



Linux Ubuntu 26.04 Manual Pages on command 'semver.1'

\$ man semver.1

SEMVER(1) General Commands Manual SEMVER(1)

NAME

semver - The semantic versioner for npm

Install

npm install semver

Usage

As a node module:

```
const semver = require('semver')

semver.valid('1.2.3') // '1.2.3'

semver.valid('a.b.c') // null

semver.clean(' =v1.2.3 ') // '1.2.3'

semver.satisfies('1.2.3', '1.x || >=2.5.0 || 5.0.0 - 7.2.3') // true

semver.gt('1.2.3', '9.8.7') // false

semver.lt('1.2.3', '9.8.7') // true

semver.minVersion('>=1.0.0') // '1.0.0'

semver.valid(semver.coerce('v2')) // '2.0.0'

semver.valid(semver.coerce('42.6.7.9.3-alpha')) // '42.6.7'
```

You can also just load the module for the function that you care about if you'd like to minimize your footprint.

```
// load the whole API at once in a single object

const semver = require('semver')

// or just load the bits you need

// all of them listed here, just pick and choose what you want
```

```
// classes

const SemVer = require('semver/classes/semver')
const Comparator = require('semver/classes/comparator')
const Range = require('semver/classes/range')

// functions for working with versions

const semverParse = require('semver/functions/parse')
const semverValid = require('semver/functions/valid')
const semverClean = require('semver/functions/clean')
const semverInc = require('semver/functions/inc')
const semverDiff = require('semver/functions/diff')
const semverMajor = require('semver/functions/major')
const semverMinor = require('semver/functions/minor')
const semverPatch = require('semver/functions/patch')
const semverPrerelease = require('semver/functions/prerelease')
const semverCompare = require('semver/functions/compare')
const semverRcompare = require('semver/functions/rcompare')
const semverCompareLoose = require('semver/functions/compare-loose')
const semverCompareBuild = require('semver/functions/compare-build')
const semverSort = require('semver/functions/sort')
const semverRsort = require('semver/functions/rsort')

// low-level comparators between versions

const semverGt = require('semver/functions/gt')
const semverLt = require('semver/functions/lt')
const semverEq = require('semver/functions/eq')
const semverNeq = require('semver/functions/neq')
const semverGte = require('semver/functions/gte')
const semverLte = require('semver/functions/lte')
const semverCmp = require('semver/functions/cmp')
const semverCoerce = require('semver/functions/coerce')

// working with ranges

const semverSatisfies = require('semver/functions/satisfies')
const semverMaxSatisfying = require('semver/ranges/max-satisfying')
const semverMinSatisfying = require('semver/ranges/min-satisfying')
```

```
const semverToComparators = require('semver/ranges/to-comparators')
const semverMinVersion = require('semver/ranges/min-version')
const semverValidRange = require('semver/ranges/valid')
const semverOutside = require('semver/ranges/outside')
const semverGtr = require('semver/ranges/gtr')
const semverLtr = require('semver/ranges/ltr')
const semverIntersects = require('semver/ranges/intersects')
const semverSimplifyRange = require('semver/ranges/simplify')
const semverRangeSubset = require('semver/ranges/subset')
```

As a command-line utility:

```
$ semver -h
```

A JavaScript implementation of the <https://semver.org/> specification

Copyright Isaac Z. Schlueter

Usage: semver [options] <version> [<version> [...]]

Prints valid versions sorted by SemVer precedence

Options:

`-r --range <range>`

Print versions that match the specified range.

`-i --increment [<level>]`

Increment a version by the specified level. Level can be one of: major, minor, patch, premajor, preminor, prepatch, or prerelease. Default level is 'patch'.

Only one version may be specified.

`--preid <identifier>`

Identifier to be used to prefix premajor, preminor, prepatch or prerelease version increments.

`-l --loose`

Interpret versions and ranges loosely

`-n <0|1>`

This is the base to be used for the prerelease identifier.

`-p --include-prerelease`

Always include prerelease versions in range matching

`-c --coerce`

Coerce a string into SemVer if possible

(does not imply --loose)

--rtl

Coerce version strings right to left

--ltr

Coerce version strings left to right (default)

Program exits successfully if any valid version satisfies

all supplied ranges, and prints all satisfying versions.

If no satisfying versions are found, then exits failure.

Versions are printed in ascending order, so supplying

multiple versions to the utility will just sort them.

Versions

A "version" is described by the v2.0.0 specification found at

<https://semver.org/>

A leading "v" or "v" character is stripped off and ignored.

Ranges

A version range is a set of comparators that specify versions that satisfy the range.

A comparator is composed of an operator and a version. The set of primitive operators is:

? < Less than

? <= Less than or equal to

? > Greater than

? >= Greater than or equal to

? = Equal. If no operator is specified, then equality is assumed, so this operator is optional but MAY be included.

For example, the comparator >=1.2.7 would match the versions

1.2.7, 1.2.8, 2.5.3, and 1.3.9, but not the versions 1.2.6

or 1.1.0. The comparator >1 is equivalent to >=2.0.0 and

would match the versions 2.0.0 and 3.1.0, but not the versions

1.0.1 or 1.1.0.

Comparators can be joined by whitespace to form a comparator set,

which is satisfied by the intersection of all of the comparators

it includes.

A range is composed of one or more comparator sets, joined by `||`. A version matches a range if and only if every comparator in at least one of the `||`-separated comparator sets is satisfied by the version.

For example, the range `>=1.2.7 <1.3.0` would match the versions 1.2.7, 1.2.8, and 1.2.99, but not the versions 1.2.6, 1.3.0, or 1.1.0.

The range `1.2.7 || >=1.2.9 <2.0.0` would match the versions 1.2.7, 1.2.9, and 1.4.6, but not the versions 1.2.8 or 2.0.0.

Prerelease Tags

If a version has a prerelease tag (for example, 1.2.3-alpha.3) then it will only be allowed to satisfy comparator sets if at least one comparator with the same [major, minor, patch] tuple also has a prerelease tag.

For example, the range `>1.2.3-alpha.3` would be allowed to match the version 1.2.3-alpha.7, but it would not be satisfied by 3.4.5-alpha.9, even though 3.4.5-alpha.9 is technically "greater than" 1.2.3-alpha.3 according to the SemVer sort rules. The version range only accepts prerelease tags on the 1.2.3 version.

Version 3.4.5 would satisfy the range because it does not have a prerelease flag, and 3.4.5 is greater than 1.2.3-alpha.7.

The purpose of this behavior is twofold. First, prerelease versions frequently are updated very quickly, and contain many breaking changes that are (by the author's design) not yet fit for public consumption. Therefore, by default, they are excluded from range-matching semantics.

Second, a user who has opted into using a prerelease version has indicated the intent to use that specific set of alpha/beta/rc versions. By including a prerelease tag in the range, the user is indicating that they are aware of the risk. However, it is still not appropriate to assume that they have opted into taking a similar risk on the next set of prerelease versions.

Note that this behavior can be suppressed (treating all prerelease

versions as if they were normal versions, for range-matching)
by setting the includePrerelease flag on the options
object to any
functions that do
range matching.

Prerelease Identifiers

The method `.inc` takes an additional identifier string argument that
will append the value of the string as a prerelease identifier:

```
semver.inc('1.2.3', 'prerelease', 'beta')  
// '1.2.4-beta.0'
```

command-line example:

```
$ semver 1.2.3 -i prerelease --preid beta  
1.2.4-beta.0
```

Which then can be used to increment further:

```
$ semver 1.2.4-beta.0 -i prerelease  
1.2.4-beta.1
```

Prerelease Identifier Base

The method `.inc` takes an optional parameter `'identifierBase'` string
that will let you let your prerelease number as zero-based or one-based.

Set to false to omit the prerelease number altogether.

If you do not specify this parameter, it will default to zero-based.

```
semver.inc('1.2.3', 'prerelease', 'beta', '1')  
// '1.2.4-beta.1'  
  
semver.inc('1.2.3', 'prerelease', 'beta', false)  
// '1.2.4-beta'
```

command-line example:

```
$ semver 1.2.3 -i prerelease --preid beta -n 1  
1.2.4-beta.1  
  
$ semver 1.2.3 -i prerelease --preid beta -n false  
1.2.4-beta
```

Advanced Range Syntax

Advanced range syntax desugars to primitive comparators in
deterministic ways.

Advanced ranges may be combined in the same way as primitive comparators using white space or ||.

Hyphen Ranges X.Y.Z - A.B.C

Specifies an inclusive set.

? 1.2.3 - 2.3.4 := >=1.2.3 <=2.3.4

If a partial version is provided as the first version in the inclusive range, then the missing pieces are replaced with zeroes.

? 1.2 - 2.3.4 := >=1.2.0 <=2.3.4

If a partial version is provided as the second version in the inclusive range, then all versions that start with the supplied parts of the tuple are accepted, but nothing that would be greater than the provided tuple parts.

? 1.2.3 - 2.3 := >=1.2.3 <2.4.0-0

? 1.2.3 - 2 := >=1.2.3 <3.0.0-0

X-Ranges 1.2.x 1.X 1.2.* *

Any of X, x, or * may be used to "stand in" for one of the numeric values in the [major, minor, patch] tuple.

? * := >=0.0.0 (Any non-prerelease version satisfies, unless includePrerelease is specified, in which case any version at all satisfies)

? 1.x := >=1.0.0 <2.0.0-0 (Matching major version)

? 1.2.x := >=1.2.0 <1.3.0-0 (Matching major and minor versions)

A partial version range is treated as an X-Range, so the special character is in fact optional.

? "" (empty string) := * := >=0.0.0

? 1 := 1.x.x := >=1.0.0 <2.0.0-0

? 1.2 := 1.2.x := >=1.2.0 <1.3.0-0

Tilde Ranges ~1.2.3 ~1.2 ~1

Allows patch-level changes if a minor version is specified on the comparator. Allows minor-level changes if not.

? ~1.2.3 := >=1.2.3 <1.(2+1).0 := >=1.2.3 <1.3.0-0

? ~1.2 := >=1.2.0 <1.(2+1).0 := >=1.2.0 <1.3.0-0 (Same as 1.2.x)

? ~1 := >=1.0.0 <(1+1).0.0 := >=1.0.0 <2.0.0-0 (Same as 1.x)

? ~0.2.3 := >=0.2.3 <0.(2+1).0 := >=0.2.3 <0.3.0-0

? ~0.2 := >=0.2.0 <0.(2+1).0 := >=0.2.0 <0.3.0-0 (Same as 0.2.x)

? ~0 := >=0.0.0 <(0+1).0.0 := >=0.0.0 <1.0.0-0 (Same as 0.x)

? ~1.2.3-beta.2 := >=1.2.3-beta.2 <1.3.0-0 Note that prereleases in the 1.2.3 version will be allowed, if they are greater than or equal to beta.2. So, 1.2.3-beta.4 would be allowed, but 1.2.4-beta.2 would not, because it is a prerelease of a different [major, minor, patch] tuple.

Caret Ranges ^1.2.3 ^0.2.5 ^0.0.4

Allows changes that do not modify the left-most non-zero element in the [major, minor, patch] tuple. In other words, this allows patch and minor updates for versions 1.0.0 and above, patch updates for versions 0.X >=0.1.0, and no updates for versions 0.0.X.

Many authors treat a 0.x version as if the x were the major "breaking-change" indicator.

Caret ranges are ideal when an author may make breaking changes between 0.2.4 and 0.3.0 releases, which is a common practice.

However, it presumes that there will not be breaking changes between 0.2.4 and 0.2.5. It allows for changes that are presumed to be additive (but non-breaking), according to commonly observed practices.

? ^1.2.3 := >=1.2.3 <2.0.0-0

? ^0.2.3 := >=0.2.3 <0.3.0-0

? ^0.0.3 := >=0.0.3 <0.0.4-0

? ^1.2.3-beta.2 := >=1.2.3-beta.2 <2.0.0-0 Note that prereleases in the 1.2.3 version will be allowed, if they are greater than or equal to beta.2. So, 1.2.3-beta.4 would be allowed, but 1.2.4-beta.2 would not, because it is a prerelease of a different [major, minor, patch] tuple.

? ^0.0.3-beta := >=0.0.3-beta <0.0.4-0 Note that prereleases in the 0.0.3 version only will be allowed, if they are greater than or equal to beta. So, 0.0.3-pr.2 would be allowed.

When parsing caret ranges, a missing patch value desugars to the number 0, but will allow flexibility within that value, even if the

major and minor versions are both 0.

? ^1.2.x := >=1.2.0 <2.0.0-0

? ^0.0.x := >=0.0.0 <0.1.0-0

? ^0.0 := >=0.0.0 <0.1.0-0

A missing minor and patch values will desugar to zero, but also allow flexibility within those values, even if the major version is zero.

? ^1.x := >=1.0.0 <2.0.0-0

? ^0.x := >=0.0.0 <1.0.0-0

Range Grammar

Putting all this together, here is a Backus-Naur grammar for ranges, for the benefit of parser authors:

range-set ::= range (logical-or range) *

logical-or ::= (' ') * '|' (' ') *

range ::= hyphen | simple (' ' simple) * | "

hyphen ::= partial ' - ' partial

simple ::= primitive | partial | tilde | caret

primitive ::= ('<' | '>' | '>=' | '<=' | '=') partial

partial ::= xr (' ' xr (' ' xr qualifier ?) ?) ?

xr ::= 'x' | 'X' | '*' | nr

nr ::= '0' | ['1'-9'] (['0'-9']) *

tilde ::= '~' partial

caret ::= '^' partial

qualifier ::= ('-' pre) ? ('+' build) ?

pre ::= parts

build ::= parts

parts ::= part (' ' part) *

part ::= nr | [-0-9A-Za-z]+

Functions

All methods and classes take a final options object argument. All options in this object are false by default. The options supported are:

? loose: Be more forgiving about not-quite-valid semver strings.

(Any resulting output will always be 100% strict compliant, of course.) For backwards compatibility reasons, if the options argument is a boolean value instead of an object, it is interpreted to be the loose param.

? includePrerelease: Set to suppress the default behavior of excluding prerelease tagged versions from ranges unless they are explicitly opted into.

Strict-mode Comparators and Ranges will be strict about the SemVer strings that they parse.

? valid(v): Return the parsed version, or null if it's not valid.

? inc(v, release, options, identifier, identifierBase):

Return the version incremented by the release type (major, premajor, minor, preminor, patch, prepatch, or prerelease), or null if it's not valid

? premajor in one call will bump the version up to the next major version and down to a prerelease of that major version. preminor, and prepatch work the same way.

? If called from a non-prerelease version, prerelease will work the same as prepatch. It increments the patch version and then makes a prerelease. If the input version is already a prerelease it simply increments it.

? identifier can be used to prefix premajor, preminor, prepatch, or prerelease version increments. identifierBase is the base to be used for the prerelease identifier.

? prerelease(v): Returns an array of prerelease components, or null if none exist. Example: prerelease('1.2.3-alpha.1') -> ['alpha', 1]

? major(v): Return the major version number.

? minor(v): Return the minor version number.

? patch(v): Return the patch version number.

? intersects(r1, r2, loose): Return true if the two supplied ranges or comparators intersect.

? parse(v): Attempt to parse a string as a semantic version, returning either a SemVer object or null.

Comparison

? `gt(v1, v2)`: `v1 > v2`

? `gte(v1, v2)`: `v1 >= v2`

? `lt(v1, v2)`: `v1 < v2`

? `lte(v1, v2)`: `v1 <= v2`

? `eq(v1, v2)`: `v1 == v2` This is true if they're logically equivalent, even if they're not the same string. You already know how to compare strings.

? `neq(v1, v2)`: `v1 != v2` The opposite of `eq`.

? `cmp(v1, comparator, v2)`: Pass in a comparison string, and it'll call the corresponding function above. `"==="` and `"!=="` do simple string comparison, but are included for completeness. Throws if an invalid comparison string is provided.

? `compare(v1, v2)`: Return 0 if `v1 == v2`, or 1 if `v1` is greater, or -1 if `v2` is greater. Sorts in ascending order if passed to `Array.sort()`.

? `rcompare(v1, v2)`: The reverse of `compare`. Sorts an array of versions in descending order when passed to `Array.sort()`.

? `compareBuild(v1, v2)`: The same as `compare` but considers build when two versions are equal. Sorts in ascending order if passed to `Array.sort()`.

? `compareLoose(v1, v2)`: Short for `compare(v1, v2, { loose: true })`.

? `diff(v1, v2)`: Returns the difference between two versions by the release type (major, premajor, minor, preminor, patch, prepatch, or prerelease), or null if the versions are the same.

Sorting

? `sort(versions)`: Returns a sorted array of versions based on the `compareBuild` function.

? `rsort(versions)`: The reverse of `sort`. Returns an array of versions based on the `compareBuild` function in descending order.

Comparators

? `intersects(comparator)`: Return true if the comparators intersect

Ranges

? `validRange(range)`: Return the valid range or null if it's not valid

? `satisfies(version, range)`: Return true if the version satisfies the

range.

? `maxSatisfying(versions, range)`: Return the highest version in the list that satisfies the range, or null if none of them do.

? `minSatisfying(versions, range)`: Return the lowest version in the list that satisfies the range, or null if none of them do.

? `minVersion(range)`: Return the lowest version that can match the given range.

? `gtr(version, range)`: Return true if the version is greater than all the versions possible in the range.

? `ltr(version, range)`: Return true if the version is less than all the versions possible in the range.

? `outside(version, range, hilo)`: Return true if the version is outside the bounds of the range in either the high or low direction. The hilo argument must be either the string `'>'` or `'<'`. (This is the function called by `gtr` and `ltr`.)

? `intersects(range)`: Return true if any of the range comparators intersect.

? `simplifyRange(versions, range)`: Return a "simplified" range that matches the same items in the versions list as the range specified. Note that it does not guarantee that it would match the same versions in all cases, only for the set of versions provided. This is useful when generating ranges by joining together multiple versions with `||` programmatically, to provide the user with something a bit more ergonomic. If the provided range is shorter in string-length than the generated range, then that is returned.

? `subset(subRange, superRange)`: Return true if the subRange range is entirely contained by the superRange range.

Note that, since ranges may be non-contiguous, a version might not be greater than a range, less than a range, or satisfy a range! For example, the range `1.2 <1.2.9 || >2.0.0` would have a hole from 1.2.9 until 2.0.0, so version 1.2.10 would not be greater than the range (because 2.0.1 satisfies, which is higher), nor less than the range (since 1.2.8 satisfies, which is lower), and it also does not satisfy the range.

If you want to know if a version satisfies or does not satisfy a range, use the `satisfies(version, range)` function.

Coercion

? `coerce(version, options)`: Coerces a string to semver if possible

This aims to provide a very forgiving translation of a non-semver string to semver. It looks for the first digit in a string and consumes all remaining characters which satisfy at least a partial semver (e.g., 1, 1.2, 1.2.3) up to the max permitted length (256 characters). Longer versions are simply truncated (4.6.3.9.2-alpha2 becomes 4.6.3). All surrounding text is simply ignored (v3.4 replaces v3.3.1 becomes 3.4.0). Only text which lacks digits will fail coercion (version one is not valid). The maximum length for any semver component considered for coercion is 16 characters; longer components will be ignored (10000000000000000.4.7.4 becomes 4.7.4). The maximum value for any semver component is `Number.MAX_SAFE_INTEGER || (2**53 - 1)`; higher value components are invalid (9999999999999999.4.7.4 is likely invalid). If the `options.rtl` flag is set, then `coerce` will return the right-most coercible tuple that does not share an ending index with a longer coercible tuple. For example, 1.2.3.4 will return 2.3.4 in rtl mode, not 4.0.0. 1.2.3/4 will return 4.0.0, because the 4 is not a part of any other overlapping SemVer tuple. If the `options.includePrerelease` flag is set, then the `coerce` result will contain prerelease and build parts of a version. For example, 1.2.3.4-rc.1+rev.2 will preserve prerelease rc.1 and build rev.2 in the result.

Clean

? `clean(version)`: Clean a string to be a valid semver if possible

This will return a cleaned and trimmed semver version. If the provided version is not valid a null will be returned. This does not work for ranges.

ex.

? `s.clean('v 2.1.5foo')`: null

? `s.clean('v 2.1.5foo', { loose: true })`: '2.1.5-foo'

? `s.clean('v 2.1.5-foo')`: null

```
? s.clean(= v 2.1.5-foo, { loose: true }): 2.1.5-foo;
? s.clean(=v2.1.5): 2.1.5;
? s.clean(=v2.1.5): 2.1.5;
? s.clean( 2.1.5 ): 2.1.5;
? s.clean(~1.0.0): null
```

Constants

As a convenience, helper constants are exported to provide information about what node-semver supports:

RELEASE_TYPES

```
? major
? premajor
? minor
? preminor
? patch
? prepatch
? prerelease

const semver = require('semver');

if (semver.RELEASE_TYPES.includes(arbitraryUserInput)) {
  console.log('This is a valid release type!');
} else {
  console.warn('This is NOT a valid release type!');
}
```

SEMVER_SPEC_VERSION

```
2.0.0

const semver = require('semver');

console.log('We are currently using the semver specification version:', semver.SEMVER_SPEC_VERSION);
```

Exported Modules

<!-- TODO: Make sure that all of these items are documented (classes aren't, eg), and then pull the module name into the documentation for that specific thing. -->

You may pull in just the part of this semver utility that you need if you are sensitive to packing and tree-shaking concerns. The main `require('semver')` export uses getter functions to lazily load the parts of the API that are used.

The following modules are available:

- ? require('semver')
- ? require('semver/classes')
- ? require('semver/classes/comparator')
- ? require('semver/classes/range')
- ? require('semver/classes/semver')
- ? require('semver/functions/clean')
- ? require('semver/functions/cmp')
- ? require('semver/functions/coerce')
- ? require('semver/functions/compare')
- ? require('semver/functions/compare-build')
- ? require('semver/functions/compare-loose')
- ? require('semver/functions/diff')
- ? require('semver/functions/eq')
- ? require('semver/functions/gt')
- ? require('semver/functions/gte')
- ? require('semver/functions/inc')
- ? require('semver/functions/lt')
- ? require('semver/functions/lte')
- ? require('semver/functions/major')
- ? require('semver/functions/minor')
- ? require('semver/functions/neq')
- ? require('semver/functions/parse')
- ? require('semver/functions/patch')
- ? require('semver/functions/prerelease')
- ? require('semver/functions/rcompare')
- ? require('semver/functions/rsort')
- ? require('semver/functions/satisfies')
- ? require('semver/functions/sort')
- ? require('semver/functions/valid')
- ? require('semver/ranges/gtr')
- ? require('semver/ranges/intersects')
- ? require('semver/ranges/ltr')

? require('semver/ranges/max-satisfying')
? require('semver/ranges/min-satisfying')
? require('semver/ranges/min-version')
? require('semver/ranges/outside')
? require('semver/ranges/simplify')
? require('semver/ranges/subset')
? require('semver/ranges/to-comparators')
? require('semver/ranges/valid')

7.6.1

November 2024

SEMVER(1)