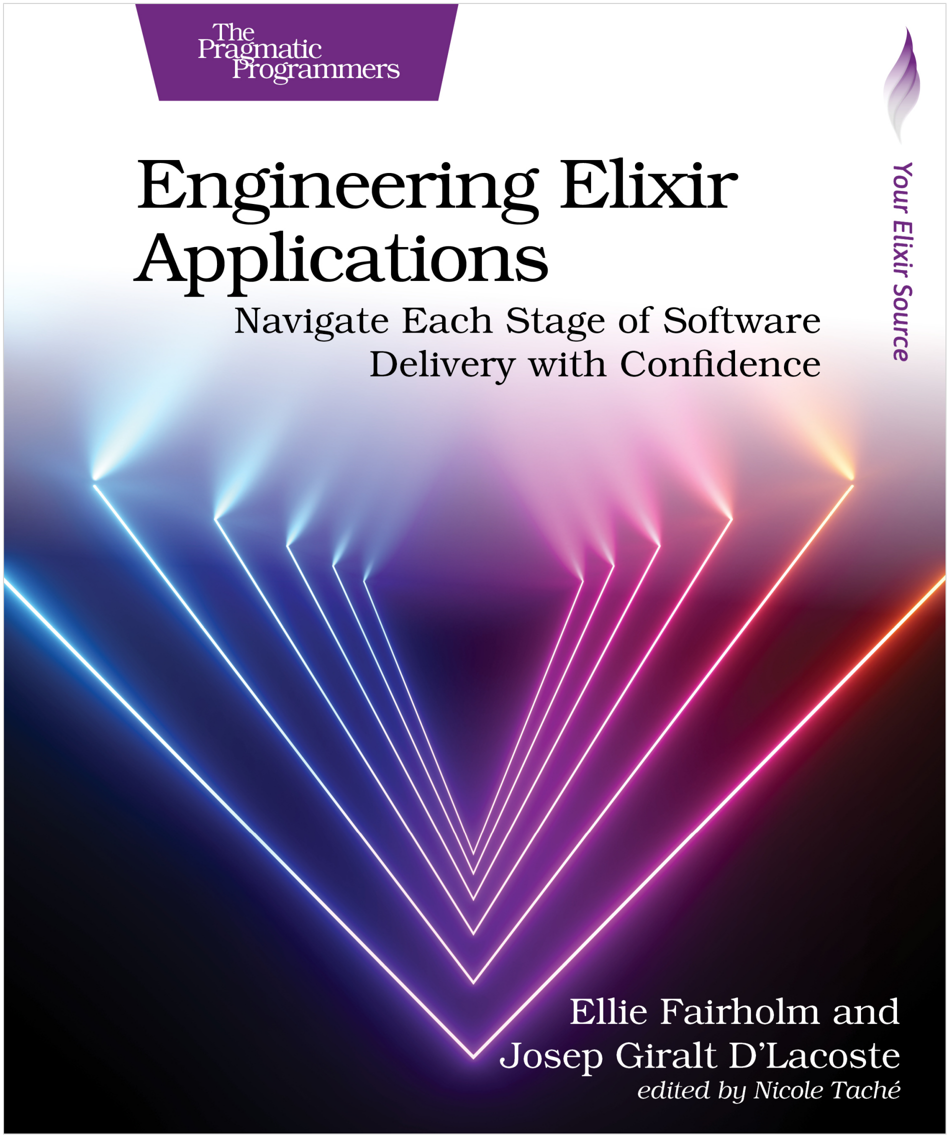
The Pragmatic Programmers
Engineering Elixir Applications
Navigate Each Stage of Software Delivery with Confidence
Your Elixir Source
Ellie Fairholm and Josep Giralt D'Lacoste
edited by Nicole Taché

Resource Dependencies

Infrastructure resources are often reliant on each other and can only exist if a separate resource exists first. This is what is called a *resource dependency*. There are two ways that you can tell Terraform that one resource depends on another: implicit or explicit dependencies. Implicit, or hidden, dependencies are created by referencing one resource's exported attribute as an argument when declaring another. Terraform itself will then automatically handle the link between resources. You'll see an example of this later in the chapter.

The second method is more explicit and uses the depends_on meta-argument. As the name would suggest, adding the depends_on meta-argument to a resource tells the Terraform Core that this resource is reliant on another and either needs to be created after or deleted before the dependent resource. So, let's now execute the IMPROVE part of the CREATE–DESTROY–IMPROVE–REPEAT cycle and add the depends_on meta-argument to both your milestone and label resources in your main.tf file:

```
# in modules/integrations/github/project_management/main.tf

resource "github_repository_milestone" "epics" {
  depends_on = [github_repository.kanban]
}

resource "github_issue_label" "issues_labels" {
  depends_on = [github_repository.kanban]
}
```

In the example, the depends_on argument uses a resource address rather than the local.github_repository value that you created earlier. This is because the depends on meta-argument expects a resource address as its value to be able to link specific resources. If you were to have used the local value, the Terraform Core wouldn't be able to create the dependency.

Now that you've added this meta-argument, you can rerun terraform destroy and there will be no dependency errors as Terraform now knows in which order the resources should be created and deleted. Similarly, you can then recreate all of your resources by running another terraform apply. Go ahead and run both of these commands.

Great! You've just ensured that your Terraform configuration is working, is reusable, and is idempotent. This CREATE–DESTROY–IMPROVE–REPEAT cycle is the way to go. Get used to it, pragmatic engineer!

Now let's move on to creating the last resource: GitHub issues. We'll do so by looking at a slightly different way of using Terraform variables.

Create Your Fourth Resource: GitHub Issues

Earlier in this chapter, you used a map to define your variable block for the milestones and labels as a way to keep your Terraform configuration lean and clean. However, a Terraform variable doesn't need to be a map, so let's look at another way in which you can use variables to parametrize a Terraform resource. Rather than using a map to create the issues, you're going to use a list. Each item in the list will have a title, a body, a list of labels, and a parent milestone. As you did when defining your milestones and issues variables, you must first create your variable block. Open up your main.tf file and define your new issues variable under your github_issue_label resource as we've done here:

```
# in modules/integrations/github/project_management/main.tf

variable "issues" {
  type = list(object({
    title     = string
    body      = string
    labels    = list(string)
    milestone = string
  }))
}
```

As you can see in the previous example, the issues variable is a list of objects. Each issue has a title, a body, a list of labels, and a milestone. This milestone will be used in the resource definition to assign each issue to a specific milestone. We've chosen to use a list rather than a map so that you can keep your .auto.tfvars file tidy. In the next section, we'll discuss how to link your list variable to your issue resource.

The Count Meta-Argument for GitHub Issues

As you saw previously when using a map or set variable, you have to use the for_each meta-argument in your resource definition to iterate over our map in a key-value fashion, but when dealing with a list, you must use the count meta-argument.

The count argument takes a number as its value. This number should be equal to the length of the variable you wish to iterate over, which in this case is the issues list. You can obtain the length of the list using the built-in Terraform length[12] function. This count argument exports an index attribute that can be accessed in the rest of that resource block to refer to the object at that index of the issue list.

12. https://developer.hashicorp.com/terraform/language/functions/length

Now, create a GitHub issue resource in your main.tf file using the github_issue resource.[13] You'll use the count meta-argument and add the properties of the issues variable to your resource using count.index. You'll also link your issue to a certain milestone and add a label. We've given you our example code. Copy the following snippet to your issues variable and then we'll go over the complicated bits:

```
# in modules/integrations/github/project_management/main.tf

resource "github_issue" "tasks" {
  count      = length(var.issues)
  repository = github_repository.kanban.name
  title      = var.issues[count.index].title
  body       = var.issues[count.index].body
  milestone_number = github_repository_milestone.epics[
    var.issues[count.index].milestone
  ].number
  labels = [for l in var.issues[count.index].labels :
    github_issue_label.issues_labels[l].name
  ]
}
```

As you can see, we've set the count argument value to be the length of the issues variable. We've also learned from our resource dependency errors, and we've made each issue dependent on our repository. Instead of explicitly defining the resource dependency using the 'depends_on' meta-argument like last time, we've chosen to create the resource dependency implicitly and use the exported name argument. In the case of the title and body arguments, we've set these as the title and body keys of the object at that index of the issues list using count.index.

The milestone_number and labels arguments may seem a bit more complicated, but we promise they aren't. Let's break them both down.

The milestone_number argument to link each issue to a specific milestone in the Terraform state:

1. We accessed the var.issues[count.index].milestone to get the milestone related to that issue. An example of this value would be infrastructure.
2. We used the value of the accessed key in step 1 to access github_repository_milestone.epic[*].number. number is an attribute exported by the github_repository_milestone resource that refers to the number that was allocated to the milestone resource when it was created in the Terraform state. By using this argument, we've linked the issue to a certain milestone.

13. https://registry.terraform.io/providers/integrations/github/latest/docs/resources/issue

The labels argument links each issue to certain labels in the Terraform state:

1. A for expression iterates over var.issues[count.index].labels, which is the list of labels declared on the object at that index of the issues list. This gives us access to each label with the variable l.

2. We used this l variable as a key to access github_issue_label.issues_labels[l].name. name is an attribute exported by the resource github_issue_label.issues_labels in the Terraform state. If we follow our infrastructure example, the name would be Kind:Infrastructure, and the resulting label that would be linked from our code snippets would be kind-infrastructure.

Now that you've defined the issues variable and the issue resource, it's time to add the issues to your .auto.tfvars file. Add the following issues to the end of your .auto.tfvars file:

```
# in modules/integrations/github/project_management/.auto.tfvars

issues = [
  {
    title     = "Implement the Dockerfile's builder stage"
    body      = <<EOT
The builder stage packages all the tools and compile-time dependencies
for your application. It has to build the mix release that will be
copied in the running stage.
EOT
    labels    = ["kind-infrastructure", "dockerfile"]
    milestone = "infrastructure"
  },
  {
    title     = "Implement the Dockerfile's runner stage"
    body      = <<EOT
This stage copies the release built in the builder stage and uses it as
the entrypoint of your Docker image with the minimum system requirement
to run it.
EOT
    labels    = ["kind-infrastructure", "dockerfile"]
    milestone = "infrastructure"
  },
  {
    title     = "Elixir integration pipelines"
    body      = <<EOT
Implement a CI pipeline that includes all of the necessary steps when
delivering an Elixir application: code compilation, dependency caching,
testing, code formatting, an unused dependency check.
EOT
    labels    = ["kind-ci-cd", "tech-elixir"]
    milestone = "ci-cd"
  }
]
```

The previous example doesn't include all of the issues required to deliver this project. You can find the complete list in the .auto.tfvars file inside the 07_issues folder of the source code for this chapter on the Pragmatic Programmers website.

Now that you have everything prepared, repeat the Terraform development cycle. Destroy the whole infrastructure and recreate it. Remember, CREATE–DESTROY–IMPROVE–REPEAT. Once you've done that and you're sure your configuration is idempotent, commit your code to your newly created kanban repository. To do so, cd out of your modules/integrations/github/project_management back to the root of your project and initialize a GitHub repository. Then add the remote URL to your kanban repository and pull the README.md and .gitignore files you created earlier. Finally add, commit, and push your main.tf and .terraform .lock.hcl files. You can also include the asdf_plugins.sh and .tool-versions files that you created in the previous chapter. And that's it! You have finished setting up your project. Let's sum up what you've learned so far.

What Have You Learned?

In this chapter, you used Terraform to create project management resources in GitHub. You created a single file main.tf that contains your Terraform resources and provider. You also did some tidying of your code by using both local values and variables. Finally, you learned the differences between implicit and explicit resource dependencies.

You should feel confident using Terraform configuration files, the Terraform state, and a Terraform console. You can now quickly spin up new project management resources for future projects. We encourage you to take ownership of the planning part of a project. Project management is for everyone. Collaboration in this stage makes everyone on the team more productive and committed to the project. Now that you have an infrastructure as code template, you can spend more time writing requirements and great descriptions that will help you and your team get back to the fun stuff.

As you continue reading this book, you'll close each of the issues that you have created in this chapter. So, with the project management for your application set up, in the next chapter, you'll start the first milestone: Infrastructure. There, you'll learn how to package your Phoenix LiveView application with Docker. Doing this will allow you to run your application under the same conditions on any machine and ensure environment integrity.

The Extra Mile

If you wish to deepen your knowledge of the themes we've covered in this chapter and refine your BEAMOps developer superpowers, we've created a few tasks for you to do:

1. Refactor the resource blocks we've covered to get rid of the `depends_on` meta-argument and use implicit dependencies everywhere instead.
2. Refactor your `main.tf` file and split it into two different files: `main.tf` and `variables.tf`. Doing this will make your configuration easier to read. It won't affect the result of running `terraform apply` because Terraform, by default, merges all `.tf` files in a directory together when applying a configuration.
3. Explore the `outputs <BLOCK TYPE>` and add a few to your module, for example the milestones.