Creating and Customizing Rails Generators & Templates

Rails generators are an essential tool if you plan to improve your workflow. With this guide you will learn how to create generators and customize existing ones.

After reading this guide, you will know:

- How to see which generators are available in your application.
- How to create a generator using templates.
- How Rails searches for generators before invoking them.
- How Rails internally generates Rails code from the templates.
- How to customize your scaffold by creating new generators.
- How to customize your scaffold by changing generator templates.
- How to use fallbacks to avoid overwriting a huge set of generators.
- How to create an application template.

First Contact

When you create an application using the `rails` command, you are in fact using a Rails generator. After that, you can get a list of all available generators by just invoking `bin/rails generate`:

```
$ rails new myapp
$ cd myapp
$ bin/rails generate
```

NOTE: To create a rails application we use the `rails` global command, the rails gem installed via `gem install rails`. When inside the directory of your application, we use the command `bin/rails` which uses the bundled rails inside this application.

You will get a list of all generators that come with Rails. If you need a detailed description of the helper generator, for example, you can simply do:

```
$ bin/rails generate helper --help
```

Creating Your First Generator

Since Rails 3.0, generators are built on top of Thor. Thor provides powerful options for parsing and a great API for manipulating files. For instance, let's build a generator that creates an initializer file named `initializer.rb` inside `config/initializers`.

The first step is to create a file at `lib/generators/initializer_generator.rb` with the following content:

```
class InitializerGenerator < Rails::Generators::Base
  def create_initializer_file
    create_file "config/initializers/initializer.rb", "# Add initialization content here"
  end
end
```

NOTE: `create_file` is a method provided by `Thor::Actions`. Documentation for `create_file` and other Thor methods can be found in Thor's documentation

Our new generator is quite simple: it inherits from `Rails::Generators::Base` and has one method definition. When a generator is invoked, each public method in the generator is executed sequentially in the order that it is defined. Finally, we invoke the `create_file` method that will create a file at the given destination with the given content. If you are familiar with the Rails Application Templates API, you'll feel right at home with the new generators API.

To invoke our new generator, we just need to do:

```
$ bin/rails generate initializer
```

Before we go on, let's see our brand new generator description:

```
$ bin/rails generate initializer --help
```

Rails is usually able to generate good descriptions if a generator is namespaced, as `ActiveRecord::Generators::ModelGenerator`, but not in this particular case. We can solve this problem in two ways. The first one is calling `desc` inside our generator:

```
class InitializerGenerator < Rails::Generators::Base
  desc "This generator creates an initializer file at config/initializers"
  def create_initializer_file
    create_file "config/initializers/initializer.rb", "# Add initialization content here"
  end
end
```

Now we can see the new description by invoking `--help` on the new generator. The second way to add a description is by creating a file named `USAGE` in the same directory as our generator. We are going to do that in the next step.

Creating Generators with Generators

Generators themselves have a generator:

```
$ bin/rails generate generator initializer
      create  lib/generators/initializer
      create  lib/generators/initializer/initializer_generator.rb
      create  lib/generators/initializer/USAGE
      create  lib/generators/initializer/templates
      invoke  test_unit
      create    test/lib/generators/initializer_generator_test.rb
```

This is the generator just created:

```ruby
class InitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path('templates', __dir__)
end
```

First, notice that we are inheriting from `Rails::Generators::NamedBase` instead of `Rails::Generators::Base`. This means that our generator expects at least one argument, which will be the name of the initializer, and will be available in our code in the variable `name`.

We can see that by invoking the description of this new generator (don't forget to delete the old generator file):

```
$ bin/rails generate initializer --help
Usage:
  bin/rails generate initializer NAME [options]
```

We can also see that our new generator has a class method called `source_root`. This method points to where our generator templates will be placed, if any, and by default it points to the created directory `lib/generators/initializer/templates`.

In order to understand what a generator template means, let's create the file `lib/generators/initializer/templates/initializer.rb` with the following content:

```ruby
# Add initialization content here
```

And now let's change the generator to copy this template when invoked:

```ruby
class InitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path('templates', __dir__)

  def copy_initializer_file
    copy_file "initializer.rb", "config/initializers/#{file_name}.rb"
  end
end
```

And let's execute our generator:

```
$ bin/rails generate initializer core_extensions
```

We can see that now an initializer named core_extensions was created at `config/initializers/core_extensions.rb` with the contents of our template. That means that `copy_file` copied a file in our source root to the destination path we gave. The method `file_name` is automatically created when we inherit from `Rails::Generators::NamedBase`.

The methods that are available for generators are covered in the final section of this guide.

Generators Lookup

When you run `bin/rails generate initializer core_extensions` Rails requires these files in turn until one is found:

```
rails/generators/initializer/initializer_generator.rb
generators/initializer/initializer_generator.rb
rails/generators/initializer_generator.rb
generators/initializer_generator.rb
```

If none is found you get an error message.

INFO: The examples above put files under the application's `lib` because said directory belongs to `$LOAD_PATH`.

Customizing Your Workflow

Rails own generators are flexible enough to let you customize scaffolding. They can be configured in `config/application.rb`, these are some defaults:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: true
end
```

Before we customize our workflow, let's first see what our scaffold looks like:

```
$ bin/rails generate scaffold User name:string
      invoke  active_record
      create    db/migrate/20130924151154_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
      invoke  resource_route
       route    resources :users
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
      invoke    test_unit
      create      test/controllers/users_controller_test.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke    jbuilder
      create      app/views/users/index.json.jbuilder
      create      app/views/users/show.json.jbuilder
      invoke  test_unit
      create    test/application_system_test_case.rb
      create    test/system/users_test.rb
```

Looking at this output, it's easy to understand how generators work in Rails 3.0 and above. The scaffold generator doesn't actually generate anything; it just invokes others to do the work. This allows us to add/replace/remove any of those invocations. For instance, the scaffold generator invokes the `scaffold_controller` generator, which invokes `erb`, `test_unit`, and `helper` generators. Since each generator has a single responsibility, they are easy to reuse, avoiding code duplication.

The next customization on the workflow will be to stop generating stylesheet and test fixture files for scaffolds altogether. We can achieve that by changing our configuration to the following:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: false
end
```

If we generate another resource with the scaffold generator, we can see that stylesheet, JavaScript, and fixture files are not created anymore. If you want to customize it further, for example to use DataMapper and RSpec instead of Active Record and TestUnit, it's just a matter of adding their gems to your application and configuring your generators.

To demonstrate this, we are going to create a new helper generator that simply adds some instance variable readers. First, we create a generator within the rails namespace, as this is where rails searches for generators used as hooks:

```
$ bin/rails generate generator rails/my_helper
      create  lib/generators/rails/my_helper
      create  lib/generators/rails/my_helper/my_helper_generator.rb
      create  lib/generators/rails/my_helper/USAGE
      create  lib/generators/rails/my_helper/templates
      invoke  test_unit
      create    test/lib/generators/rails/my_helper_generator_test.rb
```

After that, we can delete both the `templates` directory and the `source_root` class method call from our new generator, because we are not going to need them. Add the method below, so our generator looks like the following:

```
# lib/generators/rails/my_helper/my_helper_generator.rb
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
module #{class_name}Helper
  attr_reader :#{plural_name}, :#{plural_name.singularize}
end
    FILE
  end
end
```

We can try out our new generator by creating a helper for products:

```
$ bin/rails generate my_helper products
      create  app/helpers/products_helper.rb
```

And it will generate the following helper file in `app/helpers`:

```
module ProductsHelper
  attr_reader :products, :product
end
```

Which is what we expected. We can now tell scaffold to use our new helper generator by editing `config/application.rb` once again:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: false
  g.stylesheets     false
  g.helper          :my_helper
end
```

and see it in action when invoking the generator:

```
$ bin/rails generate scaffold Article body:text
      [...]
      invoke    my_helper
      create      app/helpers/articles_helper.rb
```

We can notice on the output that our new helper was invoked instead of the Rails default. However one thing is missing, which is tests for our new generator and to do that, we are going to reuse old helpers test generators.

Since Rails 3.0, this is easy to do due to the hooks concept. Our new helper does not need to be focused in one specific test framework, it can simply provide a hook and a test framework just needs to implement this hook in order to be compatible.

To do that, we can change the generator this way:

```
# lib/generators/rails/my_helper/my_helper_generator.rb
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
module #{class_name}Helper
  attr_reader :#{plural_name}, :#{plural_name.singularize}
end
    FILE
  end

  hook_for :test_framework
end
```

Now, when the helper generator is invoked and TestUnit is configured as the test framework, it will try to invoke both `Rails::TestUnitGenerator` and `TestUnit::MyHelperGenerator`. Since none of those are defined, we can tell our generator to invoke `TestUnit::Generators::HelperGenerator` instead, which is defined since it's a Rails generator. To do that, we just need to add:

```
# Search for :helper instead of :my_helper
hook_for :test_framework, as: :helper
```

And now you can re-run scaffold for another resource and see it generating tests as well!

Customizing Your Workflow by Changing Generators Templates

In the step above we simply wanted to add a line to the generated helper, without adding any extra functionality. There is a simpler way to do that, and it's by replacing the templates of already existing generators, in that case `Rails::Generators::HelperGenerator`.

In Rails 3.0 and above, generators don't just look in the source root for templates, they also search for templates in other paths. And one of them is `lib/templates`. Since we want to customize `Rails::Generators::HelperGenerator`, we can do that by simply making a template copy inside `lib/templates/rails/helper` with the name `helper.rb`. So let's create that file with the following content:

```
module <%= class_name %>Helper
  attr_reader :<%= plural_name %>, :<%= plural_name.singularize %>
end
```

and revert the last change in `config/application.rb`:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: false
end
```

If you generate another resource, you can see that we get exactly the same result! This is useful if you want to customize your scaffold templates and/or layout by just creating `edit.html.erb`, `index.html.erb` and so on inside `lib/templates/erb/scaffold`.

Scaffold templates in Rails frequently use ERB tags; these tags need to be escaped so that the generated output is valid ERB code.

For example, the following escaped ERB tag would be needed in the template (note the extra `%`)…

```
<%%= stylesheet_link_tag :application %>
```

…to generate the following output:

```
<%= stylesheet_link_tag :application %>
```

Adding Generators Fallbacks

One last feature about generators which is quite useful for plugin generators is fallbacks. For example, imagine that you want to add a feature on top of TestUnit like shoulda does. Since TestUnit already implements all generators required by Rails and shoulda just wants to overwrite part of it, there is no need for shoulda to reimplement some generators again, it can simply tell Rails to use a `TestUnit` generator if none was found under the `Shoulda` namespace.

We can easily simulate this behavior by changing our `config/application.rb` once again:

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :shoulda, fixture: false

  # Add a fallback!
  g.fallbacks[:shoulda] = :test_unit
end
```

Now, if you create a Comment scaffold, you will see that the shoulda generators are being invoked, and at the end, they are just falling back to TestUnit generators:

```
$ bin/rails generate scaffold Comment body:text
      invoke  active_record
      create    db/migrate/20130924143118_create_comments.rb
      create    app/models/comment.rb
      invoke    shoulda
      create      test/models/comment_test.rb
      create      test/fixtures/comments.yml
      invoke  resource_route
       route    resources :comments
      invoke  scaffold_controller
      create    app/controllers/comments_controller.rb
      invoke    erb
      create      app/views/comments
      create      app/views/comments/index.html.erb
      create      app/views/comments/edit.html.erb
      create      app/views/comments/show.html.erb
      create      app/views/comments/new.html.erb
      create      app/views/comments/_form.html.erb
      invoke    shoulda
      create      test/controllers/comments_controller_test.rb
      invoke    my_helper
      create      app/helpers/comments_helper.rb
      invoke    jbuilder
      create      app/views/comments/index.json.jbuilder
      create      app/views/comments/show.json.jbuilder
      invoke  test_unit
      create    test/application_system_test_case.rb
      create    test/system/comments_test.rb
```

Fallbacks allow your generators to have a single responsibility, increasing code reuse and reducing the amount of duplication.

Application Templates

Now that you've seen how generators can be used *inside* an application, did you know they can also be used to *generate* applications too? This kind of generator is referred to as a "template". This is a brief overview of the Templates API. For detailed documentation see the Rails Application Templates guide.

```
gem "rspec-rails", group: "test"
gem "cucumber-rails", group: "test"

if yes?("Would you like to install Devise?")
  gem "devise"
  generate "devise:install"
  model_name = ask("What would you like the user model to be called? [user]")
  model_name = "user" if model_name.blank?
  generate "devise", model_name
end
```

In the above template we specify that the application relies on the `rspec-rails` and `cucumber-rails` gem so these two will be added to the `test` group in the `Gemfile`. Then we pose a question to the user about whether or not they would like to install Devise. If the user replies “y” or “yes” to this question, then the template will add Devise to the `Gemfile` outside of any group and then runs the `devise:install` generator. This template then takes the users input and runs the `devise` generator, with the user’s answer from the last question being passed to this generator.

Imagine that this template was in a file called `template.rb`. We can use it to modify the outcome of the `rails new` command by using the `-m` option and passing in the filename:

```
$ rails new thud -m template.rb
```

This command will generate the `Thud` application, and then apply the template to the generated output.

Templates don’t have to be stored on the local system, the `-m` option also supports online templates:

```
$ rails new thud -m https://gist.github.com/radar/722911/raw/
```

Whilst the final section of this guide doesn’t cover how to generate the most awesome template known to man, it will take you through the methods available at your disposal so that you can develop it yourself. These same methods are also available for generators.

Adding Command Line Arguments

Rails generators can be easily modified to accept custom command line arguments. This functionality comes from Thor:

```
class_option :scope, type: :string, default: 'read_products'
```

Now our generator can be invoked as follows:

```
$ bin/rails generate initializer --scope write_products
```

The command line arguments are accessed through the `options` method inside the generator class. e.g:

```
@scope = options['scope']
```

Generator methods

The following are methods available for both generators and templates for Rails.

NOTE: Methods provided by Thor are not covered this guide and can be found in Thor’s documentation

`gem`

Specifies a gem dependency of the application.

```
gem "rspec", group: "test", version: "2.1.0"
gem "devise", "1.1.5"
```

Available options are:

- `:group` - The group in the `Gemfile` where this gem should go.
- `:version` - The version string of the gem you want to use. Can also be specified as the second argument to the method.
- `:git` - The URL to the git repository for this gem.

Any additional options passed to this method are put on the end of the line:

```
gem "devise", git: "https://github.com/plataformatec/devise.git", branch: "master"
```

The above code will put the following line into `Gemfile`:

```
gem "devise", git: "https://github.com/plataformatec/devise.git", branch: "master"
```

`gem_group`

Wraps gem entries inside a group:

```
gem_group :development, :test do
  gem "rspec-rails"
end
```

`add_source`

Adds a specified source to `Gemfile`:

```
add_source "http://gems.github.com"
```

This method also takes a block:

```
add_source "http://gems.github.com" do
  gem "rspec-rails"
end
```

`inject_into_file`

Injects a block of code into a defined position in your file.

```
inject_into_file 'name_of_file.rb', after: "#The code goes below this line. Don't forget th
  puts "Hello World"
RUBY
end
```

`gsub_file`

Replaces text inside a file.

```
gsub_file 'name_of_file.rb', 'method.to_be_replaced', 'method.the_replacing_code'
```

Regular Expressions can be used to make this method more precise. You can also use `append_file` and `prepend_file` in the same way to place code at the beginning and end of a file respectively.

`application`

Adds a line to `config/application.rb` directly after the application class definition.

```ruby
application "config.asset_host = 'http://example.com'"
```

This method can also take a block:

```ruby
application do
  "config.asset_host = 'http://example.com'"
end
```

Available options are:

- `:env` - Specify an environment for this configuration option. If you wish to use this option with the block syntax the recommended syntax is as follows:

```ruby
application(nil, env: "development") do
  "config.asset_host = 'http://localhost:3000'"
end
```

`git`

Runs the specified git command:

```ruby
git :init
git add: "."
git commit: "-m First commit!"
git add: "onefile.rb", rm: "badfile.cxx"
```

The values of the hash here being the arguments or options passed to the specific git command. As per the final example shown here, multiple git commands can be specified at a time, but the order of their running is not guaranteed to be the same as the order that they were specified in.

`vendor`

Places a file into `vendor` which contains the specified code.

```ruby
vendor "sekrit.rb", '#top secret stuff'
```

This method also takes a block:

```ruby
vendor "seeds.rb" do
  "puts 'in your app, seeding your database'"
end
```

`lib`

Places a file into `lib` which contains the specified code.

```
lib "special.rb", "p Rails.root"
```

This method also takes a block:

```
lib "super_special.rb" do
  "puts 'Super special!'"
end
```

`rakefile`

Creates a Rake file in the `lib/tasks` directory of the application.

```
rakefile "test.rake", 'task(:hello) { puts "Hello, there" }'
```

This method also takes a block:

```
rakefile "test.rake" do
  %Q{
    task rock: :environment do
      puts "Rockin'"
    end
  }
end
```

`initializer`

Creates an initializer in the `config/initializers` directory of the application:

```
initializer "begin.rb", "puts 'this is the beginning'"
```

This method also takes a block, expected to return a string:

```
initializer "begin.rb" do
  "puts 'this is the beginning'"
end
```

`generate`

Runs the specified generator where the first argument is the generator name and the remaining arguments are passed directly to the generator.

```
generate "scaffold", "forums title:string description:text"
```

`rake`

Runs the specified Rake task.

```
rake "db:migrate"
```

Available options are:

- `:env` - Specifies the environment in which to run this rake task.
- `:sudo` - Whether or not to run this task using `sudo`. Defaults to `false`.

`route`

Adds text to the `config/routes.rb` file:

```
route "resources :people"
```

`readme`

Output the contents of a file in the template's `source_path`, usually a README.

```
readme "README"
```