(shouldn't be!) an admin, so you'll need to explicitly grant your user `sts:AssumeRole` permissions on the IAM role(s) you want to be able to assume.

Warning 2: Use aliases sparingly

I said this in the multiregion example, but it bears repeating: although it's easy to use aliases with Terraform, I would caution against using them too often, including with multi-account code. Typically, you use multiple accounts to create separation between them, so if something goes wrong in one account, it doesn't affect the other. Modules that deploy across multiple accounts go against this principle. Only do it when you *intentionally* want to have resources in multiple accounts coupled and deployed together.

Creating Modules That Can Work with Multiple Providers

When working with Terraform modules, you typically work with two types of modules:

Reusable modules

These are low-level modules that are not meant to be deployed directly but instead are to be combined with other modules, resources, and data sources.

Root modules

These are high-level modules that combine multiple reusable modules into a single unit that is meant to be deployed directly by running `apply` (in fact, the definition of a root module is it's the one on which you run `apply`).

The multiprovider examples you've seen so far have put all the `provider` blocks into the root module. What do you do if you want to create a reusable module that works with multiple providers? For example, what if you wanted to turn the multi-account code from the previous section into a

reusable module? As a first step, you might put all that code, unchanged, into the *modules/multi-account* folder. Then, you could create a new example to test it with in the *examples/multi-account-module* folder, with a *main.tf* that looks like this:

```
module "multi_account_example" {
  source = "../../modules/multi-account"
}
```

If you run `apply` on this code, it'll work, but there is a problem: all of the `provider` configuration is now hidden in the module itself (in *modules/multi-account*). Defining `provider` blocks within reusable modules is an antipattern for several reasons:

Configuration problems

If you have `provider` blocks defined in your reusable module, then that module controls all the configuration for that `provider`. For example, the IAM role ARN and regions to use are currently hardcoded in the *modules/multi-account* module. You could, of course, expose input variables to allow users to set the regions and IAM role ARNs, but that's only the tip of the iceberg. If you browse the AWS Provider documentation, you'll find that there are roughly *50* different configuration options you can pass into it! Many of these parameters are going to be important for users of your module, as they control how to authenticate to AWS, what region to use, what account (or IAM role) to use, what endpoints to use when talking to AWS, what tags to apply or ignore, and much more. Having to expose 50 extra variables in a module will make that module very cumbersome to maintain and use.

Duplication problems

Even if you expose those 50 settings in your module, or whatever subset you believe is important, you're creating code duplication for users of your module. That's because it's common to combine multiple modules together, and if you have to pass in some subset of 50 settings into each of those modules in order to get them to all authenticate correctly,

you're going to have to copy and paste a lot of parameters, which is tedious and error prone.

Performance problems

Every time you include a `provider` block in your code, Terraform spins up a new process to run that provider, and communicates with that process via RPC. If you have a handful of `provider` blocks, this works just fine, but as you scale up, it may cause performance problems. Here's a real-world example: a few years ago, I created reusable modules for CloudTrail, AWS Config, GuardDuty, IAM Access Analyzer, and Macie. Each of these AWS services is supposed to be deployed into every region in your AWS account, and as AWS had ~25 regions, I included 25 `provider` blocks in each of these modules. I then created a single root module to deploy all of these as a "baseline" in my AWS accounts: if you do the math, that's 5 modules with 25 `provider` blocks each, or 125 `provider` blocks total. When I ran `apply`, Terraform would fire up 125 processes, each making hundreds of API and RPC calls. With thousands of concurrent network requests, my CPU would start thrashing, and a single `plan` could take 20 minutes. Worse yet, this would sometimes overload the network stack, leading to intermittent failures in API calls, and `apply` would fail with sporadic errors.

Therefore, as a best practice, you should *not* define any `provider` blocks in your reusable modules and instead allow your users to create the `provider` blocks they need solely in their root modules. But then, how do you build a module that can work with multiple providers? If the module has no `provider` blocks in it, how do you define provider aliases that you can reference in your resources and data sources?

The solution is to use *configuration aliases*. These are very similar to the provider aliases you've seen already, except they aren't defined in a `provider` block. Instead, you define them in a `required_providers` block.

Open up *modules/multi-account/main.tf*, remove the nested `provider` blocks, and replace them with a `required_providers` block with configuration aliases as follows:

```
terraform {
  required_providers {
    aws = {
      source                = "hashicorp/aws"
      version               = "~> 4.0"
      configuration_aliases = [aws.parent, aws.child]
    }
  }
}
```

Just as with normal provider aliases, you can pass configuration aliases into resources and data sources using the `provider` parameter:

```
data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}
```

The key difference from normal provider aliases is that configuration aliases don't create any providers themselves; instead, they force users of your module to explicitly pass in a provider for each of your configuration aliases using a `providers` map.

Open up *examples/multi-account-module/main.tf*, and define the `provider` blocks as before:

```
provider "aws" {
  region = "us-east-2"
  alias  = "parent"
}

provider "aws" {
  region = "us-east-2"
  alias  = "child"
```