Guides

If you just want to know our recommended workflow and don't care about the rationale, feel free to jump to the summary below.

Bundler's Purpose and Rationale

First, you declare these dependencies in a file at the root of your application called `Gemfile`. It looks something like this:

```
source 'https://rubygems.org'
gem 'rails', '4.1.0.rc2'
gem 'rack-cache'
gem 'nokogiri', '~> 1.6.1'
```

This `Gemfile` says a few things. First, it says that bundler should look for gems declared in the `Gemfile` at `https://rubygems.org` by default. If some of your gems need to be fetched from a private gem server, this default source can be overridden for those gems.

Next, you declare a few dependencies:

- on version `4.1.0.rc2` of `rails`
- on any version of `rack-cache`
- on a version of `nokogiri` that is `>= 1.6.1` but `< 1.7.0`

After declaring your first set of dependencies, you tell bundler to go get them:

```
$ bundle install   # 'bundle' is a shortcut for 'bundle install'
```

Bundler will connect to `rubygems.org` (and any other sources that you declared) and find a list of all of the required gems that meet the requirements you specified. Because all of the gems in your `Gemfile` have dependencies of their own (and some of those have their own dependencies), running `bundle install` on the `Gemfile` above will install quite a few gems.

```
$ bundle install
Fetching gem metadata from https://rubygems.org/........
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rake 10.3.1
Using json 1.8.1
Installing minitest 5.3.3
Installing i18n 0.6.9
Installing thread_safe 0.3.3
Installing builder 3.2.2
Installing rack 1.5.2
Installing erubis 2.7.0
Installing mime-types 1.25.1
Using bundler 1.6.2
Installing polyglot 0.3.4
Installing arel 5.0.1.20140414130214
Installing hike 1.2.3
Installing mini_portile 0.5.3
Installing multi_json 1.9.3
Installing thor 0.19.1
Installing tilt 1.4.1
Installing tzinfo 1.1.0
Installing rack-test 0.6.2
Installing rack-cache 1.2
Installing treetop 1.4.15
Installing sprockets 2.12.1
Installing activesupport 4.1.0.rc2
Installing mail 2.5.4
Installing actionview 4.1.0.rc2
Installing activemodel 4.1.0.rc2
Installing actionpack 4.1.0.rc2
Installing activerecord 4.1.0.rc2
Installing actionmailer 4.1.0.rc2
Installing sprockets-rails 2.0.1
Installing railties 4.1.0.rc2
Installing rails 4.1.0.rc2
Installing nokogiri 1.6.1
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

If any of the needed gems are already installed, Bundler will use them. After installing any needed gems to your system, bundler writes a snapshot of all of the gems and versions that it installed to `Gemfile.lock`.

Setting Up Your Application to Use Bundler

Bundler makes sure that Ruby can find all of the gems in the `Gemfile` (and all of their dependencies). If your app is a Rails app, your default application already has the code necessary to invoke bundler.

For another kind of application (such as a Sinatra application), you will need to set up bundler before trying to require any gems. At the top of the first file that your application loads (for Sinatra, the file that calls `require 'sinatra'`), put the following code:

```
require 'bundler/setup'
```

This will automatically discover your `Gemfile` and make all of the gems in your `Gemfile` available to Ruby (in technical terms, it puts the gems "on the load path"). You can think of it as adding some extra powers to `require 'rubygems'`.

Now that your code is available to Ruby, you can require the gems that you need. For instance, you can `require 'sinatra'`. If you have a lot of dependencies, you might want to say "require all of the gems in my `Gemfile`." To do this, put the following code immediately following `require 'bundler/setup'`:

```
Bundler.require(:default)
```

For our example Gemfile, this line is exactly equivalent to:

```
require 'rails'
require 'rack-cache'
require 'nokogiri'
```

For such a small `Gemfile`, we'd advise you to skip `Bundler.require` and just require the gems by hand. For much larger `Gemfile`s, using `Bundler.require` allows you to skip repeating a large stack of requirements.

Checking Your Code into Version Control

After developing your application for a while, check in the application together with the `Gemfile` and `Gemfile.lock` snapshot. Now, your repository has a record of the exact versions of all of the gems that you used the last time you know for sure that the application worked. Keep in mind that while your `Gemfile` lists only three gems (with varying degrees of version strictness), your application depends on dozens of gems, once you take into consideration all of the implicit requirements of the gems you depend on.

This is important: **the `Gemfile.lock` makes your application a single package of both your own code and the third-party code it ran the last time you know for sure that everything worked**. Specifying exact versions of the third-party code you depend on in your `Gemfile` would not provide the same guarantee, because gems usually declare a range of versions for their dependencies.

The next time you run `bundle install` on the same machine, bundler will see that it already has all of the dependencies you need and skip the installation process.

Do not check in the `.bundle` directory or any of the files inside it. Those files are specific to each particular machine and are used to persist installation options between runs of the `bundle install` command.

If you have run `bundle pack`, the gems (although not the git gems) required by your bundle will be downloaded into `vendor/cache`. Bundler can run without connecting to the internet (or the RubyGems server) if all the gems you need are present in that folder and checked in to your source control. This is an **optional** step and not recommended due to the increase in size of your source control repository.

Sharing Your Application With Other Developers

When your co-developers (or you on another machine) check out your code, it will come with the exact versions of all of the third-party code your application used on the machine that you last developed on the in the `Gemfile.lock`). When **they** run `bundle install`, bundler will find the `Gemfile.lock` and skip the dependency resolution step. Instead, it will install all of the same gems that you used on the original machine.

In other words, you don't have to guess which versions of the dependencies you should install. In the example we've been using, even though `rack-cache` declares a dependency on `rack >= 0.4`, we know for sure it works with `rack 1.5.2`. Even if the Rack team releases `rack 1.5.3`, bundler will always install `1.5.2`, the exact version of the gem that we know works. This relieves a large maintenance burden from application developers as all machines always run the exact same third-party code.

Updating a Dependency

Of course, at some point, you might want to update the version of a particular dependency your application relies on. For instance, you might want to update `rails` to `4.1.0` final. Importantly, just because you're updating one dependency, it doesn't mean you want to re-resolve all of your dependencies and use the latest version of everything. In our example, you only have three dependencies, but even in this case, updating everything can cause complications.

To illustrate, the `rails 4.1.0.rc2` gem depends on `actionpack 4.1.0.rc2` gem, which depends on `rack ~> 1.5.2` (which means `>= 1.5.2` and `< 1.6.0`) The `rack-cache` gem depends on `rack >= 0.4`. Let's assume that the `rails 4.1.0` final gem also depends on `rack ~> 1.5.2`, and that since the release of `rails 4.1.0`, the Rack team released `rack 1.5.3`.

If we naively update all of our gems in order to update Rails, we'll get `rack 1.5.3`, which satisfies the requirements of both `rails 4.1.0` and `rack-cache`. However, we didn't specifically ask to update `rack-cache`, which may not be compatible with `rack 1.5.3` (for whatever reason). And while an update from `rack 1.5.2` to `rack 1.5.3` probably won't break anything, similar scenarios can happen that involve much larger jumps. (see (i) below for a larger discussion)

In order to avoid this problem, when you update a gem, bundler will not update a dependency of that gem if another gem still depends on it. In this example, since `rack-cache` still depends on `rack`, bundler will not update the `rack` gem. This ensures that updating `rails` doesn't inadvertently break `rack-cache`. Since `rails 4.1.0`'s dependency `actionpack 4.1.0` remains compatible with `rack 1.5.2`, bundler leaves it alone and `rack-cache` continues to work even in the face of an incompatibility with `rack 1.5.3`.

Since you originally declared a dependency on `rails 4.1.0.rc2`, if you want to update to `rails 4.1.0`, simply update your `Gemfile` to `gem 'rails', '4.1.0'` and run:

```
$ bundle install
```

As described above, the `bundle install` command always does a conservative update, refusing to update gems for their dependencies) that you have not explicitly changed in the `Gemfile`. This means that if you do not modify `rack-cache` in your `Gemfile`, bundler will not trust **and its dependencies** (`rack`) as a single, unmodifiable unit. If `rails 4.1.0` was incompatible with `rack-cache`, bundler will report a conflict between your snapshotted dependencies (`Gemfile.lock`) and your updated `Gemfile`.

If you update your `Gemfile`, and your system already has all of the needed dependencies, bundler will transparently update the `Gemfile.lock` when you boot your application. For instance, if you add `mysql` to your `Gemfile` and have already installed it in your system, you can boot your application without running `bundle install`, and bundler will persist the "last known good" configuration to the `Gemfile.lock` snapshot.

This can come in handy when adding or updating gems with minimal dependencies (database drivers, `wirble`, `ruby-debug`). It will probably fail if you update gems with significant dependencies (`rails`) or that a lot of gems depend on (`rack`) If a transparent update fails, your application will fail to boot, and bundler will print out an error instructing you to run `bundle install`.

Updating a Gem Without Modifying the Gemfile

Sometimes, you want to update a dependency without modifying the Gemfile. For example, you might want to update to the latest version of `rack-cache`. Because you did not declare a specific version of `rack-cache` in the `Gemfile`, you might want to periodically get the latest version of `rack-cache`. To do this, you want to use the `bundle update` command:

```
$ bundle update rack-cache
```

This command will update `rack-cache` and its dependencies to the latest version allowed by the `Gemfile` (in this case, the latest version available). It will not modify any other dependencies.

It will, however, update dependencies of other gems if necessary. For instance, if the latest version of `rack-cache` specifies a dependency on `rack >= 1.5.2`, bundler will update `rack` to `1.5.2` even though you have not asked bundler to update `rack`. If bundler needs to update a gem that another gem depends on, it will let you know after the update has completed.

If you want to update every gem in the Gemfile to the latest possible versions, run:

```
$ bundle update
```

This will resolve dependencies from scratch, ignoring the `Gemfile.lock`. If you do this, keep `git reset --hard` and your test suite in your back pocket. Resolving all dependencies from scratch can have surprising results, especially if a number of the third-party packages you depend on have released new versions since you last did a full update.

Summary

A Simple Bundler Workflow

- When you first create a Rails application, it already comes with a `Gemfile`. For another kind of application (such as Sinatra), run:

  ```
  $ bundle init
  ```

  The `bundle init` command creates a simple `Gemfile` which you can edit.

- Next, add any gems that your application depends on. If you care which version of a particular gem that you need, be sure to include an appropriate version restriction:

  ```
  source 'https://rubygems.org'
  gem 'sinatra', '~> 1.3.6'
  gem 'rack-cache'
  gem 'rack-bug'
  ```

- If you don't have the gems installed in your system yet, run:

  ```
  $ bundle install
  ```

- To update a gem's version requirements, first modify the Gemfile:

  ```
  source 'https://rubygems.org'
  gem 'sinatra', '~> 1.4.5'
  gem 'rack-cache'
  gem 'rack-bug'
  ```

  and then run:

  ```
  $ bundle install
  ```

- If `bundle install` reports a conflict between your `Gemfile` and `Gemfile.lock`, run:

  ```
  $ bundle update sinatra
  ```

  This will update just the Sinatra gem, as well as any of its dependencies.

- To update all of the gems in your `Gemfile` to the latest possible versions, run:

  ```
  $ bundle update
  ```

- Whenever your `Gemfile.lock` changes, always check it in to version control. It keeps a history of the exact versions of all third-party code that you used to successfully run your application.
- When deploying your code to a staging or production server, first run your tests for boot your local development server), make sure you have checked in your `Gemfile.lock` to version control. On the remote server, run:

  ```
  $ bundle install --deployment
  ```

Notes

(i) For instance, if `rails 4.1.0` depended on `rack 2.0`, that gem would still satisfy the requirement of `rack-cache`, which declares `>= 0.4` as a dependency. Of course, you could argue that `rack-cache` is silly for depending on open-ended versions, but these situations exist extensively in the wild, and projects often find themselves between a rock and a hard place when deciding what version to depend on. Constrain it too narrowly (`rack >=1.5.1`) and you make it hard to use your project in other compatible projects. Constrain it too little (`rack >= 1.0`) and a new release of Rack may break your code. Using dependencies like `rack ~> 1.5.2` and versioning code in a SemVer compliant way mostly solves this problem, but it assumes universal compliance. Since RubyGems has over 100,000 packages, this assumption simply doesn't hold in practice.