Rocky Enterprise Linux 9.2 Manual Pages on command 'bundle-install.1'

$ man bundle-install.1

BUNDLE-INSTALL(1) BUNDLE-INSTALL(1)

NAME

bundle-install - Install the dependencies specified in your Gemfile

SYNOPSIS

bundle install [--binstubs[=DIRECTORY]] [--clean] [--deployment] [--frozen] [--full-index] [--gemfile=GEMFILE] [--jobs=NUMBER] [--local] [--no-cache] [--no-prune] [--path PATH] [--quiet] [--redownload] [--retry=NUMBER] [--shebang] [--standalone[=GROUP[GROUP...]]] [--sys? tem] [--trust-policy=POLICY] [--with=GROUP[GROUP...]] [--with? out=GROUP[GROUP...]]

DESCRIPTION

Install the gems specified in your Gemfile(5). If this is the first time you run bundle install (and a Gemfile.lock does not exist), Bundler will fetch all remote sources, resolve dependencies and install all needed gems.

If a Gemfile.lock does exist, and you have not updated your Gemfile(5), Bundler will fetch all remote sources, but use the dependencies speci? fied in the Gemfile.lock instead of resolving dependencies.

If a Gemfile.lock does exist, and you have updated your Gemfile(5), Bundler will use the dependencies in the Gemfile.lock for all gems that you did not update, but will re-resolve the dependencies of gems that you did update. You can find more information about this update process below under CONSERVATIVE UPDATING.

OPTIONS

The --clean, --deployment, --frozen, --no-prune, --path, --shebang, --system, --without and --with options are deprecated because they only make sense if they are applied to every subsequent bundle install run automatically and that requires bundler to silently remember them. Since bundler will no longer remember CLI flags in future versions, bundle config (see bundle-config(1)) should be used to apply them per?manently.

--binstubs[=<directory>]

Binstubs are scripts that wrap around executables. Bundler cre?ates a small Ruby file (a binstub) that loads Bundler, runs the command, and puts it in bin/. This lets you link the binstub in?side of an application to the exact gem version the application needs.

Creates a directory (defaults to ~/bin) and places any executa?bles from the gem there. These executables run in Bundler?s con?text. If used, you might add this directory to your environ?ment?s PATH variable. For instance, if the rails gem comes with a rails executable, this flag will create a bin/rails executable that ensures that all referred dependencies will be resolved us?ing the bundled gems.

--clean

On finishing the installation Bundler is going to remove any gems not present in the current Gemfile(5). Don?t worry, gems currently in use will not be removed.

This option is deprecated in favor of the clean setting.

--deployment

In deployment mode, Bundler will ?roll-out? the bundle for pro?

duction or CI use. Please check carefully if you want to have this option enabled in your development environment.

This option is deprecated in favor of the deployment setting.

--redownload

Force download every gem, even if the required versions are al? ready available locally.

--frozen

Do not allow the Gemfile.lock to be updated after this install. Exits non-zero if there are going to be changes to the Gem? file.lock.

This option is deprecated in favor of the frozen setting.

--full-index

Bundler will not call Rubygems? API endpoint (default) but down? load and cache a (currently big) index file of all gems. Perfor? mance can be improved for large bundles that seldom change by enabling this option.

--gemfile=<gemfile>

The location of the Gemfile(5) which Bundler should use. This defaults to a Gemfile(5) in the current working directory. In general, Bundler will assume that the location of the Gemfile(5) is also the project?s root and will try to find Gemfile.lock and vendor/cache relative to this location.

--jobs=[<number>], -j[<number>]

The maximum number of parallel download and install jobs. The default is 1.

--local

Do not attempt to connect to rubygems.org. Instead, Bundler will use the gems already present in Rubygems? cache or in ven? dor/cache. Note that if an appropriate platform-specific gem ex? ists on rubygems.org it will not be found.

--no-cache

Do not update the cache in vendor/cache with the newly bundled gems. This does not remove any gems in the cache but keeps the

newly bundled gems from being cached during the install.

--no-prune

Don?t remove stale gems from the cache when the installation finishes.

This option is deprecated in favor of the no_prune setting.

--path=<path>

The location to install the specified gems to. This defaults to Rubygems? setting. Bundler shares this location with Rubygems, gem install ... will have gem installed there, too. Therefore, gems installed without a --path ... setting will show up by calling gem list. Accordingly, gems installed to other locations will not get listed.

This option is deprecated in favor of the path setting.

--quiet

Do not print progress information to the standard output. In? stead, Bundler will exit using a status code ($?).

--retry=[<number>]

Retry failed network or git requests for number times.

--shebang=<ruby-executable>

Uses the specified ruby executable (usually ruby) to execute the scripts created with --binstubs. In addition, if you use --bin? stubs together with --shebang jruby these executables will be changed to execute jruby instead.

This option is deprecated in favor of the shebang setting.

--standalone[=<list>]

Makes a bundle that can work without depending on Rubygems or Bundler at runtime. A space separated list of groups to install has to be specified. Bundler creates a directory named bundle and installs the bundle there. It also generates a bun? dle/bundler/setup.rb file to replace Bundler?s own setup in the manner required. Using this option implicitly sets path, which is a [remembered option][REMEMBERED OPTIONS].

--system

Installs the gems specified in the bundle to the system?s Rubygems location. This overrides any previous configuration of --path.

This option is deprecated in favor of the system setting.

--trust-policy=[<policy>]

Apply the Rubygems security policy policy, where policy is one of HighSecurity, MediumSecurity, LowSecurity, AlmostNoSecurity, or NoSecurity. For more details, please see the Rubygems signing documentation linked below in SEE ALSO.

--with=<list>

A space-separated list of groups referencing gems to install. If an optional group is given it is installed. If a group is given that is in the remembered list of groups given to --without, it is removed from that list.

This option is deprecated in favor of the with setting.

--without=<list>

A space-separated list of groups referencing gems to skip during installation. If a group is given that is in the remembered list of groups given to --with, it is removed from that list.

This option is deprecated in favor of the without setting.

DEPLOYMENT MODE

Bundler?s defaults are optimized for development. To switch to defaults optimized for deployment and for CI, use the --deployment flag. Do not activate deployment mode on development machines, as it will cause an error when the Gemfile(5) is modified.

1. A Gemfile.lock is required.

   To ensure that the same versions of the gems you developed with and tested with are also used in deployments, a Gemfile.lock is re? quired.

   This is mainly to ensure that you remember to check your Gem? file.lock into version control.

2. The Gemfile.lock must be up to date

   In development, you can modify your Gemfile(5) and re-run bundle

install to conservatively update your Gemfile.lock snapshot.

In deployment, your Gemfile.lock should be up-to-date with changes made in your Gemfile(5).

3. Gems are installed to vendor/bundle not your default system loca? tion

In development, it?s convenient to share the gems used in your ap? plication with other applications and other scripts that run on the system.

In deployment, isolation is a more important default. In addition, the user deploying the application may not have permission to in? stall gems to the system, or the web server may not have permission to read them.

As a result, bundle install --deployment installs gems to the ven? dor/bundle directory in the application. This may be overridden us? ing the --path option.

SUDO USAGE

By default, Bundler installs gems to the same location as gem install.

In some cases, that location may not be writable by your Unix user. In that case, Bundler will stage everything in a temporary directory, then ask you for your sudo password in order to copy the gems into their system location.

From your perspective, this is identical to installing the gems di? rectly into the system.

You should never use sudo bundle install. This is because several other steps in bundle install must be performed as the current user:

? Updating your Gemfile.lock

? Updating your vendor/cache, if necessary

? Checking out private git repositories using your user?s SSH keys

Of these three, the first two could theoretically be performed by chowning the resulting files to $SUDO_USER. The third, however, can only be performed by invoking the git command as the current user.

Therefore, git gems are downloaded and installed into ~/.bundle rather than $GEM_HOME or $BUNDLE_PATH.

As a result, you should run bundle install as the current user, and Bundler will ask for your password if it is needed to put the gems into their final location.

INSTALLING GROUPS

By default, bundle install will install all gems in all groups in your Gemfile(5), except those declared for a different platform.

However, you can explicitly tell Bundler to skip installing certain groups with the --without option. This option takes a space-separated list of groups.

While the --without option will skip installing the gems in the speci? fied groups, it will still download those gems and use them to resolve the dependencies of every gem in your Gemfile(5).

This is so that installing a different set of groups on another machine (such as a production server) will not change the gems and versions that you have already developed and tested against.

Bundler offers a rock-solid guarantee that the third-party code you are running in development and testing is also the third-party code you are running in production. You can choose to exclude some of that code in different environments, but you will never be caught flat-footed by different versions of third-party code being used in different environ? ments.

For a simple illustration, consider the following Gemfile(5):

```
source ?https://rubygems.org?
gem ?sinatra?
group :production do
  gem ?rack-perftools-profiler?
end
```

In this case, sinatra depends on any version of Rack (>= 1.0), while rack-perftools-profiler depends on 1.x (~> 1.0).

When you run bundle install --without production in development, we look at the dependencies of rack-perftools-profiler as well. That way, you do not spend all your time developing against Rack 2.0, using new APIs unavailable in Rack 1.x, only to have Bundler switch to Rack 1.2

when the production group is used.

This should not cause any problems in practice, because we do not at? tempt to install the gems in the excluded groups, and only evaluate as part of the dependency resolution process.

This also means that you cannot include different versions of the same gem in different groups, because doing so would result in different sets of dependencies used in development and production. Because of the vagaries of the dependency resolution process, this usually affects more than the gems you list in your Gemfile(5), and can (surprisingly) radically change the gems you are using.

THE GEMFILE.LOCK

When you run bundle install, Bundler will persist the full names and versions of all gems that you used (including dependencies of the gems specified in the Gemfile(5)) into a file called Gemfile.lock.

Bundler uses this file in all subsequent calls to bundle install, which guarantees that you always use the same exact code, even as your appli? cation moves across machines.

Because of the way dependency resolution works, even a seemingly small change (for instance, an update to a point-release of a dependency of a gem in your Gemfile(5)) can result in radically different gems being needed to satisfy all dependencies.

As a result, you SHOULD check your Gemfile.lock into version control, in both applications and gems. If you do not, every machine that checks out your repository (including your production server) will resolve all dependencies again, which will result in different versions of third-party code being used if any of the gems in the Gemfile(5) or any of their dependencies have been updated.

When Bundler first shipped, the Gemfile.lock was included in the .git? ignore file included with generated gems. Over time, however, it became clear that this practice forces the pain of broken dependencies onto new contributors, while leaving existing contributors potentially un? aware of the problem. Since bundle install is usually the first step towards a contribution, the pain of broken dependencies would discour?

age new contributors from contributing. As a result, we have revised our guidance for gem authors to now recommend checking in the lock for gems.

CONSERVATIVE UPDATING

When you make a change to the Gemfile(5) and then run bundle install, Bundler will update only the gems that you modified.

In other words, if a gem that you did not modify worked before you called bundle install, it will continue to use the exact same versions of all dependencies as it used before the update.

Let?s take a look at an example. Here?s your original Gemfile(5):

```
source ?https://rubygems.org?
gem ?actionpack?, ?2.3.8?
gem ?activemerchant?
```

In this case, both actionpack and activemerchant depend on activesup? port. The actionpack gem depends on activesupport 2.3.8 and rack ~> 1.1.0, while the activemerchant gem depends on activesupport >= 2.3.2, braintree >= 2.0.0, and builder >= 2.0.0.

When the dependencies are first resolved, Bundler will select ac? tivesupport 2.3.8, which satisfies the requirements of both gems in your Gemfile(5).

Next, you modify your Gemfile(5) to:

```
source ?https://rubygems.org?
gem ?actionpack?, ?3.0.0.rc?
gem ?activemerchant?
```

The actionpack 3.0.0.rc gem has a number of new dependencies, and up? dates the activesupport dependency to = 3.0.0.rc and the rack depen? dency to ~> 1.2.1.

When you run bundle install, Bundler notices that you changed the ac? tionpack gem, but not the activemerchant gem. It evaluates the gems currently being used to satisfy its requirements:

```
activesupport 2.3.8
    also used to satisfy a dependency in activemerchant, which is
    not being updated
```

rack ~> 1.1.0

not currently being used to satisfy another dependency

Because you did not explicitly ask to update activemerchant, you would not expect it to suddenly stop working after updating actionpack. How? ever, satisfying the new activesupport 3.0.0.rc dependency of action? pack requires updating one of its dependencies.

Even though activemerchant declares a very loose dependency that theo? retically matches activesupport 3.0.0.rc, Bundler treats gems in your Gemfile(5) that have not changed as an atomic unit together with their dependencies. In this case, the activemerchant dependency is treated as activemerchant 1.7.1 + activesupport 2.3.8, so bundle install will re? port that it cannot update actionpack.

To explicitly update actionpack, including its dependencies which other gems in the Gemfile(5) still depend on, run bundle update actionpack (see bundle update(1)).

Summary: In general, after making a change to the Gemfile(5) , you should first try to run bundle install, which will guarantee that no other gem in the Gemfile(5) is impacted by the change. If that does not work, run bundle update(1) bundle-update.1.html.

SEE ALSO

? Gem install docs http://guides.rubygems.org/rubygems-basics/#in? stalling-gems

? Rubygems signing docs http://guides.rubygems.org/security/