Engineer

Linter

🖋 Werkzeug, das Code ohne Ausführung auf mögliche Fehler prüft

Was es ist

Ein Linter ist ein Programm, das Quellcode liest und Stellen meldet, die nach Fehlern aussehen, ohne das Programm zu bauen oder auszuführen. Jede Meldung nennt eine Regel, eine Datei und eine Zeile: eine Variable, die einen Wert erhält, aber nie verwendet wird, einen immer wahren Vergleich, eine Shellvariable ohne Anführungszeichen oder eine aus Nutzereingaben aufgebaute Formatzeichenkette.

Der Name stammt von lint, einem Prüfprogramm für C, das NetBSD weiterhin mitliefert. Sein Handbuch beschreibt die Aufgabe als Erkennen von Eigenschaften in C-Programmdateien, „die wahrscheinlich fehlerhaft, nicht portabel oder verschwenderisch sind“. Es ergänzt, dass die Typprüfung strenger ist als bei herkömmlichen Compilern vor C90. Das Werkzeug arbeitet in zwei Durchgängen: Zuerst wird jede Datei als eigene Einheit geprüft, danach werden die Ergebnisse auf Widersprüche zwischen Dateien abgeglichen.

Die meisten Sprachen haben inzwischen eigene Linter, oft mehrere:

- **ESLint** für JavaScript.
- **Ruff** für Python.
- **ShellCheck** für Shellskripte. Die erklärten Ziele reichen von typischen Syntaxproblemen bei Einsteigern bis zu subtilen Fallstricken, die ein funktionierendes Skript unter künftigen Bedingungen scheitern lassen können.
- **Stylelint** für CSS, **markdownlint** für Markdown und **yamllint** für YAML.

Jeder Linter besteht aus Regeln. Jede Regel hat eine Kennung, etwa F841 in Ruff oder SC2059 in ShellCheck, und einen Schweregrad. Eine Konfigurationsdatei legt fest, welche Regeln aktiv sind und wie schwer jede wiegt. Viele Regeln

bieten eine automatische Korrektur.

Ein Formatter schreibt das Layout um und bewertet die Bedeutung nicht; ein Typprüfer untersucht, was jeder Wert ist. Ein Linter liegt dazwischen und betrachtet die Schreibweise des Codes: Fehlgebrauch der Sprache, riskante Muster und die Konventionen, auf die sich ein Projekt geeinigt hat.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist ein Linter?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Ein Team führt eine Idee von der Skizze bis zu einem Produkt, bei dem Ihre Kunden sich anmelden können, statt sie zwischen drei Dienstleistern weiterzureichen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. GIS-Kartenprodukt zu 10-fachem Umsatz geführt — 10x Umsatzwachstum durch die Leitung des Kartenprodukts

GIS-Kartenprodukt zu 10-fachem Umsatz geführt Kontakt aufnehmen →

Warum es wichtig ist

Ein Linter liest jeden Zweig, auch die Fehlerbehandlung, die einmal im Jahr läuft und die kein Test erreicht. Für ein ganzes Repository braucht er Sekunden. Damit kann er bei jedem Speichern und jedem Commit laufen, und am wenigsten kostet ein Befund in dem Moment, in dem die Zeile geschrieben wird.

Regeln entscheiden Streitfragen einmal. Ob ein bloßes `except:` zulässig ist oder jede Shellvariable in Anführungszeichen stehen muss, wird in der Konfiguration festgelegt und danach bei jedem Autor und jeder Überprüfung gleich angewandt. Menschliche Reviewer behalten ihre Aufmerksamkeit für die Fragen, die keine Regel ausdrücken kann.

Der Exitcode ist die Schnittstelle zwischen einem Linter und dem, was ihn ausführt. Laut ShellChecks README verwendet das Werkzeug „Exitcodes auf die vorgesehene Weise“. Ein Build kann es deshalb wie einen weiteren Befehl aufrufen. Ruff und ESLint dokumentieren dieselbe Konvention mit drei Ergebnissen: 0 bei fehlerfreier Prüfung, 1 bei gefundenen Verstößen und 2, wenn das Werkzeug selbst scheitert, etwa an einer ungültigen Konfiguration. Eine Qualitätsprüfung kann damit „der Code ist falsch“ von „das Prüfprogramm ist defekt“ unterscheiden. Beides sollte den Lauf stoppen.

Der Schweregrad entscheidet, welche Befunde diesen Exitcode beeinflussen. ESLint endet erfolgreich, wenn es Warnungen und keine Fehler findet, sofern `--max-warnings` keine Grenze setzt. lint(1) bietet `-w`, um Warnungen als Fehler zu behandeln. Ein Projekt, dessen Linter Fehler verhindern und nicht nur beschreiben sollen, hält diese Entscheidung dort fest, wo jeder Lauf sie verwendet:

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

Diese Website wählt die strenge Variante. Ihre eigene Qualitätsprüfung zählt jede Diagnose als Fehler und verbietet Unterdrückungen direkt im Code wie `# noqa`. Ein Befund wird deshalb an seiner Ursache behoben.

Wie es in der Praxis aussieht

Die Konfiguration eines Linters beginnt oft mit den Regeln, die an dem Tag, als jemand ihn eingerichtet hat, zufällig nichts meldeten. Eine Seite kehrt das um: jede Regel eines Python-Linters als Fehler aktiviert. Der erste Lauf lieferte 1.815 Befunde. Sie wurden nach Kategorie statt nach Datei abgearbeitet, weil die Kategorien Informationen enthalten: Ungenutzte Argumente und verdeckte Builtins waren Rauschen, während die Kategorien Sicherheit, veränderliche Standardwerte und Ausnahmebehandlung jeweils auf echtes Verhalten hinwiesen. Die wenigen Ausnahmen tragen eine schriftliche Begründung an der jeweiligen Zeile. Zwei davon sind durch einen Test abgesichert, der die Eigenschaft prüft, die die abgeschaltete Regel durchgesetzt hätte. Parallel lief eine strenge Typprüfung und fand eine andere Art von Fehler: was ein Wert ist, statt wie er geschrieben wird. Auch die Kosten werden genannt: Jemand muss 1.815-mal eine Entscheidung treffen.

Eine andere Seite beginnt mit einer unangenehmen Entdeckung. Die Qualitätsprüfung der Website, zweiunddreißig Skripte und 10.242 Zeilen JavaScript, war damals der größte Codebestand im Repository und der einzige, den weder Formatter noch Linter oder Typprüfer lasen. Den eigenen Code der Qualitätsprüfung unter einen Linter zu stellen begann mit dem Formatter. Für diese Dateien wurde er auf Einrückung mit zwei Leerzeichen eingestellt, sodass sich 38 Zeilen statt 2.173 änderten. Der Linter lieferte anschließend dreizehn

echte Befunde, alle behoben und keiner unterdrückt. Ein Typprüfer fand zwei Fehler, die der Linter hatte passieren lassen. Der Grund für die Arbeit steht dabei: Ein Fehler in einem Prüfprogramm zeigt sich als Prüfung, die besteht, obwohl sie nicht bestehen dürfte.

Die Idee funktioniert auch bei Prosa. Eine dritte Seite beschreibt einen Linter für die eigene Dokumentation des Repositorys mit verbindlichen Grenzen, bei deren Verletzung die Prüfung scheitert, statt nur Empfehlungen auszugeben. Seine acht mechanischen Regeln umfassen ein Zeilenbudget je Datei, das nur kleiner werden darf, eine Regel, nach der der Status eines Plans zu seinen eigenen Kontrollkästchen passen muss, und eine Prüfung, ob jeder in einem Dokument genannte Pfad und jedes Codesymbol im Verzeichnisbaum existiert. Der erste Lauf fand sechs als erledigt markierte Pläne mit offenen Kästchen, vierzehn bei jedem Commit vorgeschaltete Aufgaben, die in keinem Dokument vorkamen, drei defekte Links, ein verwaistes Dokument und drei tote Pfade.

Wo es schiefgeht

Ein Linter versagt meist unauffällig. Der übliche Fehler ist ein Lauf, der mit 0 endet, obwohl er etwas hätte stoppen sollen.

Befunde, die den Exitcode nie verändern. ESLints `--max-warnings` steht standardmäßig auf -1 und schaltet damit die Grenze aus. Das Handbuch von lint(1) erklärt, dass sein dateiübergreifender Durchgang immer erfolgreich endet. Ein Befund, der den Lauf nicht scheitern lassen kann, wird einmal gelesen und danach weggescrollt. Entscheiden Sie je Regel: ein Fehler, der den Lauf scheitern lässt, oder ausgeschaltet. Setzen Sie jeden Schalter zur Behandlung von Warnungen als Fehler in der gemeinsamen Konfiguration.

Eine Korrektur meldet Erfolg. Ruff endet mit 0, wenn alle gefundenen Verstöße automatisch korrigiert wurden. Am Arbeitsplatz ist das bequem. Eine Qualitätsprüfung mit eingeschalteten Korrekturen repariert aber nur ihre eigene Arbeitskopie; der eingegangene Commit enthält weiterhin jeden Verstoß. Führen Sie die Qualitätsprüfung ohne Korrekturen aus oder ergänzen Sie `--exit-non-zero-on-fix`, damit eine Korrektur als Fehlschlag zählt. Suchen Sie auch nach `--exit-zero`: Damit endet Ruff selbst bei gefundenen Verstößen mit 0. Ein für eine Migration mit vielen Befunden ergänzter Schalter bleibt leicht bestehen.

Ein Muster wählt keine Dateien aus. Ein umbenanntes Verzeichnis oder eine verschobene Konfiguration kann einen Linter auf eine leere Menge richten. Null geprüfte Dateien gelten als fehlerfreies Ergebnis. ESLint behandelt ein in Anführungszeichen gesetztes Glob-Muster ohne Treffer standardmäßig als Fehler. `--no-error-on-unmatched-pattern` macht daraus stillschweigendes Bestehen, und `--pass-on-no-patterns` endet mit 0, wenn keine Dateimuster angegeben sind. Behalten Sie den Standard bei und halten Sie eine Datei mit einem bekannten Verstoß vor, den die Qualitätsprüfung melden muss.

Ein Regelsatz richtet sich danach, was nichts meldete. Eine Konfiguration, die alles abschaltet, was am ersten Tag anschlug, dokumentiert die Gewohnheiten des Codes statt eines Standards. Beginnen Sie mit einer bewussten Auswahl: entweder alles auf einmal oder Ruffs empfohlene kleine Basis aus `E` und `F`, die Sie jeweils um eine Gruppe erweitern. Halten Sie fest, warum jede abgeschaltete Regel aus ist. Ruff warnt, dass `ALL` bei jeder Aktualisierung des Werkzeugs implizit neue Regeln einschaltet. Fixieren Sie deshalb die Version des Linters und behandeln Sie seine Aktualisierung als eigene Änderung.

Unterdrückungen breiten sich Zeile für Zeile aus. Jeder Linter hat einen Ausweg: `/* LINTED */` in lint(1), `# noqa: F841` in Ruff, `# shellcheck disable=SC2059` in ShellCheck und `// eslint-disable-next-line` in ESLint. Jeder hält fest, was stummgeschaltet wurde, nicht warum. Verlangen Sie neben jeder Ausnahme eine schriftliche Begründung. Schalten Sie bei den wichtigen Fällen die Regel aus und ergänzen Sie einen Test für die geschützte Eigenschaft. Auch Ausnahmen an einer Stelle zu bündeln hilft: ESLints `--no-inline-config` macht Abschaltkommentare wirkungslos, und ShellCheck liest projektweite `disable`-Zeilen aus einer Datei `.shellcheckrc`.

Weiterführende Lektüre

- lint(1) — NetBSDs Handbuch des C-Prüfprogramms, von dem der Begriff stammt: seine zwei Durchgänge, `-w` und seine Unterdrückungskommentare.
- Der Ruff-Linter — Regelauswahl, Sicherheit von Korrekturen, Unterdrückungskommentare und Exitcodes.
- Referenz der Kommandozeilenschnittstelle — ESLints `--max-warnings`, Konfiguration direkt im Code, Muster ohne Treffer und Exitcodes.
- ShellCheck — die README des Projekts: wonach es sucht und wie es in einen Build passt.
- Ignorieren — ShellChecks Möglichkeiten, einen Befund je Zeile, je Lauf oder für das ganze Projekt zu ignorieren.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist ein Linter?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/linter/