```
  value = data.external.echo.result.foo
}
```

This example uses the `external` data source to execute a Bash script that echoes back to stdout any data it receives on stdin. Therefore, any data you pass in via the `query` argument should come back as is via the `result` output attribute. Here's what happens when you run `terraform apply` on this code:

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

echo = {
  "foo" = "bar"
}
echo_foo = "bar"
```

You can see that `data.external.<NAME>.result` contains the JSON returned by the external program and that you can navigate within that JSON using the syntax `data.external.<NAME>.result.<PATH>` (e.g., `data.external.echo.result.foo`).

The `external` data source is a lovely escape hatch if you need to access data in your Terraform code and there's no existing data source that knows how to retrieve that data. However, be conservative with your use of `external` data sources and all of the other Terraform "escape hatches," since they make your code less portable and more brittle. For example, the `external` data source code you just saw relies on Bash, which means you won't be able to deploy that Terraform module from Windows.

Conclusion

Now that you've seen all of the ingredients of creating production-grade Terraform code, it's time to put them together. The next time you begin to work on a new module, use the following process:

1. Go through the production-grade infrastructure checklist in Table 8-2, and explicitly identify the items you'll be implementing and the items you'll be skipping. Use the results of this checklist, plus Table 8-1, to come up with a time estimate for your boss.

2. Create an *examples* folder, and write the example code first, using it to define the best user experience and cleanest API you can think of for your modules. Create an example for each important permutation of your module, and include enough documentation and reasonable defaults to make the example as easy to deploy as possible.

3. Create a *modules* folder, and implement the API you came up with as a collection of small, reusable, composable modules. Use a combination of Terraform and other tools like Docker, Packer, and Bash to implement these modules. Make sure to pin the versions for all your dependencies, including Terraform core, your Terraform providers, and Terraform modules you depend on.

4. Create a *test* folder, and write automated tests for each example.

That last bullet point—writing automated tests for your infrastructure code—is what we'll focus on next, as we move on to Chapter 9.

---

[1] Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, 20th anniversary ed. (New York: Basic Books, 1999).

[2] Seth Godin, "Don't Shave That Yak!" Seth's Blog, March 5, 2005, *https://bit.ly/2OK45uL*.

[3] Frederick P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, anniversary ed. (Reading, MA: Addison-Wesley Professional, 1995).

[4] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. (Upper Saddle River, NJ: Prentice Hall, 2008).

[5] Peter H. Salus, *A Quarter-Century of Unix* (New York: Addison-Wesley Professional, 1994).

[6] You can find the full details on publishing modules on the Terraform website.

7 You can find the full list of provisioners on the Terraform website.