Guides

Frequently Asked Questions

Why Can't I Just Specify Only `=` Dependencies?

Q: I understand the value of locking my gems down to specific versions, but why can't I just specify `=` versions for all my dependencies in the `Gemfile` and forget about the `Gemfile.lock`?

A: Many of your gems will have their own dependencies, and they are unlikely to specify `=` dependencies. Moreover, it is probably unwise for gems to lock down all of *their* dependencies so strictly. The `Gemfile.lock` allows you to specify the versions of the dependencies that your application needs in the `Gemfile`, while remembering all of the exact versions of third-party code that your application used when it last worked correctly.

By specifying looser dependencies in your `Gemfile` (such as `nokogiri ~> 1.4.2`), you gain the ability to run `bundle update nokogiri`, and let bundler handle updating **only** `nokogiri` and its dependencies to the latest version that still satisfied the `~> 1.4.2` version requirement. This also allows you to say "I want to use the current version of nokogiri" (`gem 'nokogiri'` in your `Gemfile`) without having to look up the exact version number, while still getting the benefits of ensuring that your application always runs with exactly the same versions of all third-party code.

Why Can't I Just Submodule Everything?

Q: I don't understand why I need bundler to manage my gems in this manner. Why can't I just get the gems I need and stick them in submodules, then put each of the submodules on the load path?

A: Unfortunately, that solution requires that you manually resolve all of the dependencies in your application, including dependencies of dependencies. And even once you do that successfully, you would need to redo that work if you wanted to update a particular gem. For instance, if you wanted to update the `rails` gem, you would need to find all of the gems that depended on dependencies of Rails (`rack`, `erubis`, `i18n`, `tzinfo`, etc.), and find new versions that satisfy the new versions of Rails' requirements.

Frankly, this is the sort of problem that computers are good at, and which you, a developer, should not need to spend time doing.

More concerningly, if you made a mistake in the manual dependency resolution process, you would not get any feedback about conflicts between different dependencies, resulting in subtle runtime errors. For instance, if you accidentally stuck the wrong version of `rack` in a submodule, it would likely break at runtime, when Rails or another dependency tried to rely on a method that was not present.

Bottom line: even though it might seem simpler at first glance, it is decidedly significantly more complex.

Why Is Bundler Downloading Gems From `--without` Groups?

Q: I ran `bundle install --without production` and bundler is still downloading the gems in the `:production` group. Why?

A: Bundler's `Gemfile.lock` has to contain exact versions of all dependencies in your `Gemfile`, regardless of any options you pass in. If it did not, deploying your application to production might change all your dependencies, eliminating the benefit of Bundler. You could no longer be sure that your application uses the same gems in production that you used to develop and test with. Additionally, adding a dependency in production might result in an application that is impossible to deploy.

For instance, imagine you have a production-only gem (let's call it `rack-debugging`) that depends on `rack =1.1`. If we did not evaluate the production group when you ran `bundle install --without production`, you would deploy your application, only to receive an error that `rack-debugging` conflicted with `rails` (which depends on `actionpack`, which depends on `rack ~> 1.2.1`).

Another example: imagine a simple Rack application that has `gem 'rack'` in the `Gemfile`. Again, imagine that you put `rack-debugging` in the `:production` group. If we did not evaluate the `:production` group when you installed via `bundle install --without production`, your app would use `rack 1.2.1` in development, and you would learn, at deployment time, that `rack-debugging` conflicts with the version of Rack that you tested with.

In contrast, by evaluating the gems in **all** groups when you call `bundle install`, regardless of the groups you actually want to use in that environment, we will discover the `rack-debugger` requirement, and install `rack 1.1`, which is also compatible with the `gem 'rack'` requirement in your `Gemfile`.

In short, by always evaluating all of the dependencies in your Gemfile, regardless of the dependencies you intend to use in a particular environment, you avoid nasty surprises when switching to a different set of groups in a different environment. And because we just download (but do not install) the gems, you won't have to worry about the possibility of a difficult **installation** process for a gem that you only use in production (or in development).

I Have a C Extension That Requires Special Flags to Install

Q: I have a C extension gem, such as `mysql`, which requires special flags in order to compile and install. How can I pass these flags into the installation process for those gems?

A: First of all, this problem does not exist for the `mysql2` gem, which is a drop-in replacement for the `mysql` gem. In general, modern C extensions properly discover the needed headers.

If you really need to pass flags to a C extension, you can use the `bundle config` command:

```
$ bundle config build.mysql --with-mysql-config=/usr/local/mysql/bin/mysql_config
```

Bundler will store this configuration in `~/.bundle/config`, and bundler will use the configuration for any `bundle install` performed by the same user. As a result, once you specify the necessary build flags for a gem, you can successfully install that gem as many times as necessary.

I Do Not Have an Internet Connection and Bundler Keeps Trying to Connect to the Gem Server

Q: I do not have an internet connection but I have installed the gem before. How do I get bundler to use my local gem cache and not connect to the gem server?

A: Use the `--local` flag with bundle install. The `--local` flag tells bundler to use the local gem cache instead of reaching out to the remote gem server.

```
$ bundle install --local
```

Bundling From RubyGems is Really Slow

Q: When I bundle from RubyGems.org, it is really slow. Is there anything I can do to make it faster?

A: First, update to the latest version of Bundler by running `gem install bundler`. We have added many, many improvements that make installing gems faster over the years. If you have an extremely high-latency connection, you might also see an improvement by using the `--full-index` flag. This downloads gem information all at once, instead of making many small HTTP requests.

```
$ bundle install --full-index
```

Using Gemfiles inside gems

Q: What happens if I put a `Gemfile` in my gem?

A: When someone installs your gem, the `Gemfile` and `Gemfile.lock` files are completely ignored, even if you include them inside the `.gem` file you upload to rubygems.org. The `Gemfile` inside your gem is only to make it easy for developers (like you) to install the dependencies needed to do development work on your gem. The `Gemfile` also provides an easy way to track and install development-only or test-only gems. Read about Gemfiles in gems from the Bundler in gems page and the How to create a gem with Bundler guide.

Q: Should I commit my `Gemfile.lock` when writing a gem?

A: Yes, you should commit it. The presence of a `Gemfile.lock` in a gem's repository ensures that a fresh checkout of the repository uses the exact same set of dependencies every time. We believe this makes repositories more friendly towards new and existing contributors. Ideally, anyone should be able to clone the repo, run `bundle install`, and have passing tests. If you don't check in your `Gemfile.lock`, new contributors can get different versions of your dependencies, and run into failing tests that they don't know how to fix.

Q: But I have read that gems should not check in the Gemfile.lock!

A: The main advantage of not checking in your Gemfile.lock is that new checkouts (including CI) will immediately have failing tests if one of your dependencies changes in a breaking way. Instead of forcing every fresh checkout (and possible new contributor) to encounter broken builds, the Bundler team recommends either using a tool like Dependabot to automatically create a PR and run the test suite any time your dependencies release new versions. If you don't want to use a dependency monitoring bot, we suggest creating an additional daily CI build that deletes the Gemfile.lock before running `bundle install`. That way you, and others monitoring your CI status, will be the first to know about any failures from dependency changes. Finally, when publishing your gem, consider deleting and regenerating your Gemfile.lock with the latest dependencies just before running your tests.