Protected Branches auf GitHub – Dein Sicherheitsnetz für kritische Branches

Protected Branches sind eines der wichtigsten Features, um die Integrität deines Codes zu schützen. Sie verhindern, dass versehentlich (oder absichtlich) schädliche Änderungen direkt in wichtige Branches wie `main` oder `develop` gelangen. Stell dir Protected Branches als **Türsteher** vor, die genau prüfen, wer unter welchen Bedingungen Änderungen einbringen darf.

Was sind Protected Branches? □□

Ein Protected Branch ist ein Branch, für den du auf GitHub spezielle Regeln definierst, die eingehalten werden müssen, bevor Änderungen akzeptiert werden. Ohne Schutzregeln kann *jeder* mit Schreibrechten direkt auf `main` pushen – ein einziger falscher Befehl wie `git push --force origin main` könnte die gesamte Projekthistorie zerstören.

Mit Protected Branches kannst du unter anderem festlegen:

- **Wer** überhaupt auf den Branch pushen darf
- **Ob** direkte Pushes erlaubt sind oder nur über Pull Requests
- **Welche Prüfungen** (Tests, Reviews) bestanden sein müssen
- **Ob** die Branch-Historie überschrieben werden darf (Force Push)

Warum sind sie so wichtig?

Szenario ohne Schutz	Mögliche Konsequenz
Entwickler pusht ungetesteten Code direkt auf `main`	Produktionssystem fällt aus
Jemand führt versehentlich `git push --force` aus	Commit-Historie geht verloren

Szenario ohne Schutz	Mögliche Konsequenz
Merge ohne Code Review	Bugs und Sicherheitslücken gelangen unbemerkt in die Produktion
Commits von unverifizierten Accounts	Supply-Chain-Angriffe werden möglich

---

Die wichtigsten Schutzregeln im Detail

GitHub bietet eine Vielzahl von Schutzregeln, die du kombinieren kannst. Hier sind die wichtigsten mit Erklärungen und Empfehlungen:

1. „Require a pull request before merging"

Diese Regel ist das **Herzstück** der Branch Protection. Sie verhindert, dass irgendjemand direkt auf den geschützten Branch pushen kann. Alle Änderungen *müssen* über einen Pull Request laufen.

Unteroptionen:

- **„Require approvals"** – Legt fest, wie viele Personen den PR genehmigen müssen, bevor er gemerged werden kann. Du kannst 1 bis 6 erforderliche Approvals einstellen.
  Empfehlung: Für kleine Teams (1–3 Entwickler) reicht **1 Approval**. Für größere Teams oder kritische Projekte empfehle ich **2 Approvals**.
- **„Dismiss stale pull request approvals when new commits are pushed"** – Wenn diese Option aktiviert ist, werden bestehende Approvals *ungültig*, sobald neue Commits zum PR hinzugefügt werden. Das ist wichtig, weil ein Reviewer vielleicht Code genehmigt hat, der danach noch verändert wurde.
  Empfehlung: **Unbedingt aktivieren!** Sonst könnte jemand nach dem Approval noch problematischen Code hinzufügen.
- **„Require review from Code Owners"** – Wenn dein Repository eine `CODEOWNERS`-Datei hat, müssen die dort definierten Verantwortlichen den PR genehmigen. Mehr dazu später.
- **„Require approval of the most recent reviewable push"** – Verhindert, dass der Autor des letzten Commits seinen eigenen Code genehmigt (relevant, wenn Maintainer selbst zum PR beitragen).

2. „Require status checks to pass before merging"

Diese Regel stellt sicher, dass automatisierte Prüfungen (wie Tests oder Linting) erfolgreich durchlaufen müssen, bevor ein Merge möglich ist. Das ist dein **Qualitätstor**.

Unteroptionen:

- **„Require branches to be up to date before merging"** – Der Feature-Branch muss auf dem aktuellen Stand des Ziel-Branches sein. Das verhindert, dass Code gemerged wird, der zwar mit einer *alten* Version von `main` funktioniert, aber mit den neuesten Änderungen kollidieren könnte.
  Empfehlung: **Aktivieren**, auch wenn es manchmal nervig ist, den Branch aktualisieren zu müssen. Es verhindert das „aber auf meinem Branch hat es funktioniert"-Problem.
- **Status Checks auswählen** – Du wählst aus, welche Checks bestanden sein müssen. Diese erscheinen erst in der Liste, nachdem sie mindestens einmal gelaufen sind (z.B. durch einen GitHub Actions Workflow).

```
# Beispiel: Workflow, der als Required Check verwendet werden kann
name: Tests
on: [push, pull_request]
jobs:
  phpunit:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Run PHPUnit
        run: vendor/bin/phpunit
```

3. „Require conversation resolution before merging"

Wenn Reviewer Kommentare oder Änderungswünsche hinterlassen, müssen diese als „resolved" markiert werden, bevor der PR gemerged werden kann. Das stellt sicher, dass **kein Feedback ignoriert** wird.

Empfehlung: **Aktivieren.** Es ist frustrierend als Reviewer, wenn deine Anmerkungen einfach übergangen werden.

4. „Require signed commits"

Mit dieser Regel müssen alle Commits kryptografisch signiert sein (GPG, SSH oder S/MIME). Signierte Commits zeigen auf GitHub ein grünes „Verified"-Badge und beweisen, dass der Commit wirklich von der angegebenen Person stammt.

Empfehlung: Für Open-Source-Projekte oder sicherheitskritische Anwendungen **empfohlen**. Für kleine, interne Projekte optional. Die Einrichtung erfordert etwas Aufwand (siehe Kapitel 10 des Kurses).

5. „Require linear history"

Diese Regel erzwingt eine lineare Commit-Historie, indem sie *nur* Squash-Merges oder Rebase-Merges erlaubt – keine regulären Merge-Commits mit zwei Parents.

Empfehlung: **Geschmackssache.** Eine lineare Historie ist übersichtlicher, aber manche Teams bevorzugen explizite Merge-Commits, weil sie zeigen, wann welcher Feature-Branch integriert wurde. Ich empfehle es für Projekte, die eine saubere, leicht lesbare Historie priorisieren.

6. „Do not allow bypassing the above settings"

Normalerweise können Repository-Administratoren alle Schutzregeln umgehen. Mit dieser Option wird **auch Admins** das Umgehen verboten.

Empfehlung: Für die meisten Projekte **nicht aktivieren**, da Admins manchmal legitime Gründe haben, Regeln zu umgehen (z.B. bei Notfall-Hotfixes). Bei sehr kritischen Projekten oder aus Compliance-Gründen kann es sinnvoll sein.

7. „Restrict who can push to matching branches"

Du kannst explizit definieren, welche Personen, Teams oder Apps überhaupt auf den Branch pushen dürfen (selbst über PRs). Das ist nützlich, wenn du z.B. nur bestimmten Maintainern erlauben möchtest, PRs zu mergen.

8. „Allow force pushes" und „Allow deletions"

Diese Optionen sind standardmäßig **deaktiviert** bei Protected Branches – und das sollte auch so bleiben!

- **Force Pushes** können die gesamte Commit-Historie überschreiben

- **Deletions** würden erlauben, den Branch komplett zu löschen

Empfehlung: **Niemals aktivieren** für `main` oder andere kritische Branches.

Schritt-für-Schritt-Anleitung: Branch Protection einrichten

So richtest du die Schutzregeln für deinen `main`-Branch ein:

1. **Navigiere zu den Repository-Einstellungen**
   Öffne dein Repository auf GitHub und klicke auf **„Settings"** (Zahnrad-Symbol) in der oberen Navigationsleiste.
2. **Öffne die Branch-Einstellungen**
   In der linken Seitenleiste findest du unter „Code and automation" den Punkt **„Branches"**. Klicke darauf.
3. **Füge eine Branch Protection Rule hinzu**
   Unter „Branch protection rules" klickst du auf **„Add branch protection rule"** (oder „Add rule" bei neueren Rulesets).
4. **Definiere das Branch-Muster**
   Im Feld „Branch name pattern" gibst du den Namen des zu schützenden Branches ein. Du kannst:
   - Einen exakten Namen eingeben: `main`
   - Wildcards verwenden: `release/*` schützt alle Branches, die mit „release/" beginnen
5. **Wähle deine Schutzregeln**
   Aktiviere die gewünschten Optionen (siehe detaillierte Beschreibungen oben).
6. **Speichere die Regel**
   Scrolle nach unten und klicke auf **„Create"** oder **„Save changes"**.

Empfohlene Konfiguration für verschiedene Szenarien

□□□ Solo-Entwickler (persönliches Projekt)

Auch als Solo-Entwickler können Protected Branches sinnvoll sein – sie schützen dich vor dir selbst!

Regel	Empfehlung
Require pull request	☐ Optional (kann Workflow verlangsamen)
Require status checks	☐ Aktivieren, wenn du CI/CD hast
Require signed commits	☐ Optional
Allow force pushes	☐ Deaktiviert lassen
Allow deletions	☐ Deaktiviert lassen

Als Minimalkonfiguration empfehle ich: **Keine Force Pushes, keine Deletions, Status Checks falls vorhanden.**

□□ Kleines Team (2–5 Entwickler)

Regel	Empfehlung
Require pull request	☐ Aktivieren
Require approvals	☐ 1 Approval
Dismiss stale approvals	☐ Aktivieren
Require status checks	☐ Aktivieren
Require branch up to date	☐ Aktivieren
Require conversation resolution	☐ Aktivieren
Allow force pushes	☐ Deaktiviert
Allow deletions	☐ Deaktiviert

□□ Größeres Team oder Open-Source-Projekt

Regel	Empfehlung
Require pull request	☐ Aktivieren
Require approvals	☐ 2 Approvals
Dismiss stale approvals	☐ Aktivieren
Require review from Code Owners	☐ Aktivieren
Require status checks	☐ Aktivieren (mehrere Checks)
Require branch up to date	☐ Aktivieren
Require conversation resolution	☐ Aktivieren

Regel	Empfehlung
Require signed commits	□ Aktivieren
Require linear history	⚠ Team-Entscheidung
Do not allow bypassing	⚠ Bei hohen Compliance-Anforderungen

CODEOWNERS – Automatische Review-Zuweisung □□

Die `CODEOWNERS`-Datei ist ein mächtiges Feature, das perfekt mit Protected Branches zusammenarbeitet. Du definierst darin, wer für welche Teile des Codes verantwortlich ist. Diese Personen werden automatisch als Reviewer zu PRs hinzugefügt, die „ihre" Dateien betreffen.

So erstellst du eine CODEOWNERS-Datei

Die Datei muss an einem dieser Orte liegen:

- Repository-Root: `CODEOWNERS`
- `.github/CODEOWNERS` *(empfohlen)*
- `docs/CODEOWNERS`

Syntax und Beispiele

```
# Dies ist ein Kommentar

# Standard-Owner für alles, was nicht anders definiert ist
* @standard-reviewer

# Bestimmte Dateien/Ordner einem Team zuweisen
/src/api/           @backend-team
/src/frontend/      @frontend-team

# Bestimmte Dateitypen
*.js                @javascript-expert
*.css               @design-team
```

```
# Kritische Konfigurationsdateien
/config/            @team-lead @senior-developer
.github/workflows/  @devops-team

# Bestimmte Dateien mehreren Reviewern zuweisen (alle müssen reviewen)
/security/          @security-team @team-lead
```

CODEOWNERS in Kombination mit Branch Protection

Wenn du in den Branch Protection Rules **„Require review from Code Owners"** aktivierst, *müssen* die definierten Code Owners den PR genehmigen. Das stellt sicher, dass z.B.:

- Änderungen an der Datenbank-Schicht immer vom DBA geprüft werden
- Sicherheitskritischer Code vom Security-Team abgesegnet wird
- Frontend-Änderungen vom Frontend-Lead reviewed werden

Rulesets – Die moderne Alternative □□

GitHub hat 2023 **Rulesets** eingeführt, eine modernere und flexiblere Alternative zu den klassischen Branch Protection Rules. Rulesets bieten einige Vorteile:

- **Auf Organisations-Ebene definierbar** – Eine Regel für alle Repositories
- **Bessere Wildcards** – Komplexere Muster möglich
- **Tag Protection** – Nicht nur Branches, auch Tags schützen
- **Bypass-Listen** – Feingranulare Kontrolle, wer Regeln umgehen darf

Rulesets vs. klassische Branch Protection

Feature	Branch Protection	Rulesets
Repository-spezifisch	□	□
Organisations-weit	□	□

Feature	Branch Protection	Rulesets
Tag Protection	❌	✅
Bypass-Ausnahmen	Eingeschränkt	Flexibel
API-Steuerung	❌	✅

Empfehlung: Für einzelne Repositories sind die klassischen Branch Protection Rules einfacher. Für Organisationen mit vielen Repositories lohnt es sich, Rulesets zu evaluieren.

Visualisierung: Typischer PR-Workflow mit Branch Protection

```
flowchart TD
    A["Entwickler erstellt\nFeature-Branch"] --> B["Entwickler pusht\nCommits"]
    B --> C["Pull Request\nwird erstellt"]
    C --> D{"Status Checks\nerfolgreich?"}
    D -->|Nein| E["Entwickler fixt\nProbleme"]
    E --> B
    D -->|Ja| F{"Code Review\nbestanden?"}
    F -->|Nein| G["Entwickler arbeitet\nFeedback ein"]
    G --> B
    F -->|Ja| H{"Branch\naktuell?"}
    H -->|Nein| I["Branch mit main\naktualisieren"]
    I --> D
    H -->|Ja| J{"Alle Conversations\nresolved?"}
    J -->|Nein| K["Offene Diskussionen\nklaeren"]
    K --> J
    J -->|Ja| L["Merge in main\nmoeglich"]
    L --> M["Feature-Branch\nloeschen"]

    style A fill:#e1f5fe,color:#000000
    style L fill:#c8e6c9,color:#000000
    style M fill:#c8e6c9,color:#000000
    style E fill:#ffcdd2,color:#000000
    style G fill:#ffcdd2,color:#000000
```

Häufige Fragen und Probleme □□

„Ich bin Admin, kann aber nicht pushen – warum?"

Wenn **„Do not allow bypassing the above settings"** aktiviert ist, gelten die Regeln auch für dich. Du musst dann ebenfalls einen PR erstellen.

„Mein Status Check taucht nicht in der Auswahlliste auf"

Status Checks erscheinen erst, nachdem sie mindestens einmal gelaufen sind. Erstelle einen Test-PR oder pushe auf einen Testbranch, um den Workflow auszulösen.

„Wie kann ich in einem Notfall trotzdem direkt pushen?"

Wenn du Admin bist und **„Do not allow bypassing"** *nicht* aktiviert ist, kannst du die Regeln umgehen. Alternativ:

1. Die Regel temporär deaktivieren
2. Den Notfall-Push machen
3. Die Regel sofort wieder aktivieren
4. **Dokumentieren**, warum das nötig war

Besser: Auch Notfall-Hotfixes über PRs, aber mit minimaler Review-Zeit und einem speziellen `hotfix`-Label.

„Wie schütze ich mehrere Branches gleichzeitig?"

Nutze Wildcards im Branch-Pattern:

- `release/*` – Schützt alle Release-Branches

- `main` und `develop` – Erstelle zwei separate Regeln
- Mit Rulesets kannst du auch `main` und `develop` in einer Regel kombinieren

---

Checkliste für deine Branch Protection □□

Nutze diese Checkliste, um deine `main`-Branch-Protection einzurichten:

- ☐ Branch Protection Rule für `main` erstellt
- ☐ „Require pull request before merging" aktiviert
- ☐ Anzahl der erforderlichen Approvals festgelegt
- ☐ „Dismiss stale approvals" aktiviert
- ☐ Status Checks definiert (Tests, Linting)
- ☐ „Require branch to be up to date" aktiviert
- ☐ „Require conversation resolution" aktiviert
- ☐ Force Pushes deaktiviert (Standard)
- ☐ Branch-Deletion deaktiviert (Standard)
- ☐ CODEOWNERS-Datei erstellt (optional)
- ☐ Team über die neuen Regeln informiert

Fazit

Protected Branches sind **kein bürokratisches Hindernis**, sondern ein essentielles Werkzeug für professionelle Softwareentwicklung. Sie schützen nicht nur vor böswilligen Änderungen, sondern vor allem vor Versehen und menschlichen Fehlern. Die initiale Einrichtung dauert nur wenige Minuten, kann aber stundenlange Debugging-Sessions oder sogar Datenverlust verhindern.

Selbst als Solo-Entwickler empfehle ich dir, zumindest Force Pushes und Deletions für `main` zu verbieten. Sobald du im Team arbeitest, sollten Pull Requests mit mindestens einem Review zur Pflicht werden. Die Zeit, die du in Reviews „verlierst", gewinnst du mehrfach zurück durch früh entdeckte Bugs und bessere Code-Qualität.

Revision #1
Created 2026-06-29 15:37:27 UTC by art10m
Updated 2026-06-29 15:37:45 UTC by art10m