In general, you should prefer to use first-class, native deployment options like instance refresh whenever possible. Although such options weren't always available in the earlier days of Terraform, these days, many resources support native deployment options. For example, if you're using Amazon Elastic Container Service (ECS) to deploy Docker containers, the `aws_ecs_service` resource natively supports zero-downtime deployments via the `deployment_maximum_percent` and `deployment_minimum_healthy_percent` parameters; if you're using Kubernetes to deploy Docker containers, the `kubernetes_deployment` resource natively supports zero-downtime deployments by setting the `strategy` parameter to `RollingUpdate` and providing configuration via the `rolling_update` block. Check the docs for the resources you're using, and make use of native functionality when you can!

Valid Plans Can Fail

Sometimes, you run the `plan` command and it shows you a perfectly valid-looking plan, but when you run `apply`, you'll get an error. For example, try to add an `aws_iam_user` resource with the exact same name you used for the IAM user you created manually in Chapter 2:

```
resource "aws_iam_user" "existing_user" {
  # Make sure to update this to your own user name!
  name = "yevgeniy.brikman"
}
```

If you now run the `plan` command, Terraform will show you a plan that looks reasonable:

```
Terraform will perform the following actions:

  # aws_iam_user.existing_user will be created
  + resource "aws_iam_user" "existing_user" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
```

```
      + name          = "yevgeniy.brikman"
      + path          = "/"
      + unique_id     = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

If you run the `apply` command, you'll get the following error:

```
Error: Error creating IAM User yevgeniy.brikman:
EntityAlreadyExists:
User with name yevgeniy.brikman already exists.

  on main.tf line 10, in resource "aws_iam_user" "existing_user":
  10: resource "aws_iam_user" "existing_user" {
```

The problem, of course, is that an IAM user with that name already exists. This can happen not just with IAM users but with almost any resource. Perhaps someone created that resource manually or via CLI commands, but either way, some identifier is the same, and that leads to a conflict. There are many variations on this error, and Terraform newbies are often caught off guard by them.

The key realization is that `terraform plan` looks only at resources in its Terraform state file. If you create resources *out of band*—such as by manually clicking around the AWS Console—they will not be in Terraform's state file, and, therefore, Terraform will not take them into account when you run the `plan` command. As a result, a valid-looking plan will still fail.

There are two main lessons to take away from this:

After you start using Terraform, you should only use Terraform.

> When a part of your infrastructure is managed by Terraform, you should never manually make changes to it. Otherwise, you not only set yourself up for weird Terraform errors, but you also void many of the benefits of using infrastructure as code in the first place, given that the code will no longer be an accurate representation of your infrastructure.

If you have existing infrastructure, use the `import` command.

If you created infrastructure before you started using Terraform, you can use the `terraform import` command to add that infrastructure to Terraform's state file so that Terraform is aware of and can manage that infrastructure. The `import` command takes two arguments. The first argument is the "address" of the resource in your Terraform configuration files. This makes use of the same syntax as resource references, such as `<PROVIDER>_<TYPE>.<NAME>` (e.g., `aws_iam_user.existing_user`). The second argument is a resource-specific ID that identifies the resource to import. For example, the ID for an `aws_iam_user` resource is the name of the user (e.g., yevgeniy.brikman), and the ID for an `aws_instance` is the EC2 Instance ID (e.g., i-190e22e5). The documentation at the bottom of the page for each resource typically specifies how to import it.

For example, here is the `import` command that you can use to sync the `aws_iam_user` you just added in your Terraform configurations with the IAM user you created back in Chapter 2 (obviously, you should replace "yevgeniy.brikman" with your own username in this command):

```
$ terraform import aws_iam_user.existing_user yevgeniy.brikman
```

Terraform will use the AWS API to find your IAM user and create an association in its state file between that user and the `aws_iam_user.existing_user` resource in your Terraform configurations. From then on, when you run the `plan` command, Terraform will know that an IAM user already exists and not try to create it again.

Note that if you have a lot of existing resources that you want to import into Terraform, writing the Terraform code for them from scratch and importing them one at a time can be painful, so you might want to look into tools such as terraformer and terracognita, which can import both code and state from supported cloud environments automatically.