Writing Version Constraints

Now that you have an idea of how Composer sees versions, let's talk about how to specify version constraints for your project dependencies.

Exact Version Constraint

You can specify the exact version of a package. This will tell Composer to install this version and this version only. If other dependencies require a different version, the solver will ultimately fail and abort any install or update procedures.

Example: `1.0.2`

Version Range

By using comparison operators you can specify ranges of valid versions. Valid operators are `>`, `>=`, `<`, `<=`, `!=`.

You can define multiple ranges. Ranges separated by a space (` `) or comma (`,`) will be treated as a **logical AND**. A double pipe (`||`) will be treated as a **logical OR**. AND has higher precedence than OR.

> **Note:** Be careful when using unbounded ranges as you might end up unexpectedly installing versions that break backwards compatibility. Consider using the caret operator instead for safety.

Examples:

- `>=1.0`
- `>=1.0 <2.0`
- `>=1.0 <1.1 || >=1.2`

Hyphenated Version Range (-)

Inclusive set of versions. Partial versions on the right include are completed with a wildcard. For example `1.0 - 2.0` is equivalent to `>=1.0.0 <2.1` as the `2.0` becomes `2.0.*`. On the other hand `1.0.0 - 2.1.0` is equivalent to `>=1.0.0 <=2.1.0`.

Example: `1.0 - 2.0`

Wildcard Version Range (.*)

You can specify a pattern with a `*` wildcard. `1.0.*` is the equivalent of `>=1.0 <1.1`.

Example: `1.0.*`

Next Significant Release Operators

Tilde Version Range (~)

The `~` operator is best explained by example: `~1.2` is equivalent to `>=1.2 <2.0.0`, while `~1.2.3` is equivalent to `>=1.2.3 <1.3.0`. As you can see it is mostly useful for projects respecting semantic versioning. A common usage would be to mark the minimum minor version you depend on, like `~1.2` (which allows anything up to, but not including, 2.0). Since in theory there should be no backwards compatibility breaks until 2.0, that works well. Another way of looking at it is that using `~` specifies a minimum version, but allows the last digit specified to go up.

Example: `~1.2`

> **Note:** Although `2.0-beta.1` is strictly before `2.0`, a version constraint like `~1.2` would not install it. As said above `~1.2` only means the `.2` can change but the `1.` part is fixed.
> **Note:** The `~` operator has an exception on its behavior for the major release number. This means for example that `~1` is the same as `~1.0` as it will not allow the major number to increase trying to keep backwards compatibility.

Caret Version Range (^)

The `^` operator behaves very similarly but it sticks closer to semantic versioning, and will always allow non-breaking updates. For example `^1.2.3` is equivalent to `>=1.2.3 <2.0.0` as none of the releases until 2.0 should break backwards compatibility. For pre-1.0 versions it also acts with safety in mind and treats `^0.3` as `>=0.3.0 <0.4.0`.

This is the recommended operator for maximum interoperability when writing library code.

Example: `^1.2.3`

Stability Constraints

If you are using a constraint that does not explicitly define a stability, Composer will default internally to `-dev` or `-stable`, depending on the operator(s) used. This happens transparently.

If you wish to explicitly consider only the stable release in the comparison, add the suffix `-stable`.

Examples:

Constraint	Internally
`1.2.3`	`=1.2.3.0-stable`
`>1.2`	`>1.2.0.0-stable`
`>=1.2`	`>=1.2.0.0-dev`
`>=1.2-stable`	`>=1.2.0.0-stable`
`<1.3`	`<1.3.0.0-dev`
`<=1.3`	`<=1.3.0.0-stable`
`1 - 2`	`>=1.0.0.0-dev <3.0.0.0-dev`
`~1.3`	`>=1.3.0.0-dev <2.0.0.0-dev`
`1.4.*`	`>=1.4.0.0-dev <1.5.0.0-dev`

To allow various stabilities without enforcing them at the constraint level however, you may use stability-flags like `@<stability>` (e.g. `@dev`) to let composer know that a given package can be installed in a different stability than your default minimum-stability setting. All available stability flags are listed on the minimum-stability section of the schema page.

Summary

```
"require": {
    "vendor/package": "1.3.2", // exactly 1.3.2

    // >, <, >=, <= | specify upper / lower bounds
    "vendor/package": ">=1.3.2", // anything above or equal to 1.3.2
    "vendor/package": "<1.3.2", // anything below 1.3.2

    // * | wildcard
    "vendor/package": "1.3.*", // >=1.3.0 <1.4.0

    // ~ | allows last digit specified to go up
    "vendor/package": "~1.3.2", // >=1.3.2 <1.4.0
    "vendor/package": "~1.3", // >=1.3.0 <2.0.0

    // ^ | doesn't allow breaking changes (major version fixed - following semver)
    "vendor/package": "^1.3.2", // >=1.3.2 <2.0.0
    "vendor/package": "^0.3.2", // >=0.3.2 <0.4.0 // except if major version is 0
}
```

Testing Version Constraints

You can test version constraints using semver.mwl.be. Fill in a package name and it will autofill the default version constraint which Composer would add to your `composer.json` file. You can adjust the version constraint and the tool will highlight all releases that match.