Development Thoughts

What is a Gemfile

Jul 19, 2015

Ruby developers use Gemfiles all the time, and most of us know how to do the basics. In this post I want to dive deep into everything we can do with a Gemfile.

What is a Gemfile?

A Gemfile is a file we create which is used for describing gem dependencies for Ruby programs. A gem is a collection of Ruby code that we can extract into a "collection" which we can call later.

Your Gemfile should always be in the root of your project directory, this is where Bundler expects it to be and it is the standard place for any package manager style files to live.

It is useful to note that your Gemfile is evaluated as Ruby code. When it is evaluated by Bundler the context it is in allows us access to certain methods that we will use to explain our gem requirements.

Setting up a Gemfile

The first thing we need to do is tell the Gemfile where to

look for gems, this is called the source.

We use the `#source` method for doing this.

```
source "https://rubygems.org"
```

It isn't recommended to have more than one source per project. For 99% of projects that require a Gemfile your source will be set to `https://rubygems.org`. The only requirement for a source is that it must be a valid Rubygems repository.

Source Priority

Now seems like a good time to discuss source priority.

As well as defining a `source` at the top of our Gemfile we can define a source against each gem we are loading in. We can also define a path for a local gem or a git path for a gem hosted somewhere like GitHub (we will get to these later).

When Bundler attempts to locate a gem it will look first at what has been explicitly set on the gem and use that.

If you set up a gem using `source`, `path`, or `git` any dependencies for that gem will look in those locations first before trying anywhere else.

If nothing has been explicitly set Bundler will look at the sources you have defined, starting at the first one and

working down.

If a gem is found in more than one global source (this should rarely be the case because you should only really have one source) then you will get a warning explaining which gem source has been used.

You can call `#source` as a block;

```
source "https://my_awesome_source.com" do
  gem "my_gem"
  gem "my_other_gem"
end
```

Sources with Credentials

Some sources you use will require credentials to be set.

Bundle has a config option that allows you to set a username and password for each source;

```
bundle config my_gem_source.com my_username:my_p
```

This will need to be set up by anyone who wants access to install gems using Bundler as it doesn't go into version control (which is one of the nice things about doing it this way).

You can also set you credentials straight in the Gemfile, of course when you do it this way your details will be committed into version control.

```
source "https://username:password@my_gem_source.
```

Anything you specify in the source will override anything you have set using `bundle config`.

Setting up Ruby information

If the application you are creating requires a specific Ruby version or engine we can set this in the Gemfile.

```
ruby "1.9.3", :patchlevel => "247", :engine => "
```

When setting this up the only required bit of information is the ruby version (in our example `1.9.3`).

- The `:patchlevel` specifies the patch level for Ruby.
- The `:engine` specifies the Ruby engine to be used.
- The `:engine_version` specifies the version of the engine being used. If this is set then `:engine` also needs to be set.

Setting up your Gems

Now onto the main point of using a Gemfile, setting up the gems!

The most basic syntax is;

```
gem "my_gem"
```

In this case `my_gem` is the name of the gem. The name is

the only thing that is required, there are several optional parameters that you can use.

Setting the version of a Gem

The most common thing you will want to do with a gem is set its version.

If you don't set a version you are basically saying any version will do;

```
gem "my_gem", ">= 0.0"
```

There are seven operators you can use when specifying your gems.

- `=` Equal To `"=1.0"`
- `!=` Not Equal To `"!=1.0"`
- `>` Greater Than `">1.0"`
- `<` Less Than `"<1.0"`
- `>=` Greater Than or Equal To `">=1.0"`
- `<=` Less Than or Equal To `"<=1.0"`
- `~>` Pessimistically Greater Than or Equal To `"~>1.0"`

Pessimistically Greater Than or Equal To

The `~>` operator allows you to say that your application will work with future versions of a gem in a safe way.

If you feel that the gem you are including is safe for an entire version you can specify;

```
gem "my_gem", "~> 2.0"
```

This will allow any version of 2.x to be installed, but nothing from version 3.x

Perhaps you don't feel comfortable giving a gem such a wide remit, in that case you can specify a more specific version;

```
gem "my_gem", "~> 2.5.0"
```

This will allow anything from 2.5.0 up to anything below 2.6.0.

The following conversions might help you to understand it better;

- `gem "my_gem", "~> 1.0"` -> `gem "my_gem", ">= 1.0", "< 2.0"`
- `gem "my_gem", "~> 1.5.0"` -> `gem "my_gem", ">= 1.5.0", "< 1.6.0"`
- `gem "my_gem", "~> 1.5.5"` -> `gem "my_gem", ">= 1.5.5", "< 1.6.0"`

Setting your Gem to be Required

If you are using Rails this bit of "magic" may have been hidden from you, but inside your `config/application.rb` you will see the following line;

```
Bundler.require(:default, Rails.env)
```

This means require all the gems that haven't been assigned a group and require all the gems that have been assigned a group of the same name as your Rails environment (for example test, or development).

By default if you include a gem in your Gemfile it will be included when `Bundler.require` is called. We can stop this by setting require to false;

```
gem "my_gem", require: false
```

You can also specify which folder(s) should be required when your gem is included;

```
gem "my_gem", require: ["my_gem/specific_module/
```

This is useful when your gem has a lot of functionality that you need to manually require each time you want to call it.

Grouping your Gem

As I mentioned a gem can belong to one or more groups. When it doesn't belong to any groups it is put into the `:default` group.

There are two ways you group a gem. The first is by assigning a value to the `:group` property;

```
gem "my_gem", group: :development
```

This means it will only be required when the development environment is running.

It also means when you are installing gems (with `bundle install`) that you can specify certain groups to not install. This can speed up the install time for new projects considerably.

```
bundle install --without development test
```

Would install everything except gems in the development or test group.

The second way you can decide a grouping for a gem is by setting your gems up inside a block;

```
group :development do
  gem "my_gem"
  gem "my_other_gem"
end
```

This is visually more pleasing, and you can combine groups;

```
group :development, :test do
  gem "my_gem"
  gem "my_other_gem"
end
```

If there is a group you want to be optional you can pass `optional: true` before the block;

```
group :development, optional: true do
  gem "my_gem"
  gem "my_other_gem"
end
```

When this is set in order for it to be installed the user has to perform `bundle install --with development`

Setting a Platform for your Gem

If a gem should only be used on a particular platform (or set of platforms) then you can specify so in the Gemfile.

Platforms work in much the same way as groups, expect that you do not need to run calls with the `--without` flag as this will happen automatically.

```
gem "my_gem", platform: :jruby
gem "my_other_gem", platform: [:ruby, :mri_18]
```

Here is a list of all the different platforms you can ask your gem to install under.

- ruby – C Ruby (MRI) or Rubinius, but not Windows
- ruby_18 to ruby_22 – ruby & (version 1.8 .. version 2.2)
- mri – Same as ruby, but not Rubinius
- mri_18 to mri_22 – mri & (version 1.8 .. version 2.2)

- rbx – Same as ruby, but only Rubinius (not MRI)
- jruby – JRuby
- mswin – Windows
- mingw – Windows 32 bit mingw32 platform (aka RubyInstaller)
- mingw_18 to mingw_22 – mingw & (version 1.8 .. version 2.2)
- x64_mingw – Windows 64 bit mingw32 platform
- x64_mingw_20 to x64_mingw_22 – x64_mingw & (version 2.0 .. version 2.2)

I have found platforms really helpful when a development team are working across different platforms, if one of your developers is running Windows you might need different versions of gems depending on what is supported.

I normally use the block syntax when using platforms;

```
platforms :jruby do
  gem "my_gem"
  gem "my_other_gem"
end
```

Setting a source for you Gem

As I mentioned earlier there is a notion of setting sources for your gems.

This is done in the following way;

```
gem "my_gem", source: "https://my_awesome_gemsit
```

If the gem isn't found at this source it will not fall back to being searched for in the default source, it just will not install.

Installing a Gem from Git

You can set your install location to be from a git repository (such as Github). This acts in much the same way as changing the `:source` parameter.

```
gem "my_gem", git: "ssh@githib.com/tosbourn/my_g
```

Whilst you can link to the repository using `HTTP(S)`, `SSH` and `git` protocols it is highly recommended that you use only `HTTPS` and `SSH` since the others could leave you victim to a man-in-the-middle attack.

If you are storing your gem in a repository it should contain at least one file at the root of the directory with a `.gemspec` extension. This should contain a valid gem specification.

If you don't provide this file then Bundler will try and create one, but it shouldn't be relied upon. If you do try and include a gem hosted on a git repository without a `.gemspec` you need to have a version for your gem specified.

You can set either a `branch`, `tag`, or `ref` for your gem. The default is `branch: "master"`

You can also force Bundler to expand any submodules hosted in the git repository by passing in `submodules: true`

```
gem "my_gem", git: "ssh@githib.com/tosbourn/my_g
```

If you have several gems you want to load in from the same git repository you can use a block;

```
git "git@github.com:tosbourn/my_gems.git" do
  gem "my_gem"
  gem "my_other_gem"
end
```

Setting Git as a Source

You can set a URL to act as a more generalised source of information for your gems. You do this by calling `#git_source` and passing in a name as an argument and a block which receives one argument and returns a string for the full repository address.

```
git_source(:custom_git){ |repo| "https://my_secr
gem "my_gem", custom_git: "tosbourn/test_repo"
```

Bitbucket and Github helper methods

Since both Bitbucket and Github are popular places to

house git repositories there are helper methods made for them.

In both cases Bundler assumes the repositories are public.

```
gem "my_gem", github: "tosbourn/my_gem"
gem "my_gem", bitbucket: "tosbourn/my_gem"
```

You can specify a branch by using the `:branch` parameter.

If both the username and the repository name are the same (as is the case with projects like Rails) then you can omit one.

```
gem "rails", github: "rails"
gem "rails", bitbucket: "rails"
```

Warning – You shouldn't use the `:github` parameter until Bundler 2 comes out as right now it defaults to using the `git://` protocol, which as we have already heard can leave you open to man-in-the-middle attacks.

Another helper is `:gist`, this can be used if your project is hosted on Github as a gist. You can just use the gist ID as the path. Like `:github` and `:bitbucket` you can pass a `:branch` parameter into the method.

```
gem "my_gem", :gist => "5935162112", branch: "my
```

Include local Gem with the Path parameter

You can specify that your gem lives locally on your system by passing in the `:path` parameter.

```
gem "my_gem", :path => "../my_path/my_gem"
```

If you specify a relative path (like I did above) they will be relative to the directory containing the Gemfile.

If you have an entire directory full of gems you want to include locally you can call this directory as block;

```
path "../my_path/gems" do
  gem "my_gem"
  gem "my_other_gem"
end
```

One thing to note is that Bundler will not compile C extensions for gems that have been specified using `:path`.

Conditionally Installing Gems

Sometimes you want to only install a gem if some prerequisite is true, for example if there is a program available on your system.

This method accepts a proc or a lambda. In this example

we only want to install a gem if we are on a Mac.

```
install_if -> { RUBY_PLATFORM =~ /darwin/ } do
  gem "my_osx_gem"
end
```

Fin

Thanks for reading this guide, hopefully you found it useful. Do let me know in the comments if I have missed anything out or if you have any questions.

Translations

- Chinese – Thanks Xu Jianyong!
- If you would like to translate this (or any article) please get in touch.

Share this on:

Facebook Twitter G+ LinkedIn Reddit HN

Did you find this post helpful? If so I would really appreciate it if you could look at 5 ways you could help the site for free in under 5 minutes.

16 Comments Toby's Development Blog 1 Login

Recommend 2 Share Sort by Best

Join the discussion…

Xu Jianyong · 23 days ago
Hi, Toby, it is really a good article for the beginner. Can I translate it into Chinese to help the developer in China to read it, I will point the original source.
1 · Reply · Share ›

> **Toby Osbourn** Mod ↗ Xu Jianyong · 23 days ago
> Hey Xu, that would be amazing! Share the link here and I will add it to the article.
> · Reply · Share ›
>
> > **Xu Jianyong** ↗ Toby Osbourn · 21 days ago
> > Hi, Toby, here is the Chinese version.
> > https://ruby-china.org/topics/...
> > 1 · Reply · Share ›
> >
> > > **Toby Osbourn** Mod ↗ Xu Jianyong · 21 days ago
> > > Thanks Xu, it will be in the article in ~5 mins
> > > · Reply · Share ›
> >
> > **Xu Jianyong** ↗ Toby Osbourn · 23 days ago
> > ok, when I finish it, I will paste the link here. Thank you!
> > · Reply · Share ›

Alexandre Formagio · 24 days ago
Hi Toby, nice to meet you!
Thank you, You helped me a lot with this article, you give a very good explanation.
I'd like to make a question.

I created a gem and I installed in the system using gem install my_gem-1.0.4.gem. So in my Gemfile I only put "gem 'my_gem' , '>=1.0.4' ".

Is this the right way?

Or is better save my gems in a folder and in my Gemfile put "gem 'my_gem' , '>=1.0.4' :path => '../my_path/my_gem' "?
1 · Reply · Share ›

GitHub
Twitter
Facebook
Email

Privacy Policy
RSS
I backup with Dropbox and Backblaze

Subscribe to infrequent updates from me

Email Address

*

Subscribe