Engineer

Semantisk versjonering

versjonsnumre som viser om en oppdatering er kompatibel

Hva det er

Semantisk versjonering, på engelsk semantic versioning og som regel forkortet SemVer, er et sett regler for hvordan utgivelser nummereres, slik at selve nummeret forteller hva slags endring en utgivelse inneholder. Spesifikasjonen, som nå er i versjon 2.0.0 og opprinnelig ble skrevet av Tom Preston-Werner, bruker tre tall på formen MAJOR.MINOR.PATCH. En utgivelse som bryter det offentlige API-et, øker major-tallet. En utgivelse som legger til funksjonalitet på en bakoverkompatibel måte, øker minor-tallet, og en som bare retter feil uten å røre API-et, øker patch-tallet. Når et tall øker, settes tallene til høyre for det tilbake til null, og hvert tall sammenlignes som et tall og ikke som tekst, så `1.10.0` kommer etter `1.9.0`.

Alt hviler på ett krav: prosjektet må erklære et offentlig API, i koden eller i dokumentasjonen, fordi tallene beskriver endringer i nettopp det API-et og ingenting annet. Når en versjon først er utgitt, skal innholdet ikke endres; en rettelse er en ny versjon. Major-versjon null, `0.y.z`, er for den innledende utviklingen, der alt kan endre seg når som helst, og der API-et ikke bør regnes som stabilt. Versjon `1.0.0` er punktet der det offentlige API-et blir definert, og derfra følger tallene hvordan det endrer seg.

To valgfrie merkelapper utvider formatet. En forhåndsutgivelse (pre-release) skrives etter en bindestrek og sorteres før utgivelsen den leder fram til, så `1.0.0-alpha` kommer før `1.0.0-alpha.1`, som igjen kommer før `1.0.0-beta`, `1.0.0-rc.1` og til slutt `1.0.0`. Byggmetadata skrives etter et plusstegn, som i `1.0.0+20130313144700`, og ignoreres når versjoner settes i rekkefølge, så to versjoner som bare skiller seg der, rangeres likt. En v foran, som i Git-taggen `v1.2.3`, er ikke en del av den semantiske versjonen; spesifikasjonen kaller det en vanlig måte å vise at det dreier seg om et versjonsnummer. Go gjør det til et krav: en kanonisk versjon av en Go-modul er bokstaven v fulgt av en semantisk versjon.

Bruk begrepet i en beslutning

Bruk «Hva er semantisk versjonering?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Testet, utgitt og mulig å reversere, slik at endringer når kundene en tirsdag ettermiddag i stedet for sent en fredagskveld.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Automatiserte GIS-SaaS-utrulling i GitHub Actions — produksjonssetting, data og rapportering, alt uten manuelle grep

Automatiserte GIS-SaaS-utrulling i GitHub Actions Ta kontakt →

Hvorfor det er viktig

Spesifikasjonen tar utgangspunkt i det den kaller avhengighetshelvete (dependency hell). Angir du avhengigheter for stramt, får du versjonslås (version lock), der oppgradering av én pakke krever nye versjoner av alt som er avhengig av den. Angir du dem for løst, får du det spesifikasjonen kaller version promiscuity: du antar kompatibilitet med flere framtidige versjoner enn det er rimelig. Et versjonsnummer med avtalt betydning lar et prosjekt si hva det godtar: et bibliotek som trenger en funksjon som først kom i 3.1.0, kan godta alt fra 3.1.0 og opp til, men ikke med, 4.0.0, og få senere rettelser og tillegg uten at noen redigerer kravet.

Pakkebehandlere handler ut fra dette løftet uten å spørre. I Cargo, pakkebehandleren for Rust, er et vanlig krav som `"1.2.3"` et caret-krav og betyr minst 1.2.3, men under 2.0.0. Når Cargo skriver `Cargo.lock`, velger det den høyeste versjonen innenfor det intervallet, og det blir værende på 1.x etter at 2.0.0 er publisert, fordi det regner 2.0.0 som inkompatibel. I node-semver, som kaller seg den semantiske versjonsbehandleren for npm, betyr `^1.2.3` det samme som `>=1.2.3 <2.0.0-0`, og `~1.2.3` tillater bare endringer på patch-nivå, `>=1.2.3 <1.3.0-0`. Go bygger regelen inn i importstiene: fra major-versjon 2 og oppover må stien til en modul slutte med et suffiks som `/v2`, så to inkompatible major-versjoner er to forskjellige importstier. Minimal version selection i Go tar deretter, for hver modul, den høyeste versjonen som noe krav ber om, og det er den laveste versjonen som oppfyller alle kravene. Resultatet lagres ikke i noen låsefil; hver kommando som arbeider med moduler, regner det ut på nytt.

Ingenting av dette gjør nummeret til en garanti. Det er et løfte fra dem som publiserer pakken, og det er aldri bedre enn deres vurdering av hva som regnes som brytende. Derfor publiserer Cargo en lang kompatibilitetsguide som sorterer endringer i major, minor og muligens brytende (possibly-breaking). Mesteparten av guiden handler om endringer som gjør at kode som kompilerte i

går, ikke kompilerer i dag. Guiden påpeker selv at nesten enhver endring innebærer en viss risiko for oppførselen under kjøring, og at om en slik endring er inkompatibel, som regel er et skjønnsspørsmål for vedlikeholderne.

Slik ser det ut i praksis

Følg et oppdiktet Rust-bibliotek, tally, gjennom utgivelsene. Versjon 0.4.2 er fortsatt innledende utvikling. Versjon 1.0.0 erklærer det offentlige API-et stabilt. Versjon 1.0.1 retter en avrundingsfeil og er en patch. Versjon 1.1.0 legger til en funksjon, `parse_strict`, og merker den gamle `parse` som foreldet (deprecated): spesifikasjonen krever en minor-utgivelse for det og ber om minst én slik utgivelse før funksjonen fjernes. Versjon 2.0.0 fjerner `parse` og er en major-utgivelse. Versjon 2.1.0-rc.1 er en forhåndsutgivelse av den neste minor-versjonen.

En applikasjon som er avhengig av tally og tre andre oppdiktede crates, angir kravene sine i `Cargo.toml`. Kommentarene viser intervallet Cargo leser ut av 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 for `ledger` viser en konvensjon i verktøyene og ikke i spesifikasjonen: for Cargo er det første tallet fra venstre som ikke er null, det som signaliserer brudd, mens spesifikasjonen sier at alt under 1.0.0 kan endre seg når som helst. Cargo holder forhåndsutgivelser utenfor et krav med mindre kravet nevner en, så et krav på `"2.0"` ville aldri plukke opp utgivelseskandidaten til tally. Den eksakte `=`-formen på siste linje er det tilfellet der Cargos dokumentasjon sier at et fullt spesifisert krav kan være berettiget: å være avhengig av en forhåndsutgivelse og samtidig unngå brytende endringer mellom forhåndsutgivelser. Når versjonene er valgt, registreres de i `Cargo.lock`, og `cargo update` flytter dem innenfor disse intervallene. Med `--locked` returnerer et bygg som måtte ha endret låsefilen, en feil i stedet.

Det nummeret ikke dekker, er alt utenfor API-et. Lisensrevisjonen av 644 Rust-crates, utført i september 2026, sjekker lisensen til hver crate ved hver bygging mot en liste over vilkår prosjektet godtar, fordi en rutinemessig

versjonsoppgradering dypt nede i avhengighetsgrafen i det stille kan endre hvilke vilkår en app kan distribueres under, i en låsefil ingen leser linje for linje. For å bevise at sjekken virker, ble den provosert til å feile med vilje, to ganger. Den ene gangen ble lisensuttrykket til en crate skrevet om, slik det ser ut når en crate endrer vilkårene sine oppstrøms mellom versjoner, noe ingen menneskelig gjennomgang fanger, fordi selve craten ikke er ny. Den andre gangen ble den låste kjernerevisjonen flyttet uten at lisenskontrollen ble generert på nytt. En semantisk versjon kan fortelle deg at 1.4.0 er ment å virke med kode skrevet mot 1.0.0. Den kan ikke fortelle deg om du fortsatt har lov til å levere den.

Hvor det går galt

En brytende endring i en minor-utgivelse. Det skjer ved et uhell eller fordi man ser forskjellig på hva som er brytende. Cargos guide lister opp endringer som ser harmløse ut, men ikke er det: å legge til en variant i en offentlig enum bryter kode som matcher uttømmende på den, med mindre enumen var merket `#[non_exhaustive]` fra starten, og i sjeldne tilfeller med glob-importer gir til og med et nytt offentlig element en kompileringsfeil. Spesifikasjonens svar, når feilen er oppdaget, er å gi ut en ny minor-versjon som gjenoppretter kompatibiliteten, aldri å endre utgivelsen som allerede er publisert, og å fortelle brukerne hvilken versjon som var feil. På mottakersiden er det en låsefil som er sjekket inn i repoet, og tester som kjører ved hver oppdatering, som fanger feilen før den når produksjon.

Å bli stående på 0.x i årevis. Etter spesifikasjonen har major-versjon null ikke noe stabilt API i det hele tatt. FAQ-en sier det rett ut: brukes programvaren allerede i produksjon, bør den etter all sannsynlighet allerede være 1.0.0. Verktøyene fyller tomrommet med sin egen konvensjon: Cargo og node-semver regner det første tallet fra venstre som ikke er null, som det brytende, så `^0.2.3` godtar bare flere 0.2.x-utgivelser, og `^0.0.3` godtar ingen oppdateringer i det hele tatt. Et bibliotek som har stått på 0.x i årevis, har brukere som er avhengige av det, men ikke noe definert løfte til dem. Gi ut 1.0.0 når folk først stoler på API-et; det endrer et nummer, ikke koden.

Å lese mer ut av nummeret enn det sier. Kompatibilitet etter spesifikasjonen handler om det offentlige API-et. Cargos guide klassifiserer det å heve den minste Rust-versjonen en crate trenger, som muligens brytende og anbefaler generelt å behandle det som en minor-endring, så en minor-oppdatering kan stoppe bygget på en eldre verktøykjede. Når `package.rust-version` er satt, kan Cargos avhengighetsoppløser prøve eldre versjoner av craten ved behov.

Lisensvilkår, ytelse og oppførsel som API-et aldri har dokumentert, ligger helt utenfor nummeret. Sjekk dem for seg, med en lisensrevisjon og tester, i stedet for å utlede dem av en minor-utgivelse.

To major-versjoner i ett bygg. Når to avhengigheter trenger inkompatible versjoner av samme crate, for eksempel 0.6 og 0.7, bygger Cargo begge. Kompilatoren behandler typene deres som forskjellige, selv når navnene er like. Som regel viser det seg som en kompileringsfeil, og noen ganger først under kjøring: Cargos dokumentasjon beskriver en downcast som feiler fordi objektet ble laget av et bibliotek som brukte versjon 1.0.0, mens koden som kaller, prøver å downcaste til typen fra 2.0.0. Kommandoen `cargo tree -d` viser dupliserte versjoner og hvor de kommer fra. En bibliotekforfatter kan dempe en major-utgivelse med det Cargos dokumentasjon kaller «the semver trick»: publiser den nye major-versjonen, og deretter en punktutgivelse av den gamle som re-eksporterer typene fra den nye.

Les mer

- Semantic Versioning 2.0.0 — spesifikasjonen og FAQ-en, blant annet om hva man gjør etter å ha gitt ut en brytende endring ved en feil.
- SemVer Compatibility — Cargo-bokens guide til hvilke endringer i et Rust-bibliotek som er major, minor eller muligens brytende, hver med et eksempel.
- Specifying Dependencies — Cargos kravsyntaks: standard-, caret-, tilde- og sammenligningskrav, og hvordan forhåndsutgivelser matches.
- Dependency Resolution — hvordan Cargo velger versjoner, hva låsefilen inneholder, og farene ved to inkompatible versjoner i én graf.
- node-semver — den semantiske versjonsbehandleren for npm, med reglene for caret, tilde og intervaller med forhåndsutgivelser.
- Go Modules Reference — suffikser for major-versjoner, regelen for importkompatibilitet og minimal version selection.

Ordliste

Velg ut fra deres faktiske rammer

Ta med utrullingsløpet, driftsrisikoene og dem som har vakt. Vi kan avgrense en endring med kontroller, en vei tilbake og tydelig ansvar.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er semantisk versjonering?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/semantic-versioning/