Engineer

Semantisk versionering

versionsnumre, der viser, om en opdatering er kompatibel

Hvad det er

Semantisk versionering, på engelsk semantic versioning og som regel forkortet SemVer, er et sæt regler for, hvordan udgivelser nummereres, så selve nummeret fortæller, hvilken slags ændring en udgivelse indeholder. Specifikationen, der nu er i version 2.0.0 og oprindeligt blev skrevet af Tom Preston-Werner, bruger tre tal på formen MAJOR.MINOR.PATCH. En udgivelse, der bryder det offentlige API, hæver major-tallet. En udgivelse, der tilføjer funktionalitet på en bagudkompatibel måde, hæver minor-tallet, og en, der kun retter fejl uden at røre ved API'et, hæver patch-tallet. Når et tal stiger, sættes tallene til højre for det tilbage til nul, og hvert tal sammenlignes som et tal og ikke som tekst, så `1.10.0` kommer efter `1.9.0`.

Det hele hviler på ét krav: Projektet skal erklære et offentligt API, i koden eller i dokumentationen, for tallene beskriver ændringer i netop det API og intet andet. Når en version først er udgivet, må dens indhold ikke ændres; en rettelse er en ny version. Major-version nul, `0.y.z`, er til den indledende udvikling, hvor alt kan ændre sig når som helst, og hvor API'et ikke bør betragtes som stabilt. Version `1.0.0` er det punkt, hvor det offentlige API bliver defineret, og derfra følger tallene, hvordan det ændrer sig.

To valgfrie tilføjelser udvider formatet. En pre-release skrives efter en bindestreg og sorteres før den udgivelse, den fører frem til, så `1.0.0-alpha` kommer før `1.0.0-alpha.1`, der igen kommer før `1.0.0-beta`, `1.0.0-rc.1` og til sidst `1.0.0`. Build-metadata skrives efter et plustegn, som i `1.0.0+20130313144700`, og ignoreres, når versioner sættes i rækkefølge, så to versioner, der kun adskiller sig dér, har samme forrang. Et foranstillet v, som i Git-tagget `v1.2.3`, er ikke en del af den semantiske version; specifikationen kalder det en almindelig måde at vise på, at der er tale om et versionsnummer. Go gør det til et krav: En kanonisk version af et Go-modul er bogstavet v efterfulgt af en semantisk version.

Brug begrebet i en beslutning

Brug »Hvad er semantisk versionering?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Testet, udgivet og til at rulle tilbage, så ændringer når kunderne en tirsdag eftermiddag i stedet for sent en fredag aften.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har automatiseret GIS-SaaS-udrulning med CI/CD — udrulning, data og rapportering, alt sammen uden manuelt arbejde

Har automatiseret GIS-SaaS-udrulning med CI/CD Kontakt os →

Hvorfor det betyder noget

Specifikationen tager udgangspunkt i det, den kalder afhængighedshelvede. Er afhængighederne angivet for stramt, ender man i versionslåsning, hvor man ikke kan opgradere én pakke uden at udgive nye versioner af alt, der afhænger af den. Er de angivet for løst, ender man i versionspromiskuitet og antager kompatibilitet med flere fremtidige versioner, end der er rimeligt. Et versionsnummer med en aftalt betydning gør det muligt for et projekt at sige, hvad det vil acceptere: Et bibliotek, der bruger en funktion, som først kom med i 3.1.0, kan acceptere alt fra 3.1.0 og op til, men ikke med, 4.0.0 og få senere rettelser og tilføjelser med, uden at nogen skal redigere kravet.

Pakkehåndteringsværktøjerne handler på det løfte uden at spørge. I Cargo, Rusts pakkehåndtering, er et almindeligt krav som `"1.2.3"` et caret-krav og betyder mindst 1.2.3, men under 2.0.0. Når Cargo skriver `Cargo.lock`, vælger den den højeste version inden for det interval, og den bliver på 1.x-linjen, efter at 2.0.0 er udgivet, fordi den betragter 2.0.0 som inkompatibel. I node-semver, der præsenterer sig som npm's værktøj til semantiske versioner, betyder `^1.2.3` det samme som `>=1.2.3 <2.0.0-0`, og `~1.2.3` tillader kun ændringer på patch-niveau, `>=1.2.3 <1.3.0-0`. Go bygger reglen ind i importstierne: Fra major-version 2 og frem skal et moduls sti ende med et suffiks som `/v2`, så to inkompatible major-versioner er to forskellige importstier. Derefter tager Go med minimal version selection for hvert modul den højeste version, som et krav beder om, og det er den laveste version, der opfylder dem alle. Resultatet gemmes ikke i en låsefil; hver modulbevidst kommando beregner det igen.

Intet af dette gør nummeret til en garanti. Det er et løfte fra dem, der udgiver pakken, og det er aldrig bedre end deres vurdering af, hvad der tæller som et brud. Derfor udgiver Cargo en lang kompatibilitetsguide, der inddeler ændringer i major, minor og muligvis brydende. Det meste af guiden handler om ændringer, der får kode, som byggede i går, til ikke at bygge i dag. Guiden skriver selv, at

næsten enhver ændring indebærer en vis risiko for at påvirke opførslen under kørsel, og at det som regel er op til vedligeholdernes skøn, om sådan en ændring er inkompatibel.

Sådan ser det ud i praksis

Følg et opdigtet Rust-bibliotek, tally, gennem dets udgivelser. Version 0.4.2 er stadig indledende udvikling. Version 1.0.0 erklærer det offentlige API for stabilt. Version 1.0.1 retter en afrundingsfejl og er en patch. Version 1.1.0 tilføjer funktionen `parse_strict` og markerer den gamle `parse` som forældet: Specifikationen kræver en minor-udgivelse for at markere noget som forældet og beder om mindst én sådan udgivelse, før funktionen fjernes. Version 2.0.0 fjerner `parse` og er en major-udgivelse. Version 2.1.0-rc.1 er en pre-release af den næste minor-version.

Et program, der afhænger af tally og tre andre opdigtede crates, angiver sine krav i `Cargo.toml`. Kommentarerne viser det interval, Cargo læser ud af hver linje:

```
[dependencies]
tally = "1.0.1"         # >=1.0.1, <2.0.0: takes 1.1.0, never 2.0.0
ledger = "0.4.2"        # >=0.4.2, <0.5.0: 0.5.0 counts as breaking
stamp = "~1.2.3"        # >=1.2.3, <1.3.0: patch releases only
quill = "=2.1.0-rc.1"   # exactly this pre-release
```

Linjen med `ledger` viser en konvention i værktøjerne og ikke i specifikationen: Cargo betragter det første tal fra venstre, der ikke er nul, som det, der signalerer et brud, mens specifikationen siger, at alt under 1.0.0 kan ændre sig når som helst. Cargo udelader pre-releases fra et krav, medmindre kravet selv nævner en, så kravet `"2.0"` ville aldrig hente tallys release candidate. Den eksakte `=`-form på sidste linje er det tilfælde, hvor Cargos dokumentation siger, at et fuldt specificeret krav kan være berettiget: når man afhænger af en pre-release og vil undgå brydende ændringer mellem pre-releases. Når versionerne er fundet, registreres de i `Cargo.lock`, og `cargo update` flytter dem inden for de intervaller. Med `--locked` giver et build, der ville skulle ændre låsefilen, i stedet en fejl.

Det, nummeret ikke dækker, er alt uden for API'et. Licensgennemgangen af 644 Rust-crates, udført i september 2026, tjekker licensen på hver crate ved hvert build mod en tilladelsesliste. Grunden er, at en rutinemæssig versionsopdatering dybt nede i afhængighedsgrafen kan ændre, hvilke vilkår et program må distribueres under, uden at nogen opdager det, i en låsefil, der aldrig bliver læst i

sin helhed. For at bevise, at tjekket virker, blev det bevidst provokeret til at fejle to gange. Den ene gang blev en crates licensudtryk omskrevet, hvilket svarer til en crate, der ændrer sine vilkår upstream mellem versioner, og det fanger en menneskelig gennemgang ikke, fordi selve craten ikke er ny. Den anden gang blev den fastlåste kernerevision flyttet, uden at gennemgangen blev genereret igen. En semantisk version kan fortælle dig, at 1.4.0 er ment til at virke med kode, der er skrevet mod 1.0.0. Den kan ikke fortælle dig, at du stadig må sende den ud.

Hvor det går galt

Et brud i en minor-udgivelse. Det sker ved et uheld, eller fordi man ser forskelligt på, hvad der tæller som et brud. Cargos guide nævner ændringer, der ser harmløse ud, men ikke er det: At tilføje en variant til en offentlig enum bryder kode, der matcher udtømmende på den, medmindre enum'en var markeret med `#[non_exhaustive]` fra starten, og i sjældne tilfælde med glob-importer kan selv et nyt offentligt element give en kompileringsfejl. Specifikationens svar, når fejlen bliver opdaget, er at udgive en ny minor-version, der genopretter kompatibiliteten, aldrig at ændre den udgivelse, der allerede er ude, og at fortælle brugerne, hvilken version der var skyld i det. Hos dem, der bruger pakken, er det en låsefil, der ligger i repositoryet, og test, der kører ved hver opdatering, som fanger bruddet, før produktionen gør det.

At blive på 0.x i årevis. Ifølge specifikationen har major-version nul slet ikke et stabilt API. Dens FAQ siger det ligeud: Bruges softwaren allerede i produktion, bør den efter al sandsynlighed allerede være 1.0.0. Værktøjerne udfylder hullet med deres egen konvention: Cargo og node-semver betragter det første tal fra venstre, der ikke er nul, som det brydende, så `^0.2.3` accepterer kun flere 0.2.x-udgivelser, og `^0.0.3` accepterer slet ingen opdateringer. Et bibliotek, der har stået på 0.x i årevis, har brugere, der afhænger af det, men intet defineret løfte til dem. Udgiv 1.0.0, når folk er afhængige af API'et; det ændrer et tal, ikke koden.

At læse mere ind i nummeret, end det siger. Kompatibilitet efter specifikationen handler om det offentlige API. Cargos guide placerer det at hæve den mindste Rust-version, en crate kræver, under muligvis brydende og anbefaler generelt at behandle det som en minor-ændring, så en minor-opdatering kan stoppe buildet på en ældre toolchain. Sætter man `package.rust-version`, kan Cargos resolver forsøge med ældre versioner af craten, når det er

nødvendigt. Licensvilkår, ydeevne og opførsel, som API'et aldrig har dokumenteret, ligger helt uden for nummeret. Tjek dem for sig, med en licensgennemgang og test, i stedet for at udlede dem af en minor-udgivelse.

To major-versioner i ét build. Når to afhængigheder kræver inkompatible versioner af samme crate, for eksempel 0.6 og 0.7, bygger Cargo dem begge. Compileren betragter deres typer som forskellige, selv når navnene er ens. Det viser sig som regel som en kompileringsfejl, men nogle gange først under kørsel: Cargos dokumentation beskriver en downcast, der fejler, fordi objektet blev oprettet af et bibliotek, der bruger version 1.0.0, mens den kaldende kode downcaster til typen fra 2.0.0. `cargo tree -d` viser dublerede versioner, og hvor de kommer fra. En biblioteksforfatter kan gøre en major-udgivelse mindre smertefuld med det, Cargos dokumentation kalder »the semver trick«: Udgiv den nye major-version, og udgiv derefter en punktudgivelse af den gamle, der reeksporterer typerne fra den nye.

Læs mere

- Semantic Versioning 2.0.0 — specifikationen og dens FAQ, blandt andet om, hvad man gør, når man ved en fejl har udgivet en brydende ændring. Findes også i dansk oversættelse.
- SemVer Compatibility — Cargo Books guide til, hvilke ændringer i et Rust-bibliotek der er major, minor eller muligvis brydende, hver med et eksempel.
- Specifying Dependencies — Cargos syntaks for krav: standard, caret, tilde og sammenligninger, og hvordan pre-releases bliver matchet.
- Dependency Resolution — hvordan Cargo vælger versioner, hvad låsefilen indeholder, og farerne ved to inkompatible versioner i samme graf.
- node-semver — npm's værktøj til semantiske versioner med reglerne for caret, tilde og pre-releases i intervaller.
- Go Modules Reference — suffikser for major-versioner, reglen om importkompatibilitet og minimal version selection.

Ordliste

Vælg ud fra jeres faktiske vilkår

Medbring udrulningsforløbet, driftsrisiciene og dem, der har vagten. Vi kan afgrænse en ændring med kontroller, en vej tilbage og tydeligt ansvar.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er semantisk versionering?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/semantic-versioning/