Entwickler Themen

Dieser Bereich enthält Anleitungen und Dokumentation für verschiedenste relevante Themen aus dem Alltag als Entwickler.

Conventional Commits anwenden

Disclaimer

Die **Conventional Commits** sind ein Standard zur einheitlichen Benennung von Commit-Nachrichten. Sie helfen Teams, Versionsänderungen besser nachzuvollziehen, automatisch Changelogs zu generieren und semantische Versionierung zu automatisieren.

Weitere Informationen: conventionalcommits.org

Die Vorteile

1. Erleichtert die automatische Generierung von Changelogs
2. Unterstützt die semantische Versionierung vom Projekt
3. Fördert ein einheitliches und verständliches Commit-Format im Team
4. Verbessert die Zusammenarbeit zwischen Entwicklern, Release-Automatisierungstools und CI/CD-Systemen

---

Anwendungsformat

Ein konformer Commit folgt diesem Format:

```
Typ(optionaler Scope): Beschreibung
```

Beispiel:

```
feat(auth): added login functionality
```

Commit-Typen

- **feat**: Fügt eine neue Funktion hinzu (führt zu einem `minor`-Version-Sprung bei semantischer Versionierung)
- **fix**: Behebt einen Bug (führt zu einem `patch`-Version-Sprung)

- **docs**: Änderungen an der Dokumentation (z. B. README, Kommentare)
- **style**: Formatierungsänderungen (keine Code-Logik-Änderungen, z. B. Einrückung, Leerzeichen, Semikolon)
- **refactor**: Codeänderungen ohne Fehlerbehebung oder neue Funktion (z. B. Umstrukturierung)
- **perf**: Leistungsverbesserungen (Performance-Optimierungen)
- **test**: Hinzufügen oder Anpassen von Tests
- **chore**: Wartungsaufgaben, die nichts mit dem Quellcode oder Tests zu tun haben (z. B. Build-Prozess, Abhängigkeiten)

Optional: Scope

Der Scope beschreibt, auf welchen Teil der Codebasis sich der Commit bezieht. Er steht in Klammern direkt hinter dem Typ.

Beispiel:

```
fix(api): resolved error when fetching user data
```

Breaking Changes

Für Änderungen, die inkompatibel zur bisherigen API sind, wird `BREAKING CHANGE:` in der Fußzeile des Commits verwendet.

```
feat(auth): added two-factor authentication
BREAKING CHANGE: login process has been restructured
```

Semantische Versionierung anwenden

Disclaimer

Die **semantische Versionierung** ist ein gängiger Standard in der Softwareentwicklung, um Versionsnummern zu vergeben. Sie macht Änderungen im Code nachvollziehbar, erleichtert den Umgang mit Abhängigkeiten und verbessert die Kommunikation über den Entwicklungsstand.

Weitere Informationen: semver.org

Was ist die semantische Versionierung?

Die semantische Versionierung beschreibt ein dreiteiliges Versionsschema:

```
MAJOR.MINOR.PATCH
```

- **MAJOR**: Inkompatible Änderungen der öffentlichen API
- **MINOR**: Neue, abwärtskompatible Funktionalitäten
- **PATCH**: Abwärtskompatible Fehlerbehebungen

Beispiele aus der Praxis

- `1.2.3`: Erste stabile Version, zwei kleinere Feature-Erweiterungen, drei Bugfixes
- `2.0.0`: Führt zu inkompatiblen API-Änderungen, z. B. durch Entfernen oder Umbenennen von Funktionen
- `1.3.0`: Fügt neue Funktionalitäten hinzu, ohne vorhandene zu verändern
- `1.3.1`: Behebt einen Fehler aus der Version 1.3.0

Regeln der semantischen Versionierung

1. Erhöhe die **MAJOR**-Version bei inkompatiblen API-Änderungen
2. Erhöhe die **MINOR**-Version bei neuen, abwärtskompatiblen Funktionen
3. Erhöhe die **PATCH**-Version bei abwärtskompatiblen Bugfixes

---

Vorabversionen und Build-Metadaten

Zusätzlich können Vorabversionen und Build-Metadaten angegeben werden:

```
1.0.0-alpha
1.0.0-beta+exp.sha.5114f85
```

- **Vorabversionen:** `-alpha`, `-beta`, `-rc` usw. – für nicht stabile Entwicklungsstände
- **Build-Metadaten:** `+build` – zusätzliche Informationen wie Git-Hashes oder Build-Zeitpunkte

---

Zusammenfassung

- Die semantische Versionierung gibt eine klare Struktur für Versionen vor
- Sie erleichtert das Management von Abhängigkeiten in Projekten
- Sie wird häufig in Kombination mit Conventional Commits verwendet