Engineer

Linter

🚀 Outil signalant des bogues probables sans exécuter le code

Ce que c’est

Un linter est un programme qui lit le code source et signale les parties ressemblant à des erreurs, sans construire ni exécuter le programme. Chaque signalement nomme une règle, un fichier et une ligne : variable affectée mais jamais utilisée, comparaison toujours vraie, variable shell laissée sans guillemets, chaîne de format construite à partir d’une entrée utilisateur.

Le nom vient de lint, un vérificateur de programmes C que NetBSD fournit encore. Son manuel définit le travail comme la détection de caractéristiques des fichiers C « susceptibles d’être des bogues, de manquer de portabilité ou de gaspiller des ressources » et précise que son contrôle des types est plus strict que celui des compilateurs traditionnels antérieurs à C90. Il fonctionne en deux passes : chaque fichier est d’abord vérifié séparément, puis les résultats sont croisés pour repérer les incohérences entre fichiers.

La plupart des langages ont désormais leurs propres linters, souvent plusieurs :

- **ESLint** pour JavaScript.
- **Ruff** pour Python.
- **ShellCheck** pour les scripts shell. Ses objectifs déclarés vont des erreurs de syntaxe de débutant aux subtilités susceptibles de faire échouer plus tard un script qui fonctionne aujourd’hui.
- **Stylelint** pour CSS, **markdownlint** pour Markdown et **yamllint** pour YAML.

Chaque linter est construit à partir de règles. Chacune possède un identifiant, comme F841 dans Ruff ou SC2059 dans ShellCheck, et un niveau de gravité. Un fichier de configuration décide quelles règles sont actives et de leur gravité, et beaucoup proposent une correction automatique.

Un outil de formatage réécrit la disposition sans se prononcer sur le sens ; un vérificateur de types raisonne sur la nature de chaque valeur. Le linter se situe entre les deux et examine la façon dont le code est écrit : mauvais usage du langage, motifs risqués et conventions choisies par le projet.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'un linter ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Une seule équipe transforme une idée en un produit auquel vos clients peuvent se connecter, sans le faire passer entre trois prestataires.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Application GIS : revenus multipliés par 10 — 10x croissance du chiffre d'affaires portée par le produit cartographique

Application GIS : revenus multipliés par 10 Contactez-nous →

Pourquoi c’est important

Un linter lit chaque branche, y compris le gestionnaire d’erreur exécuté une fois par an qu’aucun test n’atteint. Quelques secondes suffisent pour un dépôt entier : il peut donc s’exécuter à chaque sauvegarde et chaque commit, et c’est au moment où la ligne est écrite que sa correction coûte le moins.

Les règles règlent les débats une fois pour toutes. L’acceptabilité d’un `except:` sans précision ou l’obligation de mettre chaque variable shell entre guillemets se décident dans la configuration, puis s’appliquent de la même façon à chaque auteur et à chaque revue. Les personnes chargées de relire gardent leur attention pour les questions qu’une règle ne peut pas exprimer.

Le code de sortie est l’interface entre un linter et ce qui l’exécute. Le README de ShellCheck précise que l’outil « fait un usage canonique des codes de sortie » : le processus de construction peut donc l’appeler comme toute autre commande. Ruff et ESLint documentent la même convention à trois états : 0 si le contrôle ne trouve rien, 1 s’il relève des violations et 2 si l’outil lui-même échoue, par exemple sur une configuration invalide. Un contrôle qualité peut ainsi distinguer « le code est incorrect » de « le vérificateur est défectueux » ; les deux doivent l’arrêter.

La gravité détermine quels signalements influent sur ce code de sortie. ESLint réussit s’il trouve des avertissements mais aucune erreur, sauf si `--max-warnings` impose une limite, et lint(1) possède `-w` pour traiter les avertissements comme

des erreurs. Un projet qui veut que ses linters bloquent les erreurs, au-delà de les décrire, écrit ce choix à l'endroit utilisé à chaque exécution :

```
# every finding fails the run, warnings included
lint -w -abcFghprSxz source.c
eslint --max-warnings 0 src/
ruff check src/
shellcheck scripts/*.sh
```

Ce site fait le choix le plus strict. Son contrôle qualité traite chaque diagnostic comme une erreur et interdit les suppressions en ligne comme `# noqa` : on corrige donc la cause du défaut signalé.

À quoi cela ressemble en pratique

La configuration d'un linter commence souvent par les règles qui ne signalaient rien le jour de son installation. Une page renverse cette approche : toutes les règles d'un linter Python activées comme erreurs. La première exécution a produit 1 815 signalements, traités par catégorie plutôt que par fichier, car ces catégories apportent des informations : les arguments inutilisés et les noms prédéfinis masqués étaient du bruit, tandis que la sécurité, les valeurs par défaut mutables et la gestion des exceptions révélaient chacune des comportements réels. Les rares exceptions portent une justification écrite sur la ligne ; deux sont appuyées par un test qui vérifie par assertion la propriété que la règle désactivée aurait imposée. Un contrôle de types strict s'exécutait en parallèle et détectait un autre genre de défaut, lié à ce qu'est une valeur plutôt qu'à la façon dont elle est écrite. Le coût est indiqué aussi : quelqu'un doit exercer son jugement 1 815 fois.

Une autre page part d'une découverte gênante. Le contrôle qualité du site, trente-deux scripts et 10 242 lignes de JavaScript, était le plus grand ensemble de code du dépôt et le seul qu'aucun outil de formatage, linter ou vérificateur de types ne lisait. Soumettre le code du contrôle qualité lui-même à un linter a commencé par l'outil de formatage, réglé à deux espaces d'indentation pour ces fichiers afin de modifier 38 lignes plutôt que 2 173. Le linter a ensuite produit treize signalements réels, tous corrigés, aucun supprimé, puis un vérificateur de types a trouvé deux défauts que le linter avait laissés passer. La raison de cet effort est donnée : un défaut de vérificateur se manifeste par un contrôle qui réussit à tort.

L'idée s'applique aussi à la prose. Une troisième page décrit un linter pour la documentation propre au dépôt, avec des limites bloquantes plutôt que de simples conseils. Ses huit politiques mécaniques comprennent un budget de lignes par fichier qui ne peut que diminuer, une règle imposant l'accord entre l'état d'un plan et ses cases à cocher, et un contrôle de l'existence, dans l'arbre, de chaque chemin et symbole de code nommé par un document. La première exécution a trouvé six plans marqués terminés avec des cases non cochées, quatorze tâches bloquant chaque commit mais absentes de toute documentation, trois liens cassés, un document orphelin et trois chemins morts.

Où les problèmes surviennent

Un linter échoue rarement bruyamment. Le défaut habituel est une exécution qui se termine avec 0 alors qu'elle aurait dû arrêter quelque chose.

Des signalements qui ne changent jamais le code de sortie. `--max-warnings` d'ESLint vaut -1 par défaut, ce qui désactive la limite, et le manuel de lint(1) indique que sa passe entre fichiers se termine toujours avec succès. Un signalement incapable de faire échouer l'exécution est lu une fois, puis ignoré en faisant défiler. Décidez pour chaque règle : erreur bloquant l'exécution ou désactivation, et placez toute option transformant les avertissements en erreurs dans la configuration partagée.

Une correction qui annonce un succès. Ruff se termine avec 0 lorsque toutes les violations trouvées ont été corrigées automatiquement. C'est commode au bureau, mais un contrôle qualité avec les corrections activées ne répare que sa copie de travail ; le commit reçu contient encore toutes les violations. Exécutez le contrôle sans corrections ou ajoutez `--exit-non-zero-on-fix` pour qu'une correction compte comme un échec. Recherchez aussi `--exit-zero` : Ruff se termine avec 0 malgré les violations, et une option ajoutée pour une migration bruyante a tendance à rester.

Un motif qui ne sélectionne aucun fichier. Un répertoire renommé ou une configuration déplacée peut laisser un linter pointer vers un ensemble vide, et vérifier zéro fichier donne un résultat sans erreur. ESLint traite par défaut comme erreur un motif glob entre guillemets qui ne correspond à aucun fichier. `--no-error-on-unmatched-pattern` le transforme en réussite silencieuse, et `--pass-on-no-patterns` se termine avec 0 si aucun motif n'est fourni. Gardez le comportement par défaut et un fichier contenant une violation connue que le contrôle doit signaler.

Un jeu de règles choisi selon ce qui ne signalait rien. Une configuration élaborée en désactivant tout ce qui réagissait le premier jour enregistre les habitudes du code plutôt qu'une norme. Partez d'une sélection délibérée, soit tout à la fois, soit la petite base `E` et `F` proposée par Ruff avec un groupe ajouté à la fois, et justifiez par écrit chaque désactivation. Ruff prévient que `ALL` active implicitement de nouvelles règles à chaque mise à jour : figez donc sa version et traitez sa mise à jour comme une modification distincte.

Les suppressions se propagent ligne après ligne. Chaque linter possède une échappatoire : `/* LINTED */` dans lint(1), `# noqa: F841` dans Ruff, `# shellcheck disable=SC2059` dans ShellCheck et `// eslint-disable-next-line` dans ESLint. Chacune indique ce qui a été rendu silencieux, pas pourquoi. Exigez une justification écrite près de chaque exception et, pour celles qui comptent, désactivez la règle puis ajoutez un test de la propriété qu'elle protégeait. Regrouper les exceptions aide aussi : `--no-inline-config` d'ESLint rend les commentaires de désactivation inopérants, et ShellCheck lit les lignes `disable` communes au projet dans `.shellcheckrc` .

Pour aller plus loin

- lint(1) — le manuel NetBSD du vérificateur C à l'origine du nom : ses deux passes, `-w` et les commentaires de suppression.
- Le linter Ruff — choix des règles, sûreté des corrections, commentaires de suppression et codes de sortie.
- Référence de l'interface en ligne de commande — `--max-warnings` d'ESLint, configuration en ligne, motifs sans correspondance et codes de sortie.
- ShellCheck — le README du projet : les défauts recherchés et son intégration au processus de construction.
- Ignorer — les moyens de ShellCheck pour ignorer un signalement par ligne, exécution ou projet.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce qu'un linter ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/linter/