FRICTIONLESS GENERATORS

Effortlessly automate repetitive code generation in your Rails apps.

Garrett Dimon

Frictionless Generators

by Garrett Dimon

Copyright © 2024 Start & Sustain LLC

1 Leveraging Templates & Testing

We covered the basics and laid the groundwork for our own custom generator to handle arguments and options. Now we'll use those values to generate files. This is where we pick up the pace and start doing more heavy lifting with all of the powers that generators provide us.

Since we benefit the most when generators do as much work as possible while writing as little code as possible, we'll lean heavily on templates because that's where we get the most leverage. We'll start by digging into how templates work, and then we'll see how automated testing reduces the steps needed to verify the generated results.

We briefly addressed some of this at a high level earlier, but just as templates streamline the generation portion of our work, automated testing streamlines the rest of it. We can use a few test commands and assertions to automate verifying the existence and contents of our generated files, and we get all the standard benefits of automated tests.

We don't need to get bogged down in the configuration weeds, but understanding how and why configuration plays such a central role will shed light on the big picture. Fortunately, the built-in generator generator handles most of the configuration lines for us, but just because we don't *have* to understand them doesn't mean we *shouldn't*.

1.1 Templates

The core of the templates functionality originates from *Thor*, but Rails adds some magic to ensure they work smoothly with ActiveRecord, the file system, and tests. Structurally speaking, generators have three core elements: a source directory for template files, the generator itself, and a destination directory for the generated files.

The `template` action coordinates these elements by performing the file

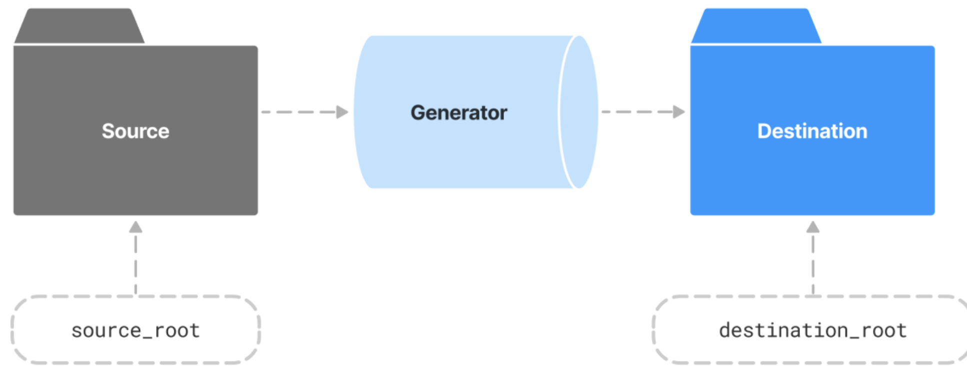

***Figure 1.1:** Generators look for templates in their source directory via the `source_root` variable and then create the generated files in the destination directory via the `destination_root` variable.*

system work and ERb parsing to generate the files. For a method that does so much work, the `template` action only accepts two parameters: the name of the template file and the relative path and file name for the destination.

```ruby
template "file.rb", "app/models/file.rb"
```

Conceptually, the arguments for the template method map nicely to the work it's doing, but it oversimplifies what's happening since Rails handles so much of the configuration for us. In day-to-day use, that frees us from having to think about how it works, but understanding how it works helps when it comes to troubleshooting.

template Source , Destination

***Figure 1.2:** The `template` action only needs a source and destination to generate a new file from an ERb template.*

Sources & Destinations

All of this templating functionality pivots on the source and destination configuration. Earlier, we saw how the values for the source (`source_root`) and destination (`destination_root`) are pre-configured when we use the built-in generator to create our generator. So now we'll do a quick refresher and dive into the details of how it all works.

When we use the built-in generator to start a generator, Rails automatically creates an empty `templates` directory to serve as the default `source_root` value for each new generator. And if we look in a generator's initial *Class* definition, we see one line of code which sets the `source_root` by pointing it at the `templates` directory.

lib/generators/obj/obj_generator.rb

```
1    source_root File.expand_path("templates", __dir__)
```

The `destination_root` represents our corollary to `source_root`. While the source is mostly relevant only for template-related actions, the destination plays a role in almost every file system change performed by a generator. The destination root points to `Rails.root` by default when running a generator, but the test cases re-route the destination root to the `tmp/generators` directory during tests.

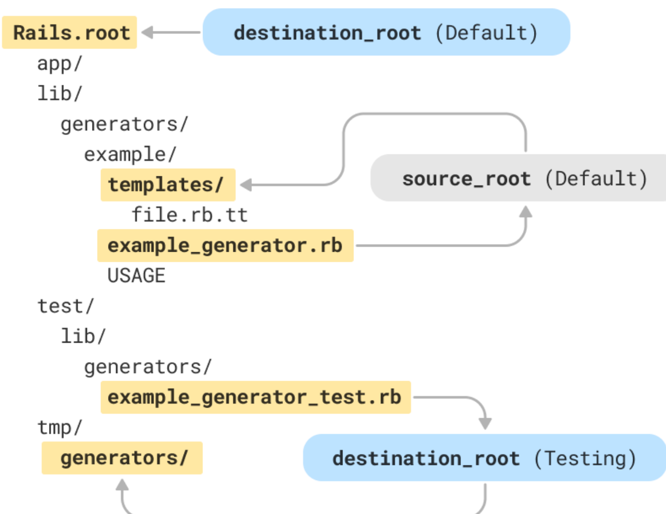

Figure 1.3: *New Rails generators will set the* source_root *set to the generator's templates directory and the* destination_root *to* Rails.root*. Tests override the* destination_root *put test-generated files into 'tmp/generators'.*

This customizable nature of the destination root helps with automated testing to ensure files generated by tests are disposed of properly. Thanks to that tiny change, we don't have to worry about junk files accidentally being generated within the application or committed to the repository.

These source and destination settings also save us from typing something complex and unreadable to something much easier to type and read. If Rails

generators weren't using `source_root` and `destination_root` values, the call to the `template` method would be much more verbose and much less readable.

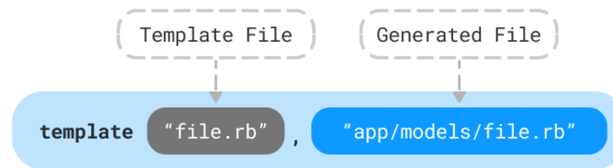

Figure 1.4: *The `template` method allows the source and destination parameter values to be as simple as they can be for the template file and the resulting generated file.*

```
# If we had to explicitly specify everything...
template "lib/generators/#{generator_name}/templates/file.rb",
         "#{Rails.root}/app/models/file.rb"

# vs. what generators enables us to specify...
template "file.rb", "app/models/file.rb"
```

Thanks to a combination of *Thor* and some Rails generators conventions leveraging `source_root` and `destination_root`, the `template` call is easier to type while also being more readable. And it makes testing easier as well because that `destination_root` value can be redirected to a disposable location during testing.

On the surface, all of this may look like it only saves some typing, but it also makes that `template` call much more readable. Without the directory and file extension obfuscating the template file name, we can more quickly recognize which file we're referring to.

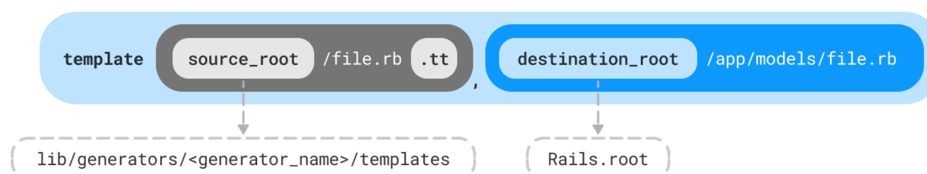

Figure 1.5: *The `template` method automatically prepends the `source_root` value and appends the `.tt` extension to locate a the source file. It also prepends the `destination_root` value for the destination.*

If you're underwhelmed so far, I get it. This all looks relatively simple once you see how it works, but *that's* the beauty of generators. They *are* that simple. The magic stems from how all the conveniences and conventions add up and work together to make it all seamless.

Now that we've established some basic context on how templates fit into the big picture, we can talk about the template files themselves, file names, and ERb.

Writing ERb in Templates

Generators will recognize the relevant files as templates to be read and parsed into the resulting generated file. In that way the `template` action serves as the glue within the generator because it touches all of the key elements. (We can also use the `js_template` action for JavaScript templates and the `migration_template` for migrations, but we'll cover those later.)

The ERb support in templates rounds out the magic, and the syntax should be familiar to anyone that's written ERb views in Rails. Any dynamic files in the templates folder need to have the `.tt` (short for "Thor Template") extension after the primary extension. But we don't have to specify the `.tt` extension when using the `template` action. Our generators just know, and they handle the extension for us automatically.

For example, with a file named `model.rb`, the `template` action expects the template file to be named `model.rb.tt` in the `templates` folder. The extension helps identify the file as a template, but more importantly, it prevents Rails from trying to autoload template files that also happen to be Ruby.

While the file name bits are handy, the real power of templates comes from the ERb. For simplicity's sake, you can think of any template file with the `tt` extension as ERb. Most templates should be simple enough that they won't require extensive or complex logic in the ERb since t friction of writing complex ERb in generator templates helps discourage getting too far into the weeds with the template files.

For most file types, ERb will work just like it does anywhere else, but things get a little meta when we use an ERb template to generate an ERb file. In that context, we need to escape the ERb delimiters (`<%`, `<%=`, `<%#`, `%>`, and `-%>`) that we want in the generated file.

To accomplish that, we add an extra percent sign to the opening ERb delimiter. So instead of `<%` to open an ERb block, we use `<%%`, which gets converted to the standard ERB delimiters. The same goes for the expression delimiter

(`<%%=` turns into `<%=` and `<%%#` turns into `<%#`) The generator will process any unescaped ERb, and while processing, it will convert the escaped delimiters to standard delimiters in the resulting file so we end up with working ERb.

We also have to be mindful of whitespace. Any whitespace or tabs in our template files will be carried over into the generated file. Unfortunately, where extra whitespace won't create significant problems in most documents, it can cause all sorts of sloppy indention in the generated files. While we normally wouldn't worry too much about extra whitespace in our views, generator templates need to be more precise since someone will need to read and edit the file at some point. And if we have arbitrary whitespace everywhere, that's a little more challenging.

With generator templates, this has two implications. First, if we indent our ERb, the whitespace in front of the ERb will be carried through to the generated file. As a result, ERb templates tend to be a little less elegant than a standard ERb file would be. For an example, let's look at the built-in partial scaffolding template file where we can see all of the executable EbB fully left-aligned. With simpler templates, this isn't horribly inconvenient, but with more complex templates, the lack of indention can make the files much more difficult to read and understand.

railties/lib/rails/generators/erb/scaffold/templates/partial.html.erb.tt

```
1  <div id="<%%= dom_id <%= singular_name %> %>">
2  <% attributes.reject(&:password_digest?).each do |attribute| -%>
3    <p>
4      <strong><%= attribute.human_name %>:</strong>
5  <% if attribute.attachment? -%>
6      <%%= link_to
7        <%= singular_name %>.<%= attribute.column_name %>.filename,
8        <%= singular_name %>.<%= attribute.column_name %> if <%= singular_name %>.<%=
   ↪  attribute.column_name %>.attached? %>
9  <% elsif attribute.attachments? -%>
10     <%% <%= singular_name %>.<%= attribute.column_name %>.each do |<%= attribute.singular_name
   ↪  %>| %>
11       <div><%%= link_to <%= attribute.singular_name %>.filename, <%= attribute.singular_name %>
   ↪  %></div>
12     <%% end %>
13 <% else -%>
14     <%%= <%= singular_name %>.<%= attribute.column_name %> %>
15 <% end -%>
16   </p>
17 <% end -%>
18 </div>
```

The other key to managing whitespace in our ERb files involves suppressing unneeded whitespace and newlines by adding a hyphen just before the closing tag delimiter. Looking back at the example, we can see that in addition to not being indented, the executable ERb tags end with `-%>` instead of `%>`. That ensures that our generated file doesn't have extra blank lines everywhere we have ERb. This only works if there are no other characters—visible or invisible—before the opening ERb delimiter. So we have to remove any extra leading whitespace and fully left-align our ERb if we want this to work.

Collectively, these caveats can be managed once we know to pay attention to them, but because complex ERb files quickly become difficult to read without indentation, it's as if they're whispering to us that our generator is trying to do too much when a generator becomes challenging to understand. And since we're already working with ERb inside of ERb, when the templates whisper to us, we should definitely listen.

By itself, using ERb in the templates is great, but generators make it a little nicer because any methods available within the generator are *also* available within the template files. So, to some degree, we can refactor complex ERb templates by moving some of the logic into the generator itself and then using the method name as a simplified placeholder in the template. That way, we reduce the amount of logic in the template, but it can also obfuscate what's happening because some of the template content will be defined in the generator file rather than the template file. When doing this, we have to remember to make the method private so that the generator doesn't try to run the method as part of its process.

Full access to the generator's public and private methods provides additional string-oriented conveniences by virtue of all of the available Rails' inflections that mean we have a plethora of name variations to use in both the generator and the template.

Rails Inflections

We briefly touched on the availability of inflections and related methods for the `name` argument earlier. Now we can fully expand on what that provides for us to work with both inside our generator and our templates. Let's revisit our contrived overly-name-spaced example again based on running `rails g my_generator Animal::Pet::Dog`.

Primary Inflections

```
human_name           # => "Dog"
singular_name        # => "dog"
plural_name          # => "dogs"
file_path            # => "animal/pet/dog"
file_name            # => "dog"
plural_file_name     # => "dogs"
fixture_file_name    # => "dogs"
class_name           # => "Animal::Pet::Dog"
class_path           # => ["animal", "pet"]
regular_class_path   # => ["animal", "pet"]
table_name           # => "animal_pet_dogs"
singular_table_name  # => "animal_pet_dog"
plural_table_name    # => "animal_pet_dogs"
i18n_scope           # => "animal.pet.dog"
```

Routes & URL Inflections

```
# Routes & Redirects
route_url                # => "/animal/pet/dogs"
singular_route_name      # => "animal_pet_dog"
plural_route_name        # => "animal_pet_dogs"
redirect_resource_name   # => "@animal_pet_dog"

# URL Helpers
url_helper_prefix                # => "animal_pet_dog"
index_helper(type: ...)          # => "animal_pet_dogs"
new_helper(type: ...)            # => "new_animal_pet_dog_url"
show_helper(arg=..., type: ...)  # => "animal_pet_dog_url(@animal_pet_dog)"
edit_helper(...)                 # => "edit_animal_pet_dog_url(@animal_pet_dog)"
```

Controller Inflections

```
controller_class_name    # => "Animal::Pet::Dogs"
controller_class_path    # => ["animal", "pet"]
controller_file_name     # => "dogs"
controller_file_path     # => "animal/pet/dogs"
controller_i18n_scope    # => "animal.pet.dogs"
controller_name          # => "Animal::Pet::Dogs"
```

If the patterns of the resulting strings seem familiar, that's because Rails uses them heavily for the built-in generators to create models, migrations, fixtures, tests, and controllers. When necessary, we can also include the `Rails::Generators::ResourceHelpers` module for the additional controller-related convenience methods in the last group.

Entire Directories as Templates

The `template` method is handy, but the `directory` method can do even more heavy lifting with a single method call. It does everything the `template` method does, but it does so recursively for every file and, optionally, for every sub-directory. Plus, it offers the added bonus of dynamically generating file names.

Any time I create generators and have the opportunity to use the `directory` action, I'm absolutely thrilled because it simplifies so much tedious work. It's a good sign that whatever I'm building is a perfect candidate for a generator. But enough build up. Let's look at an example.

```
1  #           Source    Destination
2  directory "icons", "app/assets/icons"
```

The method signature looks similar to the `template` method, but calling `directory` results in copies of each and every file in the `icons` folder having copies within `#{Rails.root}/app/assets/icons`. That is, the generator is copying all the contents rather than creating the source directory nested within the destination directory.

But the secret superpower comes from the fact that file names can use a `%method_name%` syntax, and the generator will run the specified method and replace that portion of the file name. So if we have a file named `%file_name%.rb` in the source directory, the generator will copy it and rename it according to the generator's value for `file_name`. So if we passed 'email' as the first argument to the generator, the generated copy of the file would be `email.rb`.

This isn't particularly amazing since the same thing could fairly easily be achieved with a call to `template` from within the generator, but there's something about one line in our generator recursively generating files from an entire directory and giving them the correct file name in the process that feels so efficient.

But that's not all. The destination path is technically optional, and leaving it off will create the copy in `destination_root` if not provided. Again, this is small, but when we're generating an entire directory, it's often not necessary to specify the destination. Revisiting our example, if we placed our files in the templates

folder using the directory structure we want in our destination, we could provide a single argument.

```
#         From templates...    -> Rails.root.join('app/assets/icons')
directory "app/assets/icons" # -> Destination Implied
```

And lastly, we can tell it not to run recursively. That way, we can put multiple directories in our generator's *templates* directory but copy each directory individually to a unique destination.

```
directory "images", "public/images", recursive: false
```

Let's look at a more complete example. Since our generated files frequently have tests, I'll often use sub-directories within the templates directory so the primary files and test files each correspond to a `directory` call.

Then in my generator, instead of three complex `template` calls, I'm able to use two simple `directory` calls, and the files will automatically be named when they're generated as well. Let's look at a rough example of handling several files with the `template` method, and then we'll see how a similar setup would work with `directory` method.

```
# Assuming three files in the templates folder...
# - model.rb.tt
# - fixtures.yml.tt
# - model_test.rb.tt

# With the template action...
template "model.rb",      "app/models/#{file_name}.rb"
template "fixtures.yml",  "tests/fixtures/#{fixture_file_name}.yml"
template "model_test.rb", "tests/models/#{file_name}_test.rb"
```

With `directory`, we create two directories in the templates folder and then add the files with their relative paths based on where we'd like them to end up. This requires more effort to organize the files within the templates folder, but it drastically simplifies the work within our generator. For just a few files, it's

likely not worth it, but as soon as we're using multiple files and target directories, it can be really nice to simplify the generator. As an added bonus with `directory`, we can also add new template files to either of the two primary directories in the future without having to modify any generator code.

```
# Assuming this directory structure in the templates folder...
# - model_files/app/models/%file_name%.rb.tt
# - model_test_files/tests/fixtures/%fixture_file_name%.yml.tt
# - model_test_files/tests/models/%file_name%_test.rb.tt

# With the directory action...
directory "model_files"
directory "model_test_files"
```

Like most scenarios, we usually have choices how we want to approach something. Both approaches handle a fair amount of heavy lifting, but `directory` can become very handy in cases where we're planning on working with numerous files or multiple distinct destination directories. But while `template` and `directory` represent the core examples, we have a couple of additional options for generating files from templates.

JavaScript & Migration Templates

In addition to the `template` action, we can also use `js_template` which can read from a template file relative to the generator's `templates` directory, process any ERb in the file, and save the generated file to the specified path relative to the `destination_root`.

It's not quite as flexible as the `template` method, and even the built-in Rails generators don't use it extensively as it's primarily a thin wrapper around `template` but removes the need to specify the `js` extension on the source and destination. It's not significantly more convenient, but it can be more intention-revealing.

```
def create_js_file_from_template
  #            Source    Destination
  js_template 'sample', 'app/assets/javascripts/sample'
end

# Roughly equivalent to...
def create_file_from_template
  template 'sample.js', 'app/assets/javascripts/sample.js'
end
```

In addition to `js_template`, Rails also provides a *Rails::Generators::Migration* module to include a few extra methods related to migrations. These methods simplify the process of generating migration files by removing the need for us to deal with the ever-changing timestamp at the beginning of migration file names.

We get `migration_template` action as well as `create_migration` that accepts a block representing the content and works similarly to `create_file`. We also get a some additional attributes related to migrations: `migration_class_name`, `migration_file_name`, and `migration_number`. We're not going to cover these in detail because they're not as widely useful in the average generator, but they're worth being aware of in case you ever need a generator that works with migrations.

While all of the template-oriented functionality provides the most magic, it's only part of the story. We can also create and modify files *without* using templates, and we can even run system commands or scripts directly from our generators. And for the icing on the cake, Rails provides us with dedicated test cases and assertions for generators that streamline the testing process.

We've seen how Rails handles sources and destinations when running a generator, but we also receive some generator-specific testing utilities and configuration values to help streamline testing.

1.2 Testing Generators

Now that we have a solid understanding of how and where files are generated, we can start to appreciate how automating testing provides just the right tools for ensuring our generators do what we expect. That means we'll see how to instantiate and run generators as well as look at the various assertions and

techniques we can use to verify that our generated files exist and to verify that they contain the content we want to generate.

Before we get into the actual testing, we need to talk about RSpec and generators. Like all of Rails, generators default to *Minitest*. *RSpec* options exist, but going deep down the customization rabbit hole just to use RSpec with generators can cut into our time savings. On the other hand, maintaining two separate automated test suites isn't a great long-term solution either.

Given that Rails provides dedicated *Minitest* assertions for generators, there's value and some convenience in sticking with the defaults. If your team is heavily invested in RSpec, however, you'll likely encounter some friction. That can be mitigated by using external gems or building some of your own tools to streamline testing generators, but both approaches have pros and cons.

Based on my research, the best resource is Thoughtbot's `suspenders` gem that they use to bootstrap new Rails applications. Looking through how they approach testing Rails generators with RSpec, it definitely provides the best examples I've found. It also includes quite a few generators and RSpec examples that leverage their custom generator matchers, generator test helpers, and file operations to streamline testing generators with RSpec.

That said, while the rest of this chapter will focus on *Minitest*, the context should still be useful in terms of understanding how it all works. Having a solid mental model of the internals can also help map the tooling to any custom *RSpec* equivalents you may choose to build.

Testing by Pretending

Before we get to automated testing, let's talk about how "pretending" can help with testing. We've briefly discussed how the `--pretend` option can streamline testing usability and interactivity (when a generator inherits from `NamedBase`), but it can also work as a sort of smoke test as we're building our generator. While automated tests are great at verifying results and quantitative things, sometimes we just want a quick lightweight smoke test to make sure it can run without errors or try out any interactivity we may have added.

Even thought it may not be listed as an official testing tool, running a generator with the `--pretend` option streamlines the process of seeing what it will feel like to actually run our generator without bogging us down in manual cleanup after every run. And if there's something really wrong with our syntax, we'll usually be able to make sense of the error more easily if we're running it

directly from the command line than if we're making assertions in our tests.

Automated Preparation & Cleanup

Now that we have the lay of the land, we can dive into how our automated tests integrate with generators. Revisiting our generator's initial test case from the test file, we have the three lines of configuration that work together to save us from dealing directly with our file system. We covered them earlier, but let's look at them again in light of everything we've learned up to this point. Together, they specify the generator to be tested while ensuring we have a clean destination directory before each test case.

test/lib/generators/obj_generator_test.rb

```
1  tests ObjGenerator
2  destination Rails.root.join("tmp/generators")
3  setup :prepare_destination
```

The first line establishes which generator we'll be testing. That way, the test helpers can reference the `generator_class` method and know which generator to run for the test. Then the `destination` declaration takes the first step in routing all the test-generated files to `tmp/generators` where we can be confident they won't pollute our codebase.

Note that even though it's setting `destination_root`, the declaration uses the `destination` class method for the test case. This line doesn't do all the work by itself, but we'll cover that shortly.

The `setup` method in the next line ensures that we have a clean destination directory before each test. By running it *before* rather than *after* the test, any files generated during the test stick around if we need to perform any manual inspection in our `tmp/generators` directory. That's really handy to remember because it enables us to run a specific test and see the actual results in `tmp/generators` when our automated tests fail. As inefficient as swimming around the file system is, sometimes manual inspection of results is the best way to troubleshoot.

Moreover, by re-routing test-generated files, we never have to worry about garbage files sneaking into our repository, and we never have to manually clean up the generated files and changes. Since `tmp` is *.gitignore*'d, the repository stays clean, and testing our generators takes milliseconds instead of seconds or minutes.

Running vs. Instantiating Generators

We've seen how our configuration told our test suite that it's testing the *ObjGenerator* so we don't need to specify the generator name in each of our tests. Instead, generator tests provide two methods to facilitate testing. One lets us create an instance of the generator so we can inspect state, and the other lets us run the generator so we can inspect results.

The `run_generator` method will be our primary option. It performs a full generator run so our tests can inspect the `destination_root` and generated files. The `generator` method, on the other hand, lets us create an instance of our generator without running it. That gives us a way to inspect the values to ensure they were translated from the command line to our arguments and options as we expected.

Other than saving us from needing to specify the generator name over and over again, these two methods also handle the last piece of our automated-cleanup puzzle. They override our `destination_root` so that files generated during tests end up in `tmp/generators` instead of `Rails.root`.

The key takeaway from this is that overriding the `destination_root` *only* happens if we test our generator using one of these two methods. If we were to use a direct system call to run our generator (i.e. `system "rails g model NAME"`), that call would use the generator's actual `destination_root` value and would no longer be re-routing the generated files to `tmp/generators` for us. In that scenario, not only will our files be generated outside of the `tmp` folder, but our tests won't pass since they'll still be looking in the `tmp` folder.

With that out of the way, let's see how they work. With `run_generator`, we can pass the values just like we would if we were calling it from the command-line. And if we use Ruby's `%w` syntax for creating a string array, we can pass the arguments and options precisely how we'd type them on the command line.

```
1  # Equivalent to the command line version:
2  #   rails g generator_name myapp --skip-active-record
3  run_generator %w(myapp --skip-active-record)
```

We'll usually want `run_generator` because it correlates more closely with how the generator would be called from the command line. But occasionally,

we want an instance of our generator so we can inspect its state prior to running it. We *can* technically use `generator` with tests and assertions, but I've found that the end-to-end nature of testing a complete generator run will create the mots reliable tests in the long run.

Unlike `run_generator`, the `generator` command accepts an array of arguments and a hash for options. Since we provide those values directly in Ruby, it can unintentionally mask issues where something might otherwise be lost in translation between the command-line string and the generator's parsing of that string. Since `run_generator` mimics the command line approach most closely, it provides the more complete integration test similar to how people will actually use the generator.

```
# Since the values are already Ruby, this can mask issues that
#   might otherwise arise from the generator parsing the values
#   from the command line and converting them.
args = [arg_one, arg_two]
opts = { one: 1, two: 2 }
g = generator(args, opts)

# A more accurate test of day-to-day usage...
run_generator %w[arg_one arg_two one:1 two:2]
```

Default Testing Arguments

When working with a generator where most of the test cases can use the same set of arguments, we can use the `arguments` method to declare default values for calls to `run_generator`. To accomplish that, we add the declaration right after the `setup :prepare_destination` line, and calls to `run_generator` no longer need arguments specified every time.

Default Arguments for Generator Tests

```
arguments %w(app_name --skip-active-record)
```

Note that even though it's called "arguments", it handles both arguments and options. Essentially, it provides the array of values passed to `run_generator` and `generator` calls. Keep in mind that it creates some indirection since the argument declaration won't be inline with the relevant method calls. It isn't a huge problem, but it can obfuscate things a bit in the future when debugging issues.

While all of the test configuration sets the table for us rather nicely, it's not only about getting a purpose-built testing setup for our generators. We also get some extra generator-specific assertions. In addition to the custom assertions, the *Rails::Generators::TestCase* class includes Ruby's *FileUtils* for us automatically. So when the assertions aren't enough, we can quickly turn to those lower-level tools as well. Fortunately, all the additional assertions have extensive documentation, but we'll go through them here to help get a feel for how they improve testing.

File and Directory Assertions

Rails provides some foundational assertions related to files and directories that provide convenient ways to verify that a file was (or was *not*) generated with `assert_file` and `assert_no_file`. These are also aliased as `assert_directory` and `assert_no_directory`. These assertions look in the specified destination for the test—which will be `tmp/generators` unless we change the default value provided by the generator generator. When not explicitly specified otherwise, any of the path-based values can be absolute paths or paths relative to the `destination_root`.

While they do verify the presence or absence of files/directories, that's not all they do. They can also verify content *within* the file—assuming a file is found. All together, we have three options for using these assertions. With the simplest version, we can easily verify that an expected file or directory exists. And like other path-related aspects of generators, the assertions play nicely with `destination_root`.

Testing only for Existence

```
1  assert_file "dir/file.rb"
2  assert_directory "dir"
```

To verify the presence of a string in a file's contents, we can check for the existence of the file and the presence of the string by passing the string as a second argument. Even better, the second argument can be a string *or* a regular expression. With a string, it expects an exact match, and with the regular expression, it performs a standard regex match.

Testing for Existence and Content

```
1  assert_file "dir/file.rb", "class Query; end"
2  assert_file "dir/file.rb", /Query/
```

These options can get us pretty far, but we can also pass a block to handle more complex assertions about the content—including multiple different assertions about the content. And for cases where we expect our generator to skip some files under certain conditions, we can also use `assert_no_file` and `assert_no_directory` to verify that they were not generated.

```
1  assert_file "model/user.rb" do |file_content|
2    assert_match(/class User/, file_content)
3    assert_match(/def first_name/, file_content)
4    assert_match(/def last_name/, file_content)
5  end
```

While the generic `assert_file` variations are great for most cases, we don't always have predictable file names in the case of things like migrations. For that, Rails provides `assert_migration` and `assert_no_migration` when we need to test them. Like the other migration-related utilities, the `assert_migration` method is similar to its siblings like `assert_file` but with a trick up its sleeve. Since migration files prepend timestamps to the migration name, we can make the assertion without having to specify the exact generated file name.

Verify that a Migration was Generated

```
1  assert_migration "db/migrate/migration_name.rb"
```

That way, the assertion will succeed regardless of the generated timestamp. If you're curious about the flexibility, it's using the `migration_file_name` method behind the scenes and checking to see if there's a file name in the directory that starts with a series of digits followed by the underscore and followed by the file name with the same extension.

File Content Assertions

While `assert_match` isn't a generator-specific assertion, it is useful when verifying the contents of generated files. Most likely, you've encountered or used it previously, but given its usefulness with generated content, it frequently helps in generator tests.

In addition to `assert_match`, we also get some specialized assertions to help cover common scenarios. For example, we have `assert_instance_method`

(aliased as `assert_method`) and `assert_class_method` to explicitly verify that the relevant method is defined. Even better, both assertions optionally yield the content of the method to a block for more detailed verification if a match is found.

As handy as `assert_instance_method` is, it only performs a naïve text search for the method name preceded by `def`. It is *not* inspecting the available methods for the generator. As a result, it will not discover methods defined dynamically via `method_messing` or other similar techniques.

Verify the Existence and Content of a Method

```
1    assert_instance_method :full_name, "[first_name, last_name].join(' ')"
```

Like `assert_file`, `assert_instance_method` can also yield the contents of the method to a block for more detailed verification of the method's internals, and we can even nest these to verify multiple methods in the same file.

Verify a File, Method, and the Method Contents

```
1    assert_file "model/user.rb" do |file_content|
2      assert_instance_method :full_name, file_content do |method_content|
3        assert_match(/first_name/, method_content)
4        assert_match(/last_name/, method_content)
5      end
6    end
```

In the case of `assert_class_method`, it works just like `assert_instance_method`, but it uses a naïve text search looking for methods defined with `self.` preceding the method name. Unfortunately, that means it will not recognize class methods defined in `class << self` blocks. We can, however, still use `assert_match` for cases where we define the class methods in a block. And like its siblings, `assert_class_method` can also yield the contents of the method to a block for more detailed verification.

```
1    assert_class_method :first, file_content_string
```

```
assert_file "model/user.rb" do |file_content|
  assert_class_method :first, file_content do |method_content|
    assert_match(/all/, method_content)
  end
end
```

Keep it Simple

Thanks to this minimal set of assertions, we're able to maintain a high level of confidence with very little effort. Of course, there's no real limit to what we can test or verify, but we'll see the most significant time savings the less code we write.

So whenever we find ourselves starting to use other assertions or creating complex custom tests in our generators, we may find that we're having our generators try to do more than those few tasks for which they're ideally suited. Just like with utilities, the lower-level we get, the more likely we are to waste time building things that won't be saving us time. So if a test starts to feel complicated, that's often a good sign that the generator is trying too hard.

1.3 Take Care with Locations

We've seen how our `destination_root` is a moving target depending on whether we're running our generator from the command line or running it within our tests. Regardless of the context, though, knowing that it can change depending on context might provide a hint at how crucial it is to ensure we reference it in path and file names in most situations.

Joining Pathname Elements

For simplicity, I've used basic string concatenation for the destination path up to this point, but there's one more thing worth keeping in mind. Let's look at an example `template` call for context. The source parameter is simple enough because it already knows *where* to look for that (the generator's `templates` directory), and we only need to provide the template file name. The destination, however, will be slightly more involved if we want to strive for cross-platform compatibility. While the path to the source file is handled automatically, we'll want to use `File.join` to specify the pathname for the destination in order to maintain maximum compatibility across operating systems.

```
1 template "task.rb", File.join("lib", "tasks", "#{file_name}.rake")
```

In this example, the destination path uses a `File.join` call instead of pure string concatenation. This is one of those small things that helps ensure maximum compatibility with Windows, Unix, or Mac because `File.join` uses the appropriate directory delimiter for the current system. ("\" on Windows and "/" on Unix/Mac.)

Navigating the File System

We've already seen how `destination_root` and `source_root` work with generators. Generally, the primary actions will be enough, and we can trust them to handle these locations for us. Every so often, however, we'll need to fall back to lower-level tools like *File* or *FileUtils*. Or we might need to work with destinations outside of `Rails.root`. In those cases, we want to have a good grasp of what's going on in the background.

For example, when I generate a new *Markdown* file for a blog post on my personal site, the generator checks to see if I've added images for the post. Since new images for posts will ultimately live in a directory that doesn't exist yet, I place them in `tmp/scratch`, and the generator knows to look there. Since these images are always unique to each blog post, the original images don't belong in the generator's templates folder—or version control for that matter. So I put them in a temporary folder for the generator to check and then optimize and move the images where they need to go.

With my custom generator knowing to check for images, it can then iterate over the images without knowing how many to expect. It can also optimize each one, move it to the correct long-term destination, and generate the corresponding code for each image directly in the newly generated *Markdown* file. But since these files live somewhat outside the standard generator workflow, we need to make sure our file system commands reference `source_root` and `destination_root` in the relevant locations when handling the paths. Otherwise, our tests will be overriding `destination_root`, and those tests will know nothing about the custom locations our generator is using.

We'll cover the details shortly, but the primary thing to remember about locations is that if we want to start tinkering with files *outside* the standard

source or destination locations, we'll want to explicitly specify paths relative to these locations.

Depending on the context, we might want to use the `inside` utility action to specify a working directory and pass a block that has the relevant commands, but when sources and destinations are involved, it can sometimes be handy to explicitly specify both source and destination since `inside` only affects the source directory. So we have options, but for explanatory purposes, we'll take this example all the way down to clarify the challenges with lower-level tools.

If we find we're doing this frequently, we can create some helper methods for them. On the other hand, frequently using tools other than those integrated into generators may be one of those signs that we're trying to shoehorn generators into doing something they're not necessarily great at. Any time I find myself using lower-level tools in generators, I'll often step back to think about whether there's a better way to handle it—or if it should be automated at all.

1.4 Our Generator: Adding Templates & Tests

Now we're able to start building out the core functionality of our generator. We can create templates and define the behavior that will generate our files, and we can add the tests that ensure they work as expected. We have endless options on how to approach this, so we'll start with the shell of our model template and iteratively make some enhancements that take advantage of the full capabilities of generators.

lib/generators/obj/templates/model.rb.tt

```
1 class <%= class_name %>
2 end
```

It's not much to look at, but we have a template file that uses the `class_name` inflection. But for our generator to do anything with this file, we need to add some logic within the generator. We already have our validation method `validate_accessors_count`, and since generators automatically run all of the public methods in the order they're defined, we'll want to add our model-generation method right after that with a call to `template`.

lib/generators/obj/obj_generator.rb

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

  # ... Constant, Argument & Option

  def validate_accessors_count
    # ...
  end

  def generate_object_file
    template "model.rb", File.join("app", "models", "#{file_name}.rb")
  end
end
```

We could run this right now with `rails g obj fancy name`, and we'd get our generated class file. But since we're far from done, we'll run it with `--pretend` to check-in and make sure we haven't made any significant mistakes. That way, we can be reasonably confident it works without having to worry about cleaning up incorrectly-generated files if something is broken.

```
$ rails g obj Fancy name --pretend
      create  app/models/fancy.rb
```

Now that we've got some basic confirmation that our generator can run, let's add a proper test. We'll want to verify that our file was generated and contains our class declaration. For that, we'll turn to `assert_file` and `assert_match`.

```ruby
test "generates the object file" do
  run_generator %w[Point x y]
  assert_file "app/models/point.rb" do |content|
    assert_match(/class Point/, content)
  end
end
```

Now we can run our generator tests with `rails t test/lib/generators/obj_generator_test.rb` to verify that it not only runs but actually generates the desired file. And since we'll want to

also generate a corresponding test file, we can go ahead and add a test for that as well.

```
test "generates the object test file" do
  run_generator %w[Point x y]
  assert_file "test/models/point_test.rb" do |content|
    assert_match(/class PointTest < ActiveSupport::TestCase/, content)
  end
end
```

And with a simpler generator like the one we're building, we'll see that with only two tests we're already repeating our arguments every time. That's perfectly fine, but we'll go ahead and pull those out into defaults using the `arguments` class method so we don't have to specify them every time. And with these changes, we've got our first tests—one passing and one failing.

test/lib/generators/obj_generator_test.rb

```
require "test_helper"
require "generators/obj/obj_generator"

class ObjGeneratorTest < Rails::Generators::TestCase
  tests ObjGenerator
  destination Rails.root.join("tmp/generators")
  setup :prepare_destination
  arguments %w[Point x y]

  test "generates the object file" do
    run_generator
    assert_file "app/models/point.rb" do |content|
      assert_match(/class Point/, content)
    end
  end

  test "generates the object test file" do
    run_generator
    assert_file "test/models/point_test.rb" do |content|
      assert_match(/class PointTest < ActiveSupport::TestCase/, content)
    end
  end
end
```

We're on a roll, but we want both of our tests to pass. So let's add the template for our test file and update the generator to process that template as well. We'll

start by filling in a little more information in the test file template, but we'll limit our ERb to the included inflections for now.

lib/generators/obj/templates/model_test.rb.tt

```
require "test_helper"

class <%= class_name %>Test < ActiveSupport::TestCase
  setup do
    @<%= singular_name %> = <%= class_name %>.new
  end

  test "the truth" do
    assert_predicate @<%= singular_name %>, present?
  end
end
```

And in our generator, we'll add a method to generate the test file. We could just as easily include it within the `generate_object_file` method, but I find generators are a little more self-documenting if each piece of logic has its own method and each method has a corresponding test.

lib/generators/obj/obj_generator.rb

```
class ObjGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("templates", __dir__)

  # ... Constant, Argument & Option

  def validate_accessors_count
    # ...
  end

  def generate_object_file
    template "model.rb", File.join("app", "models", "#{file_name}.rb")
  end

  def generate_object_test_file
    template "model_test.rb", File.join("test", "lib", "models", "#{file_name}_test.rb")
  end
end
```

Now if we run our tests again, everything should pass. (`rails t test/lib/generators/obj_generator_test.rb`) Our generator is finally starting to do some work, but it's not yet doing anything with our arguments or our `:comparable` option. So let's upgrade our templates by

expanding our generator's tests first. We'll start by having our test look for the `attr_accessor` declaration and the `initialize` method within our generated object.

```
test "supports specifying attr_accessor fields for the model" do
  run_generator
  assert_file "app/models/point.rb" do |content|
    assert_match "attr_accessor :x, :y", content
    assert_method :initialize, content do |method_content|
      assert_match "@x = x", method_content
      assert_match "@y = y", method_content
    end
  end
end
```

Now our tests will be failing again. So let's verify that they're failing, and we'll update our template to handle these new expectations. Our `attr_accessor` and `def initialize` lines can map our array values to comma-separated strings, but we'll want to loop through the assignments in the class `initialize` method since they each get their own line. And remember, since our whitespace will carry through to the generated file, all of our ERb blocks need to be left-aligned and terminated with `-%>` so they don't add unnecessary line breaks.

lib/generators/obj/templates/model.rb.tt

```
class <%= class_name %>
  attr_accessor <%= accessors.map { |name| ":#{name}" }.join(', ') %>

  def initialize(<%= accessors.map { |name| "#{name}:" }.join(', ') %>)
<% accessors.each do |name| -%>
    @<%= name %> = <%= name %>
<% end -%>
  end
end
```

With those changes, our tests should be passing again. Now we can add in our handling for our `comparable` option. That means we need to include the `Comparable` module and create a placeholder definition for the `<=>` method. And since our `comparable` option is boolean, we'll add one test with the option included and one specifying not to include it.

Since we specified defaults arguments for our tests as `%w[Point x y]`, we'll need to explicitly specify the arguments to `run_generator` for these tests so they include `--comparable` and `--no-comparable` respectively. With aliases, it can often be handy to test them as well purely to increase the chances that a test will fail if there are conflicts with other aliases. Let's add some failing tests, and then we can update our template to satisfy the tests.

```
test "supports specifying --comparable for the model" do
  run_generator %w[Point x y --comparable]
  assert_file "app/models/point.rb" do |content|
    assert_match(/include Comparable/, content)
    assert_instance_method '<=>', content
  end
end

test "supports specifying or --no-comparable for the model" do
  run_generator %w[Point x y --no-comparable]
  assert_file "app/models/point.rb" do |content|
    assert_no_match(/include Comparable/, content)
    assert_no_match("def <=>", content)
  end
end
```

We're adding two `if` statements to the template and checking `comparable?` so we know whether our generated model should include the relevant code or not.

lib/generators/obj/templates/model.rb.tt

```
class <%= class_name %>
<% if options.comparable? -%>
  include Comparable

<% end -%>

# ... Accessors and Initializer

<% if options.comparable? -%>
  def <=>(other)
    # self.<value> <=> other.<value>
  end
<% end -%>
end
```

With these template updates, our tests should be passing again. We've skipped one minor detail from our original object design, though. We want or objects to

define `to_h`, `to_a`, and `to_s` by default. So we'll add those in along with the corresponding assertions so we can see our finished model object. In the case of the tests, instead of adding a new test case, we'll add a few `assert_instance_method` calls in our object test.

test/lib/generators/obj_generator_test.rb

```
require "test_helper"
require "generators/obj/obj_generator"

class ObjGeneratorTest < Rails::Generators::TestCase
  # ... Configuration

  test "generates the model file" do
    run_generator
    assert_file "app/models/point.rb" do |content|
      assert_match(/class Point/, content)
      assert_instance_method(:to_h, content)
      assert_instance_method(:to_a, content)
      assert_instance_method(:to_s, content)
    end
  end

  # ... Other Tests
end
```

And since our tests will be failing without defining those methods, our model template needs some updates to ensure all of relevant methods will be generated. We'll add the methods, and between all three methods, we only need a single stretch of ERb. The rest can be plain text because it will always be the same as far as the generator is concerned.

lib/generators/obj/templates/model.rb.tt

```
class <%= class_name %>

  # ...

  def to_h
    { <%= accessors.map { |name| "#{name}: #{name}" }.join(', ') %> }
  end

  def to_a
    to_h.values
  end

  def to_s
    to_h.values.to_sentence
  end

  # ...

end
```

This is all great, but *something* is still missing. Given the relatively narrow scope of our generator, we could proactively generate some tests to go along with our generated object. For many scenario's, auto-generating tests would fall into the category of over-doing things, but we'll go ahead and create some as an example of generated related code in separate files. Let's go back and add a little bit to our object's test template.

lib/generators/obj/templates/model_test.rb.tt

```
require "test_helper"

class <%= class_name %>Test < ActiveSupport::TestCase
  setup do
    @<%= singular_name %> = <%= class_name %>.new(<%= accessors.map { |name| "#{name}: nil"
↪  }.join(', ')%>)
  end

  test "the truth" do
    assert_predicate @<%= singular_name %>, :present?
  end

  test "supports common Ruby functionality" do
    assert @<%= singular_name %>.to_h.is_a?(Hash)
    assert @<%= singular_name %>.to_a.is_a?(Array)
    assert @<%= singular_name %>.to_s.is_a?(String)
  end

<% accessors.each do |name| -%>
  test "exposes getter/setter for <%= name %>" do
    assert @<%= singular_name %>.respond_to?(:<%= name %>)
    assert @<%= singular_name %>.respond_to?(:<%= name %>=)
  end
<% end -%>
end
```

These aren't the most robust tests, but they provide a little coverage of the key elements. In most cases, we probably wouldn't generate tests that do much more than ensure the generated file works. So if it's generating a Ruby class, we'd probably want to make sure an instance can be successfully instantiated. Going much further often results in people having to delete a bunch of irrelevant boilerplate code. So if ever there's a case where less is more, this is it.

Proactively generating tests often works best for providing minimal placeholders that serve as an outline for tests that are likely to be necessary. For example, if we create a generator for query objects that builds the class with an initializer, a `run` method, and a `results` accessor, we might define placeholder tests for instantiating a query, running it, and then one test each for successful results and and empty results. Then, the generator is still helping, but it's not pre-defining a bunch of code that's likely to just be deleted.

At this point, we've built a simple but useful generator, and we've covered all of our foundational bases. From here on out, we'll get into some of the lower-level functionality and tools that provide more granular control over our generator's behavior.

1.5 Summary & Review: Templates & Testing

If we're serious about saving time, generator template actions hold the key. They require some up-front effort before they become second nature, but once they do, generators can save time in endless scenarios. Let's run through what we've covered and how they play such a significant role.

- **The `template` and `directory` actions are superpowers.** They handle an incredible amount of work under the hood so we barely even have to think about it. Reach for them first, and only fall back on lower-level utilities when absolutely necessary.
- **Remember that the `directory` action supports dynamic file names.** Using sub-directories in the `templates` folder—one for code and one for tests/fixtures—can make it more practical, but sometimes multiple `template` calls are good enough.
- **Make the most of higher-level actions when possible.** Since the built-in actions handle sources and destinations seamlessly, straying from those puts more burden on you to use `source_root` and `destination_root` correctly with lower-level tools.
- **Remember to escape ERb delimiters when generating ERb.** Add an extra `%` in ERb delimiters when you want to generate an ERb file.
- **Pay close attention to whitespace and use `-%>` to avoid unnecessary new lines.** ERb templates will render any additional whitespace into the generated file, so ERb sections need to be fully left-aligned and end with `-%>` to avoid generating excess whitespace.
- **Templates can access everything the generator can.** Anything that we can access from within the generator code can be accessed from within the template as well. That includes all of the Rails inflections as well as public or private methods from the generator.

While templates are great, they aren't a superpower unto themselves without the wide array of generating-testing tools we get. With generators, automated tests are the real superhero because testing file generation requires an absurd amount of tedious file and content verification without tests.

- **Use `--pretend` to test manually.** Automated tests might be the best option, but occasionally we have to get our hands dirty with manual testing and verification—especially with generators that use prompts.

- **You can always dig into the `tmp/generators` folder.** When possible, running a single test will leave the generated files around since they're only cleaned up before each test. That makes it a little easier to investigate manually when something's not working.
- **Keep your eyes on the `destination_root`.** The tests override it for good reason, but even using a `FileUtils` method once can create chaos if we're not careful to apply the `destination_root`.
- **Tests only override the `destination_root` when the `generator` and `run_generator` commands.** Even though test cases specify a different destination root, the change will only affect generators run using the test helpers.
- **Running generators using `run_generator` is the most complete option.** Running a generator behaves more like an end-to-end test since the arguments and options are passed similarly to how the command would be entered from the command line.
- **Instantiate generators using `generator` to inspect state.** Running a generator can make it challenging to inspect state and ensure everything checks out before it runs. In those cases, we can instantiate a generator to peek inside without running it.
- **Make use of default arguments where possible.** For simple generators, providing default arguments can streamline testing a bit, but separating the arguments from the test can make it less obvious when an argument causes problems.
- **The file and directory assertions are location-aware.** Just like with the actions, the generator-specific assertions seamlessly handle source and destinations for us. The more we drift from the built-in tools in the generator or tests, the more likely we are to have some wires crossed.
- **Always verify there's at least some content from a file.** Writing assertions for every specific type of content isn't necessary, but we want to at least make sure the files aren't blank. So adding even a single content check can help catch indirect problems.

These tools are powerful as long as we go with the flow, but listen to the tests and generator if things start to feel complex. Straying from conventions can bog us down in ways that lower the value of the generator if we have to spend too much time debugging complexity. We can't go wrong with simple and easy-to-test generators. When a creating a generator starts to feel slow, focus on generating the basics, and leave the rest to the humans. Remember, we only need to give people a springboard rather than carry them the whole way.

…that's just a brief extract.