permet un design cohérent (sidebar Git, DaisyUI) sans compromis avec l'UI générique du framework.

SSR + HTMX, zéro CDN

L'interface est rendue côté serveur via Askama (templates Rust compilés). HTMX gère les interactions dynamiques (mises à jour partielles de page, SSE pour les notifications et les logs CI). Aucun asset externe : tout CSS/JS est servi depuis `static/` — imposé par la Content Security Policy du framework.

Bare repos sur le système de fichiers

gitrust-git est **découplé de la base de données**. Les opérations Git (navigation d'arbre, lecture de blobs, pack protocol) travaillent directement sur les bare repos (`{GIT_REPOS_BASE_PATH}/{owner}/{slug}.git/`) via libgit2. La base ne stocke que les métadonnées (nom, description, visibilité, ownership). Ce découplage permet de remplacer ou tester gitrust-git indépendamment.

CI hybride : Dagger comme abstraction d'exécution

Plutôt que d'implémenter un runner CI from scratch, gitrust délègue l'exécution à **Dagger**, qui garantit l'isolation (containers), la reproductibilité (cache) et la portabilité (local ou runner distant via SSH+rsync). Le `CiWorker` est une tâche Tokio dans le processus principal — pas un daemon séparé — ce qui simplifie le déploiement (un seul binaire).

Pour aller plus loin

- Référence structurelle complète (tables de modules, routes, dépendances) : Architecture des crates
- Diagrammes de séquence détaillés (clone SSH, push, permissions) : Flux de requêtes
- Conception de la couche de durcissement SSH : Conception de ssh-guard et Crate gitrust-ssh-guard
- Règles de code et gates QA : Règles QA et conformité ANSSI

© Gitrust

1.5.2 Flux de requêtes

Diagrammes de séquence couvrant les opérations principales de gitrust : création de dépôt, clone et push SSH, enregistrement de clé SSH, résolution des permissions, navigation web, inscription utilisateur.

1. Création d'un dépôt (Web)

```
sequenceDiagram
    actor User as Utilisateur
    participant Web as gitrust-web
    participant RepoSvc as RepositoryService
    participant ResSvc as ResourceService
    participant DB as PostgreSQL
    participant Git as gitrust-git
    participant FS as Système de fichiers

    User->>Web: POST /new {name, description}
    Web->>Web: Extraire user_id (session JWT)

    Web->>RepoSvc: create(db, owner_id, username, base_path, input)
    RepoSvc->>RepoSvc: RepoSlug::from_name(input.name)
    RepoSvc->>DB: SELECT ... WHERE owner_id AND slug
    DB-->>RepoSvc: None (pas de doublon)

    RepoSvc->>RepoSvc: compute_disk_path(base, owner, slug)
    Note over RepoSvc: Validation anti path-traversal

    RepoSvc->>ResSvc: register(db, "repository", repo_id, owner_id)
    ResSvc->>DB: INSERT INTO resources
    DB-->>ResSvc: Ok

    RepoSvc->>DB: INSERT INTO repositories
    DB-->>RepoSvc: repository::Model

    RepoSvc-->>Web: RepositoryOutput

    Web->>Git: BareRepo::init(disk_path)
    Git->>FS: git init --bare
    FS-->>Git: Ok
    Git-->>Web: Ok
```


© Gitrust

```
    Web-->>User: 302 Redirect /{owner}/{slug}
```

2. Clone SSH (git clone)

```
sequenceDiagram
    actor Dev as Développeur
    participant Client as Git CLI
    participant SSH as gitrust-ssh
    participant Auth as SshKeyService
    participant Access as AccessService
    participant Git as gitrust-git
    participant DB as PostgreSQL
    participant FS as Bare repo (.git)

    Dev->>Client: git clone ssh://git@host:2222/alice/my-repo.git
    Client->>SSH: Connexion TCP :2222
    SSH-->>Client: Banner SSH + échange clés

    Client->>SSH: Auth publickey (signature)
    SSH->>Auth: find_by_fingerprint(db, fingerprint)
    Auth->>DB: SELECT FROM ssh_keys WHERE fingerprint = ?
    DB-->>Auth: ssh_key::Model {user_id, ...}
    Auth-->>SSH: user_id

    SSH->>Auth: update_last_used(db, key_id)
    Auth->>DB: UPDATE ssh_keys SET last_used_at = now()

    SSH-->>Client: Auth OK

    Client->>SSH: exec "git-upload-pack 'alice/my-repo.git'"
    SSH->>SSH: CommandHandler::parse("git-upload-pack alice/my-repo.git")
    Note over SSH: Extraction owner=alice, repo=my-repo

    SSH->>Access: effective_role(db, user_id, repo_id)
    Access->>DB: Check ownership + shares + teams
    DB-->>Access: Role::Developer
    Access-->>SSH: can_read = true

    SSH->>Git: pack_protocol::advertise_refs(repo_path)
    Git->>FS: Lecture refs
```


© Gitrust

```
    FS-->>Git: refs/heads/main, ...
    Git-->>SSH: advertise output

    SSH-->>Client: refs advertisement
    Client->>SSH: want/have negotiation

SSH->>Git: pack_protocol::serve_pack(repo_path, wants, haves)
    Git->>FS: Pack objects
    FS-->>Git: pack data
    Git-->>SSH: pack stream

    SSH-->>Client: pack data
Client-->>Dev: Dépôt cloné avec succès
```

3. Push SSH (git push)

```
sequenceDiagram
    actor Dev as Développeur
    participant Client as Git CLI
    participant SSH as gitrust-ssh
    participant Access as AccessService
    participant Git as gitrust-git
    participant Hooks as gitrust-hooks
    participant DB as PostgreSQL
    participant FS as Bare repo (.git)

    Note over Client,SSH: Session SSH déjà authentifiée

    Client->>SSH: exec "git-receive-pack 'alice/my-repo.git'"
    SSH->>SSH: CommandHandler::parse(...)
    Note over SSH: Extraction owner=alice, repo=my-repo

    SSH->>Access: effective_role(db, user_id, repo_id)
    Access->>DB: Check ownership + shares + teams
    DB-->>Access: Role::Developer
    Access-->>SSH: can_push = true

    SSH->>Git: pack_protocol::advertise_refs(repo_path)
    Git->>FS: Lecture refs
    Git-->>SSH: refs advertisement
```


© Gitrust

```
                SSH-->>Client: refs
            Client->>SSH: pack data + commands

        SSH->>Git: pack_protocol::serve_pack(repo_path, receive-pack)
            Git->>FS: Écriture objets + update refs
                FS-->>Git: Ok
                Git-->>SSH: Ok

    SSH->>DB: UPDATE repositories SET is_empty = false, updated_at = now()

                SSH-->>Client: Push OK
            Client-->>Dev: Push réussi
```

4. Enregistrement d'une clé SSH

```
                sequenceDiagram
                actor User as Utilisateur
            participant Web as gitrust-web
            participant Svc as SshKeyService
                participant DB as PostgreSQL

        User->>Web: POST /settings/keys {title, key_data}
                Web->>Web: Extraire user_id (session)

                Web->>Svc: create(db, user_id, input)

                Svc->>Svc: parse_public_key(key_data)
            Note over Svc: Extraire type + décoder base64

        Svc->>Svc: validate_key_type_and_size(type, bytes)
                    Note over Svc: ed25519 OK
                        RSA >= 4096 bits
                    ecdsa-p256/p384 OK

                Svc->>Svc: compute_fingerprint(bytes)
        Note over Svc: SHA256(key_bytes) -> base64-no-pad

    Svc->>DB: SELECT FROM ssh_keys WHERE fingerprint = ?
                DB-->>Svc: None (pas de doublon)

                Svc->>DB: INSERT INTO ssh_keys
```


© Gitrust

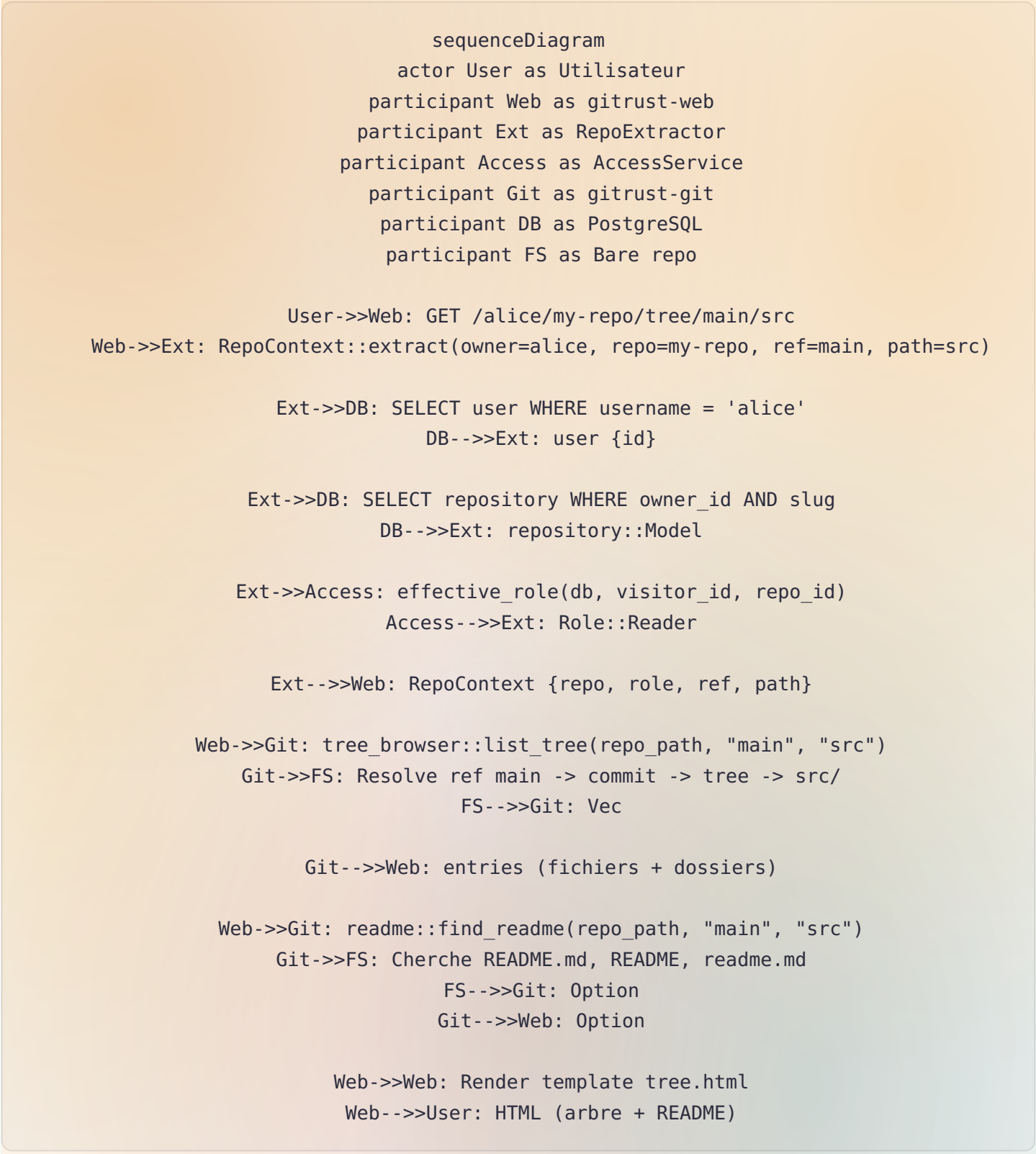
```
DB-->>Svc: ssh_key::Model

Svc-->>Web: SshKeyOutput
Web-->>User: 200 Clé ajoutée (fingerprint affiché)
```

5. Résolution des permissions (AccessService)

```
sequenceDiagram
participant Caller as Handler / SSH Session
participant AS as AccessService
participant RS as ResourceService
participant DB as PostgreSQL

Caller->>AS: effective_role(db, user_id, repo_id)

AS->>RS: effective_permission(db, user_id, "repository", repo_id)
RS->>DB: SELECT FROM resources WHERE type AND id
DB-->>RS: resource {owner_id, is_public}

alt user_id == owner_id
RS-->>AS: "owner"
else Share exists
RS->>DB: SELECT FROM resource_shares WHERE resource AND user
DB-->>RS: share {permission_level}
RS-->>AS: permission_level
else Resource is public
RS-->>AS: "read"
else No access
RS-->>AS: None
end

AS->>AS: Role::from_permission(individual_level)

AS->>DB: SELECT team_members + team_repository_access
WHERE user_id AND repository_id
DB-->>AS: Vec

AS->>AS: team_role = max(team_permissions)
AS->>AS: effective = max(individual_role, team_role)

AS-->>Caller: Role (Reader / Developer / Maintainer / Owner)
```

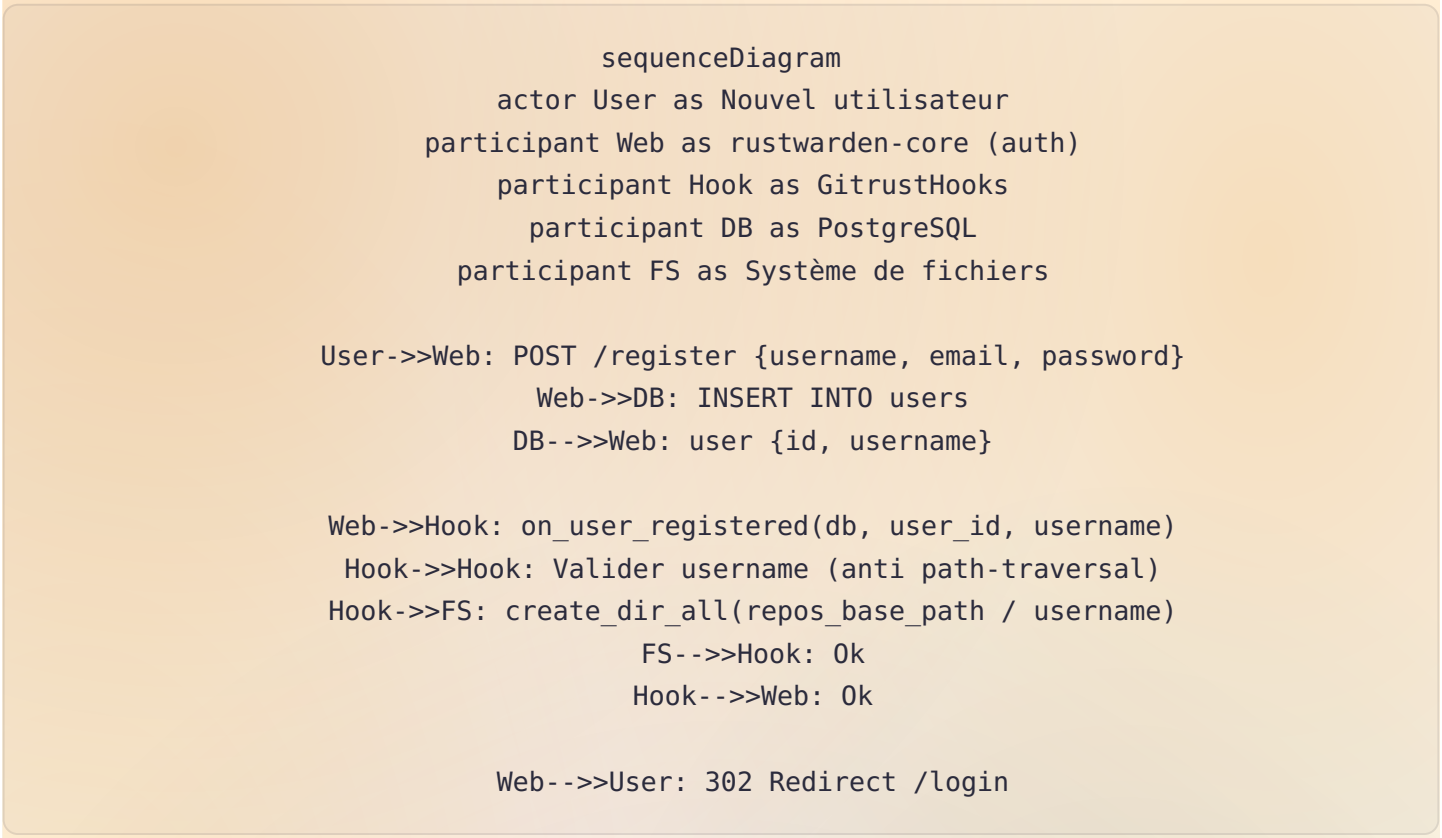

© Gitrust

6. Navigation dans un dépôt (Web)



© Gitrust

7. Inscription utilisateur + création répertoire



© Gitrust

1.5.3 Comprendre le modèle de permissions RBAC de gitrust

Ce que vous allez comprendre

- Analyser comment les 4 rôles (Reader, Developer, Maintainer, Owner) s'appliquent à un dépôt.
- Évaluer la différence entre accès direct et accès via équipe, et comment le rôle effectif est calculé.
- Comparer le modèle gitrust avec les ACL Unix (user/group/other).

Le problème concret

Vous avez un dépôt privé `alice/myproject`. Alice souhaite que : - Bob puisse lire le code (pas pusher). - L'équipe `backend-devs` puisse pusher. - Charlie, membre de `backend-devs`, soit quand même restreint en lecture seule.

Sans un modèle de permissions précis, impossible de savoir quel accès Charlie obtient réellement.

L'analogie

Le modèle gitrust ressemble aux **ACL Unix**, mais à deux dimensions.

Sur Unix, chaque fichier a trois niveaux : `user` (propriétaire), `group`, `other`. Le niveau effectif est le premier qui correspond, dans cet ordre.

Gitrust fonctionne de manière similaire, avec deux axes :

Axe Unix	Axe gitrust
`user`	accès individuel direct (`resource_shares`)
`group`	accès via équipe (`team_repository_access`)
`other`	accès public (`resources.is_public`)

La différence clé : gitrust prend le **maximum** entre l'accès individuel et l'accès équipe, pas le premier correspondant.

Le modèle

Les 4 rôles et leur hiérarchie

```
graph LR
R[Reader] --> D[Developer]
```


© Gitrust

```
D --> M[Maintainer]
M --> O[Owner]
```

Rôle	Permissions cumulatives
Reader	Clone, browse (lecture seule)
Developer	Reader + push sur branches non protégées
Maintainer	Developer + gestion dépôt (settings, collaborateurs, labels, protection de branches)
Owner	Maintainer + suppression, transfert

La hiérarchie est **strictement cumulative** : un Maintainer a toutes les permissions d'un Developer, qui a toutes celles d'un Reader.

Sources d'accès

Un utilisateur peut accéder à un dépôt par trois chemins indépendants :

```
flowchart TD
User[Utilisateur]

subgraph "Accès individuel"
Owner["Owner (resources.owner_id)"]
Share["resource_shares
(read / write / admin)"]
Public["Accès public
(resources.is_public = true)"]
end

subgraph "Accès équipe"
TeamMember["team_members
(l'utilisateur est membre)"]
TeamAccess["team_repository_access
(permission de l'équipe sur le dépôt)"]
end

Effective["Rôle effectif
= max(individuel, équipe)"]

User --> Owner
```


© Gitrust

```
                User --> Share
                User --> Public
              User --> TeamMember
            TeamMember --> TeamAccess

          Owner -->|"owner (full)"| Effective
        Share -->|"read/write/admin"| Effective
          Public -->|"read only"| Effective
      TeamAccess -->|"read/write/admin"| Effective
```

Calcul du rôle effectif

Le rôle effectif d'un utilisateur sur un dépôt est le **maximum** de tous ses accès :

```
rôle_effectif = max(
    owner_directement ? owner : none,
    resource_shares.permission_level,
    is_public ? read : none,
    max(team_repository_access.permission pour chaque équipe dont il est membre)
)
```

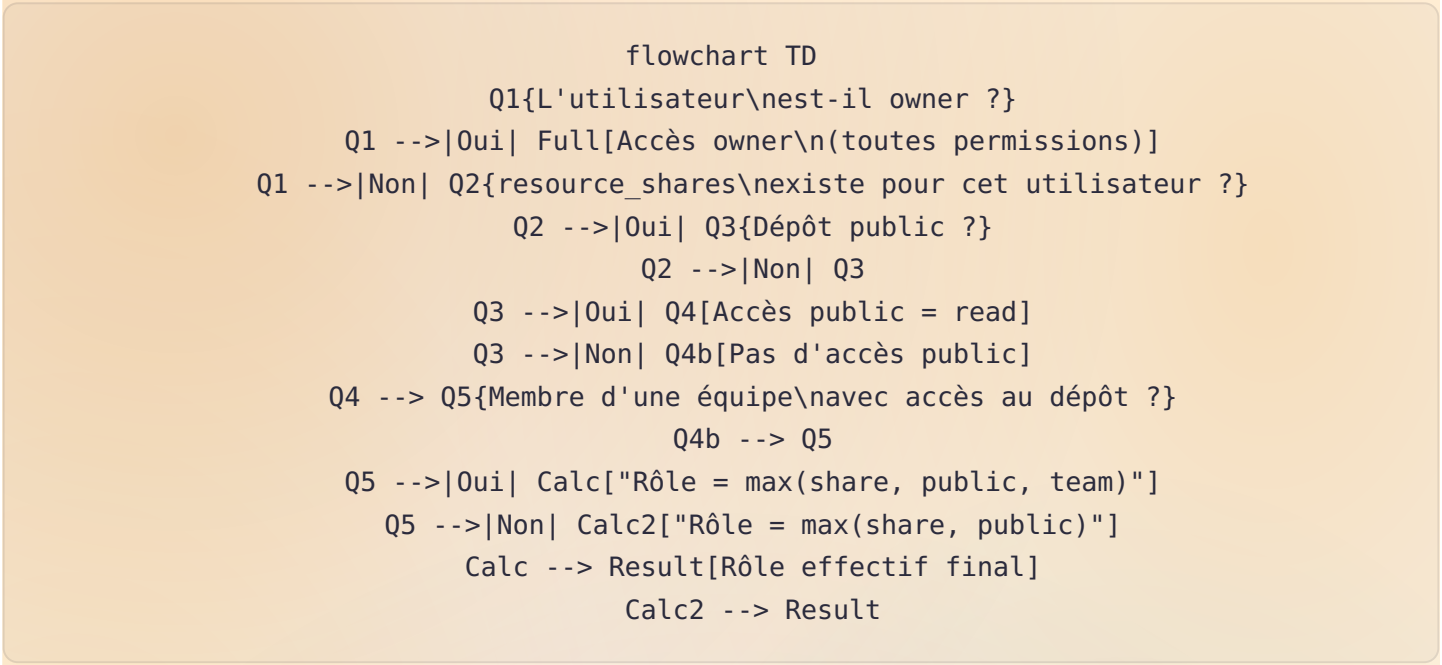
Exemple — réponse au problème concret :

Utilisateur	Accès individuel	Accès via équipe	Rôle effectif
Alice	owner	—	owner
Bob	read (partagé directement)	—	read
Charlie	read (partagé directement)	write (via backend-devs)	**write** (max)
Dave (membre backend-devs)	—	write	write

Charlie obtient `write` car le max(read, write) = write. Pour restreindre Charlie en lecture seule malgré son appartenance à `backend-devs`, il faudrait le retirer de l'équipe ou utiliser une protection de branche.

© Gitrust

Arbre de décision



Mapping vers les niveaux ResourceService

`ResourceService` (rustwarden-core) utilise les niveaux `"read"`, `"write"`, `"admin"`. Le mapping gitrust est :

Niveau ResourceService	Rôle gitrust
`"read"`	Reader
`"write"`	Developer
`"admin"`	Maintainer
`owner` (ownership directe)	Owner

Alternatives et compromis

Pourquoi pas un modèle RBAC pur avec des rôles nommés par dépôt ? Gitea/Forgejo utilisent des rôles nommés (`reporter`, `developer`, `maintainer`). Gitrust utilise des niveaux numériques (`read < write < admin`) héritables, ce qui simplifie le calcul du maximum mais rend impossible l'attribution de permissions non hiérarchiques (ex : « peut créer des issues mais pas pusher »).

Pourquoi pas des ACL par fichier ? La granularité fichier (comme dans Perforce) crée une complexité de maintenance considérable pour des équipes de 3-20 personnes. Les protections de branches couvrent 90 % des besoins de contrôle granulaire.

© Gitrust

Vérifier votre compréhension

1. Alice est owner de `myrepo`. Elle partage le dépôt avec Bob en `read`. L'équipe `ops` a accès en `admin`. Bob rejoint `ops`. Quel est le rôle effectif de Bob ?

2. Un visiteur anonyme accède à un dépôt dont `resources.is_public = true`. Peut-il pusher ? Pourquoi ?

Pour aller plus loin

- Référence ResourceService
- Schéma base de données — team_repository_access
- Gérer les équipes et permissions

© Gitrust

1.5.4 Comprendre le fonctionnement du 2FA TOTP dans gitrust

Ce que vous allez comprendre

- Décrire le flux cryptographique TOTP (RFC 6238) de la génération du secret à la vérification du code.
- Analyser les choix de sécurité : chiffrement du secret, codes de secours, challenges temporaires.
- Évaluer pourquoi le 2FA ne protège pas contre le vol de session et comment gitrust atténue ce risque.

Le problème concret

Alice utilise un mot de passe fort. Mais si sa base de mots de passe est compromise, ou si un attaquant intercepte son mot de passe (phishing), il peut se connecter à son compte gitrust et pousser du code malveillant. Un second facteur garantit que la possession du mot de passe seul ne suffit pas.

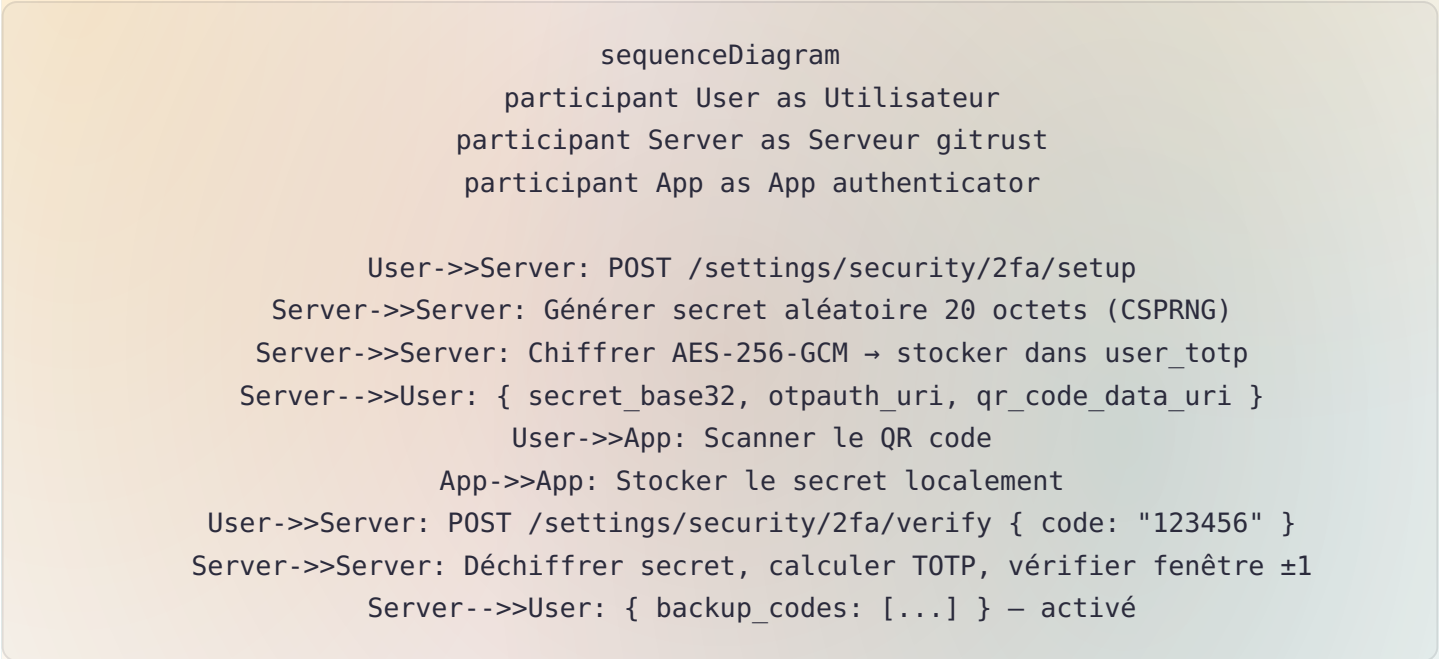
L'analogie

TOTP ressemble à un **générateur de codes de distributeur bancaire** qui change toutes les 30 secondes. La banque et le distributeur partagent un secret commun (la clé de votre carte). À partir de ce secret et de l'heure actuelle, les deux parties calculent indépendamment le même code. Personne n'a besoin de transmettre le code sur le réseau.

La différence avec un SMS OTP : le secret n'est jamais transmis après la configuration initiale. Un attaquant qui intercepte votre réseau ne voit jamais le code à venir.

Le modèle

Génération du secret (RFC 6238)



© Gitrust

L'URI OTPAuth encode tout ce qu'une app authenticator a besoin :

```
otpauth://totp/gitrust:alice?secret=JBSWY3DPEHPK3PXP&issuer=gitrust&algorithm=SHA1&digi
```

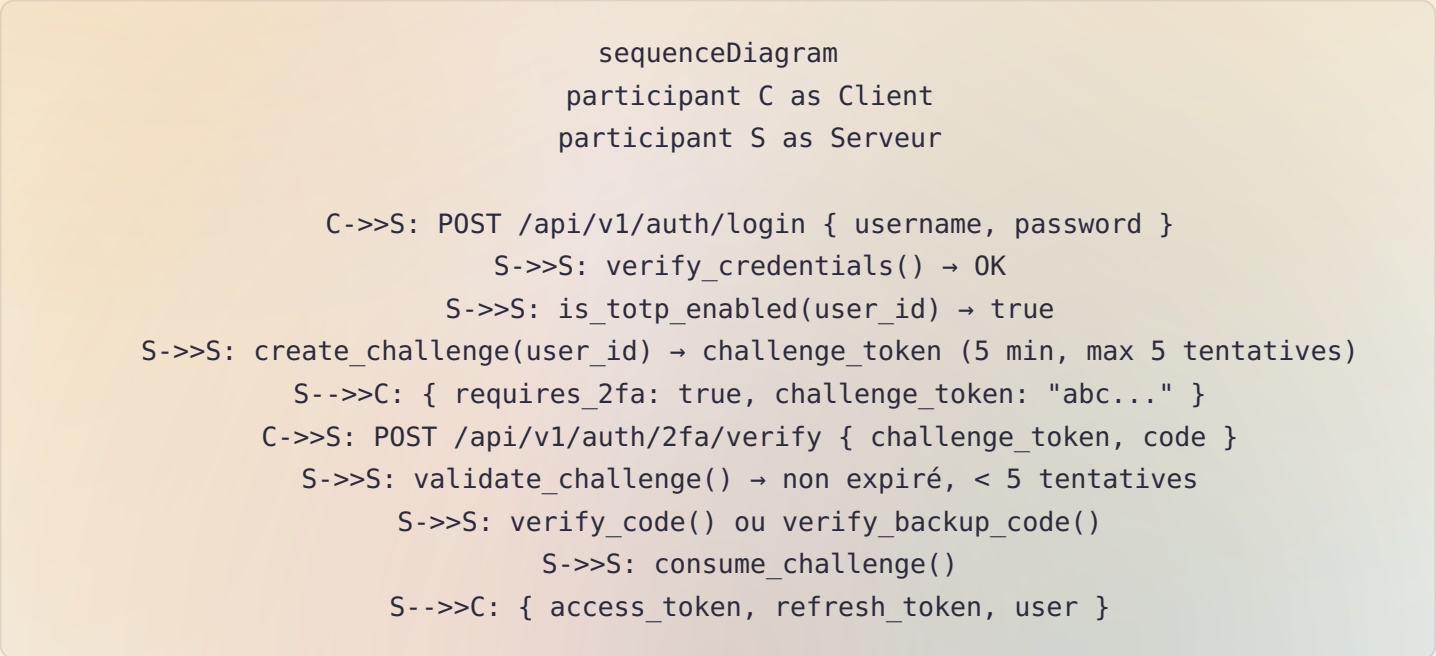
Calcul du code TOTP

TOTP est HOTP (RFC 4226) avec un compteur basé sur le temps :

```
counter = floor(unix_timestamp / 30)
hmac    = HMAC-SHA1(secret, counter_as_8_bytes_big_endian)
offset  = hmac[19] & 0x0f
code    = (hmac[offset..offset+4] & 0x7fffffff) % 1_000_000
```

Le code est valide pendant 30 secondes. Gitrust accepte une **fenêtre de tolérance ±1 période** (90 secondes totales) pour compenser les légères dérives d'horloge entre l'appareil de l'utilisateur et le serveur.

Flux de login avec 2FA activé



Le challenge expire après **5 minutes**. Maximum **5 tentatives** par challenge. Un challenge échoué 5 fois force une nouvelle authentification complète.

Stockage sécurisé du secret

Le secret TOTP **n'est jamais stocké en clair** en base de données. Il est chiffré avec AES-256-GCM via `CryptoService` avant insertion dans `user_totp.encrypted_secret`.

© Gitrust

```
// À l'activation
let encrypted = CryptoService::encrypt(&secret_base32)?;
// → stocké : "enc:base64(nonce||ciphertext)"

// À la vérification
let secret = CryptoService::decrypt(&user_totp.encrypted_secret)?;
```

La clé de chiffrement est `ENCRYPTION_KEY` (32 octets, variable d'environnement). Si elle est absente, le TOTP ne peut pas être activé.

Codes de secours

À l'activation du 2FA, 10 codes de secours de 8 caractères alphanumériques sont générés. Ils sont stockés **hachés bcrypt** dans `user_totp.backup_codes_json` (JSON chiffré).

Chaque code est **à usage unique** : une fois utilisé, il est marqué consommé. L'utilisateur peut régénérer un nouveau jeu de 10 codes depuis `/settings/security`.

Format : `XXXXXXXX` (ex : `A3B7K2M9`) — reconnaissable par sa longueur (8 chars vs 6 chiffres pour TOTP).

La détection est automatique dans `verify_backup_code` : si le code contient des lettres ou fait 8 caractères, c'est un code de secours ; sinon c'est un code TOTP.

Force 2FA par l'administrateur

Un administrateur peut forcer le 2FA sur toute l'instance via `AppSettingsService::set_setting("force_2fa_enabled", "true")`. Quand ce flag est actif :

- Les utilisateurs sans 2FA sont redirigés vers la page de configuration à chaque login.
- Les endpoints API retournent `403` avec `{"error":"totp_required"}` tant que le 2FA n'est pas configuré.

Alternatives et compromis

Pourquoi TOTP et pas WebAuthn/passkeys ? WebAuthn offre une protection supérieure contre le phishing (le domaine est lié cryptographiquement). Gitrust prévoit de l'ajouter en phase future. TOTP a été choisi pour sa compatibilité universelle (toutes les apps authenticator, hors connexion, pas de dépendance matérielle).

Pourquoi pas SMS OTP ? Les SMS OTP sont vulnérables au SIM swapping et à l'interception SS7. Gitrust ne les implémente pas.

La fenêtre de ±1 période réduit-elle la sécurité ? Marginalement. Elle augmente la fenêtre d'attaque de 30 à 90 secondes, mais un code TOTP reste à 6 chiffres (1/1 000 000 de probabilité de deviner), et les 5 tentatives max par challenge rendent le brute-force infaisable.

© Gitrust

Vérifier votre compréhension

1. Un utilisateur est dans un fuseau horaire décalé de 45 secondes par rapport au serveur. Sa tentative de connexion TOTP échoue systématiquement. Pourquoi, et comment diagnostiquer ?

2. L'`ENCRYPTION_KEY` du serveur change (rotation). Que se passe-t-il pour les utilisateurs ayant déjà activé le 2FA ? Quelle procédure de migration faudrait-il mettre en place ?

Pour aller plus loin

- TotpService — référence API
- Forcer le 2FA globalement
- RFC 6238 — TOTP

© Gitrust

1.5.5 Comprendre la hiérarchie des labels à deux niveaux

Ce que vous allez comprendre

- Analyser pourquoi gitrust utilise deux types de labels indépendants plutôt qu'un seul système plat.
- Évaluer l'impact du filtrage combinatoire sur la recherche d'issues.
- Décrire comment l'UI différencie visuellement les deux types.

Le problème concret

Sur une forge Git classique, les labels forment une liste plate par dépôt. Quand vous avez 50 dépôts avec chacun leurs propres labels `bug`, `feature`, `documentation`, vous ne pouvez pas filtrer « toutes les issues de type `bug` dans tous mes dépôts » sans dupliquer les labels à la main dans chaque dépôt.

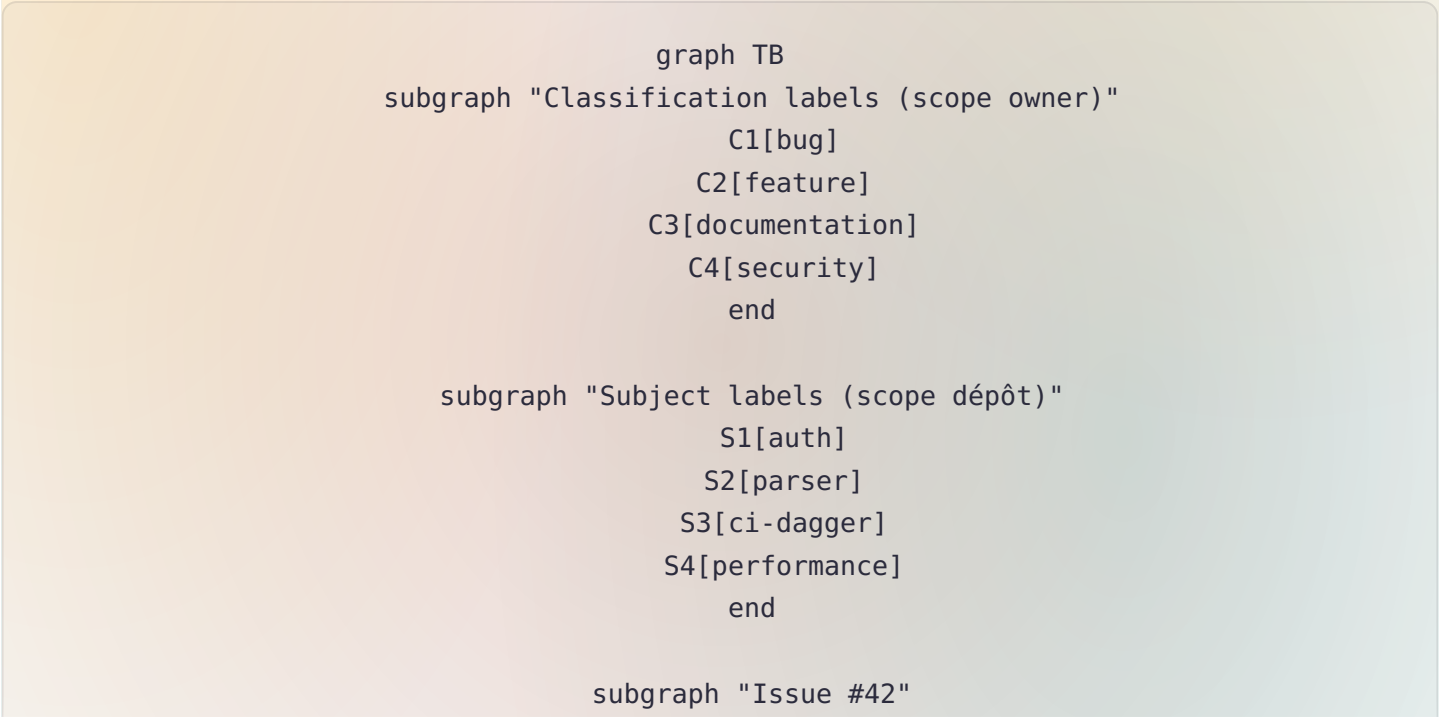
De plus, deux équipes qui utilisent le même dépôt veulent des tags libres pour annoter leurs domaines (`auth`, `ci`, `parser`) sans que ces tags exigent une validation préalable par un mainteneur.

L'analogie

Pensez aux **tags Flickr à deux niveaux** : un album appartient à une catégorie fixe définie par l'organisateur (`Voyage`, `Famille`, `Travail`), et peut recevoir des tags libres ajoutés par quiconque (`paris`, `2026`, `coucher-de-soleil`). Les deux systèmes coexistent sans se mélanger. La catégorie donne la structure ; les tags libres donnent la granularité.

Le modèle

Deux types, deux scopes



© Gitrust

```
        I1["bug (classification)"]
        I2["auth (subject)"]
        end

        C1 --> I1
        S1 --> I2
```

Caractéristique	Classification	Subject
Scope	Owner (tous ses dépôts)	Dépôt unique
Qui crée	Owner, Maintainer uniquement	Tout utilisateur connecté
Création inline	Non (CRUD dédié `/labels`)	Oui (champ autocomplete dans l'issue)
Style visuel	Badge plein (fond coloré)	Badge outline (contour coloré)
Couleur	Définie par l'auteur	Défaut `#6b7280`, personnalisable
Colonne DB	`owner_id` non-null, `repository_id` null	`repository_id` non-null, `owner_id` null

Filtrage combinatoire

L'intérêt principal des classification labels est le filtrage cross-dépôt. Sur le tableau de bord d'Alice (`/alice`) :

```
Filtrer par classification: bug
→ Affiche toutes les issues "bug" dans tous les dépôts d'Alice
  (alice/myrepo, alice/api-client, alice/cli-tool)
```

Un subject label seul ne permet pas ce filtrage car il est scopé à un seul dépôt.

La combinaison des deux offre un filtrage à deux dimensions :

```
bug (classification) × auth (subject)
→ Toutes les issues bug liées à l'authentification
```

Assignation sur une issue

La page de détail d'une issue présente deux blocs distincts :

© Gitrust

Bloc classification (géré par Owner/Maintainer uniquement) : - Checkboxes pour les labels de classification de l'owner - Bouton « Save labels » - Envoyé dans `label_ids[]` au POST `/{o}/{r}/issues/{num}/labels`

Bloc subject tags (géré par l'auteur ou Owner/Maintainer) : - Champ texte avec autocomplete HTMX (déclenché après 3 caractères, délai 300ms) - Tab sur une suggestion → sélectionne le label existant - Tab sur du texte libre → crée le label subject à la soumission (`find_or_create_subject`) - Badges removable (×) - Envoyé dans `subject_names` (comma-separated) au même endpoint

Schéma DB

```
-- Un seul champ label_type discrimine les deux cas
labels (
    id           UUID PK,
    owner_id     UUID NULL,      -- non-null pour classification
    repository_id UUID NULL,     -- non-null pour subject
    name         VARCHAR(50),
    color        VARCHAR(7),
    description  VARCHAR(255) NULL,
    label_type   VARCHAR(20),    -- 'classification' ou 'subject'
    ...
)

-- Index d'unicité composite
UNIQUE(owner_id, repository_id, name, label_type)
```

La contrainte d'unicité garantit qu'un owner ne peut pas avoir deux labels classification `bug` , et qu'un dépôt ne peut pas avoir deux subject tags `auth` .

Alternatives et compromis

Pourquoi pas un seul système de tags libres ? Les tags libres offrent une flexibilité maximale mais aucune standardisation cross-dépôt. Deux développeurs utilisant `bug` et `Bug` créent deux labels distincts.

Pourquoi pas des labels en arbre (hiérarchiques) ? Un arbre de labels (comme les tags JIRA avec sous-catégories) serait plus expressif mais beaucoup plus complexe à afficher dans des badges inline. La combinaison classification × subject couvre 95 % des besoins à coût d'implémentation minimal.

Compromis choisi : la classification donne la rigueur (contrôlée par les mainteneurs, partagée cross-dépôt) ; le subject donne la liberté (créé par n'importe qui, spécifique au dépôt). Chacun fait ce que l'autre ne fait pas.

Vérifier votre compréhension

1. Un développeur crée un subject tag `bug` dans son dépôt. L'owner a aussi un classification label `bug` . Comment l'UI les distingue-t-elle visuellement ? Peuvent-ils coexister sur la même issue ?

© Gitrust

2. Alice crée un classification label `security` sur son compte. Elle a 15 dépôts. Sans aucune action supplémentaire, dans combien de dépôts ce label est-il disponible ?

Pour aller plus loin

- LabelService — référence API
- Schéma DB — table labels
- Utiliser les labels et issues

© Gitrust

1.5.6 Comprendre la CI Dagger : Easy Mode vs Power Mode

Ce que vous allez comprendre

- Analyser pourquoi Dagger a été choisi comme moteur CI plutôt qu'une CI YAML classique.
- Comparer Easy Mode (`.gitrust-ci.yml`) et Power Mode (`.dagger/`) et évaluer quand basculer de l'un à l'autre.
- Décrire les garanties d'isolation et de reproductibilité offertes par Dagger.

Le problème concret

Une CI traditionnelle (GitHub Actions, GitLab CI) exécute des étapes dans un environnement partagé. Deux problèmes fréquents :

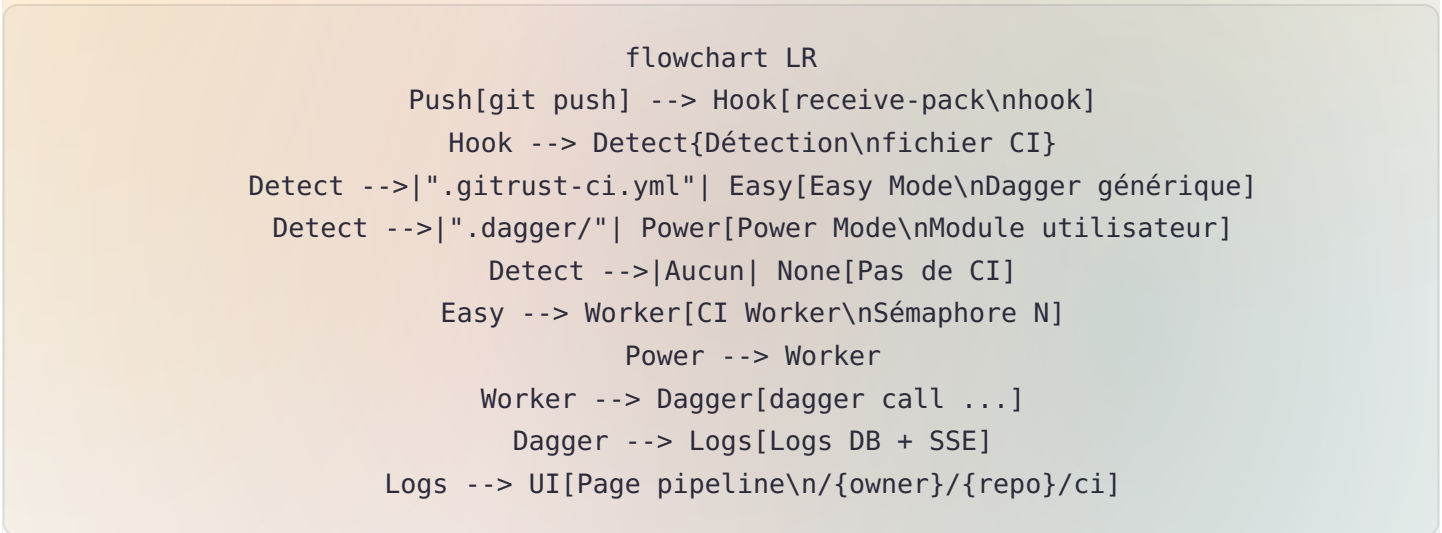
1. **Manque de reproductibilité** : « ça passe sur ma machine mais pas en CI » parce que les versions d'outils diffèrent.
2. **Couplage fort** : un pipeline GitLab CI est indissociable de GitLab. Migrer vers une autre plateforme = réécrire tous les pipelines.

L'analogie

Dagger, c'est comme **Docker pour les pipelines CI** : chaque étape s'exécute dans un container isolé, avec des dépendances déclarées et des caches portables. Le pipeline peut tourner identiquement sur votre machine, sur le serveur gitrust, ou sur n'importe quel autre runner qui a Docker installé.

L'analogie avec Docker/Kubernetes est directe : - Docker résout « ça marche sur ma machine » pour les *applications*. - Dagger résout « ça marche sur ma machine » pour les *pipelines*.

Le modèle



© Gitrust

La détection est prioritaire : si `.dagger/` et `.gitrust-ci.yml` coexistent, le Power Mode prend la priorité.

Easy Mode — `.gitrust-ci.yml`

Un fichier YAML simple décrit le pipeline. Le moteur Dagger générique de la plateforme l'interprète.

```
# .gitrust-ci.yml — exemple pour un projet Rust
build:
  command: cargo build --release
  cache:
    - path: ~/.cargo/registry
      key: cargo-registry

test:
  command: cargo test --workspace
  requires_build: true

checks:
  - name: fmt
    command: cargo fmt --all -- --check
  - name: clippy
    command: cargo clippy --workspace -- -D warnings
```

Avantages : - Accessible à tout utilisateur sans connaissance de Dagger. - Standardisé : tous les projets Easy Mode ont la même structure. - Zéro configuration côté plateforme.

Limites : - Pas de logique conditionnelle complexe. - Pas d'accès au Daggerverse (modules Dagger communautaires). - Pas de tests unitaires du pipeline lui-même. - Pas de composition (appeler un module depuis un autre).

Power Mode — `.dagger/`

Le dépôt contient un vrai module Dagger (Go, Python, ou TypeScript) :

```
.dagger/
├── dagger.json          # manifest du module
└── main.go              # fonctions Dagger
```

```
// .dagger/main.go — exemple Go
package main

import (
    "dagger/gitrust-ci/internal/dagger"
)
```


© Gitrust

```
type GitrustCi struct{}

func (m *GitrustCi) Test(ctx context.Context, source *dagger.Directory) (string, error)
    return dag.Container().
        From("rust:1.77-slim").
        WithMountedDirectory("/src", source).
        WithWorkdir("/src").
        WithExec([]string{"cargo", "test", "--workspace"}).
        Stdout(ctx)
}
```

Gitrust exécute : `dagger call -m .dagger/ test --source=.`

Avantages : - Logique conditionnelle arbitraire (Go/Python/TS complet). - Accès au Daggerverse : réutiliser des modules existants (Helm, SBOM, Trivy...). - Tests unitaires du pipeline possible (fonctions Dagger sont du vrai code). - Composition : un module peut appeler un autre module. - Isolation totale garantie par les containers Dagger. - Type-safety : le compilateur Go/TS vérifie les appels.

Limites : - Courbe d'apprentissage : il faut écrire du code Go/Python/TS. - Chaque équipe maintient son propre module.

Quand basculer de Easy Mode vers Power Mode ?

```
flowchart TD
    Q1{Pipeline simple ?\nbuild + test + lint}
    Q1 -->|Oui| Easy[Rester en Easy Mode]
    Q1 -->|Non| Q2{Logique conditionnelle ?\nex: déployer seulement sur main}
    Q2 -->|Oui| Power[Passer en Power Mode]
    Q2 -->|Non| Q3{Composition avec\nd'autres modules Dagger ?}
    Q3 -->|Oui| Power
    Q3 -->|Non| Q4{Tests du pipeline\nlui-même ?}
    Q4 -->|Oui| Power
    Q4 -->|Non| Easy
```

Règle pratique : si votre `.gitrust-ci.yml` commence à ressembler à un programme avec des `if/else` imbriqués, c'est le moment de passer en Power Mode.

© Gitrust

Tableau comparatif

Critère	Easy Mode (YAML)	Power Mode (Dagger natif)
Accessibilité	Excellent	Courbe d'apprentissage
Puissance	Limitée	Illimitée
Testabilité du pipeline	Nulle	Totale
Composition (Daggerverse)	Non	Oui
Portabilité	Via le moteur gitrust	Native (tout runner Dagger)
Alignement philosophie Dagger	Partiel (YAML = anti-pattern)	Total
Maintenance plateforme	Le moteur évolue	Chaque repo est autonome

Position philosophique de Dagger

Dagger a été créé explicitement **contre** le modèle YAML des CI traditionnelles par l'équipe fondatrice de Docker. Le YAML Easy Mode de gitrust est un **compromis assumé** : on sacrifie la pureté philosophique pour l'accessibilité des 90 % de cas simples. Le YAML Easy Mode ne doit jamais essayer de tout couvrir — dès que la complexité augmente, le Power Mode est la bonne réponse.

Alternatives et compromis

Pourquoi pas GitHub Actions ou GitLab CI directement ? Ces systèmes sont couplés à leur plateforme. Un pipeline GitHub Actions ne tourne pas localement sans outil tiers (act). Dagger tourne identiquement en local, sur le runner gitrust, et sur GitHub Actions si nécessaire.

Pourquoi pas Jenkins/Tekton ? L'objectif de gitrust est l'auto-hébergement simple pour des équipes de 3-20 personnes. Ces outils ajoutent une complexité d'administration disproportionnée.

Vérifier votre compréhension

1. Un développeur a un `.gitrust-ci.yml` avec 15 étapes et des `skip_if` conditionnels simulés via des scripts shell. Doit-il rester en Easy Mode ? Justifiez.

2. Quelle propriété de Dagger garantit qu'un pipeline Power Mode qui passe en local passera aussi sur le runner gitrust ? Comment cette propriété est-elle implémentée techniquement ?

Pour aller plus loin

- Lancer les tests unitaires
- CI Easy Mode — schéma YAML complet

© Gitrust

- Documentation officielle Dagger

© Gitrust

1.5.7 Comprendre les décisions UI : SSR, HTMX et DaisyUI

Ce que vous allez comprendre

- Analyser pourquoi gitrust utilise le rendu côté serveur (SSR) plutôt qu'un framework JavaScript front-end.
- Évaluer comment HTMX ajoute de l'interactivité sans introduire de build front-end.
- Décrire les contraintes qui ont orienté le choix de DaisyUI comme bibliothèque de composants.

Le problème concret

Une forge Git moderne doit proposer des interfaces réactives : autocomplétion, chargement partiel, mises à jour en temps réel (logs CI, progression d'import). Les approches habituelles imposent soit un SPA React/Vue avec son pipeline de build, soit un back-end qui sert une API JSON distincte de ses vues.

Pour une équipe de 3 personnes maintenant un outil en Rust, les coûts de ces approches sont disproportionnés : - Un SPA impose une deuxième codebase (TypeScript), deux langages de test, deux pipelines CI. - Un back-end JSON-only perd les avantages du rendu serveur (SEO, accessibilité, temps de chargement initial).

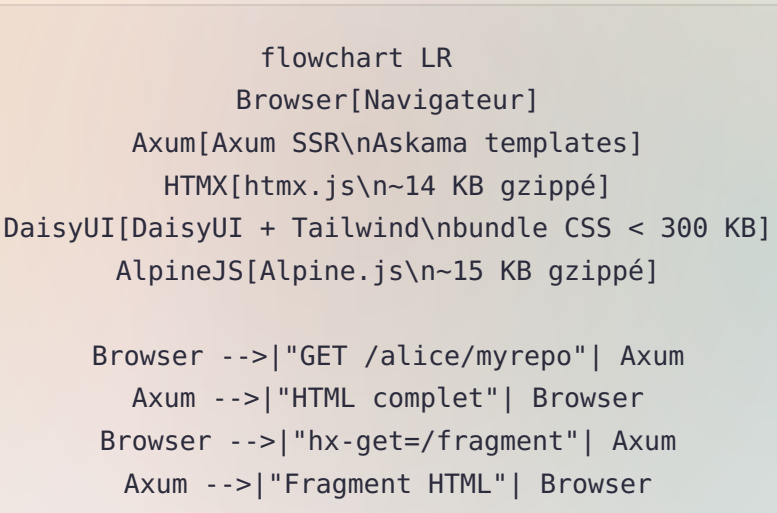
L'analogie

SSR + HTMX fonctionne comme un **serveur HTML des années 2000 avec superpouvoirs** : le serveur produit du HTML complet que le navigateur affiche directement (SSR classique), mais HTMX permet au navigateur de remplacer *des fragments* de la page sans recharger le tout. C'est la différence entre remplacer une page entière de livre et coller un post-it sur un paragraphe.

DaisyUI est la couche de présentation : elle apporte des composants CSS cohérents (boutons, badges, modales) sans JavaScript, uniquement via des classes Tailwind.

Le modèle

Architecture UI de gitrust



© Gitrust

```
Browser --- HTMX
Browser --- DaisyUI
Browser --- AlpineJS
```

Aucun CDN externe. Tous les assets sont servis localement depuis `static/`. Le navigateur ne fait jamais de requête vers un domaine tiers.

Rendu côté serveur — Askama

Gitrust utilise **Askama**, un moteur de templates Rust compilé. Les templates sont vérifiés au moment de la compilation : une clé manquante ou un type incompatible est une erreur de build, pas une erreur à l'exécution.

```
// crates/gitrust-web/src/templates.rs
#[derive(Template)]
#[template(path = "repository/show.html")]
pub struct RepositoryShowTemplate {
    pub repo: RepositoryView,
    pub branches: Vec<BranchSummary>,
    pub current_user: Option<UserView>,
    pub csrf_token: String,
}
```

Le handler construit la struct, passe-la au template, et Askama sérialise le HTML :

```
async fn repo_show(/* ... */) -> impl IntoResponse {
    let tmpl = RepositoryShowTemplate {
        repo: repo_view,
        branches,
        current_user: user.map(|u| u.into()),
        csrf_token: generate_csrf(&session),
    };
    Html(tmpl.render().unwrap())
}
```

Avantage clé : le compilateur Rust garantit que chaque champ exposé au template existe et a le bon type. Impossible d'afficher un champ `None` non géré.

HTMX — interactivité sans JavaScript custom

HTMX étend HTML avec des attributs `hx-*`. Le navigateur envoie des requêtes HTTP et remplace des fragments DOM avec la réponse.

© Gitrust

Autocomplétion des subject labels

```
<!-- templates/issues/partials/subject_tags.html -->
<input
  type="text"
  name="tag_query"
  hx-get="/{{ owner }}/{{ repo }}/labels/search"
  hx-trigger="input changed delay:300ms, keyup[key=='Tab']"
  hx-target="#tag-suggestions"
  hx-swap="innerHTML"
  placeholder="Ajouter un tag..."
/>
<ul id="tag-suggestions"></ul>
```

Le handler correspondant retourne un fragment HTML, pas du JSON :

```
async fn search_subject_labels(
    Path((owner, repo)): Path<(String, String)>,
    Query(params): Query<HashMap<String, String>>,
    State(db): State<DatabaseConnection>,
) -> impl IntoResponse {
    let query = params.get("tag_query").map(|s| s.as_str()).unwrap_or("");
    let labels = LabelService::search_subject_labels(&db, &repo_id, query).await?;
    let tmpl = LabelSuggestionsTemplate { labels };
    Html(tmpl.render().unwrap())
}
```

Logs CI en temps réel (SSE)

Pour les logs de pipeline qui s'affichent en continu, gitrust utilise les **Server-Sent Events** exposés par Axum, consommés via `hx-ext="sse"` :

```
<div
  hx-ext="sse"
  sse-connect="/{{ owner }}/{{ repo }}/ci/{{ pipeline_id }}/stream"
  sse-swap="message"
  hx-swap="beforeend"
  id="log-output"
>
</div>
```


© Gitrust

Le serveur envoie des événements `data: <ligne HTML>\n\n`. HTMX les append dans `#log-output` sans aucun JavaScript custom.

DaisyUI — composants CSS, zéro JavaScript

DaisyUI est une bibliothèque de composants construite sur Tailwind CSS. Ses composants (bouton, badge, dropdown, modal) s'activent uniquement via des classes CSS et des attributs HTML natifs (`<details>`, `<input type="checkbox">`).

```
<!-- Badge classification (fond plein) -->
<span class="badge badge-primary">{{ label.name }}</span>

<!-- Badge subject (contour) -->
<span class="badge badge-outline" style="border-color: {{ label.color }}">
  {{ label.name }}
</span>

<!-- Dropdown natif DaisyUI -->
<details class="dropdown">
  <summary class="btn btn-sm">Assignees</summary>
  <ul class="dropdown-content menu bg-base-100 rounded-box z-[1] w-52 p-2 shadow">
    {% for collaborator in collaborators %}
    <li><a hx-post="..." hx-vals='{"user_id": "{{ collaborator.id }}"}'>
      {{ collaborator.username }}
    </a></li>
    {% endfor %}
  </ul>
</details>
```

Aucun composant DaisyUI ne nécessite d'initialisation JavaScript. Les modales utilisent `<input type="checkbox" id="modal-toggle">` avec CSS `:checked` pour afficher/masquer.

Alpine.js — état local léger

Pour les rares cas nécessitant de l'état côté client (toggle d'un panneau, validation inline), gitrust utilise **Alpine.js** (~15 KB) plutôt que React ou Vue.

```
<!-- Toggle d'un panneau de settings -->
<div x-data="{ open: false }">
  <button @click="open = !open">Paramètres avancés</button>
  <div x-show="open" x-transition>
    <!-- contenu du panneau -->
```


© Gitrust

```
  </div>
</div>
```

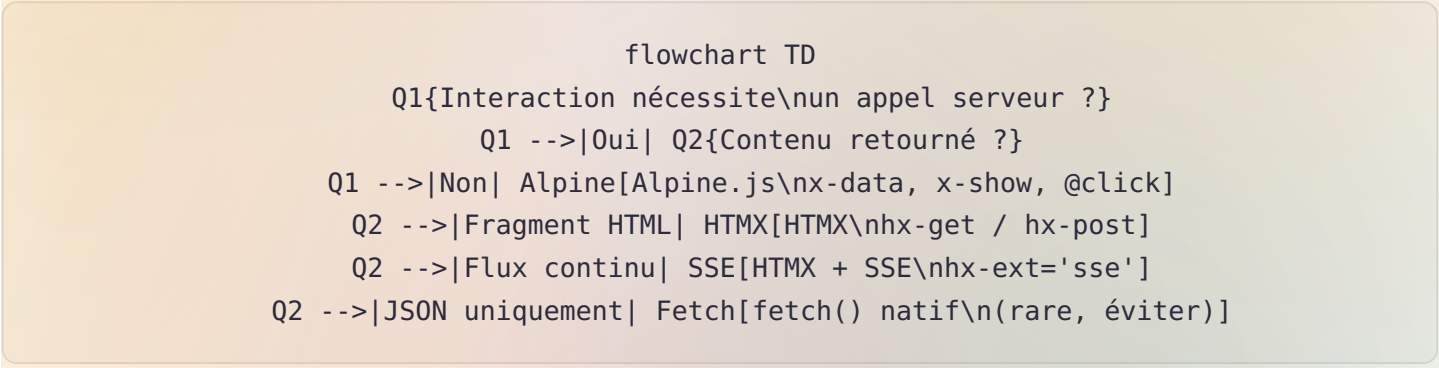
Alpine.js est réservé aux comportements purement visuels sans appel réseau. Dès qu'un appel serveur est nécessaire, c'est HTMX qui prend le relais.

Budget bundle et contraintes

Asset	Taille gzippée
htmx.min.js	~14 KB
alpine.min.js	~15 KB
gitrust.css (DaisyUI + Tailwind purged)	~280 KB
Total	**< 310 KB**

Le CSS est généré lors du build via `npx tailwindcss --input ... --output ... --minify`. Le résultat est commité dans `static/css/gitrust.css` pour éviter une dépendance Node.js en production. La reconstruction est déclenchée uniquement quand les templates changent (gate QA : `make css-rebuild`).

Quand utiliser chaque outil



Alternatives et compromis

Pourquoi pas React/Vue/Svelte ? Ces frameworks nécessitent un pipeline de build (Vite, Webpack), une API JSON séparée, et des compétences TypeScript en plus de Rust. Pour une forge auto-hébergée ciblant 3-20 développeurs, ce coût n'est pas justifié.

Pourquoi pas Inertia.js (SSR + SPA hybride) ? Inertia demande un adaptateur côté serveur. Il n'existe pas d'adaptateur Rust/Axum stable. De plus, Inertia reste JavaScript-first.

Pourquoi pas Turbo (Hotwire) ? Turbo est excellent mais son modèle Frame/Stream est plus complexe que les attributs HTMX pour des équipes qui démarrent. HTMX est plus proche de HTML pur.

© Gitrust

Compromis assumé : HTMX ne peut pas gérer des interactions très complexes (éditeur de code en ligne, diagrammes interactifs). Si gitrust devait ajouter un éditeur Monaco, un composant React isolé serait envisageable — mais ce n'est pas une décision prise aujourd'hui.

Vérifier votre compréhension

1. Un développeur propose d'ajouter une feature de « preview Markdown en temps réel » dans l'éditeur de description d'issue. Il suggère d'installer React juste pour ce composant. Quelles sont les alternatives possibles avec la stack existante ? Évaluez leurs compromis.

2. La page `/alice/myrepo/issues` charge 100 issues et est lente. Un développeur propose de migrer vers une SPA React avec pagination côté client. Quelle solution HTMX permettrait d'obtenir le même résultat sans changer de stack ?

Pour aller plus loin

- Ajouter une route web
- Patron worker async
- Documentation HTMX
- DaisyUI composants

© Gitrust

1.5.8 Comprendre le patron worker async de gitrust

Ce que vous allez comprendre

- Analyser pourquoi les opérations longues (import, export, envoi de mails) ne peuvent pas s'exécuter dans le cycle requête/réponse HTTP.
- Décrire comment le patron worker async de gitrust combine `mpsc`, `Semaphore`, et SSE pour traiter des tâches longues avec retour de progression.
- Évaluer les garanties de durabilité offertes par le modèle « état en base + canal en mémoire ».

Le problème concret

L'import d'un dépôt GitHub peut prendre 30 secondes. Une requête HTTP a un timeout. Si le serveur redémarre pendant l'import, la tâche est perdue. Si 50 utilisateurs lancent un import simultanément sans limite, le serveur sature.

Le même problème se pose pour l'envoi de mails, la génération de SBOM, ou toute opération dont la durée dépasse quelques secondes.

L'analogie

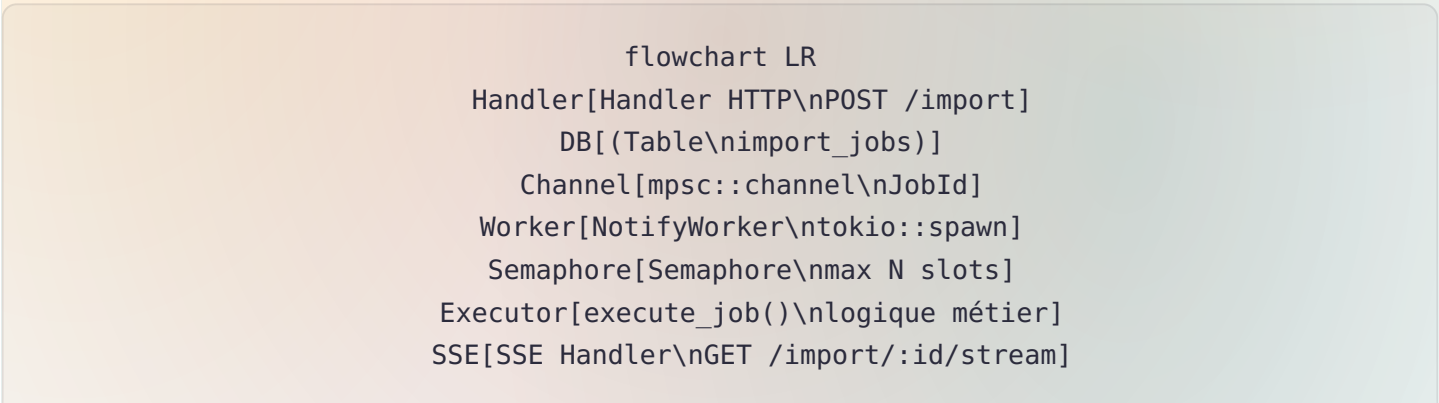
Le patron ressemble à une **cuisine de restaurant avec un tableau de commandes** :

- Le **serveur** (handler HTTP) prend la commande, l'inscrit sur le tableau (DB), et dit au client « votre numéro est le 42, suivez l'avancement ici ».
- La **cuisine** (worker) lit le tableau en continu, choisit la prochaine commande, la prépare derrière un comptoir limité à N plans de travail (Semaphore).
- Le **tableau lumineux** (SSE) indique l'état en temps réel au client sans qu'il ait besoin de redemander.

Si la cuisine ferme brutalement (crash), les commandes « en cours » sur le tableau sont remises en attente au redémarrage — elles ne disparaissent pas.

Le modèle

Vue d'ensemble du patron



© Gitrust

```
    Client[Navigateur\nhx-ext='sse']

    Handler -->|"INSERT status=Pending"| DB
    Handler -->|"tx.send(job_id)"| Channel
    Channel -->|"rx.recv()"| Worker
    Worker -->|"acquire()"| Semaphore
    Semaphore -->|"permit"| Executor
    Executor -->|"UPDATE status=..."| DB
    DB -->|"SELECT"| SSE
    SSE -->|"data: "| Client
```

La table de persistance

Chaque type de job a sa propre table. Le schéma canonique pour un import :

```
CREATE TABLE import_jobs (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id         UUID NOT NULL REFERENCES users(id),
    repository_id   UUID REFERENCES repositories(id),
    status          VARCHAR(20) NOT NULL DEFAULT 'pending',
    -- 'pending' | 'running' | 'completed' | 'failed'
    source_url      TEXT NOT NULL,
    error_message   TEXT,
    progress_pct    SMALLINT NOT NULL DEFAULT 0,
    created_at      TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    started_at      TIMESTAMPTZ,
    completed_at    TIMESTAMPTZ
);
```

Invariant fondamental : le statut en base est la source de vérité. Si le processus redémarre, un job `running` sans process vivant est remis à `pending` au démarrage.

L'état en mémoire : le canal mpsc

```
// crates/gitrust-web/src/workers/import_worker.rs

use tokio::sync::mpsc;

#[derive(Clone)]
pub struct ImportJobSender(pub mpsc::Sender<Uuid>);

pub struct ImportWorker {
```


© Gitrust

```
    db: DatabaseConnection,
    rx: mpsc::Receiver<Uuid>,
    semaphore: Arc<Semaphore>,
}

impl ImportWorker {
    pub fn new(db: DatabaseConnection) -> (Self, ImportJobSender) {
        let (tx, rx) = mpsc::channel(100); // buffer de 100 IDs en attente
        let semaphore = Arc::new(Semaphore::new(4)); // max 4 imports simultanés
        let worker = Self { db, rx, semaphore };
        (worker, ImportJobSender(tx))
    }

    pub async fn run(mut self) {
        while let Some(job_id) = self.rx.recv().await {
            let db = self.db.clone();
            let permit = Arc::clone(&self.semaphore)
                .acquire_owned()
                .await
                .expect("semaphore closed");

            tokio::spawn(async move {
                let _permit = permit; // libéré quand le spawn se termine
                if let Err(e) = execute_import(&db, job_id).await {
                    tracing::error!(?job_id, error = %e, "import job failed");
                    let _ = ImportJobRepository::mark_failed(&db, job_id, &e.to_string(
                }
            });
        }
    }
}
```

Le `Semaphore` borne la concurrence. Même si 100 IDs arrivent dans le canal, seuls 4 `execute_import` s'exécutent en parallèle. Les autres attendent leur tour dans le canal.

Le handler HTTP — découpler la réponse de l'exécution

```
// crates/gitrust-web/src/routes/import.rs

pub async fn post_import(
    State(db): State<DatabaseConnection>,
    State(sender): State<ImportJobSender>,
    user: AuthUser,
```


© Gitrust

```
    Form(form): Form<ImportForm>,
) -> Result<impl IntoResponse, AppError> {
    // 1. Valider et persister
    let job = ImportJobRepository::create(&db, user.user_id, &form.source_url).await?;

    // 2. Envoyer l'ID dans le canal (non-bloquant grâce au buffer)
    sender.0.send(job.id).await.map_err(|_| AppError::Internal("worker unavailable".int

    // 3. Répondre immédiatement avec la page de suivi
    Ok(Redirect::to(&format!("/import/{}/progress", job.id)))
}
```

Le handler retourne avant que l'import ait commencé. Le navigateur est redirigé vers une page de progression.

Le worker — exécution avec transitions d'état

```
async fn execute_import(db: &DatabaseConnection, job_id: Uuid) -> anyhow::Result<()> {
    // Transition : pending → running
    ImportJobRepository::mark_running(db, job_id).await?;

    // Récupérer le job
    let job = ImportJobRepository::find(db, job_id).await?
        .ok_or_else(|| anyhow!("job not found: {}", job_id))?;

    // Étape 1/3 — cloner
    ImportJobRepository::update_progress(db, job_id, 10).await?;
    let temp_dir = git_clone(&job.source_url).await?;

    // Étape 2/3 — créer le dépôt gitrust
    ImportJobRepository::update_progress(db, job_id, 50).await?;
    let repo = RepositoryService::create_from_import(db, &job, &temp_dir).await?;

    // Étape 3/3 — finaliser
    ImportJobRepository::update_progress(db, job_id, 90).await?;
    finalize_import(db, job.id, repo.id).await?;

    // Transition : running → completed
    ImportJobRepository::mark_completed(db, job_id, repo.id).await?;
    Ok(())
}
```


© Gitrust

Chaque `update_progress` écrit en base. Si le process redémarre entre deux étapes, la progression est connue — mais le job doit être relancé depuis zéro (pas de reprise partielle dans ce modèle).

SSE — retour de progression vers le navigateur

```
// crates/gitrust-web/src/routes/import.rs

pub async fn stream_import_progress(
    Path(job_id): Path<Uuid>,
    State(db): State<DatabaseConnection>,
    user: AuthUser,
) -> Sse<impl Stream<Item = Result<Event, Infallible>>> {
    let stream = async_stream::stream! {
        loop {
            let job = ImportJobRepository::find(&db, job_id).await
                .unwrap_or(None);

            match job {
                None => {
                    yield Ok(Event::default().data("<p>Job introuvable.</p>"));
                    break;
                }
                Some(ref j) if j.status == "completed" => {
                    yield Ok(Event::default()
                        .data(format!("<p>Import terminé — <a href='/{}/{}'>Voir le dép
                            j.owner_login, j.repo_slug)));
                    break;
                }
                Some(ref j) if j.status == "failed" => {
                    yield Ok(Event::default()
                        .data(format!("<p class='text-error'>Erreur : {}</p>",
                            j.error_message.as_deref().unwrap_or("inconnue"))));
                    break;
                }
                Some(ref j) => {
                    yield Ok(Event::default()
                        .data(format!("<progress class='progress' value='{}' max='100'>
                            j.progress_pct)));
                    tokio::time::sleep(Duration::from_millis(500)).await;
                }
            }
        }
    };
```


© Gitrust

```
    Sse::new(stream).keep_alive(KeepAlive::default())
}
```

Le template HTMX consomme ce flux :

```
<!-- templates/import/progress.html -->
<div
  hx-ext="sse"
  sse-connect="/import/{{ job_id }}/stream"
  sse-swap="message"
  hx-swap="innerHTML"
  id="progress-container"
>
  <progress class="progress" value="0" max="100"></progress>
</div>
```

Récupération au redémarrage

Au démarrage de l'application, gitrust remet en `pending` les jobs bloqués en `running` :

```
// crates/gitrust-web/src/main.rs (dans la fonction de démarrage)

async fn recover_stuck_jobs(db: &DatabaseConnection, sender: &ImportJobSender) {
    let stuck = ImportJobRepository::find_by_status(db, "running").await
        .unwrap_or_default();

    for job in stuck {
        tracing::warn!(?job.id, "recovering stuck import job");
        let _ = ImportJobRepository::reset_to_pending(db, job.id).await;
        let _ = sender.0.send(job.id).await;
    }
}
```

Cette fonction s'exécute une fois au démarrage, avant que le serveur HTTP accepte des connexions.

Enregistrement dans main.rs

```
// crates/gitrust-web/src/main.rs

let (import_worker, import_sender) = ImportWorker::new(db.clone());
```


© Gitrust

```
// Démarrer le worker dans un task dédié
tokio::spawn(async move { import_worker.run().await });

// Récupérer les jobs bloqués
recover_stuck_jobs(&db, &import_sender).await;

// Injecter le sender dans l'état Axum
let app = Router::new()
    // ... routes ...
    .with_state(AppState { db, import_sender, /* ... */ });
```

Décisions et compromis

Durabilité : base de données vs queue dédiée

Approche	Durabilité	Complexité opérationnelle
Table SQL (gitrust)	Au niveau PostgreSQL	Nulle (déjà présent)
Redis + Bull/Sidekiq	Haute (AOF/RDB)	Redis à opérer
RabbitMQ / NATS	Très haute	Broker supplémentaire
In-memory seulement	Nulle (perdu au restart)	Nulle

Gitrust choisit la table SQL : la durabilité est suffisante (perte maximale = 1 transaction), et PostgreSQL est déjà requis. Un broker externe serait disproportionné pour une forge auto-hébergée à petite échelle.

Concurrence : Semaphore vs pool de workers

Le `Semaphore(N)` est plus simple qu'un pool de threads ou de tâches fixes. Il permet au runtime Tokio de gérer l'ordonnancement. L'inconvénient : si un job prend 30 secondes et que N=4, les 5e et 6e jobs attendent dans le canal, pas dans un pool observable.

Pour des jobs très longs (> 5 minutes), un pool de workers avec heartbeat serait plus robuste — mais ce cas ne se présente pas dans gitrust aujourd'hui.

© Gitrust

SSE vs WebSocket vs polling

Méthode	Cas d'usage
Polling HTMX (`hx-trigger="every 2s"`)	Statut simple, peu de mises à jour
SSE (gitrust)	Flux unidirectionnel continu (logs, progression)
WebSocket	Bidirectionnel (chat, collaboration temps réel)

SSE est suffisant pour tous les workers actuels de gitrust (import, CI logs). WebSocket ajouterait une complexité d'état bidirectionnel non nécessaire.

Vérifier votre compréhension

1. Le serveur gitrust redémarre pendant l'import d'un dépôt. Le job était en statut `running` avec `progress_pct = 60`. Que se passe-t-il exactement au redémarrage ? L'utilisateur verra-t-il la progression reprendre à 60 % ou à 0 % ?

2. Le `Semaphore` est initialisé à 4 slots. 10 utilisateurs lancent un import simultanément. Combien de `tokio::spawn` s'exécutent en parallèle ? Où attendent les 6 jobs restants ? Que se passe-t-il si le serveur redémarre à cet instant ?

Pour aller plus loin

- Créer un worker async
- Schéma base de données
- Décisions UI HTMX
- Documentation Tokio — mpsc

© Gitrust

1.5.9 Conception de ssh-guard

Ce que vous allez comprendre

- Identifier les défauts de sécurité SSH de gitrust avant l'introduction de ssh-guard et le périmètre exact que cette nouvelle crate couvre.
- Analyser le design async/event-driven choisi : pourquoi un listener wrapper, pourquoi des détecteurs push-based, pourquoi le store hybride.
- Évaluer les compromis (fail-open, dry-run, idempotence, priorité ACL) et savoir comment les ajuster.

> ***Public** : contributeurs au code de `gitrust-ssh` et `gitrust-ssh-guard`. Pour une recette d'exploitation, voir Configurer ssh-guard.*

---

1. Le problème concret

Avant ssh-guard, le serveur SSH `russh` intégré à gitrust avait quatre angles morts opérationnels :

1. **IP cliente faussée derrière nginx stream.** Le déploiement type (« nginx en frontal sur `:22`, `proxy_protocol on;` vers `127.0.0.1:2222` ») rendait `peer_addr() = 127.0.0.1` côté gitrust. Conséquence : tous les rate-limits par IP étaient inutilisables, les logs n'identifiaient personne, fail2ban était impossible à brancher.
2. **Aucune détection de motifs d'attaque.** La couche `russh` rejette une auth invalide, mais ne corrèle rien : 1 000 tentatives en 30 secondes ressemblent à 1 000 lignes de log indépendantes.
3. **Format de logs instable.** Les messages d'erreur de `russh` évoluent entre versions, ce qui rendait les filtres fail2ban fragiles (cassés à chaque mise à jour).
4. **Pas de levier admin.** Aucune ACL CIDR (allowlist d'un VPN, denylist d'un AS abusif) ne pouvait être appliquée à l'extérieur du firewall système, donc inopérante pour la majorité des admins.

ssh-guard adresse exactement ces quatre points — pas plus. Toute la couche TLS, OAuth, JWT, RBAC reste hors de son périmètre.

---

2. L'analogie

Imaginez une porte d'immeuble (le `TcpListener`) et derrière elle un agent d'accueil (`russh`) qui contrôle les badges. ssh-guard est le **sas** qui s'intercale entre les deux. Il fait trois choses, dans cet ordre :

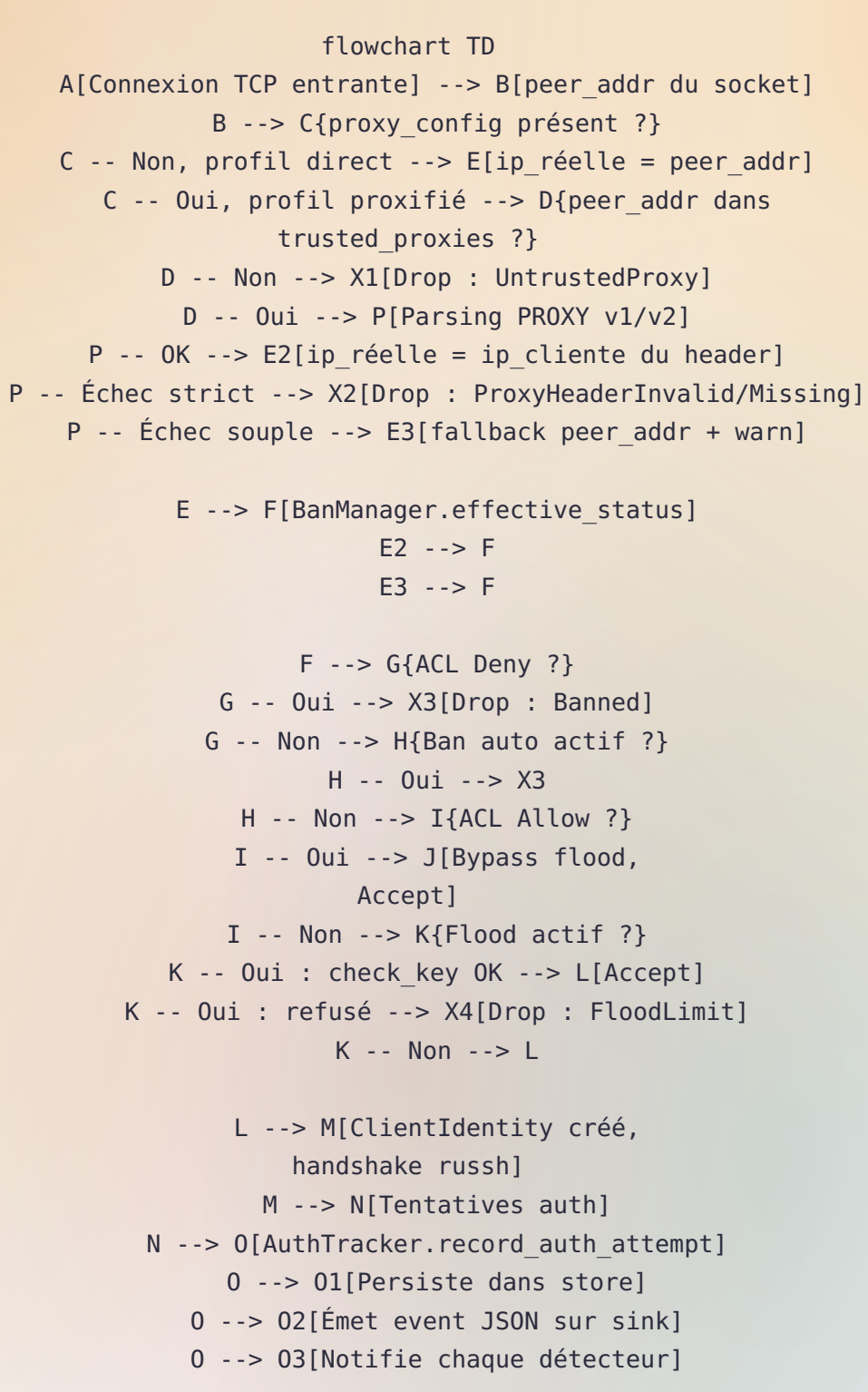
1. **Lire la pièce d'identité** correctement, qu'elle soit présentée directement ou tendue par un coursier (le proxy) — c'est la résolution d'IP réelle.
2. **Consulter une liste blanche/noire** affichée à l'intérieur du sas — c'est l'ACL admin.
3. **Compter les visiteurs récents** par identité et activer une alarme si quelqu'un sonne 5 fois en 5 minutes — c'est la détection.

© Gitrust

Le sas n'inspecte pas les badges (c'est le rôle de `russh`). Il décide juste qui a le droit d'arriver jusqu'à l'agent d'accueil.

3. Le modèle

Vie d'une connexion



© Gitrust

```
O3 --> P1[BruteForce / UserEnum / KeyScan]
P1 --> Q{Seuil atteint ?}
Q -- Oui --> R[BanManager.auto_ban]
R --> S[Insert ban + event IpBanned]
Q -- Non --> Z[Fin]
```

Trois plans qui interagissent

Plan	Composants	Caractéristique principale
Synchrone, hot path	`SecureListener.accept`, `BanManager.effective_status`, `ConnectionFloodDetector.check`	Lock-free (DashMap, GCRA), aucun appel DB tant que le store est `memory` ou `hybrid`
Asynchrone, push	`AuthTracker.record_auth_attempt` → détecteurs → `BanManager.auto_ban`	Tâches Tokio courtes, jamais bloquantes
Persistant	`HybridStore`, `PostgresStore`	Write-through périodique, `rehydrate` au boot

L'invariant clé : **le hot path ne touche jamais directement la DB**. Si PostgreSQL tombe, le listener continue à servir avec sa vue mémoire des bans/ACL, et la prochaine écriture admin sera mise en attente (HybridStore) ou échouera proprement (Postgres direct).

---

4. Les décisions de conception

4.1 Listener wrapper plutôt que middleware russh

ssh-guard intercale `SecureListener` **entre** `TcpListener` et `russh`. Trois raisons :

- Beaucoup de drops (flood, ban) doivent se faire **avant** d'allouer un handler `russh` (RAM, threads). Un middleware `russh` coûterait des allocations inutiles à chaque scan de port.
- L'extraction d'IP via PROXY protocol nécessite de **peeker** les premiers octets du `TcpStream` avant que `russh` ne le voie — un middleware de niveau session ne peut pas faire ça.
- Découplage strict : ssh-guard ne dépend pas de `russh` (testable en standalone, réutilisable si on remplace la lib SSH).

© Gitrust

4.2 Push-based detectors

Chaque détecteur expose `async fn on_event(event: &GuardEvent)`. L'`AuthTracker` les appelle après chaque événement. Avantages :

- Pas de scheduler de fond à entretenir. Le détecteur se réveille uniquement quand il y a quelque chose à analyser.
- Idempotence facile : un second appel après franchissement du seuil retombe sur un `auto_ban` no-op (déjà banni).
- Test déterministe : on injecte des événements synthétiques et on observe le sink — pas de `tokio::time::sleep` à attendre.

Coût : chaque détecteur fait une lecture du store pour calculer son agrégat. C'est acceptable car on ne déclenche qu'aux échecs d'auth (rare) et que l'agrégat est borné par la fenêtre.

4.3 Priorité ACL : `deny > auto_ban > allow > default`

L'ordre est exécuté tel quel dans `BanManager.effective_status`. Conséquences :

- `deny` **admin** > **tout** : un opérateur peut brûler une IP même si elle a été allowlistée par erreur ailleurs.
- `auto_ban` > `allow` : un ban auto déjà posé ne saute pas si on allowliste après coup. Pour le lever, il faut explicitement `unban`. C'est un choix conservateur : un allow ne doit pas effacer la trace d'un comportement passé.
- `allow` > `default` : une IP allowlistée bypasse les détecteurs (mais ses événements restent loggés pour audit).

L'invariant `auto_ban` no-op si IP allowlistée est un garde-fou de défense en profondeur : si un détecteur oublie un jour de consulter l'ACL avant d'appeler `auto_ban`, le `BanManager` refuse quand même.

4.4 Fail-open sur erreur de store

Quand `BanManager.effective_status` reçoit `Err(StoreError)`, le listener log `warn` et **laisse passer**. C'est un compromis :

- **Pour** : une panne PostgreSQL n'arrête pas le service Git en lecture/push (les opérateurs ne sont pas réveillés à 3h du matin pour une indisponibilité partielle).
- **Contre** : pendant la fenêtre de panne, un attaquant pourrait passer (mais l'auth `russh` reste opérationnelle, donc l'attaquant ne devient utilisateur que s'il connaît une clé valide).

Le compromis a été pris sciemment : il est documenté, et on peut le renverser en remplaçant `Ok(...)` par `return Ok(self.drop_reason(ip, DropReason::Banned).await)` dans la branche `Err` si une instance préfère fail-closed.

4.5 Dry-run avec émission d'événements

`SSH_GUARD_DRY_RUN=true` désactive la persistance des bans mais **continue à émettre les événements** `IpBanned`. C'est le mode parfait pour :

- valider un nouveau seuil en prod sans risquer de bannir un utilisateur ;
- alimenter fail2ban (qui ferait le ban réel via UFW/iptables) tout en laissant ssh-guard observer ;
- répliquer le comportement d'un détecteur dans un environnement de pré-prod.

© Gitrust

4.6 Stockage hybride par défaut

Trois backends, mais le défaut **hybrid** est presque toujours le bon :

- les lectures hot path passent par DashMap (latence µs) ;
- chaque écriture est **synchrone côté mémoire** + **asynchrone vers PostgreSQL** (write-through) ;
- au boot, `rehydrate` repeuple la mémoire depuis la DB en une seule transaction. Si la DB est vide ou inaccessible, on démarre avec une mémoire vide et un warn (fail-open au boot aussi).

Le mode `memory` est réservé aux tests et au profil `private` (réseau interne, on accepte de perdre l'historique au restart). Le mode `postgres` direct existe pour les rares cas où une instance multi-noeuds **n'a pas** de cache RAM cohérent et veut tout passer par la DB.

4.7 Schéma JSON stable

`#[serde(tag = "event", rename_all = "snake_case")]` produit `{"event":"<nom>","ts":"...","ip":"...",...}`. La règle inviolable :

- ajouter un nouveau variant ou un nouveau champ optionnel : OK ;
- renommer un variant ou un champ existant : breaking change → nouveau nom + period de transition.

C'est ce qui rend les filtres fail2ban et les requêtes Loki stables dans le temps. Un test (`event_name_matches_serde_tag`) garantit que le nom Rust et le tag JSON ne divergent pas.

---

5. Ce que ssh-guard ne fait pas (volontairement)

Hors périmètre	Pourquoi	Couvert par
Inspection des payloads SSH	Niveau TCP uniquement, ssh-guard n'a pas la clé d'hôte	`russh` lui-même
Authentification utilisateur	Domaine de `gitrust-core::SshKeyService`	`gitrust-ssh`
Limites de débit applicatives (push trop gros)	Concerne le pack-protocol, pas la connexion	`gitrust-git`
Distribution multi-noeuds	Pas d'algorithme de gossip, pas de Raft	À traiter par le store backend (Postgres partagé)
GeoIP / blocage par pays	Décision politique, pas technique	Reverse-proxy ou firewall en amont

© Gitrust

6. Implications pour les contributeurs

Quand ajouter un détecteur

Si un nouveau motif d'attaque émerge (ex. : « attaques par chronométrie sur les fingerprints »), suivre ce squelette :

1. Ajouter un fichier `src/detector/<nom>.rs` calqué sur `brute_force.rs`.
2. Implémenter `pub async fn on_event(&self, event: &GuardEvent)` qui filtre les événements pertinents et lit l'agrégat via `GuardStore`.
3. Ajouter une variante `BanReason::<Nom>` et un `GuardEvent::<Nom>Detected { ... }` dans `events.rs`. Mettre à jour `event_name` et le test `event_name_matches_serde_tag`.
4. Câbler le détecteur dans `runtime.rs::build` (instanciation conditionnelle si `threshold.is_disabled()`).
5. Câbler dans `tracker.rs` (ajout d'un `Option<Arc<NouveauDetector>>` + appel dans `record_auth_attempt`).
6. Documenter le nouveau seuil dans `config.rs` (env var + preset par profil) et dans la page admin de référence.

Quand modifier le format d'événement

Ne pas. Si un champ doit changer de type ou de sens, créer un **nouveau variant** et garder l'ancien jusqu'à période de transition explicite. Le test de stabilité du tag est là pour casser la PR si quelqu'un le tente sans s'en rendre compte.

Quand changer une priorité ACL

Pratiquement jamais. La priorité actuelle (`deny > auto_ban > allow > default`) est un consensus défensif. Si une PR la modifie, exiger un dossier d'opportunité avec attaques traitées et nouveaux compromis acceptés.

---

7. Vérifier votre compréhension

1. Une connexion arrive avec un en-tête PROXY v2 valide depuis `192.168.10.5`. La config est `SSH_GUARD_PROFILE=nginx` (donc `trusted_proxies` = `127.0.0.1/32, ::1/128`). Quel est l'`AcceptOutcome` ? Pourquoi ?
2. Une IP est dans la denylist admin **et** vient d'être bannie automatiquement pour brute force. Quel `EffectiveStatus` retourne `effective_status` ? L'ordre des deux lookups dans le code change-t-il quelque chose ?
3. PostgreSQL tombe pendant 30 minutes. Une attaque brute-force démarre depuis une IP nouvelle. Que se passe-t-il : (a) au moment de l'accept, (b) au moment où le détecteur lit `count_auth_failures` ?

---

8. Pour aller plus loin

- Crate gitrust-ssh-guard — référence structurelle (modules, types, API publique)
- Architecture des crates — position dans le graphe de dépendances
- Vue d'ensemble de l'architecture — où ssh-guard s'insère dans le runtime gitrust
- ssh-guard : détection d'attaques SSH — vue admin de la chaîne de détection
- Conformité ANSSI PA-074 — directives applicables au code de la crate

© Gitrust