FRICTIONLESS GENERATORS

Effortlessly automate repetitive code generation in your Rails apps.

Garrett Dimon

Frictionless Generators

by Garrett Dimon

Copyright © 2024 Start & Sustain LLC

1 Save Time and Reduce Tedium

Before we begin, I need to clarify one detail. This book isn't about *building* custom Rails generators. It's about *leveraging* custom generators to save time and reduce tedium. That may sound like a distinction without a difference, but I encourage you to take a moment and think about whether anyone would regularly build or use a generator that *didn't* save time and reduce tedium—I know I wouldn't.

So, yes, we *are* creating generators, but the results will only be valuable in the context of saving time. To that end, we won't simply discuss how to write code for generators but how to design and implement generators to be net time savers for us and everyone we work with.

And since you're reading this, the book assumes that you at least have a passing familiarity with the built-in Rails generators like scaffold, resource, model, migration, or others. And, if you've used those, you probably *continue* to use at least some of them even if you could always choose to do the same work manually.

You may not fully understand *how* they do what they do, but you likely appreciate *what* they do—save time and rescue us from tedious and mindless work. That's the gap for us to bridge. We'll break generators down and examine them so we can more easily recognize opportunities to build custom generators as well as develop the skills we need to build them efficiently. For that, we need to really think about why we build generators—both the obvious and not-so-obvious aspects.

Generators primarily save time by reducing direct manual effort, but they also save time *indirectly* in ways that aren't easily recognizable. For example, automating a particularly error-prone task can save even more time than a less error-prone task. Since the generator will have automated tests and reduce the chance of regressions or manual mistakes, we spend less time

debugging and fixing issues.

Custom generators can also provide a great way to capture common shared patterns so they're more approachable and less intimidating for junior developers and new team members who may not be as familiar with them. That can be empowering for them while also reducing the need for manual documentation or frequent assistance from other team members. And for each way they streamline processes for less-experienced team members, they also help the experienced team members stay in the zone by delegating the boring and repetitive tasks to the machines.

In many ways generators can serve as a substitute for several categories of documentation. By codifying patterns and tools in a way that manual documentation can't, generators can have automated tests that wouldn't be possible with prose-based documentation. Moreover, it will usually be faster for someone to run a command than read a page of explanatory prose.

For example, when performing a generator-friendly task manually, a new or junior team member may need assistance from other team members. When that happens, we're almost doubling the amount of time spent since a second team member becomes involved with the process. In many cases, that also involves an interruption and the related context-switching costs. So if teammates regularly require help with a specific task, automating it with a custom generator might be worth it if only to reduce their need for additional help.

While codifying practices into scripts and tools can require additional documentation and education to ensure that everyone is aware of the tools, generators are already a familiar tool for Rails developers. That means that compared to one-off scripts, generators can be more inviting and welcoming to junior and new team members since they'll already know how to look for and use generators. No need to reinvent the wheel, right?

And finally, mind-numbingly tedious and manual tasks can break our concentration and take us out of the zone. Or we might be tempted to put it off because other tasks might feel less misery-inducing. So accounting for delays, interruptions, and that return trip to the zone could lengthen the average time as well.

Despite all of the *potential* upside with generators, we also have to acknowledge the cost of the up-front time required to create one. The time has a cost,

but if we choose wisely, it's more like an investment. And like any investment, spending two hours to save five minutes would be silly. But spending thirty minutes or less to save five minutes each of the five times someone performs a task each week, and that's a multiplier where the math quickly starts to work out.

While infrequent tasks may need to be especially time-consuming to justify automation with generators, shorter high-frequency tasks can also be more time-consuming due to context-switching costs. And just as frequency can increase with the number of teammates, those context-switching costs will go up as well.

Even with a finished, well-tested, working generator, we never get a guarantee that it will see enough use to generate a good return if it's too difficult to find or use. And with every generator, we still have to remember and type the command accurately or spend time refreshing our memories by perusing the documentation. Fortunately, when compared to the time required to learn and perform a manual process, it's rare for the generator-based approach to take anywhere near as long. And since generators provide a familiar syntax for Rails developers, turning to them means they'll be inherently discoverable alongside all of the other generators.

Collectively, these considerations aren't easily quantifiable. We can't just run a stopwatch once and then multiply it by a frequency. That wouldn't account for all the other ways manual tasks can take longer and impact our productivity. And if we're talking about a team, the chances are good that each team member would perform the task differently and require differing amounts of time.

But just because generators *can* save time and reduce tedium doesn't guarantee they *will*. If we sink too much time into creating a generator or waste time over-building complex or confusing generators, we could have been better off not building one at all. If a generator is difficult to use, it may never be used enough to justify the development effort.

Clearly, we can benefit from the right generators, but that up-front development cost can get in the way. With that in mind, we can start to see how reducing the time required to create one becomes critical to making the math work. The more efficiently we can create a custom generator, the sooner we start recognizing net time savings, and if we're able to create a useful and reliable generator in thirty minutes, there are few scenarios where generators don't generate a great return. That's our main goal. Lower that up-front cost so low that the option of performing recurring tasks manually rarely merits

consideration.

If reducing the up-front time required to build a generator is the goal, then becoming knowledgeable and skilled with generators is certainly a key part of that process. If, however, we maintain too narrow of a view of which tasks can be handled with custom generators, we'll still miss out on a wide variety of opportunities. Similarly, if we spend too much time over-building, the time savings may never exceed the up-front development costs. Furthermore, if we limit our imagination to only generating application code, we won't be able to see many generator-friendly tasks that don't touch application code.

Enough with the theory and background. Let's explore some real code. If we look at the most well-known built-in generators, we might believe they need to be complex and deeply integrated with every part of Rails to save meaningful time. While the more advanced generators for models or scaffolding receive the bulk of the attention, Rails also provides some simpler generators like *benchmark* and *task* which generate one file each and don't involve *ActiveRecord* at all.

The code and behavior of smaller generators might tempt us to dismiss them as too simple, but that simplicity is precisely the point. We can create small and simple generators with much less effort, and they'll be easier to maintain as well. They're easier to test. They're easier to learn and use regularly. And if we want a simple generator to do more, we can always add on to it with very little effort.

Imagine we want to create a new rake task file that has four tasks: cleaning the cache, clear out the "tmp" folder, rebuild assets, and reset our development and test logs. To start, we'd likely want to put together a basic *Rake* file structure that has a namespace and placeholders for each of the tasks.

lib/tasks/reset.rake

```
namespace :reset do
  desc "TODO"
  task cache: :environment do
  end

  desc "TODO"
  task tmp: :environment do
  end

  desc "TODO"
  task logs: :environment do
  end

  desc "TODO"
  task assets: :environment do
  end
end
```

At a glance, this file might appear to be too simple to justify a generator, and we could probably create it manually each time without too much trouble. On the other hand, we have a fair amount of repetition in the resulting file. If we only consider the complexity of the generated content, however, we don't see the handful of other steps involved. And those steps change the equation just enough for otherwise simple files.

The first point of friction happens because we also have to create the file in the right location. That involves naming the file and knowing where to place it. For an experienced Rails developer, that may seem trivial, but that's not always the case for a junior developer or even a senior developer that's new to Rails or a given codebase. They might not know that rake files go in `lib/tasks`, might need to find similar code to copy and paste, or they might need to go read up on how to structure rake tasks. All of those little pieces of friction start to add up, and it's the collective inefficiency that contributes to making generators worthwhile.

With all of that context out of the way, we're going to start creating some code, and as we start running generators, we should sync up about the generator command and examples throughout the book so there's no confusion about running the `generate` command.

Depending on your local setup for both Ruby and Rails, you'll either use `bin/rails` or `rails` to run the generator generator. The former ensures that the call will use your application's version of Rails, but the latter is more concise and saves characters in examples. Similarly, just as most of the key Rails commands can be run with just the first letter of the command, the `generate` command works the same way. So instead of `generate`, we can use `g`.

```
# The verbose way to run a generator...
bin/rails generate generator obj

# The shorthand we'll use...
rails g generator obj
```

For the rest of the book, we'll use "g" to save space and typing, but either option works. Running our generate command creates a handful of files and directories for us. So use whichever you're most comfortable with, but recognize from here on out that we'll use `rails g` in order to save space and keystrokes. And for the examples where we run tests, we'll shorten `bin/rails test` to `rails t` as well. But don't hesitate to use `bin/rails g` if that creates more consistent results based on your configuration.

Now we can plan an ideal command-line syntax for generating our *Rake* file. Regardless of a developer's familiarity with *Rake* files, it would be nice if we could type something like `rails g task reset cache tmp logs assets` to generate that *Rake* file for us.

Fortunately, that's what the built-in `task` generator gives us, and it's a great place to start since it only generates one file. Don't worry too much about understanding the code in detail just yet. We'll get to that later. For now, focus on the fact that we have a useful generator with one declaration for an argument and a single, one-line method.

/railties/lib/rails/generators/rails/task/task_generator.rb

```
module Rails
  module Generators
    class TaskGenerator < NamedBase # :nodoc:
      argument :actions, type: :array, default: [],
        banner: "action action"

      def create_task_files
        template "task.rb", File.join("lib/tasks", "#{file_name}.rake")
      end
    end
  end
end
```

That's only the generator itself, though. The other half of the magic comes from a relatively short ERb template file. Again, don't focus on the specific code just yet. For now, just note that the template contains some basic ERb. Even better, templates have full access to all of the attributes and methods available from within the generator.

/railties/lib/rails/generators/rails/task/templates/task.rb.tt

```
namespace :<%= file_name %> do
<% actions.each do |action| -%>
  desc "TODO"
  task <%= action %>: :environment do
  end

<% end -%>
end
```

Regardless of their simplicity, those two files streamline a lot of tedious work. Not much code and barely any logic? It's like a free running start! Even better, that template file is incredibly similar to the code we'd have to write when creating the file manually anyways. So the effort to create a generator for this is only marginally more than it would be creating the file manually that first time. And once you know generators well enough, that effort is arguably trivial.

Not all generators *will* be this short, but many custom generators *can* be this short. Moreover, this is exactly how we want *all* of our custom generators to start. That approach means we can be sure they save time before we invest too much effort in them. We always have the option to enhance them later if we think it's worth it.

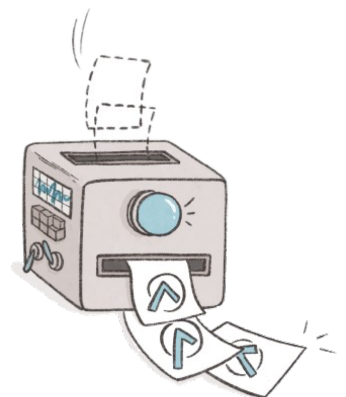
In the interest of full disclosure, those two files only represent the core of the task generator. Frictionless generators also need tests. Fortunately, Rails generators get their own test cases and assertions to streamline testing so that it doesn't drastically add to the effort. Those specialized tools for automated testing are so central to making it all work, we'll cover them extensively in a later chapter.

So it's alright for generators to be small, but limiting ourselves to small generators would be selling ourselves short. If we want to maximize our time-saving potential, we still need to expand our ability to recognize generator candidates by imagining various ways they can help us.

While most generators create application code, there's no requirement to do so, and limiting ourselves to generating application code ignores countless opportunities. As we saw, generators can be used for tasks and benchmarks. They can just as easily generate reports, documentation, test data, seed data, or update a *README*. If we ever find that we repeatedly need a certain text-based file, there's a chance we've got a generator candidate. (This book was created by a generator that converts *Markdown* files and assembles the results into a PDF and ePub.)

Even less code-like, many teams keep a "docs" directory in the root of the repository for plain-text or *Markdown* files. It provides a predictable home for basic development documentation around setting up development environments, running tests, as well as vendor and dependency knowledge.

Depending on how far we want to take it, generators can help streamline that process because we could create a generator for documentation templates. We could run a command like `rails g vendor VendorName` and get a prefilled template file at `docs/vendors/<vendor_name>.md`. We could even add logic to ensure that a list of vendors at `docs/vendors.md` automatically includes a reference to our newly added vendor. This kind of text file work is precisely where generators shine.



That vendor template could include placeholders for the website, status site, support site, and documentation links. It could include the contact information for our primary vendor contact, as well as the internal team member most familiar with the vendor. It could encourage capturing knowledge about how integral the vendor is, which other vendors were considered, why we chose the vendor, or even what might make

it worth switching to another vendor.

And to take all of this a little further, we could create a generator to upgrade the `bundle add <gem>` command. That generator could use the various Gemfile actions available to generators to handle adding the gem and relevant details to the Gemfile, and then it could automatically create a new markdown file at `/docs/dependencies/gems/<gem>.md`. Now when we go to commit our updates, we're reminded we have an otherwise empty template file that could use some information before we add another dependency to the code-base.

Not every team needs or wants to spend time capturing this kind of information, but that's not the point. These are only basic examples of tasks that nobody enjoys. And when those tasks are even remotely tedious, we'll phone it in or skip it entirely. If it's seamlessly integrated into the process of setting up new vendors using a generator and the generator does most of the tedious work, it creates new opportunities to more easily capture the types of knowledge that tend to be lost over time.

Documentation can be tedious or easily overlooked, but generators can remove the friction while nudging team members to fill in the kind of knowledge that might otherwise be lost over time.

Documentation is only one example, however. I use a generator to start new blog posts since they're just *Markdown* files in my site's repository. Generators can just as easily be used to generate YAML, CSV, CSS, JavaScript, or any other kind of plain text. We could even use them to optimize or resize images. Once we start thinking of opportunities beyond the basics and not limiting ourselves to generating Ruby, our potential time savings expand significantly.

And with all of that we have some context for creating helpful custom generators that save time and reduce tedium. And while I've been banging the time-saving drum pretty heavily, there's one more—admittedly subjective—bonus with this kind of automation. The categories of tasks that lend themselves to automation with custom generators also frequently happen to be the tasks that are the least enjoyable to perform manually.

So we're not just going to save time. We're going to (hopefully) enjoy our work more too. That may sound like I'm over-selling them, but I promise I've had way more fun building generators than I have manually creating files and

doing the copy-paste-search-replace dance. Hopefully this book can do the same for you by saving time *and* having a little more fun. Save time. Reduce tedium. Enjoy work a little more.