8 6

리마인드

기본적인 백엔드는 local working_dir에 구성된다. 이 것을 원격 백엔드로 구성해서 팀원들이 공유할 수 있다. aws s3를 이용해서 이를 구현 가능하다. 모든 팀원이 write 할 수 있으면 경합 상황이 발생할 수 있다. 따라서 lock 기능을 사용하기 위해 dynamodb를 사용한다.

실습 2 (상태관리)

terraform_remote_state 데이터 소스

이전 장 에서는 VPC의 서브넷 목록을 반환하는 aws_subnets과 같은 데이터 소스를 사용하여 AWS에서 읽기 전용 정보를 가져왔습니다. 상태 작업 시 특히 유용한 또 다른 데이터 소스인 terraform_remote_state가 있습니다. 이 데이터 소스를 사용하여 다른 Terraform 구성 세트에 저장된 Terraform 상태 파일을 가져올 수 있습니다.
예를 살펴보겠습니다. 웹 서버 클러스터가 MySQL 데이터베이스와 통신해야 한다고 상상해 보세요. 확장 가능하고 안전하며 내구성이 뛰어나고 가용성이 높은 데이터베이스를 실행하는 것은 많은 작업입니다. 그림에 표시된 것처럼 Amazon의 관계형 데이터베이스 서비스(RDS)를 사용하여 AWS가 이를 처리하도록 할 수 있습니다. RDS는 MySQL, PostgreSQL, SQL Server 및 Oracle을 포함한 다양한 데이터베이스를 지원합니다.
웹 서버 클러스터와 동일한 구성 파일 세트에 MySQL 데이터베이스를 정의하고 싶지 않을 수도 있습니다. 웹 서버 클러스터에 대한 업데이트를 훨씬 더 자주 배포하고 매번 실수로 데이터베이스가 손상되는 위험을 감수하고 싶지 않기 때문입니다.

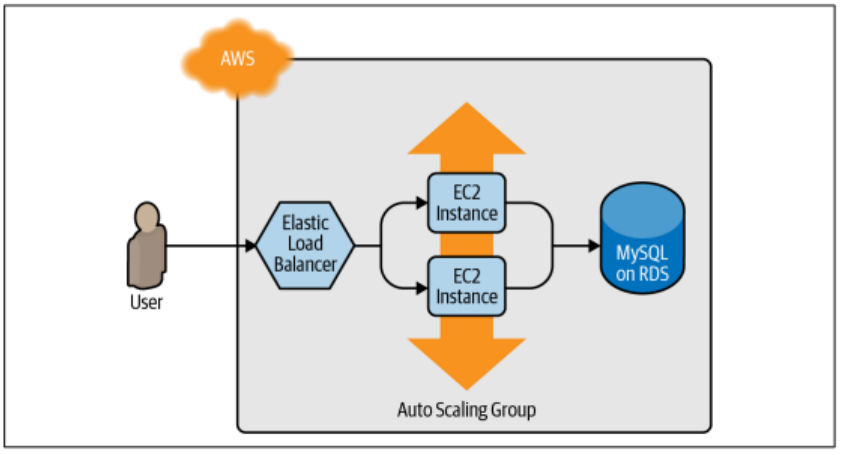


Figure 3-9. The web server cluster communicates with MySQL, which is deployed on top of Amazon RDS.

따라서 첫 번째 단계는 그림 3-10과 같이 stage/data-stores/mysql에 새 폴더를 생성하고 그 안에 기본 Terraform 파일(main.tf,variables.tf,outputs.tf)을 생성하는 것입니다.

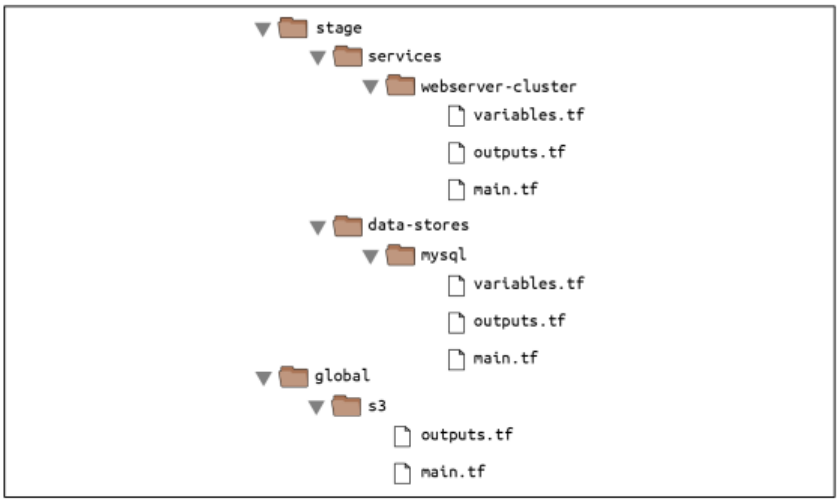


Figure 3-10. Create the database code in the stage/data-stores folder.

다음으로 stage/data-stores/mysql/main.tf에 데이터베이스 리소스를 생성합니다.

```
provider "aws" {
 region = "ap-northeast-2"
}
resource "aws_db_instance" "example" {
 identifier_prefix = "terraform-mysql"
 engine = "mysql"
 allocated_storage = 10
 instance_class = "db.t2.micro"
 skip_final_snapshot = true
 db_name = "example_database"
 # How should we set the username and password?
 username = "???"
 password = "???"
}
```

파일 상단에는 일반적인 공급자 블록이 표시되지만 바로 아래에는 새로운 리소스인 aws_db_instance가 있습니다. 이 리소스는 다음 설정을 사용하여 RDS에 데이터베이스를 생성합니다.

- MySQL을 데이터베이스 엔진으로 사용합니다.
- 10GB의 저장 공간.
- 가상 CPU 1개, 메모리 1GB를 포함하고 AWS 프리 티어의 일부인 db.t2.micro 인스턴스.
- 이 코드는 학습 및 테스트용이므로 최종 스냅샷이 비활성화됩니다. (스냅샷을 비활성화하지 않거나 final_snapshot_identifier 매개변수를 통해 스냅샷 이름을 제공하지 않으면 삭제가 실패합니다.)

aws_db_instance 리소스에 전달해야 하는 두 가지 파라미터는 마스터 사용자 이름과 마스터 비밀번호입니다. 이는 비밀이기 때문에 일반 텍스트로 코드에 직접 입력하면 안 됩니다! 비밀을 일반 텍스트로 저장하지 않고 사용하기 쉬운 옵션을 사용해 보겠습니다. 데이터베이스 비밀번호와 같은 비밀을 Terraform 외부(예: 1Password, LastPass 또는 macOS Keychain과 같은 비밀번호 관리자)에 저장합니다. 환경 변수를 통해 해당 비밀을 Terraform에 전달합니다.

이를 수행하려면 stage/datastores/mysql/variables.tf에서 db_username 및 db_password라는 변수를 선언하십시오.

```
variable "db_username" {
 description = "The username for the database"
```

```
 type = string
 sensitive = true
}

variable "db_password" {
 description = "The password for the database"
 type = string
 sensitive = true
}
```

먼저, 이러한 변수에는 비밀이 포함되어 있음을 나타내기 위해 sensitive = true로 표시되어 있습니다. 이렇게 하면 plan 또는 apply 를 실행할 때 Terraform이 값을 기록하지 않습니다. 둘째, 이러한 변수에는 기본값이 없습니다. 이는 의도적인 것입니다. 데이터베이스 자격 증명이나 민감한 정보를 일반 텍스트로 저장하면 안 됩니다. 대신 환경 변수를 사용하여 이러한 변수를 설정합니다.
그 전에 코드를 완성해 봅시다. 먼저 두 개의 새로운 입력 변수를 aws_db_instance 리소스에 전달합니다.

```
resource "aws_db_instance" "example" {
 identifier_prefix = "terraform"
 engine = "mysql"
 allocated_storage = 10
 instance_class = "db.t2.micro"
 skip_final_snapshot = true
 db_name = "example_database"
 username = var.db_username
 password = var.db_password
}
```

다음으로, 이전에 stage/data-stores/mysql/terraform.tfstate 경로에서 생성한 S3 버킷에 상태를 저장하도록 이 모듈을 구성합니다.

```
terraform {
 backend "s3" {
 # Replace this with your bucket name!
 bucket = "terraform-state-uvely"
 key = "stage/data-stores/mysql/terraform.tfstate"
 region = "us-east-2"
 # Replace this with your DynamoDB table name!
 dynamodb_table = "terraform-locks"
 encrypt = true
 }
}
```

마지막으로 stage/data-stores/mysql/outputs.tf에 두 개의 출력 변수를 추가하여 데이터베이스의 주소와 포트를 반환합니다.

```
output "address" {
 value = aws_db_instance.example.address
 description = "Connect to the database at this endpoint"
}
output "port" {
 value = aws_db_instance.example.port
```

```
 description = "The port the database is listening on"
}
```

이제 환경 변수를 사용하여 데이터베이스 사용자 이름과 비밀번호를 전달할 준비가 되었습니다. 참고로, Terraform 구성에 정의된 각 입력 변수 foo에 대해 환경 변수 TF_VAR_foo 를 사용하여 Terraform에 이 변수의 값을 제공할 수 있습니다. db_username 및 db_password 입력 변수의 경우 Linux/Unix/macOS 시스템에서 TF_VAR_db_username 및 TF_VAR_db_password 환경 변수를 설정하는 방법은 다음과 같습니다.

```
$ export TF_VAR_db_username="(YOUR_DB_USERNAME)" # testuser
$ export TF_VAR_db_password="(YOUR_DB_PASSWORD)" # testpass
```

Windows 시스템에서 이를 수행하는 방법은 다음과 같습니다.

```
$ set TF_VAR_db_username="(YOUR_DB_USERNAME)"
$ set TF_VAR_db_password="(YOUR_DB_PASSWORD)"
```

terraform init 및 terraform apply를 실행하여 데이터베이스를 만듭니다. Amazon RDS는 작은 데이터베이스라도 프로비저닝하는 데 약 10분 정도 걸릴 수 있으므로 인내심을 가지십시오. 적용이 완료되면 터미널에 출력이 표시됩니다.

```
$ terraform init
$ terraform apply
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Outputs:
address = "terraform.cowu6mts6srx.us-east-2.rds.amazonaws.com"
port = 3306
```

이제 이러한 출력은 S3 버킷의 stage/data-stores/mysql/terraform.tfstate 경로에 있는 데이터베이스의 Terraform 상태에도 저장됩니다.

웹 서버 클러스터 코드로 돌아가면 stage/services/webserver-cluster/data.tf 에 terraform_remote_state 데이터 소스를 추가하여 웹 서버가 데이터베이스의 상태 파일에서 해당 출력을 읽도록 할 수 있습니다.

```
data "terraform_remote_state" "db" {
 backend = "s3"
 config = {
 bucket = "(YOUR_BUCKET_NAME)"
 key = "stage/data-stores/mysql/terraform.tfstate"
 region = "ap-northeast-2"
 }
}
```

이 terraform_remote_state 데이터 소스는 그림과 같이 데이터베이스가 상태를 저장하는 동일한 S3 버킷 및 폴더에서 상태 파일을 읽도록 웹 서버 클러스터 코드를 구성합니다.

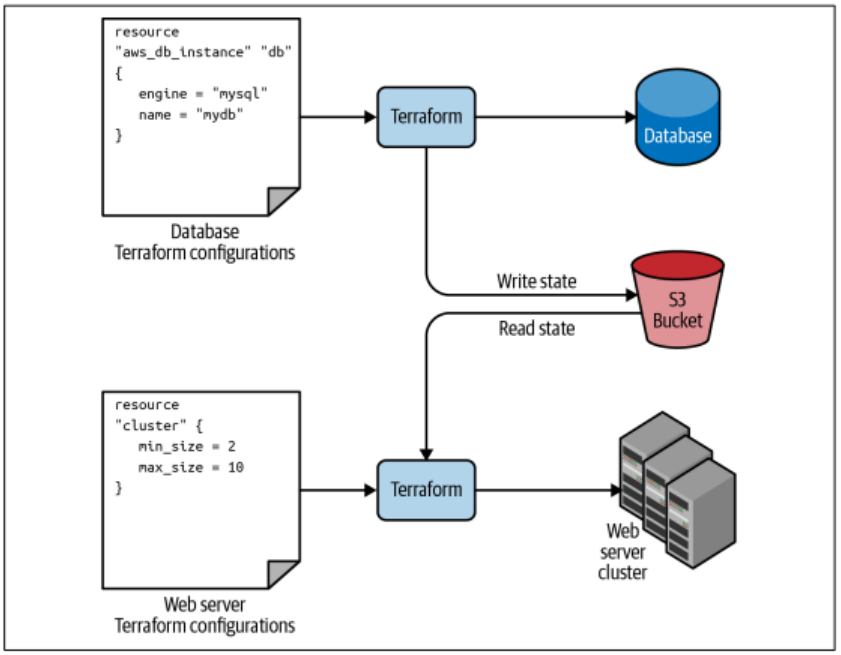


Figure 3-11. The database writes its state to an S3 bucket (top), and the web server cluster reads that state from the same bucket (bottom).

모든 데이터 소스와 마찬가지로 terraform_remote_state에서 반환된 데이터는 읽기 전용이라는 점을 이해하는 것이 중요합니다. 웹 서버 클러스터에서 사용자가 수행하는 작업은 Terraform 코드가 해당 상태를 수정할 수 없으므로 데이터베이스 자체에 문제를 일으킬 위험 없이 데이터베이스의 상태 데이터를 가져올 수 있습니다.
데이터베이스의 모든 출력 변수는 상태 파일에 저장되며 다음 형식의 속성 참조를 사용하여 terraform_remote_state 데이터 소스에서 읽을 수 있습니다.

```
data.terraform_remote_state.<NAME>.outputs.<ATTRIBUTE>
```

예를 들어 웹 서버 클러스터 인스턴스의 사용자 데이터를 업데이트하여 terraform_remote_state 데이터 소스에서 데이터베이스 주소와 포트를 가져오고 해당 정보를 HTTP 응답에 노출하는 방법은 다음과 같습니다.

```
user_data = <<EOF
#!/bin/bash
echo "Hello, World" >> index.html
echo "${data.terraform_remote_state.db.outputs.address}" >> index.html
echo "${data.terraform_remote_state.db.outputs.port}" >> index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

사용자 데이터 스크립트가 길어지면서 이를 인라인으로 정의하는 것이 점점 더 복잡해지고 있습니다. 일반적으로 하나의 프로그래밍 언어(Bash)를 다른 프로그래밍 언어(Terraform) 안에 포함하면 각 언어를 유지 관리하기가 더 어려워지므로 Bash 스크립트를 외부화하기 위해 여기서 잠시 멈추겠습니다. 이를 위해 templatefile 내장 함수를 사용할 수 있습니다. Terraform에는 다음 형식의 표현식을 사용하여 실행할 수 있는 다양한 내장 함수가 포함되어 있습니다.

```
function_name(...)
```

예를 들어 형식(format) 함수 예시는 다음과 같습니다.

```
format(<FMT>, <ARGS>, ...)
```

이 함수는 문자열 FMT 의 sprintf 구문에 따라 ARGS의 인수 형식을 지정합니다. 내장 기능을 실험하는 가장 좋은 방법은 terraform console 명령을 실행하여 Terraform 구문을 시험해 보고, 인프라 상태를 쿼리하고, 결과를 즉시 확인할 수 있는 대화형 콘솔을 얻는 것입니다.

```
$ terraform console
> format("%.3f", 3.14159265359)
3.142
```

Terraform 콘솔은 읽기 전용이므로 실수로 인프라나 상태를 변경하는 것에 대해 걱정할 필요가 없습니다.
문자열, 숫자, 목록 및 지도를 조작하는 데 사용할 수 있는 기타 내장 함수가 많이 있습니다. 그 중 하나는 templatefile 함수입니다.

```
templatefile(<PATH>, <VARS>)
```

이 함수는 PATH 에서 파일을 읽고, 이를 템플릿으로 렌더링하고, 결과를 문자열로 반환합니다. "템플릿으로 렌더링합니다"라고 말하면 PATH의 파일이 Terraform(${...})의 문자열 보간 구문을 사용할 수 있고 Terraform이 해당 파일의 콘텐츠를 렌더링하여 VARS에서 변수 참조를 채운다는 의미입니다.

이를 실제로 보려면 다음과 같이 사용자 데이터 스크립트의 내용을 stage/services/webserver-cluster/user-data.sh 파일에 넣으십시오.

```
#!/bin/bash
cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF
nohup busybox httpd -f -p ${server_port} &
```

이 Bash 스크립트에는 원본에서 몇 가지 변경 사항이 있습니다.

- Terraform의 표준 보간 구문을 사용하여 변수를 찾습니다. 단, 액세스할 수 있는 유일한 변수는 두 번째 매개변수를 통해 templatefile에 전달하는 변수이므로(곧 살펴보겠지만) 액세스하기 위해 접두사가 필요하지 않습니다. : 예를 들어 ${var.server_port} 가 아닌 ${server_port} 를 사용해야 합니다.
- 이제 스크립트에 일부 HTML 구문(예: <h1>)이 포함되어 웹 브라우저에서 출력을 좀 더 쉽게 읽을 수 있습니다.

마지막 단계는 aws_launch_configuration 리소스의 user_data 파라미터를 업데이트하여 templatefile 함수를 호출하고 필요한 변수를 맵으로 전달하는 것입니다.

```
resource "aws_launch_configuration" "example" {
 image_id = "ami-0fb653ca2d3203ac1"
 instance_type = "t2.micro"
 security_groups = [aws_security_group.instance.id]
 # Render the User Data script as a template
 user_data = templatefile("user-data.sh", {
 server_port = var.server_port
 db_address = data.terraform_remote_state.db.outputs.address
 db_port = data.terraform_remote_state.db.outputs.port
 })
 # Required when using a launch configuration with an auto scaling group.
 lifecycle {
 create_before_destroy = true
```

```
  }
}
```

이제 Bash 스크립트를 인라인으로 작성하는 것보다 훨씬 깔끔합니다!

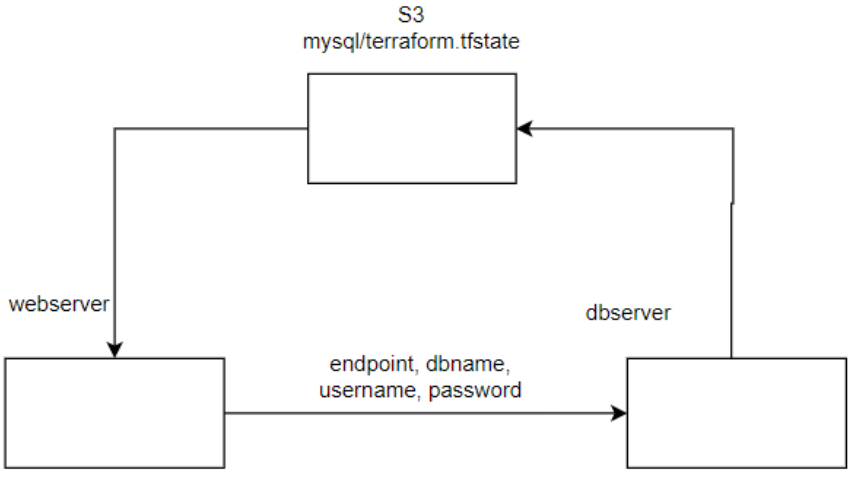
Terraform Apply를 사용하여 이 클러스터를 배포하고 인스턴스가 ALB에 등록될 때까지 기다린 다음 웹 브라우저에서 ALB URL을 열면 그림과 비슷한 내용이 표시됩니다.

축하합니다. 이제 데이터베이스와 함께 웹 서버 클러스터가 Terraform을 통해 프로그래밍 방식으로 데이터베이스 주소와 포트에 액세스할 수 있습니다. 실제 웹 프레임워크(예: Ruby on Rails)를 사용하는 경우 주소와 포트를 환경 변수로 설정하거나 구성 파일에 기록하여 데이터베이스 라이브러리(예: ActiveRecord)에서 통신에 사용할 수 있습니다.

Figure 3-12. The web server cluster can programmatically access the database address and port.

필기 및 결과

- 구조



- 코드

1. backend.tf

```
terraform {
 backend "s3" {
 # Replace this with your bucket name!
 bucket = "cloudwave-cj-std23"
 key = "stage/data-stores/mysql/terraform.tfstate"
 region = "ap-northeast-2"
 # Replace this with your DynamoDB table name!
 dynamodb_table = "terraform-locks"
 encrypt = true
 }
}
```

2. main.tf

```
provider "aws" {
 region = "ap-northeast-2"
}
resource "aws_db_instance" "example" {
 identifier_prefix = "terraform-mysql"
 engine = "mysql"
 allocated_storage = 10
 instance_class = "db.t3.micro"
 skip_final_snapshot = true
 db_name = "example_database"
 # How should we set the username and password?
 username = var.db_username
 password = var.db_password
}
```

3. output.tf

```
output "address" {
 value = aws_db_instance.example.address
 description = "Connect to the database at this endpoint"
}
output "port" {
 value = aws_db_instance.example.port
 description = "The port the database is listening on"
}
```

4. variable.tf

```
variable "db_username" {
 description = "The username for the database"
```

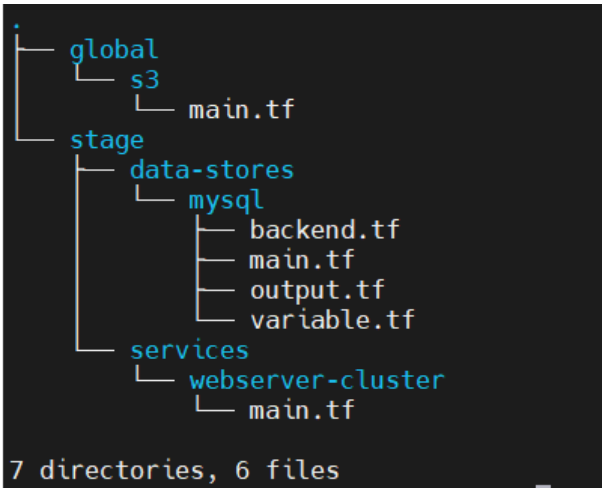
```
 type = string
 sensitive = true
}

variable "db_password" {
 description = "The password for the database"
 type = string
 sensitive = true
}
```

```
$ export TF_VAR_db_username="(YOUR_DB_USERNAME)" # testuser
$ export TF_VAR_db_password="(YOUR_DB_PASSWORD)" # testpass
```

```
terraform init ; terraform apply -auto-approve
```

yes 안누르고 자동으로 init apply 적용

- 결과



```
[root@controller mysql]# terraform init ; terraform apply -auto-approve
Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.61.0...
- Installed hashicorp/aws v5.61.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

address = "terraform-mysql20240806005425722700000001.cj0yoocecbvk.ap-northeast-2.rds.amazonaws.com
port = 3306
```

- 코드

1. main.tf 수정

user_data 변경

```
user_data = templatefile("user-data.sh", {
 server_port = var.server_port
 db_address = data.terraform_remote_state.db.outputs.address
 db_port = data.terraform_remote_state.db.outputs.port
 })
```

2. user_data.sh

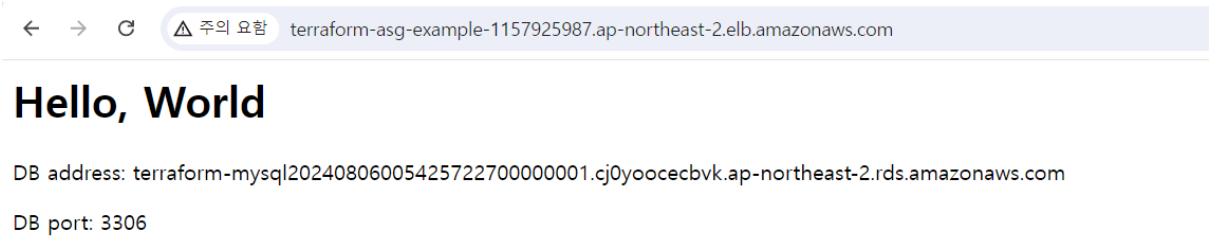
```
#!/bin/bash
cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF
nohup busybox httpd -f -p ${server_port} &
```

3. data.tf

```
data "terraform_remote_state" "db" {
 backend = "s3"
 config = {
 bucket = "cloudwave-cj-std23"
 key = "stage/data-stores/mysql/terraform.tfstate"
 region = "ap-northeast-2"
 }
}
```

- 결과

```
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = "terraform-asg-example-1157925987.ap-northeast-2.elb.amazonaws.com"
```



결론

격리, 잠금 및 상태에 대해 많은 생각을 해야 하는 이유는 코드형 인프라(IaC)가 일반 코딩과 다른 장단점이 있기 때문입니다. 일반적인 앱용 코드를 작성할 때 대부분의 버그는 상대적으로 사소하며 단일 앱의 일부만 손상됩니다. 인프라를 제어하는 코드를 작성할 때 버그는 모든 앱, 모든 데이터 저장소, 전체 네트워크 토폴로지 및 기타 모든 것을 손상시킬 수 있다는 점을 고려하면 버그가 더 심각한 경향이 있습니다. 따라서 일반적인 코드보다 IaC에서 작업할 때 더 많은 "안전 메커니즘"을 포함하는 것이 좋습니다.
권장되는 파일 레이아웃을 사용할 때 공통적으로 우려되는 점은 코드 중복이 발생한다는 것입니다. 스테이징 및 프로덕션 모두에서 웹 서버 클러스터를 실행하려는 경우 stage/services/ webserver-cluster 및 prod/services/webserver-cluster 사이에 많은 코드를 복사하여 붙여넣지 않아도 되는 방법은 무엇입니까? 그것은 바로 다음 주제인 Terraform 모듈을 사용해야 한다는 것입니다.

실습 3 (모듈로 재사용 가능한 인프라 구성)

이전장의 마지막에 그림과 같은 아키텍처를 구성했습니다.

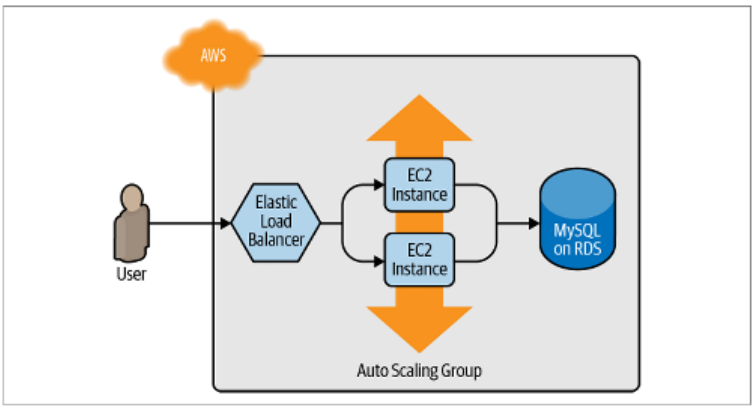


Figure 4-1. The architecture you deployed in previous chapters included a load balancer, web server cluster, and database.

이는 첫 번째 환경으로는 훌륭하게 작동하지만 일반적으로 아래 그림에 표시된 것처럼 팀의 내부 테스트용 환경("스테이징")과 실제 사용자가 액세스할 수 있는 환경("프로덕션") 두 개 이상의 환경이 필요합니다. 이상적으로 두 환경은 거의 동일하지만 비용을 절약하기 위해 준비 단계에서 약간 더 적은 수의 서버를 실행할 수도 있습니다.

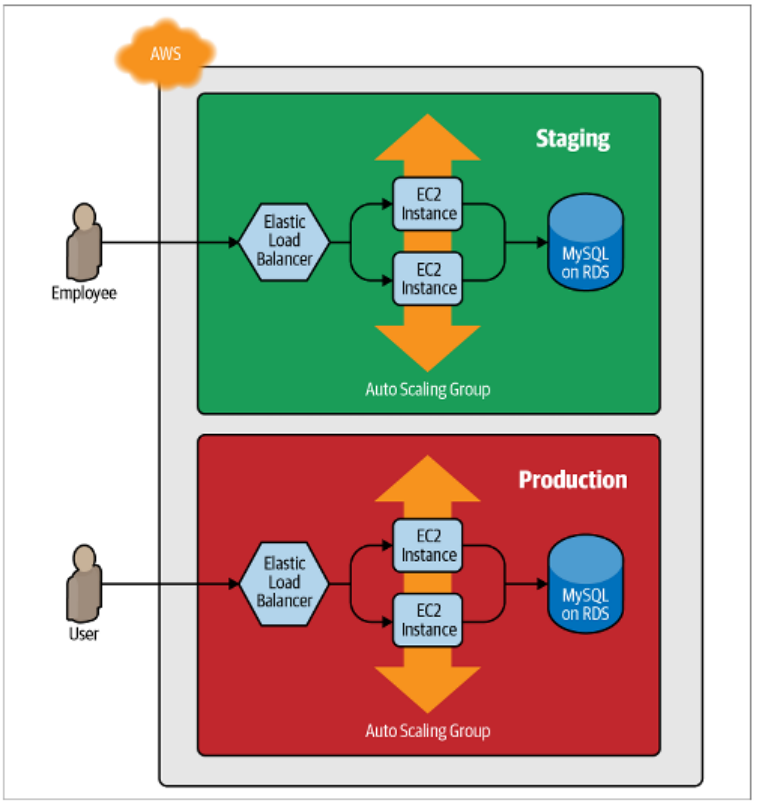


Figure 4-2. The architecture you'll deploy in this chapter will have two environments, each with its own load balancer, web server cluster, and database.

스테이징에서 모든 코드를 복사하여 붙여넣지 않고도 이 프로덕션 환경을 어떻게 추가할까요? 예를 들어, stage/services/webserver-cluster의 모든 코드를 prod/services/webserver-cluster에 복사하고 stage/data-stores/mysql의 모든 코드를 prod/data-stores/mysql에 복사하여 붙여넣지 않아도 되는 방법은 무엇입니까? Ruby와 같은 범용 프로그래밍 언어에서 동일한 코드를 여러 위치에 복사하여 붙여넣은 경우 해당 코드를 함수 내부에 배치하고 해당 함수를 어디에서나 재사용할 수 있습니다.

```
# Define the function in one place
def example_function()
 puts "Hello, World"
end
# Use the function in multiple other places
example_function()
```

모듈을 사용하면 Terraform 내부에 코드를 배치하고 코드 전체의 여러 위치에서 해당 모듈을 재사용할 수 있습니다. 스테이징 및 프로덕션 환경에서 동일한 코드를 복사하여 붙여넣는 대신 그림과 같이 두 환경 모두에서 동일한 모듈의 코드를 재사용할 수 있습니다.

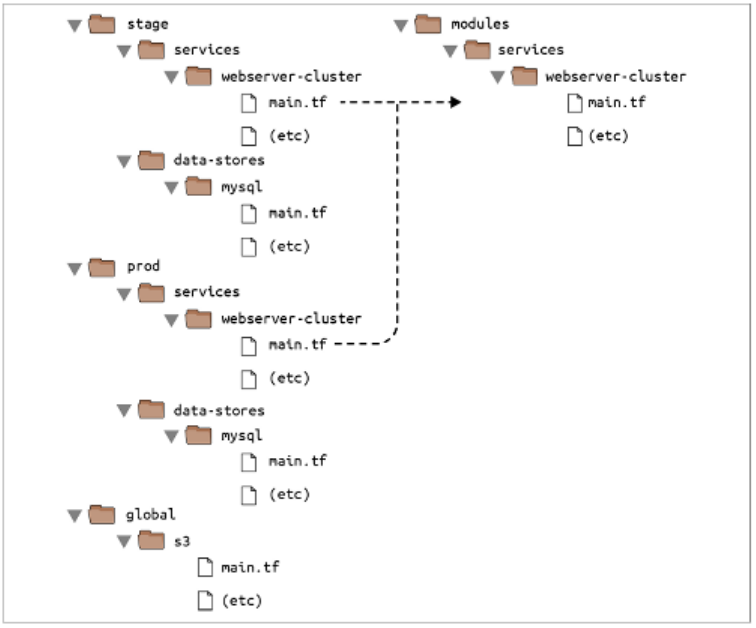


Figure 4-3. Putting code into modules allows you to reuse that code from multiple environments.

모듈은 재사용, 유지 관리 및 테스트 가능한 Terraform 코드를 작성하는 핵심 요소입니다. 일단 사용하기 시작하면 되돌릴 수 없습니다. 모든 것을 모듈로 구축하고, 회사 내에서 공유할 모듈 라이브러리를 만들고, 온라인에서 찾은 모듈을 사용하고, 전체 인프라를 재사용 가능한 모듈 모음으로 생각하기 시작합니다.

이 장에서는 다음 주제를 다루면서 Terraform 모듈을 생성하고 사용하는 방법을 보여 드리겠습니다.

- 모듈 기본
- 모듈 입력
- 모듈 지역화
- 모듈 출력
- 모듈 문제
- 모듈 버전 관리

모듈 기본

Terraform 모듈은 매우 간단합니다. 폴더에 있는 모든 Terraform 구성 파일 세트는 모듈입니다. 사실 지금까지 작성한 모든 구성은 기술적으로 모듈이었습니다. 모듈에서 직접 적용을 실행하면 루트 모듈이라고 합니다. 실제로 어떤 모듈이 가능한지 확인하려면 다른 모듈 내에서 사용하기 위한 모듈인 재사용 가능한 모듈을 만들어야 합니다. 예를 들어 ASG(Auto Scaling Group), ALB(Application Load Balancer), 보안 그룹 및 기타 여러 리소스가 포함된 stage/services/webserver-cluster 의 코드를 재사용 가능한 모듈로 바꿔 보겠습니다.

첫 번째 단계로 stage/services/webserver-cluster에서 terraform destroy를 실행하여 이전에 생성한 모든 리소스를 정리합니다. 다음으로, module이라는 새 최상위 폴더를 만들고 stage/services/webserver-cluster의 모든 파일을 module/services/webserver-cluster로 이동합니다. 그림과 같은 폴더 구조가 나타나야 합니다. module/services/webserver-cluster에서 main.tf 파일을 열고 공급자 정의를 제거합니다. 공급자는 재사용 가능한 모듈이 아닌 루트 모듈에서만 구성해야 합니다.

Figure 4-4. Move your reusable web server cluster code into a modules/services/web server-cluster folder.

이제 스테이징 환경에서 이 모듈을 사용할 수 있습니다. 모듈을 사용하는 구문은 다음과 같습니다.

```
module "<NAME>" {
 source = "<SOURCE>"
 [CONFIG ...]
}
```

여기서 NAME은 Terraform 코드 전체에서 이 모듈을 참조하는 데 사용할 수 있는 식별자(예: webserver_cluster)이고, SOURCE는 모듈 코드를 찾을 수 있는 경로(예: module/services/webserver-cluster)이며 CONFIG는 해당 모듈과 관련된 인수로 구성합니다. 예를 들어, stage/services/webserver-cluster/main.tf에 새 파일을 생성하고 다음과 같이 그 파일에서 webserver-cluster 모듈을 사용할 수 있습니다.

```
provider "aws" {
 region = "ap-northeast-2"
}
module "webserver_cluster" {
 source = "../../../modules/services/webserver-cluster"
}
```

그런 다음 다음 내용으로 새 prod/services/webserver-cluster/main.tf 파일을 생성하여 프로덕션 환경에서 정확히 동일한 모듈을 재사용할 수 있습니다.

```
provider "aws" {
 region = "ap-northeast-2"
}
module "webserver_cluster" {
```

```
 source = "../../../modules/services/webserver-cluster"
}
```

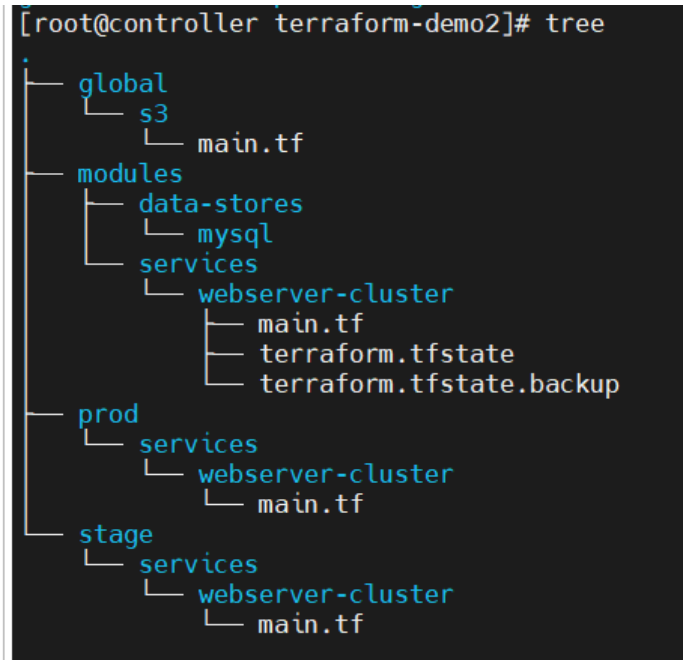
Terraform 구성에 모듈을 추가하거나 모듈의 소스 매개변수를 수정할 때마다 plan 또는 apply를 실행하기 전에 init 명령을 실행해야 합니다.

```
$ terraform init
Initializing modules...
- webserver_cluster in ../../../modules/services/webserver-cluster

Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!
```

이렇게 init 명령이 가지고 있는 자세한 내용을 살펴보았습니다. 즉, 공급자를 설치하고, 백엔드를 구성하고, 모듈을 다운로드하는 등 모두 init 이라는 하나의 편리한 명령으로 수행됩니다.

이 코드에서 적용 명령을 실행하기 전에 webserver-cluster 모듈에 문제가 있다는 점에 유의하십시오. 모든 이름이 하드코딩되어 있습니다. 즉, 보안 그룹 이름, ALB, 기타 리소스 이름이 모두 하드코딩되어 있으므로 동일한 AWS 계정에서 이 모듈을 두 번 이상 사용하면 이름 충돌 오류가 발생합니다. module/services/webserver-cluster에 복사한 main.tf 파일이 terraform_remote_state 데이터 소스를 사용하여 데이터베이스 주소와 포트를 파악하고 terraform_remote_state가 하드코딩되어 있으므로 데이터베이스 상태를 읽는 방법에 대한 세부 정보도 하드코딩되어 있습니다. 준비 환경에서.
이러한 문제를 해결하려면 webserver-cluster 모듈에 구성 가능한 입력을 추가하여 다양한 환경에서 다르게 동작할 수 있도록 해야 합니다.

결과



```
[root@controller webserver-cluster]# terraform init
Initializing the backend...
Initializing modules...
- webserver_cluster in ../../../modules/services/webserver-cluster
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.61.0...
- Installed hashicorp/aws v5.61.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

모듈입력

Ruby와 같은 범용 프로그래밍 언어로 함수를 구성 가능하게 만들려면 해당 함수에 입력 매개변수를 추가하면 됩니다.

```
# A function with two input parameters
def example_function(param1, param2)
 puts "Hello, #{param1} #{param2}"
end
# Pass two input parameters to the function
example_function("foo", "bar")
```

Terraform에서는 모듈에도 입력 매개변수가 있을 수 있습니다. 이를 정의하려면 이미 익숙한 메커니즘인 입력 변수를 사용합니다. module/services/webserver-cluster/variables.tf를 열고 세 가지 새로운 입력 변수를 추가합니다.

```
variable "cluster_name" {
 description = "The name to use for all the cluster resources"
 type = string
}
variable "db_remote_state_bucket" {
 description = "The name of the S3 bucket for the database's remote state"
 type = string
}
variable "db_remote_state_key" {
 description = "The path for the database's remote state in S3"
 type = string
}
```

다음으로, module/services/webserver-cluster/main.tf를 살펴보고 하드코딩된 이름 대신(예: "terraform-asgexample" 대신 var.cluster_name) 을 사용합니다. 예를 들어 ALB 보안 그룹에 대해 수행하는 방법은 다음과 같습니다.

```
resource "aws_security_group" "alb" {
 name = "${var.cluster_name}-alb"
```

```
 ingress {
  from_port = 80
  to_port = 80
  protocol = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
 }
 egress {
  from_port = 0
  to_port = 0
  protocol = "-1"
  cidr_blocks = ["0.0.0.0/0"]
 }
}
```

name 매개변수가 어떻게 "${var.cluster_name}-alb"로 설정되었는지 확인하세요. 다른 aws_security_group 리소스(예: "${var.cluster_name}-instance" 이름 지정), aws_alb 리소스 및 aws_autoscaling_group 리소스의 태그 섹션을 비슷하게 변경해야 합니다.
또한 db_remote_state_bucket 및 db_remote_state_key를 각각 버킷 및 키 매개변수로 사용하도록 terraform_remote_state 데이터 소스를 업데이트하여 올바른 환경에서 상태 파일을 읽고 있는지 확인해야 합니다.

```
data "terraform_remote_state" "db" {
 backend = "s3"
 config = {
 bucket = var.db_remote_state_bucket
 key = var.db_remote_state_key
 region = "us-east-2"
 }
}
```

이제 스테이징 환경의 stage/services/webserver-cluster/main.tf 에서 다음과 같은 새 입력 변수를 적절하게 설정할 수 있습니다.

```
module "webserver_cluster" {
 source = "../../../modules/services/webserver-cluster"
 cluster_name = "webservers-stage"
 db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
 db_remote_state_key = "stage/data-stores/mysql/terraform.tfstate"
}
```

또한 prod/services/webserver-cluster/main.tf의 프로덕션 환경에서 이러한 변수를 설정해야 하지만 해당 환경에 해당하는 다른 값으로 설정해야 합니다.

```
module "webserver_cluster" {
 source = "../../../modules/services/webserver-cluster"
 cluster_name = "webservers-prod"
 db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
 db_remote_state_key = "prod/data-stores/mysql/terraform.tfstate"
}
```

보시다시피 리소스에 대한 인수를 설정하는 것과 동일한 구문을 사용하여 모듈에 대한 입력 변수를 설정합니다. 입력 변수는 모듈의 API로, 모듈이 다양한 환경에서 어떻게 작동할지 제어합니다. 지금까지 이름 및 데이터베이스 원격 상태에 대한

입력 변수를 추가했지만 모듈에서 다른 매개변수도 구성 가능하게 만들고 싶을 수도 있습니다. 예를 들어, 준비 단계에서는 비용 절감을 위해 소규모 웹 서버 클러스터를 실행하고 싶지만 프로덕션 단계에서는 많은 트래픽을 처리하기 위해 더 큰 클러스터를 실행하고 싶을 수도 있습니다. 그렇게 하려면 module/services/webserver-cluster/variables.tf에 입력 변수 3개를 더 추가하면 됩니다.

```
variable "instance_type" {
 description = "The type of EC2 Instances to run (e.g. t2.micro)"
 type = string
}
variable "min_size" {
 description = "The minimum number of EC2 Instances in the ASG"
 type = number
}
variable "max_size" {
 description = "The maximum number of EC2 Instances in the ASG"
 type = number
}
```

다음으로, module/services/webserver-cluster/main.tf의 시작 구성을 업데이트하여 해당 instance_type 매개변수를 새 var.instance_type 입력 변수로 설정합니다.

```
resource "aws_launch_configuration" "example" {
 image_id = "ami-0fb653ca2d3203ac1"
 instance_type = var.instance_type
 security_groups = [aws_security_group.instance.id]
 user_data = templatefile("user-data.sh", {
  server_port = var.server_port
  db_address = data.terraform_remote_state.db.outputs.address
  db_port = data.terraform_remote_state.db.outputs.port
 })
 # Required when using a launch configuration with an auto scaling group.
 lifecycle {
  create_before_destroy = true
 }
}
```

마찬가지로, 동일한 파일의 ASG 정의를 업데이트하여 min_size 및 max_size 매개변수를 각각 새로운 var.min_size 및 var.max_size 입력 변수로 설정해야 합니다.

```
resource "aws_autoscaling_group" "example" {
 launch_configuration = aws_launch_configuration.example.name
 vpc_zone_identifier = data.aws_subnets.default.ids
 target_group_arns = [aws_lb_target_group.asg.arn]
 health_check_type = "ELB"
 min_size = var.min_size
 max_size = var.max_size
 tag {
  key = "Name"
  value = var.cluster_name
  propagate_at_launch = true
```

```
 }
}
```

이제 스테이징 환경(stage/services/webserver-cluster/main.tf)에서 instance_type을 "t2.micro"로 설정하고 min_size 및 max_size를 2로 설정하여 클러스터를 작고 저렴하게 유지할 수 있습니다.

```
module "webserver_cluster" {
 source = "../../../modules/services/webserver-cluster"
 cluster_name = "webservers-stage"
 db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
 db_remote_state_key = "stage/data-stores/mysql/terraform.tfstate"
 instance_type = "t2.micro"
 min_size = 2
 max_size = 2
}
```

반면, 프로덕션 환경에서는 m4.large와 같이 더 많은 CPU와 메모리를 갖춘 더 큰 인스턴스 유형을 사용할 수 있습니다. (이 인스턴스 유형은 AWS 프리 티어의 일부가 아니므로 단지 사용하는 경우 이는 학습용이므로 비용이 청구되지 않으니, instance_type에 "t2.micro"를 사용하세요.) max_size를 10으로 설정하면 클러스터가 로드에 따라 줄어들거나 늘어날 수 있습니다(클러스터는 처음에 두 개의 인스턴스로 시작되기 때문에 큰 걱정은 없습니다.):

모듈지역화

입력 변수를 사용하여 모듈의 입력을 정의하는 것은 훌륭하지만, 중간 수행하기 위해 모듈에서 변수를 정의하는 방법이 필요하거나 코드를 DRY로 유지하고 싶지만 해당 변수를 다음과 같이 노출하고 싶지 않은 경우 어떻게 해야 합니까? 구성 가능한 입력? 예를 들어, module/services/webserver-cluster/main.tf의 webserver-cluster 모듈에 있는 로드 밸런서는 HTTP의 기본 포트인 포트 80에서 수신 대기합니다. 이 포트 번호는 현재 로드 밸런서 리스너를 포함한 여러 위치에 복사되어 붙여넣어져 있습니다.

```
resource "aws_lb_listener" "http" {
 load_balancer_arn = aws_lb.example.arn
 port = 80
 protocol = "HTTP"
 # By default, return a simple 404 page
 default_action {
  type = "fixed-response"
  fixed_response {
   content_type = "text/plain"
   message_body = "404: page not found"
   status_code = 404
  }
 }
}
```

로드 밸런서 보안 그룹은 다음과 같습니다.

```
resource "aws_security_group" "alb" {
 name = "${var.cluster_name}-alb"
  ingress {
  from_port = 80
  to_port = 80
  protocol = "tcp"
```

```
  cidr_blocks = ["0.0.0.0/0"]
 }
 egress {
  from_port = 0
  to_port = 0
  protocol = "-1"
  cidr_blocks = ["0.0.0.0/0"]
 }
}
```

"모든 IP" CIDR 블록 0.0.0.0/0 , "모든 포트" 값 0 , "모든 프로토콜" 값 "-1"을 포함한 보안 그룹의 값도 여러 위치에 복사 되어 붙여 넣어집니다. 모듈 전반에 걸쳐. 이러한 값을 여러 위치에 하드코딩하면 코드를 읽고 유지하기가 더 어려워집니다. 값을 입력 변수로 추출할 수 있지만 그러면 모듈 사용자가 원하지 않는 이러한 값을 실수로 무시할 수 있습니다. 입력 변수를 사용하는 대신 locals 블록에서 이를 로컬 값으로 정의할 수 있습니다.

```
locals {
 http_port = 80
 any_port = 0
 any_protocol = "-1"
 tcp_protocol = "tcp"
 all_ips = ["0.0.0.0/0"]
}
```

로컬 값을 사용하면 Terraform 표현식에 이름을 할당하고 모듈 전체에서 해당 이름을 사용할 수 있습니다. 이러한 이름은 모듈 내에서만 표시되므로 다른 모듈에는 영향을 주지 않으며 모듈 외부에서는 이러한 값을 재정의할 수 없습니다. 로컬 값을 읽으려면 다음 구문을 사용하는 로컬 참조를 사용해야 합니다.

```
local . <NAME>
```

로드 밸런서 리스너의 포트 매개변수를 업데이트하려면 다음 구문을 사용하십시오.

```
resource "aws_lb_listener" "http" {
 load_balancer_arn = aws_lb.example.arn
 port = local.http_port
 protocol = "HTTP"
 # By default, return a simple 404 page
 default_action {
  type = "fixed-response"
  fixed_response {
   content_type = "text/plain"
   message_body = "404: page not found"
   status_code = 404
  }
 }
}
```

마찬가지로 로드 밸런서 보안 그룹을 포함하여 모듈의 보안 그룹에 있는 거의 모든 매개변수를 업데이트합니다.

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
  ingress {
```

```
  from_port = local.http_port
  to_port = local.http_port
  protocol = local.tcp_protocol
  cidr_blocks = local.all_ips
 }
 egress {
  from_port = local.any_port
  to_port = local.any_port
  protocol = local.any_protocol
  cidr_blocks = local.all_ips
 }
}
```

로컬은 코드를 더 쉽게 읽고 유지 관리할 수 있도록 하므로 자주 사용하세요.

모듈 출력

ASG의 강력한 기능은 로드에 대한 응답으로 실행 중인 서버 수를 늘리거나 줄이도록 구성할 수 있다는 것입니다. 이를 수행하는 한 가지 방법은 하루 중 예정된 시간에 클러스터의 크기를 변경할 수 있는 예약된 작업을 사용하는 것입니다. 예를 들어, 정규 업무 시간 동안 클러스터에 대한 트래픽이 훨씬 높은 경우 예약된 작업을 사용하여 오전 9시에 서버 수를 늘리고 오후 5시에 서버 수를 줄일 수 있습니다.

webserver-cluster 모듈에서 예약된 작업을 정의하면 스테이징 및 프로덕션 모두에 적용됩니다. 스테이징 환경에서는 이러한 종류의 조정을 수행할 필요가 없기 때문에 당분간은 프로덕션 구성에서 직접 자동 조정 일정을 정의할 수 있습니다. 예약된 작업을 webserver-cluster 모듈로 이동할 수 있습니다.
예약된 작업을 정의하려면 prod/services/webserver-cluster/main.tf에 다음 두 개의 aws_autoscaling_schedule 리소스를 추가합니다.

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
 scheduled_action_name = "scale-out-during-business-hours"
 min_size = 2
 max_size = 10
 desired_capacity = 10
 recurrence = "0 9 * * *"
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
 scheduled_action_name = "scale-in-at-night"
 min_size = 2
 max_size = 10
 desired_capacity = 2
 recurrence = "0 17 * * *"
}
```

이 코드는 aws_autoscaling_schedule 리소스 하나를 사용하여 오전 시간 동안 서버 수를 10개로 늘리고(recurrence 매개변수는 cron 구문을 사용하므로 "0 9 * * *"는 "매일 오전 9시"를 의미함) 두 번째 aws_autoscaling_schedule 리소스를 사용하여 서버 수를 줄입니다("0 17 * * *"는 "매일 오후 5시"를 의미) 그러나 aws_autoscaling_schedule의 두 가지 사용법 모두 ASG 이름을 지정하는 필수 파라미터인 autoscaling_group_name이 누락되었습니다. ASG 자체는 webserver-cluster 모듈 내에 정의되어 있으므로 해당 이름에 어떻게 액세스합니까?

Terraform은 출력 변수를 사용하여 모듈은 값을 반환할 수 있습니다. 다음과 같이 ASG 이름을 /modules/services/webserver-cluster/outputs.tf에 출력 변수로 추가할 수 있습니다.

```
output "asg_name" {
 value = aws_autoscaling_group.example.name
 description = "The name of the Auto Scaling Group"
}
```

이제 다음 구문을 사용하여 모듈 출력 변수에 액세스할 수 있습니다.

```
module . <MODULE_NAME> . <OUTPUT_NAME>
```

예를들면 다음과 같습니다.

```
module . frontend . asg_name
```

prod/services/webserver-cluster/main.tf 에서 이 구문을 사용하여 각 aws_autoscaling_schedule 리소스에 autoscaling_group_name 파라미터를 설정할 수 있습니다.

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
 scheduled_action_name = "scale-out-during-business-hours"
 min_size = 2
 max_size = 10
 desired_capacity = 10
 recurrence = "0 9 * * *"
 autoscaling_group_name = module.webserver_cluster.asg_name
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
 scheduled_action_name = "scale-in-at-night"
 min_size = 2
 max_size = 10
 desired_capacity = 2
 recurrence = "0 17 * * *"
 autoscaling_group_name = module.webserver_cluster.asg_name
}
```

webserver-cluster 모듈에서 ALB의 DNS 이름이라는 또 다른 출력을 노출하여 클러스터가 배포될 때 테스트할 URL을 알 수 있습니다. 이를 수행하려면 /modules/services/webserver-cluster/outputs.tf에 출력 변수를 다시 추가합니다.

```
output "alb_dns_name" {
 value = aws_lb.example.dns_name
 description = "The domain name of the load balancer"
}
```

그런 다음 다음과 같이 stage/services/webserver-cluster/outputs.tf 및 prod/services/webserver-cluster/outputs.tf에서 이 출력을 사용할 수 있습니다.

```
output "alb_dns_name" {
 value = module.webserver_cluster.alb_dns_name
 description = "The domain name of the load balancer"
}
```

웹 서버 클러스터를 배포할 준비가 거의 완료되었습니다.

모듈 고려사항

모듈을 만들 때 다음 내용에 주의하세요.

- 파일 경로
- 인라인 블록

파일 경로

이전 챕터에서 웹 서버 클러스터에 대한 사용자 데이터 스크립트를 외부 파일인 user-data.sh로 이동하고 템플릿 파일 내장 기능을 사용하여 디스크에서 이 파일을 읽었습니다. templatefile 함수의 문제점은 사용하는 파일 경로가 상대 경로여야 한다는 것입니다(Terraform 코드는 각기 다른 디스크 레이아웃을 가진 여러 컴퓨터에서 실행될 수 있으므로 절대 파일 경로를 피하는게 좋습니다).

기본적으로 Terraform은 현재 작업 디렉터리를 기준으로 경로를 해석합니다. 이는 terraform Apply를 실행하는 디렉터리와 동일한 디렉터리에 있는 Terraform 구성 파일에서 templatefile 함수를 사용하는 경우(즉, 루트 모듈에서 templatefile 함수를 사용하는 경우) 작동하지만 별도의 폴더에 정의된 모듈(재사용 가능한 모듈)에서 템플릿 파일을 사용하는 경우에는 작동하지 않습니다.
이 문제를 해결하려면 path.<TYPE> 형식의 경로 참조 표현식을 사용할 수 있습니다. Terraform은 다음 유형의 경로 참조를 지원합니다.

- 경로.모듈 (path.module)
  표현식이 정의된 모듈의 파일 시스템 경로를 반환합니다.
- 경로.루트 (path.root)
  루트 모듈의 파일 시스템 경로를 반환합니다.
- 경로.cwd (path.cwd)
  현재 작업 디렉터리의 파일 시스템 경로를 반환합니다. 일반적인 Terraform 사용에서는 path.root 와 동일하지만 Terraform의 일부 고급 사용에서는 루트 모듈 디렉터리가 아닌 다른 디렉터리에서 실행하므로 이러한 경로가 달라집니다.

사용자 데이터 스크립트의 경우 모듈 자체에 상대적인 경로가 필요하므로 module/services/webserver-cluster/main.tf에서 templatefile 함수를 호출할 때 path.module을 사용해야 합니다.

```
user_data = templatefile("${path.module}/user-data.sh", {
 server_port = var.server_port
 db_address = data.terraform_remote_state.db.outputs.address
 db_port = data.terraform_remote_state.db.outputs.port
}
```

인라인 블록

일부 Terraform 리소스에 대한 구성은 인라인 블록 또는 별도의 리소스로 정의될 수 있습니다. 인라인 블록은 다음 형식의 리소스 내에 설정하는 인수입니다.

```
resource "type" "name" {
 <NAME> {
 [CONFIG...]
 }
}
```

여기서 NAME은 인라인 블록의 이름(예: ingress)이고 CONFIG는 해당 인라인 블록(예: from_port 및 to_port)과 관련된 하나 이상의 인수로 구성됩니다. 예를 들어, aws_security_group_resource 를 사용하면 인라인 블록(예: ingress { ... }) 또는 별도의 aws_security_group_rule 리소스를 사용하여 수신 및 송신 규칙을 정의할 수 있습니다.

인라인 블록과 별도의 리소스를 혼합하여 사용하려고 하면 Terraform의 설계 방식으로 인해 구성이 충돌하고 서로 덮어쓰는 오류가 발생합니다. 따라서 둘 중 하나를 사용해야 하며 둘 중에서 별도의 리소스를 사용하는 것을 권장합니다. 별도의 리소스를 사용하면 어디에든 추가할 수 있다는 장점이 있지만, 인라인 블록은 리소스를 생성하는 모듈 내에서만 추가할 수 있습니다. 따라서 별도의 리소스만 사용하면 모듈이 더욱 유연하고 구성 가능해집니다.
예를 들어 webserver-cluster 모듈(modules/services/webserver-cluster/main.tf)에서는 인라인 블록을 사용하여 수신 및 송신 규칙을 정의했습니다.

```
resource "aws_security_group" "alb" {
 name = "${var.cluster_name}-alb"
 ingress {
  from_port = local.http_port
  to_port = local.http_port
  protocol = local.tcp_protocol
  cidr_blocks = local.all_ips
 }
 egress {
  from_port = local.any_port
  to_port = local.any_port
  protocol = local.any_protocol
  cidr_blocks = local.all_ips
 }
}
```

이러한 인라인 블록을 사용하면 이 모듈의 사용자가 모듈 외부에서 추가 수신 또는 송신 규칙을 추가할 수 없습니다. 모듈을 더욱 유연하게 만들려면 별도의 aws_security_group_rule 리소스를 사용하여 정확히 동일한 수신 및 송신 규칙을 정의하도록 모듈을 변경해야 합니다(모듈의 두 보안 그룹 모두에 대해 이 작업을 수행해야 합니다).

```
resource "aws_security_group" "alb" {
 name = "${var.cluster_name}-alb"
}
resource "aws_security_group_rule" "allow_http_inbound" {
 type = "ingress"
 security_group_id = aws_security_group.alb.id
 from_port = local.http_port
 to_port = local.http_port
 protocol = local.tcp_protocol
 cidr_blocks = local.all_ips
}
resource "aws_security_group_rule" "allow_all_outbound" {
 type = "egress"
 security_group_id = aws_security_group.alb.id
 from_port = local.any_port
 to_port = local.any_port
 protocol = local.any_protocol
 cidr_blocks = local.all_ips
}
```

또한 aws_security_group의 ID를 module/services/webserver-cluster/outputs.tf의 출력 변수로 내보내야 합니다.

```
output "alb_security_group_id" {
 value = aws_security_group.alb.id
```

```
 description = "The ID of the Security Group attached to the load balancer"
}
```

이제 스테이징 환경에서만(예: 테스트용) 추가 포트를 노출해야 하는 경우 aws_security_group_rule 리소스를 stage/services/webserver-cluster/main.tf 에 추가하여 이를 수행할 수 있습니다.

```
module "webserver_cluster" {
 source = "../../../modules/services/webserver-cluster"
 # (parameters hidden for clarity)
}
resource "aws_security_group_rule" "allow_testing_inbound" {
 type = "ingress"
 security_group_id = module.webserver_cluster.alb_security_group_id
 from_port = 12345
 to_port = 12345
 protocol = "tcp"
 cidr_blocks = ["0.0.0.0/0"]
}
```

단일 수신 또는 송신 규칙을 인라인 블록으로 정의한 경우 이 코드는 작동하지 않습니다. 이와 동일한 유형의 문제는 다음과 같은 여러 Terraform 리소스에 영향을 미칩니다.

- aws_security_group 그리고 aws_security_group_rule
- aws_route_table 그리고 aws_route
- aws_network_acl 그리고 aws_network_acl_rule

이제 스테이징 및 프로덕션 환경 모두에서 웹 서버 클러스터를 배포할 준비가 되었습니다. 평소처럼 Terraform Apply를 실행하고 인프라의 별도 복사본 두 개를 사용해 보세요.

네트워크 격리
이 장의 예제에서는 Terraform 코드에서 격리되고 별도의 로드 밸런서, 서버 및 데이터베이스가 있다는 측면에서 격리되지만 네트워크 수준에서는 격리되지 않는 두 가지 환경을 만듭니다. 이 책의 모든 예제를 단순하게 유지하기 위해 모든 리소스가 동일한 VPC에 배포됩니다. 즉, 스테이징 환경의 서버는 프로덕션 환경의 서버와 통신할 수 있으며 그 반대의 경우도 마찬가지입니다. 실제 사용 시 하나의 VPC에서 두 환경을 모두 실행하면 최대 두 가지 위험이 발생합니다. 첫째, 한 환경의 실수가 다른 환경에 영향을 미칠 수 있습니다. 예를 들어, 스테이징을 변경하고 실수로 라우팅 테이블의 구성을 망친 경우 프로덕션의 모든 라우팅도 영향을 받을 수 있습니다. 둘째, 공격자가 한 환경에 대한 액세스 권한을 얻으면 다른 환경에도 액세스할 수 있습니다. 스테이징을 빠르게 변경하고 실수로 포트를 노출시킨 경우 침입한 해커는 스테이징 데이터뿐만 아니라 프로덕션 데이터에도 액세스할 수 있습니다. 따라서 간단한 예제와 실험 외에 각 환경을 별도의 VPC에서 실행해야 합니다. 실제로 더욱 확실하게 하기 위해 완전히 별도의 AWS 계정에서 각 환경을 실행할 수도 있습니다.

모듈 버저닝

스테이징 환경과 프로덕션 환경이 모두 동일한 모듈 폴더를 가리키는 경우 해당 폴더를 변경하는 즉시 다음 배포 시 두 환경 모두에 영향을 미칩니다. 이러한 종류의 결합으로 인해 생산에 영향을 미칠 가능성 없이 스테이징 변경을 테스트하기가 더 어려워집니다. 더 나은 접근 방식은 그림과 같이 스테이징에서 한 버전(예: v0.0.2)을 사용하고 프로덕션에서 다른 버전(예: v0.0.1)을 사용할 수 있도록 버전이 지정된 모듈을 만드는 것입니다.
지금까지 본 모든 모듈 예제에서는 모듈을 사용할 때마다 모듈의 소스 매개변수를 로컬 파일 경로로 설정했습니다. 파일 경로 외에도 Terraform은 Git URL, Mercurial URL 및 임의 HTTP URL과 같은 다른 유형의 모듈 소스를 지원합니다.

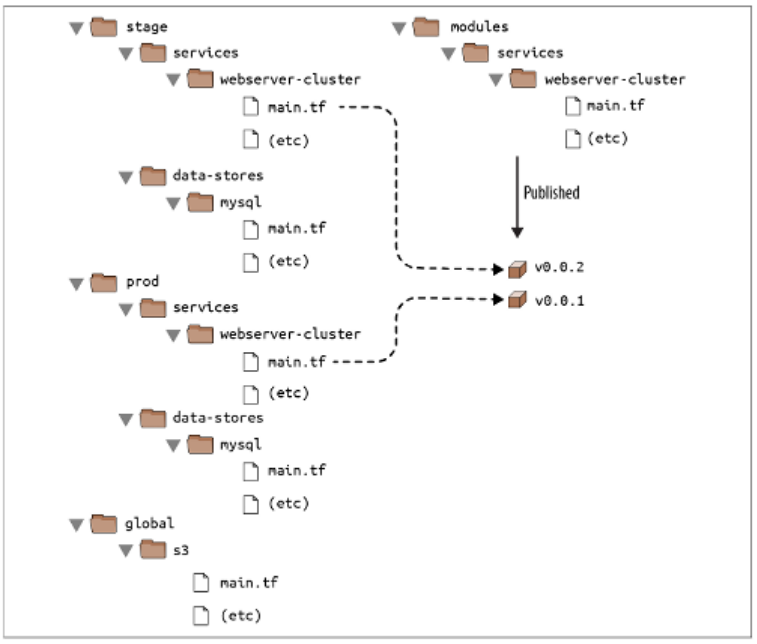


Figure 4-5. By versioning your modules, you can use different versions in different environments: e.g., v0.0.1 in prod and v0.0.2 in stage.

버전이 지정된 모듈을 생성하는 가장 쉬운 방법은 모듈의 코드를 별도의 Git 저장소에 넣고 소스 매개변수를 해당 저장소의 URL로 설정하는 것입니다. 이는 Terraform 코드가 (적어도) 두 개의 저장소에 분산된다는 의미입니다.

- 모듈
  이 저장소는 재사용 가능한 모듈을 정의합니다. 각 모듈을 인프라의 특정 부분을 정의하는 "청사진"으로 생각하십시오.
- live
  이 저장소는 각 환경(단계, 프로덕션, 관리 등)에서 실행 중인 라이브 인프라를 정의합니다. 이것을 모듈 저장소의 "청사진"에서 만든 "집"이라고 생각하세요.

Terraform 코드의 업데이트된 폴더 구조는 이제 그림과 같습니다.

Figure 4-6. You should store reusable, versioned modules in one repo (modules) and the configuration for your live environments in another repo (live).

이 폴더 구조를 설정하려면 먼저 stage, prod 및 global 폴더를 live라는 폴더로 이동해야 합니다. 다음으로, 라이브 및 모듈 폴더를 별도의 Git 리포지토리로 구성합니다. 다음은 모듈 폴더에 대해 이를 수행하는 방법의 예입니다.

```
$ cd modules
$ git init
```

```
$ git add .
$ git commit -m "Initial commit of modules repo"
$ git remote add origin "(URL OF REMOTE GIT REPOSITORY)"
$ git push origin main
```

버전 번호로 사용하기 위해 모듈 저장소에 태그를 추가할 수도 있습니다. GitHub를 사용하는 경우 GitHub UI를 사용하여 릴리스를 생성하면 내부적으로 태그가 생성됩니다.
GitHub를 사용하지 않는 경우 Git CLI를 사용할 수 있습니다.

```
$ git tag -a "v0.0.1" -m "First release of webserver-cluster module"
$ git push --follow-tags
```

이제 소스 매개변수에 Git URL을 지정하여 스테이징 및 프로덕션 모두에서 이 버전이 지정된 모듈을 사용할 수 있습니다. 모듈 저장소가 GitHub 저장소 github.com/foo/modules에 있는 경우 live/stage/services/webserver-cluster/main.tf의 모습은 다음과 같습니다(다음 Git URL에서 이중 슬래시를 사용하는 구문을 잘 확인 하세요):

```
module "webserver_cluster" {
 source = "github.com/foo/modules//services/webserver-cluster?ref=v0.0.1"
 cluster_name = "webservers-stage"
 db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
 db_remote_state_key = "stage/data-stores/mysql/terraform.tfstate"
 instance_type = "t2.micro"
 min_size = 2
 max_size = 2
}
```

ref 매개변수를 사용하면 sha1 해시, 브랜치 이름 또는 이 예에서와 같이 특정 Git 태그를 통해 특정 Git 커밋을 지정할 수 있습니다. 일반적으로 Git 태그를 모듈의 버전 번호로 사용하는 것이 좋습니다. 브랜치 이름은 항상 브랜치에 대한 최신 커밋을 가져오므로 init 명령을 실행할 때마다 변경될 수 있고 sha1 해시는 사람에게 친숙하지 않기 때문에 안정적이지 않습니다. Git 태그는 커밋만큼 안정적이지만(사실 태그는 커밋에 대한 포인터일 뿐입니다) 친숙하고 읽기 쉬운 이름을 사용할 수 있습니다.

태그는 의미적으로 알아볼수 있는 버전 관리용 정보입니다. 이는 버전 번호의 각 부분을 증가시켜야 하는 경우에 대한 특정 규칙이 있는 MAJOR.MINOR.PATCH(예: 1.0.4) 형식의 버전 관리 체계입니다. 특히 다음을 증가시켜야 합니다.

- 호환되지 않는 API 변경 시 MAJOR 버전
- 이전 버전과 호환되는 방식으로 기능을 추가하는 경우 MINOR 버전
- 이전 버전과 호환되는 버그 수정 시 PATCH 버전

시맨틱(Semantic Tag) 버전 관리를 사용하면 어떤 종류의 변경 사항을 적용했고 업그레이드에 따른 영향을 모듈 사용자에게 알릴 수 있습니다.

버전이 지정된 모듈 URL을 사용하도록 Terraform 코드를 업데이트했기 때문에 terraform init를 다시 실행하여 Terraform에 모듈 코드를 다운로드하도록 지시해야 합니다.

```
$ terraform init
Initializing modules...
Downloading git@github.com:username/repository.git?ref=v0.0.1
for webserver_cluster...
(...)
```

이번에는 Terraform이 로컬 파일 시스템이 아닌 Git에서 모듈 코드를 다운로드하는 것을 볼 수 있습니다. 모듈 코드를 다운로드한 후 평소처럼 적용 명령을 실행할 수 있습니다.

이제 버전이 지정된 모듈을 사용하고 있으므로 변경 프로세스를 살펴보겠습니다. webserver-cluster 모듈을 일부 변경했고 이를 스테이징에서 테스트하고 싶다고 가정해 보겠습니다. 먼저 해당 변경 사항을 모듈 저장소에 커밋합니다.

```
$ cd modules
$ git add .
$ git commit -m "Made some changes to webserver-cluster"
$ git push origin main
```

다음으로 모듈 저장소에 새 태그를 만듭니다.

```
$ git tag -a "v0.0.2" -m "Second release of webserver-cluster"
$ git push --follow-tags
```

이제 스테이징 환경(live/ stage/services/webserver-cluster/main.tf)에서 사용되는 소스 URL만 업데이트하여 이 새 버전을 사용할 수 있습니다.

```
module "webserver_cluster" {
 source = "github.com/foo/modules//services/webserver-cluster?ref=v0.0.2"
 cluster_name = "webservers-stage"
 db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
 db_remote_state_key = "stage/data-stores/mysql/terraform.tfstate"
 instance_type = "t2.micro"
 min_size = 2
 max_size = 2
}
```

프로덕션(live/prod/services/webserver-cluster/main.tf)에서는 v0.0.1을 변경 없이 계속 실행할 수 있습니다.

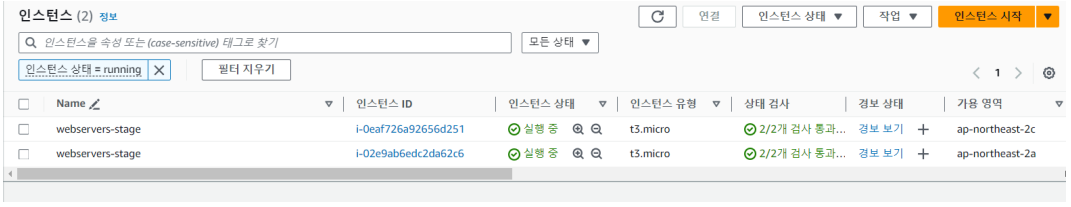
```
module "webserver_cluster" {
 source = "github.com/foo/modules//services/webserver-cluster?ref=v0.0.1"
 cluster_name = "webservers-prod"
 db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
 db_remote_state_key = "prod/data-stores/mysql/terraform.tfstate"
 instance_type = "m4.large"
 min_size = 2
 max_size = 10
}
```

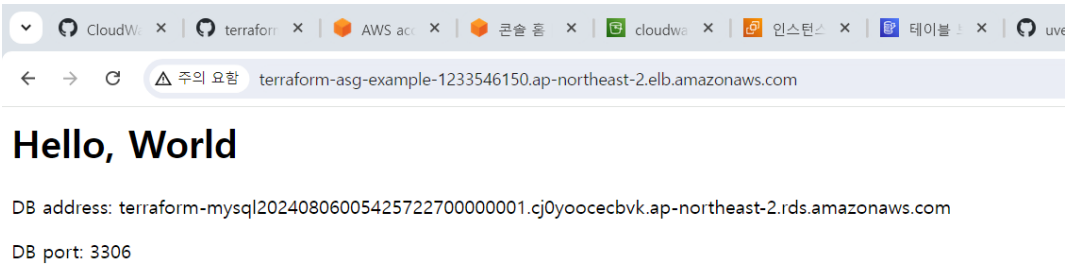
v0.0.2가 철저한 테스트를 거쳐 준비 단계에서 입증된 후에는 프로덕션 버전도 업데이트할 수 있습니다. 그러나 v0.0.2에 버그가 있는 것으로 밝혀진다고 해도 큰 문제는 아닙니다. 프로덕션 환경의 실제 사용자에게는 아무런 영향도 미치지 않기 때문입니다. 버그를 수정하고, 새 버전을 출시하고, 프로덕션에 충분히 안정적인 버전이 나올 때까지 전체 프로세스를 다시 반복하세요.

> 💡 모듈 개발
> 모듈 버전이 지정된 모듈은 공유 환경(예: 준비 또는 프로덕션)에 배포할 때 유용하지만 자신의 컴퓨터에서 테스트할 때는 로컬 파일 경로를 사용하는 것이 좋습니다. 이렇게 하면 코드를 커밋하고 새 버전을 게시하고 init를 다시 실행할 필요 없이 모듈 폴더를 변경하고 라이브 폴더에서 즉시 계획을 다시 실행하거나 명령을 적용할 수 있으므로 더 빠르게 반복할 수 있습니다. 매번. 이 책의 목표는 가능한 한 빨리 Terraform을 배우고 실험하도록 돕는 것이므로 나머지 코드 예제에서는 모듈에 대한 로컬 파일 경로를 사용합니다.

결과

```
[root@controller terraform-demo2]# tree
.
├── global
│   └── s3
│       └── main.tf
├── modules
│   ├── data-stores
│   │   └── mysql
│   │       ├── backend.tf
│   │       ├── main.tf
│   │       ├── output.tf
│   │       └── variable.tf
│   └── services
│       └── webserver-cluster
│           ├── data.tf
│           ├── local.tf
│           ├── main.tf
│           ├── terraform.tfstate
│           ├── terraform.tfstate.backup
│           ├── user-data.sh
│           └── variable.tf
├── prod
│   └── services
│       └── webserver-cluster
│           ├── main.tf
│           └── outputs.tf
└── stage
    └── services
        └── webserver-cluster
            ├── main.tf
            ├── outputs.tf
            ├── terraform.tfstate
            └── terraform.tfstate.backup

13 directories, 18 files
```






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

Outputs:

alb_dns_name = "terraform-asg-example-1233546150.ap-northeast-2.elb.amazonaws.com"
```

결론

인프라를 모듈의 코드로 정의하면 다양한 소프트웨어 엔지니어링 모범 사례를 인프라에 적용할 수 있습니다. 코드 검토 및 자동화된 테스트를 통해 모듈에 대한 각 변경 사항을 검증할 수 있고, 각 모듈의 의미론적 버전 릴리스를 생성할 수 있으며, 다양한 환경에서 모듈의 다양한 버전을 안전하게 시험해보고 문제가 발생하면 이전 버전으로 롤백할 수 있습니다. .
개발자는 입증되고, 테스트되고, 문서화된 인프라의 전체 부분을 재사용할 수 있기 때문에 이 모든 것이 인프라를 빠르고 안정적으로 구축하는 능력을 크게 향상시킬 수 있습니다. 예를 들어 클러스터 실행 방법, 로드에 따라 클러스터 크기를 조정하는 방법, 클러스터 전체에 트래픽 요청을 분산하는 방법 등 단일 마이크로서비스를 배포하는 방법을 정의하는 표준 모듈을 생성할 수 있으며, 각 팀은 다음을 사용할 수 있습니다. 이 모듈을 사용하면 단 몇 줄의 코드만으로 자체 마이크로서비스를 관리할 수 있습니다.
이러한 모듈이 여러 팀에서 작동하도록 하려면 해당 모듈의 Terraform 코드가 유연하고 구성 가능해야 합니다. 예를 들어, 한 팀에서는 모듈을 사용하여 로드 밸런서 없이 마이크로 서비스의 단일 인스턴스를 배포하려고 하는 반면, 다른 팀에서는 해당 인스턴스 간에 트래픽을 분산하기 위해 로드 밸런서가 있는 마이크로 서비스의 12개 인스턴스를 배포하기를 원할 수 있습니다.

실습 4 (제어문과 무중단 배포)

Terraform은 선언적 언어입니다. 처음에 설명한 것처럼 선언적 언어의 IaC는 절차적 언어보다 실제로 배포된 내용에 대한 더 정확한 정보를 제공하는 경향이 있으므로 추론하기 쉽고 코드베이스를 작게 유지하는 것이 더 쉽습니다.

그러나 특정 유형의 작업은 선언적 언어에서 더 어렵습니다. 예를 들어, 선언적 언어에는 일반적으로 for 루프가 없기 때문에 복사 및 붙여넣기 없이 여러 개의 유사한 리소스 생성과 같은 논리를 어떻게 반복할까요? 그리고 선언적 언어가 if 문을 지원하지 않는 경우 해당 모듈의 일부 사용자에게는 특정 리소스를 생성할 수 있지만 다른 사용자에게는 생성되지 않는 Terraform 모듈을 생성하는 등 조건부로 리소스를 어떻게 구성할 수 있을까요? 마지막으로 중단 시간 없는 배포와 같은 본질적인 절차적 아이디어를 선언적 언어로 어떻게 표현할까요?
다행스럽게도 Terraform은 메타 매개변수 count, for_each 및 for 표현식, 삼항 연산자, create_before_destroy라는 수명 주기 블록 등 특정 유형의 루프를 수행할 수 있는 다수의 함수 몇 가지를 기본 요소로 제공합니다. 명령문 및 가동 중지 시간 없는 배포. 이번 장에서 다룰 주제는 다음과 같습니다.

- 루프

- 조건부
- 다운타임 없는 배포
- Terraform 문제

루프

Terraform은 각각 약간 다른 시나리오에서 사용하도록 고안된 여러 가지 루프 구성을 제공합니다.

- 자원과 모듈을 반복하기 위한 count 매개변수
- for_each 표현식, 리소스, 리소스 내의 인라인 블록 및 모듈을 반복합니다.
- 표현식의 경우 목록과 맵을 반복합니다.
- 문자열 지시문의 경우 문자열 내의 목록과 맵을 반복합니다.

한 번에 하나씩 살펴보겠습니다.

count 매개변수를 사용한 루프

이전에 AWS Identity and Access Management(IAM) 사용자를 생성했습니다. 이제 이 사용자가 있으므로 Terraform을 사용하여 향후 모든 IAM 사용자를 생성하고 관리할 수 있습니다. live/global/iam/main.tf 에 다음 Terraform 코드를 작성합니다.

```
provider "aws" {
 region = "ap-northeast-2"
}
resource "aws_iam_user" "example" {
 name = "neo"
}
```

이 코드는 aws_iam_user 리소스를 사용하여 단일 새 IAM 사용자를 생성합니다. 세 명의 IAM 사용자를 생성하려면 어떻게 해야 합니까? 범용 프로그래밍 언어에서는 아마도 for-loop를 사용할 것입니다:

```
# This is just pseudo code. It won't actually work in Terraform.
for (i = 0; i < 3; i++) {
 resource "aws_iam_user" "example" {
 name = "neo"
 }
}
```

Terraform에는 for-loop 또는 기타 기존 절차 논리가 언어에 내장되어 있지 않으므로 이 구문은 작동하지 않습니다. 그러나 모든 Terraform 리소스에는 count 라는 사용할 수 있는 메타 매개변수가 있습니다. count는 Terraform의 가장 오래되고 단순하며 가장 제한적인 반복 구성입니다. 이것이 하는 일은 생성할 리소스의 복사본 수를 정의하는 것뿐입니다. count를 사용하여 3명의 IAM 사용자를 생성하는 방법은 다음과 같습니다.

```
resource "aws_iam_user" "example" {
 count = 3
 name = "neo"
}
```

이 코드의 한 가지 문제점은 세 명의 IAM 사용자가 모두 동일한 이름을 갖게 되어 사용자 이름이 고유해야 하기 때문에 오류가 발생한다는 것입니다. 표준 for 루프에 액세스할 수 있는 경우 for 루프의 인덱스 i 를 사용하여 각 사용자에게 고유한

이름을 지정할 수 있습니다.

```
# This is just pseudo code. It won't actually work in Terraform.
for (i = 0; i < 3; i++) {
 resource "aws_iam_user" "example" {
 name = "neo.${i}"
 }
}
```

Terraform에서 동일한 작업을 수행하려면 count.index를 사용하여 "루프"의 각 "반복" 인덱스를 가져올 수 있습니다.

```
resource "aws_iam_user" "example" {
 count = 3
 name = "neo.${count.index}"
}
```

앞의 코드에서 plan 명령을 실행하면 Terraform이 각각 다른 이름("neo.0" , "neo.1" , "neo.2")을 가진 3명의 IAM 사용자를 생성하려고 한다는 것을 알 수 있습니다.

```
Terraform will perform the following actions:
 # aws_iam_user.example[0] will be created
 + resource "aws_iam_user" "example" {
  + name = "neo.0"
  (...)
 }
 # aws_iam_user.example[1] will be created
 + resource "aws_iam_user" "example" {
  + name = "neo.1"
  (...)
 }
 # aws_iam_user.example[2] will be created
 + resource "aws_iam_user" "example" {
  + name = "neo.2"
  (...)
 }
Plan: 3 to add, 0 to change, 0 to destroy.
```

물론 "neo.0"과 같은 사용자 이름은 특별히 사용할 수 없습니다. count.index를 Terraform의 일부 내장 함수와 결합하면 "루프"의 각 "반복"을 훨씬 더 맞춤 설정할 수 있습니다.
예를 들어 live/global/iam/variables.tf의 입력 변수에 원하는 모든 IAM 사용자 이름을 정의할 수 있습니다.

```
variable "user_names" {
 description = "Create IAM users with these names"
 type = list(string)
 default = ["neo", "trinity", "morpheus"]
}
```

루프와 배열이 포함된 범용 프로그래밍 언어를 사용하는 경우 var.user_names 배열에서 인덱스 i를 조회하여 각 IAM 사용자가 다른 이름을 사용하도록 구성합니다.

```
# This is just pseudo code. It won't actually work in Terraform.
for (i = 0; i < 3; i++) {
 resource "aws_iam_user" "example" {
 name = vars.user_names[i]
 }
}
```

Terraform에서는 다음과 함께 count를 사용하여 동일한 작업을 수행할 수 있습니다.

- 배열 조회 구문
  Terraform에서 배열 멤버를 검색하는 구문은 대부분의 다른 프로그래밍 언어와 유사합니다.

  ```
  ARRAY[<INDEX>]
  ```

  예를 들어, var.user_names의 인덱스 1에서 요소를 찾는 방법은 다음과 같습니다.

  ```
  var.user_names[1]
  ```

- 길이 함수
  Terraform에는 다음 구문을 포함하는 length라는 내장 함수가 있습니다.

  ```
  length(<ARRAY>)
  ```

  length 함수는 지정된 ARRAY에 있는 항목 수를 반환합니다. 문자열과 map에서도 작동합니다.

이를 종합하면 다음과 같은 결과를 얻을 수 있습니다.

```
resource "aws_iam_user" "example" {
 count = length(var.user_names)
 name = var.user_names[count.index]
}
```

이제 plan 명령을 실행하면 Terraform이 각각 고유하고 읽을 수 있는 이름을 가진 3명의 IAM 사용자를 생성하려고 한다는 것을 알 수 있습니다.

```
Terraform will perform the following actions:
 # aws_iam_user.example[0] will be created
 + resource "aws_iam_user" "example" {
  + name = "neo"
  (...)
 }
 # aws_iam_user.example[1] will be created
 + resource "aws_iam_user" "example" {
  + name = "trinity"
  (...)
 }
 # aws_iam_user.example[2] will be created
 + resource "aws_iam_user" "example" {
  + name = "morpheus"
  (...)
```

```
 }
Plan: 3 to add, 0 to change, 0 to destroy.
```

리소스에 count를 사용한 후에는 리소스가 하나가 아닌 리소스 배열이 된다는 점에 유의하세요. aws_iam_user.example은 이제 IAM 사용자의 배열이므로 해당 리소스(<PROVIDER>_<TYPE>.<NAME>. <ATTRIBUTE>)에서 속성을 읽는 표준 구문을 사용하는 대신 어떤 IAM 사용자를 지정해야 합니다. 동일한 배열 조회 구문을 사용하여 배열에 해당 인덱스를 지정해야 합니다.

```
<PROVIDER>_<TYPE>.<NAME>[INDEX].ATTRIBUTE
```

예를 들어 목록에 있는 첫 번째 IAM 사용자의 Amazon 리소스 이름(ARN)을 출력 변수로 제공하려면 다음을 수행해야 합니다.

```
output "first_arn" {
 value = aws_iam_user.example[0].arn
 description = "The ARN for the first user"
}
```

모든 IAM 사용자의 ARN을 원할 경우 인덱스 대신 표현식 " * "을 사용해야 합니다.

```
output "all_arns" {
 value = aws_iam_user.example[*].arn
 description = "The ARNs for all users"
}
```

Apply 명령을 실행하면 first_arn 출력에는 neo 에 대한 ARN만 포함되는 반면, all_arns 출력에는 모든 ARN 목록이 포함됩니다.

```
$ terraform apply
(...)
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:
first_arn = "arn:aws:iam::123456789012:user/neo"
all_arns = [
 "arn:aws:iam::123456789012:user/neo",
 "arn:aws:iam::123456789012:user/trinity",
 "arn:aws:iam::123456789012:user/morpheus",
]
```

Terraform 0.13부터 count 매개변수를 모듈에서도 사용할 수 있습니다. 예를 들어, 단일 IAM 사용자를 생성할 수 있는 모듈이 module/landing-zone/iam-user에 있다고 가정해 보겠습니다.

```
resource "aws_iam_user" "example" {
 name = var.user_name
}
```

사용자 이름은 이 모듈에 입력 변수로 전달됩니다.

```
variable "user_name" {
 description = "The user name to use"
```

```
 type = string
}
```

그리고 모듈은 생성된 IAM 사용자의 ARN을 출력 변수로 반환합니다.

```
output "user_arn" {
 value = aws_iam_user.example.arn
 description = "The ARN of the created IAM user"
}
```

이 모듈을 count 매개변수와 함께 사용하여 다음과 같이 3명의 IAM 사용자를 생성할 수 있습니다.

```
module "users" {
 source = "../../../modules/landing-zone/iam-user"
 count = length(var.user_names)
 user_name = var.user_names[count.index]
}
```

앞의 코드는 count를 사용하여 이 사용자 이름 목록을 반복합니다.

```
variable "user_names" {
 description = "Create IAM users with these names"
 type = list(string)
 default = ["neo", "trinity", "morpheus"]
}
```

그리고 생성된 IAM 사용자의 ARN을 다음과 같이 출력합니다.

```
output "user_arns" {
 value = module.users[*].user_arn
 description = "The ARNs of the created IAM users"
}
```

리소스에 개수를 추가하면 리소스 배열이 되는 것처럼, 모듈에 개수를 추가하면 모듈 배열이 됩니다.
이 코드에 대해 Apply를 실행하면 다음과 같은 출력이 표시됩니다.

```
$ terraform apply
(...)
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:
all_arns = [
 "arn:aws:iam::123456789012:user/neo",
 "arn:aws:iam::123456789012:user/trinity",
 "arn:aws:iam::123456789012:user/morpheus",
]
```

따라서 보시다시피 count는 리소스 및 모듈과 거의 동일하게 작동합니다.
불행하게도 count에는 유용성을 크게 감소시키는 두 가지 제한 사항이 있습니다. 첫째, count를 사용하여 전체 리소스를 반복할 수 있지만 리소스 내에서 count를 사용하여 인라인 블록을 반복할 수는 없습니다.
예를 들어 aws_autoscaling_group 리소스에서 태그가 어떻게 설정되는지 생각해 보세요.

```
resource "aws_autoscaling_group" "example" {
 launch_configuration = aws_launch_configuration.example.name
 vpc_zone_identifier = data.aws_subnets.default.ids
 target_group_arns = [aws_lb_target_group.asg.arn]
 health_check_type = "ELB"
 min_size = var.min_size
 max_size = var.max_size
 tag {
  key = "Name"
  value = var.cluster_name
  propagate_at_launch = true
 }
}
```

각 태그에는 key , value 및 propagate_at_launch 값이 포함된 새 인라인 블록을 생성해야 합니다. 앞의 코드는 단일 태그를 하드코딩하지만 사용자가 사용자 정의 태그를 전달하도록 허용할 수도 있습니다. count 매개변수를 사용하여 이러한 태그를 반복하고 동적 인라인 태그 블록을 생성하려고 시도할 수도 있지만 안타깝게도 인라인 블록 내에서 count를 사용하는 것은 지원되지 않습니다.
count의 두 번째 제한은 해당 값을 변경하려고 할 때 발생합니다. 이전에 생성한 IAM 사용자 목록을 고려해보세요.

```
variable "user_names" {
 description = "Create IAM users with these names"
 type = list(string)
 default = ["neo", "trinity", "morpheus"]
}
```

이 목록에서 "trinity"를 제거했다고 상상해 보세요. Terraform plan을 실행하면 어떻게 되나요?

```
$ terraform plan
(...)
Terraform will perform the following actions:
 # aws_iam_user.example[1] will be updated in-place
 ~ resource "aws_iam_user" "example" {
     id = "trinity"
   ~ name = "trinity" -> "morpheus"
 }
 # aws_iam_user.example[2] will be destroyed
 - resource "aws_iam_user" "example" {
  - id = "morpheus" -> null
  - name = "morpheus" -> null
 }
Plan: 0 to add, 1 to change, 1 to destroy.
```

아마도 여러분이 기대했던 것이 아닐 수도 있습니다! 계획 출력에는 "trinity" IAM 사용자를 삭제하는 대신 Terraform이 "trinity" IAM 사용자의 이름을 "morpheus"로 바꾸고 원래 "morpheus" 사용자를 삭제하려고 함을 나타냅니다.
리소스에 count 매개 변수를 사용하면 해당 리소스는 리소스 배열이 됩니다. 안타깝게도 Terraform이 배열 내의 각 리소스를 식별하는 방식은 해당 배열의 위치(인덱스)를 기준으로 합니다. 즉, 세 개의 사용자 이름으로 처음 적용을 실행한 후 이러한 IAM 사용자에 대한 Terraform의 내부 표현은 다음과 같습니다.

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: trinity
```

```
aws_iam_user.example[2]: morpheus
```

배열 중간에서 항목을 제거하면 그 뒤의 모든 항목이 하나씩 뒤로 이동하므로 두 개의 버킷 이름만으로 계획을 실행한 후 Terraform의 내부 표현은 다음과 같습니다.

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: morpheus
```

"morpheus"가 인덱스 2에서 인덱스 1로 어떻게 이동했는지 확인하세요. 인덱스를 리소스의 ID인 Terraform으로 간주하기 때문에 이 변경은 대략 "인덱스 1의 버킷 이름을 morpheus로 바꾸고 인덱스 2의 버킷을 삭제"하는 것으로 해석됩니다. 즉, count를 사용하여 리소스 목록을 생성할 때마다 목록 중간에서 항목을 제거하면 Terraform은 해당 항목 이후의 모든 리소스를 삭제한 다음 해당 리소스를 처음부터 다시 생성합니다. 물론 최종 결과는 요청한 것과 정확히 일치하지만, 리소스를 삭제하는 것은 가용성을 잃을 수 있으므로 좋은 방법이 아닐 수 있습니다. 더 나쁜 경우에는 데이터가 손실될 수 있습니다(삭제하는 리소스가 데이터베이스인 경우 그 안에 있는 모든 데이터가 손실될 수 있습니다!). 이러한 두 가지 제한 사항을 해결하기 위해 Terraform 0.12에서는 for_each 표현식을 도입했습니다.

for_each 표현식을 사용한 루프

for_each 표현식을 사용하면 목록, 세트 및 맵을 반복하여 (a) 전체 리소스의 여러 복사본, (b) 리소스 내 인라인 블록의 여러 복사본 또는 (c) 모듈의 여러 복사본을 만들 수 있습니다. 먼저 for_each를 사용하여 리소스의 여러 복사본을 만드는 방법을 살펴보겠습니다.
구문은 다음과 같습니다.

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
 for_each = <COLLECTION>
 [CONFIG ...]
}
```

여기서 COLLECTION은 반복할 세트 또는 맵이고(리소스에서 for_each를 사용할 때 목록은 지원되지 않음) CONFIG는 해당 리소스에 특정한 하나 이상의 인수로 구성됩니다. CONFIG 내에서 Each.key 및 Each.value를 사용하여 COLLECTION에 있는 현재 항목의 키와 값에 액세스할 수 있습니다.

예를 들어 리소스에서 for_each를 사용하여 동일한 3명의 IAM 사용자를 생성하는 방법은 다음과 같습니다.

```
resource "aws_iam_user" "example" {
 for_each = toset(var.user_names)
 name = each.value
}
```

var.user_names 목록을 세트로 변환하려면 toset을 사용하십시오. 이는 for_each가 리소스에 사용될 때만 세트와 맵을 지원하기 때문입니다. for_each가 이 세트를 반복하면 각 사용자 이름을 each.value에서 사용할 수 있게 됩니다. 사용자 이름은 each.key 에서도 사용할 수 있지만 일반적으로 키-값 쌍의 맵에만 each.key를 사용합니다.

리소스에 for_each를 사용하면 이는 단지 하나의 리소스(또는 count와 같은 리소스 배열)가 아닌 리소스 맵이 됩니다. 이것이 의미하는 바를 확인하려면 원래 all_arns 및 first_arn 출력 변수를 제거하고 새 all_users 출력 변수를 추가합니다

```
output "all_users" {
 value = aws_iam_user.example
}
```

Terraform Apply를 실행하면 다음과 같은 일이 발생합니다.

```
$ terraform apply
(...)
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:
all_users = {
 "morpheus" = {
  "arn" = "arn:aws:iam::123456789012:user/morpheus"
  "force_destroy" = false
  "id" = "morpheus"
  "name" = "morpheus"
  "path" = "/"
  "tags" = {}
 }
 "neo" = {
  "arn" = "arn:aws:iam::123456789012:user/neo"
  "force_destroy" = false
  "id" = "neo"
  "name" = "neo"
  "path" = "/"
  "tags" = {}
 }
 "trinity" = {
  "arn" = "arn:aws:iam::123456789012:user/trinity"
  "force_destroy" = false
  "id" = "trinity"
  "name" = "trinity"
  "path" = "/"
  "tags" = {}
 }
}
```

Terraform이 세 명의 IAM 사용자를 생성했고 all_users 출력 변수에 키가 for_each의 키(이 경우 사용자 이름)이고 값이 해당 리소스에 대한 모든 출력인 맵이 포함되어 있음을 확인할 수 있습니다. all_arns 출력 변수를 다시 가져오려면 value 내장 함수(맵에서 값만 반환)와 표시 표현식을 사용하여 해당 ARN을 추출하기 위한 약간의 추가 작업을 수행해야 합니다.

```
output "all_arns" {
 value = values(aws_iam_user.example)[*].arn
}
```

그러면 예상되는 출력이 제공됩니다.

```
$ terraform apply
(...)
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
all_arns = [
 "arn:aws:iam::123456789012:user/morpheus",
 "arn:aws:iam::123456789012:user/neo",
 "arn:aws:iam::123456789012:user/trinity",
]
```

이제 count와 같은 리소스 배열 대신 for_each를 사용하여 리소스 맵을 갖게 되었다는 사실은 큰 변경점 입니다. 컬렉션 중간에서 항목을 안전하게 제거할 수 있기 때문입니다. 예를 들어 var.user_names 목록 중간에서 "trinity"를 다시 제거하고 terraform plan 을 실행하면 다음과 같이 표시됩니다.

```
$ terraform plan
Terraform will perform the following actions:
 # aws_iam_user.example["trinity"] will be destroyed
 - resource "aws_iam_user" "example" {
 - arn = "arn:aws:iam::123456789012:user/trinity" -> null
 - name = "trinity" -> null
 }
Plan: 0 to add, 0 to change, 1 to destroy.
```

바로 이거죠! 이제 다른 모든 리소스를 이동하지 않고 원하는 정확한 리소스만 삭제하고 있습니다. 이것이 리소스의 여러 복사본을 생성할 때 거의 항상 count 대신 for_each를 사용하는 것을 선호해야 하는 이유입니다.

for_each는 거의 동일한 방식으로 모듈과 작동합니다. 이전의 iam-user 모듈을 사용하면 다음과 같이 for_each를 사용하여 3명의 IAM 사용자를 생성할 수 있습니다.

```
module "users" {
 source = "../../../modules/landing-zone/iam-user"
 for_each = toset(var.user_names)
 user_name = each.value
}
```

그리고 다음과 같이 해당 사용자의 ARN을 출력할 수 있습니다.

```
output "user_arns" {
 value = values(module.users)[*].user_arn
 description = "The ARNs of the created IAM users"
}
```

이 코드에 대해 적용을 실행하면 예상되는 출력을 얻습니다.

```
$ terraform apply
(...)
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:
all_arns = [
 "arn:aws:iam::123456789012:user/morpheus",
 "arn:aws:iam::123456789012:user/neo",
 "arn:aws:iam::123456789012:user/trinity",
]
```

이제 for_each의 또 다른 장점, 즉 리소스 내에서 여러 인라인 블록을 생성하는 기능에 주목해 보겠습니다. 예를 들어 for_each를 사용하면 webserver-cluster 모듈에서 ASG에 대한 태그 인라인 블록을 동적으로 생성할 수 있습니다. 먼저, 사용자가 사용자 정의 태그를 지정할 수 있도록 하려면 module/services/webserver-cluster/variables.tf에 custom_tags라는 새 맵 입력 변수를 추가하십시오.

```
variable "custom_tags" {
 description = "Custom tags to set on the Instances in the ASG"
 type = map(string)
```

```
 default = {}
}
```

그런 다음 프로덕션 환경의 live/prod/services/ webserver-cluster/main.tf 에서 다음과 같이 일부 사용자 정의 태그를 설정하십시오.

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key = "prod/data-stores/mysql/terraform.tfstate"
  instance_type = "m4.large"
  min_size = 2
  max_size = 10
  custom_tags = {
    Owner = "team-foo"
    ManagedBy = "terraform"
  }
}
```

이전 코드는 몇 가지 유용한 태그를 설정합니다. Owner 태그는 이 ASG를 소유한 팀을 지정하고 ManagedBy 태그는 이 인프라가 Terraform을 사용하여 관리됨을 지정합니다(이 인프라를 수동으로 수정해서는 안 됨을 나타냄).

이제 태그를 지정했으므로 aws_autoscaling_group 리소스에서 태그를 실제로 어떻게 설정합니까? 필요한 것은 다음 코드와 유사한 var.custom_tags에 대한 for 루프입니다.

```
resource "aws_autoscaling_group" "example" {
 launch_configuration = aws_launch_configuration.example.name
 vpc_zone_identifier = data.aws_subnets.default.ids
 target_group_arns = [aws_lb_target_group.asg.arn]
 health_check_type = "ELB"
 min_size = var.min_size
 max_size = var.max_size
 tag {
  key = "Name"
  value = var.cluster_name
  propagate_at_launch = true
 }

 for (tag in var.custom_tags) {
  tag {
   key = tag.key
   value = tag.value
   propagate_at_launch = true
  }
 }
}
```

앞의 코드는 실제로 동작하지는 않지만 작동하지 않지만 for 대신 for_each 표현식은 작동합니다. 인라인 블록을 동적으로 생성하기 위해 for_each를 사용하는 구문은 다음과 같습니다.

```
dynamic "<VAR_NAME>" {
 for_each = <COLLECTION>
 content {
 [CONFIG...]
 }
}
```

여기서 VAR_NAME은 각 "반복"의 값을 저장할 변수에 사용할 이름이고, COLLECTION은 반복할 목록 또는 맵이며, 콘텐츠 블록은 각 반복에서 생성할 항목입니다. 콘텐츠 블록 내에서 <VAR_NAME>.key 및 <VAR_NAME>.value를 사용하여 각각 COLLECTION에 있는 현재 항목의 키와 값에 액세스할 수 있습니다. 목록과 함께 for_each를 사용하는 경우 키는 인덱스가 되고 값은 해당 인덱스에 있는 목록의 항목이 되며, 맵과 함께 for_each를 사용하는 경우 키와 값은 다음 중 하나가 됩니다. 맵의 키-값 쌍.
이 모든 것을 종합하면 aws_autoscaling_group 리소스에서 for_each를 사용하여 태그 블록을 동적으로 생성하는 방법은 다음과 같습니다.

```
resource "aws_autoscaling_group" "example" {
 launch_configuration = aws_launch_configuration.example.name
 vpc_zone_identifier = data.aws_subnets.default.ids
 target_group_arns = [aws_lb_target_group.asg.arn]
 health_check_type = "ELB"
 min_size = var.min_size
 max_size = var.max_size
 tag {
  key = "Name"
  value = var.cluster_name
  propagate_at_launch = true
 }
  dynamic "tag" {
  for_each = var.custom_tags
  content {
   key = tag.key
   value = tag.value
   propagate_at_launch = true
  }
 }
}
```

지금 Terraform 계획을 실행하면 다음과 같은 계획이 표시됩니다.

```
$ terraform plan
Terraform will perform the following actions:
 # aws_autoscaling_group.example will be updated in-place
 ~ resource "aws_autoscaling_group" "example" {
  (...)
  tag {
   key = "Name"
   propagate_at_launch = true
   value = "webservers-prod"
  }
  + tag {
   + key = "Owner"
```

```
    + propagate_at_launch = true
    + value = "team-foo"
   }
   + tag {
    + key = "ManagedBy"
    + propagate_at_launch = true
    + value = "terraform"
   }
  }
Plan: 0 to add, 1 to change, 0 to destroy.
```

for 표현식을 사용한 루프

이제 루프를 사용하여 전체 리소스 및 인라인 블록의 여러 복사본을 만드는 방법을 살펴보았습니다. 하지만 단일 변수나 매개변수를 설정하기 위해 루프가 필요한 경우 어떻게 해야 할까요?
이름 목록을 가져오는 Terraform 코드를 작성했다고 가정해 보겠습니다.

```
variable "names" {
 description = "A list of names"
 type = list(string)
 default = ["neo", "trinity", "morpheus"]
}
```

이 이름을 모두 대문자로 어떻게 변환한다고 가정하면, Python과 같은 범용 프로그래밍 언어에서는 다음 for 루프를 작성할 수 있습니다.

```
names = ["neo", "trinity", "morpheus"]
upper_case_names = []
for name in names:
 upper_case_names.append(name.upper())
print upper_case_names
```

Python은 목록 이해(list comprehension)라는 구문을 사용하여 정확히 동일한 코드를 한 줄에 작성하는 또 다른 방법을 제공합니다.

```
names = ["neo", "trinity", "morpheus"]
upper_case_names = [name.upper() for name in names]
print upper_case_names
```

Python에서는 조건을 지정하여 결과 목록을 필터링할 수도 있습니다.

```
names = ["neo", "trinity", "morpheus"]
short_upper_case_names = [name.upper() for name in names if len(name) < 5]
print short_upper_case_names
```

Terraform은 for 식의 형태로 유사한 기능을 제공합니다(이전 섹션에서 본 for_each 식과 혼동하지 마세요). for 표현식의 기본 구문은 다음과 같습니다.

```
[for <ITEM> in <LIST> : <OUTPUT>]
```

여기서 LIST는 반복할 목록이고, ITEM은 LIST의 각 항목에 할당할 지역 변수 이름이며, OUTPUT은 어떤 방식으로든 ITEM을 변환하는 표현식입니다. 예를 들어 다음은 var.names의 이름 목록을 대문자로 변환하는 Terraform 코드입니다.

```
output "upper_names" {
 value = [for name in var.names : upper(name)]
}
```

이 코드에 대해 terraform Apply를 실행하면 다음과 같은 출력이 표시됩니다.

```
$ terraform apply
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
upper_names = [
 "NEO",
 "TRINITY",
 "MORPHEUS",
]
```

Python의 목록 이해와 마찬가지로 조건을 지정하여 결과 목록을 필터링할 수 있습니다.

```
output "short_upper_names" {
 value = [for name in var.names : upper(name) if length(name) < 5]
}
```

이 코드에서 Terraform Apply를 실행하면 다음이 제공됩니다.

```
short_upper_names = [
 "NEO",
]
```

Terraform의 for 표현식을 사용하면 다음 구문을 사용하여 맵을 반복할 수도 있습니다.

```
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]
```

여기서 MAP은 반복할 맵 데이터이고, KEY와 VALUE는 MAP의 각 키-값 쌍에 할당할 로컬 변수 이름이며, OUTPUT은 KEY와 VALUE를 어떤 방식으로 변환하는 표현식입니다. 예는 다음과 같습니다.

```
variable "hero_thousand_faces" {
 description = "map"
 type = map(string)
 default = {
  neo = "hero"
  trinity = "love interest"
  morpheus = "mentor"
 }
}
output "bios" {
 value = [for name, role in var.hero_thousand_faces : "${name} is the ${role}"]
}
```

이 코드에 대해 Terraform Apply를 실행하면 다음과 같은 결과가 나타납니다.

```
bios = [
 "morpheus is the mentor",
 "neo is the hero",
 "trinity is the love interest",
]
```

다음 구문처럼 for 표현식을 사용하면 목록이 아닌 맵을 출력할 수도 있습니다.

```
# Loop over a list and output a map
{for <ITEM> in <LIST> : <OUTPUT_KEY> => <OUTPUT_VALUE>}
# Loop over a map and output a map
{for <KEY>, <VALUE> in <MAP> : <OUTPUT_KEY> => <OUTPUT_VALUE>}
```

차이점은 (a) 대괄호가 아닌 중괄호로 표현식을 묶고, (b) 반복할 때마다 단일 값을 출력하는 대신 화살표로 구분된 키와 값을 출력한다는 것입니다. 예를 들어, 모든 키와 값을 대문자로 만들기 위해 맵을 변환하는 방법은 다음과 같습니다.

```
output "upper_roles" {
 value = {for name, role in var.hero_thousand_faces : upper(name) => upper(role)}
}
```

이 코드를 실행한 결과는 다음과 같습니다.

```
upper_roles = {
 "MORPHEUS" = "MENTOR"
 "NEO" = "HERO"
 "TRINITY" = "LOVE INTEREST"
}
```

for String 지시어를 사용한 루프

이전 앞부분에서는 문자열 내에서 Terraform 코드를 참조할 수 있는 문자열 보간법에 대해 배웠습니다.

```
"Hello, ${var.name}"
```

문자열 지시문을 사용하면 문자열 보간과 유사한 구문을 사용하여 문자열 내에서 제어 문(예: for 루프 및 if 문)을 사용할 수 있지만 달러 기호 대신 백분율 기호(%)와 중괄호를 사용합니다. Terraform은 for 루프와 조건문이라는 두 가지 유형의 문자열 지시문을 지원합니다. for string 지시문은 다음 구문을 사용합니다.

```
%{ for <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

여기서 COLLECTION은 반복할 목록 또는 맵이고, ITEM은 COLLECTION의 각 항목에 할당할 지역 변수 이름이며, BODY는 각 반복을 렌더링할 항목(ITEM을 참조할 수 있음)입니다. 예는 다음과 같습니다.

```
variable "names" {
 description = "Names to render"
 type = list(string)
 default = ["neo", "trinity", "morpheus"]
}
output "for_directive" {
```

```
 value = "%{ for name in var.names }${name}, %{ endfor }"
}
```

terraform apply 를 실행하면 다음과 같은 출력이 표시됩니다.

```
$ terraform apply
(...)
Outputs:
for_directive = "neo, trinity, morpheus, "
```

for 루프에 인덱스를 제공하는 for string 지시문 구문 버전도 있습니다.

```
%{ for <INDEX>, <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

다음은 인덱스를 사용하는 예입니다.

```
output "for_directive_index" {
 value = "%{ for i, name in var.names }(${i}) ${name}, %{ endfor }"
}
```

terraform apply 를 실행하면 다음과 같은 출력이 표시됩니다.

```
$ terraform apply
(...)
Outputs:
for_directive_index = "(0) neo, (1) trinity, (2) morpheus, "
```

두 출력 모두 맨 마지막 항목 뒤에 추가 쉼표와 공백이 있다는 점에 유의하세요. 이것은 조건문을 활용해서 해결할 수 있습니다.

조건문

Terraform이 루프를 수행하는 여러 가지 방법을 제공하는 것처럼 조건을 수행하는 여러 가지 방법도 있으며 각각 약간 다른 시나리오에서 사용하도록 고안되었습니다.

- 카운트 매개변수
  조건부 리소스에 사용됩니다.
- for_each 및 for 표현식
  조건부 리소스 및 리소스 내의 인라인 블록에 사용됩니다.
- if 문자열 지시어
  문자열 내의 조건문에 사용됩니다.

한 번에 하나씩 살펴보겠습니다.

count 매개변수와 If 문

이전에 웹 서버 클러스터 배포를 위한 "청사진"으로 사용할 수 있는 Terraform 모듈을 만들었습니다. 모듈은 ASG(Auto Scaling 그룹), ALB(Application Load Balancer), 보안 그룹 및 기타 여러 리소스를 생성했습니다. 모듈이 생성하지 않은 것 중 하나는 예약된 작업이었습니다. 프로덕션에서만 클러스터를 확장하려고 하기 때문에

live/prod/services/webserver-cluster/main.tf 아래의 프로덕션 구성에서 aws_autoscaling_schedule 리소스를 직접 정의했습니다. webserver-cluster 모듈에서 aws_autoscal ing_schedule 리소스를 정의하고 모듈의 일부 사용자에 대해서는 조건부로 생성하고 다른 사용자에 대해서는 생성하지 않을 수 있는 방법이 있을까요?

한번 시도해 봅시다. 첫 번째 단계는 모듈이 자동 크기 조정을 활성화해야 하는지 여부를 지정하는 데 사용할 수 있는 부울 입력 변수를 module/services/webserver-cluster/variables.tf에 추가하는 것입니다.

```
variable "enable_autoscaling" {
 description = "If set to true, enable auto scaling"
 type = bool
}
```

이제 범용 프로그래밍 언어가 있다면 if 문에서 이 입력 변수를 사용할 수 있습니다.

```
if var.enable_autoscaling {
 resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  scheduled_action_name = "${var.cluster_name}-scale-out-during-business-hours"
  min_size = 2
  max_size = 10
  desired_capacity = 10
  recurrence = "0 9 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
 }
 resource "aws_autoscaling_schedule" "scale_in_at_night" {
  scheduled_action_name = "${var.cluster_name}-scale-in-at-night"
  min_size = 2
  max_size = 10
  desired_capacity = 2
  recurrence = "0 17 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
 }
}
```

Terraform은 if 문을 지원하지 않으므로 이 코드는 작동하지 않습니다. 그러나 count 매개변수를 사용하고 다음 두 가지 속성을 활용하여 동일한 작업을 수행할 수 있습니다.

- 리소스 개수를 1로 설정하면 해당 리소스의 복사본 하나를 얻게 됩니다. 개수를 0으로 설정하면 해당 리소스가 전혀 생성되지 않습니다.
- Terraform은 <CONDITION> 형식의 조건식을 지원합니다. <TRUE_VAL> : <FALSE_VAL> . 다른 프로그래밍 언어에서 익숙할 수 있는 이 삼항 구문은 CONDITION의 부울 논리를 평가하고, 결과가 true이면 TRUE_VAL을 반환하고, 결과가 false이면 FALSE_VAL을 반환합니다.

이 두 가지 아이디어를 종합하면 다음과 같이 webserver-cluster 모듈을 업데이트할 수 있습니다.

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
 count = var.enable_autoscaling ? 1 : 0
```

```
  scheduled_action_name = "${var.cluster_name}-scale-out-during-business-hours"
  min_size = 2
  max_size = 10
  desired_capacity = 10
  recurrence = "0 9 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
  count = var.enable_autoscaling ? 1 : 0
  scheduled_action_name = "${var.cluster_name}-scale-in-at-night"
  min_size = 2
  max_size = 10
  desired_capacity = 2
  recurrence = "0 17 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}
```

var.enable_autoscaling이 true 인 경우 각 aws_autoscaling_schedule 리소스의 count 파라미터가 1로 설정되므로 각 리소스가 하나씩 생성됩니다. var.enable_autoscaling이 false 인 경우 각 aws_autoscaling_schedule 리소스의 count 파라미터가 0으로 설정되므로 둘 다 생성되지 않습니다. 이것이 바로 조건부 논리입니다!
이제 스테이징(live/stage/services/webserver-cluster/main.tf)에서 이 모듈의 사용을 업데이트하여 enable_autoscaling을 false로 설정하여 자동 스케일링을 비활성화할 수 있습니다.

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key = "stage/data-stores/mysql/terraform.tfstate"
  instance_type = "t2.micro"
  min_size = 2
  max_size = 2
  enable_autoscaling = false
}
```

마찬가지로, 프로덕션(live/prod/services/webserver-cluster/main.tf)에서 이 모듈의 사용을 업데이트하여 활성화_autoscaling을 true로 설정하여 자동 크기 조정을 활성화할 수 있습니다.

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key = "prod/data-stores/mysql/terraform.tfstate"
  instance_type = "m4.large"
  min_size = 2
  max_size = 10
  enable_autoscaling = true
```

```
 custom_tags = {
  Owner = "team-foo"
  ManagedBy = "terraform"
 }
}
```

if-else 문

이제 if 문을 수행하는 방법을 알았으니 if-else 문은 살펴봅니다. 이 장의 앞부분에서는 EC2에 대한 읽기 전용 액세스 권한을 가진 여러 IAM 사용자를 생성했습니다. 이러한 사용자 중 한 명인 neo에게 CloudWatch에 대한 액세스 권한도 부여하고 Terraform 구성을 적용하는 사람이 neo에게 읽기 액세스 권한만 할당할지 아니면 읽기 및 쓰기 액세스 권한을 모두 할당할지 결정할 수 있도록 허용한다고 가정해 보겠습니다. 다음은 CloudWatch에 대한 읽기 전용 액세스를 허용하는 IAM 정책입니다.

```
resource "aws_iam_policy" "cloudwatch_read_only" {
 name = "cloudwatch-read-only"
 policy = data.aws_iam_policy_document.cloudwatch_read_only.json
}
data "aws_iam_policy_document" "cloudwatch_read_only" {
 statement {
  effect = "Allow"
  actions = [
  "cloudwatch:Describe*",
  "cloudwatch:Get*",
  "cloudwatch:List*"
  ]
  resources = ["*"]
 }
}
```

다음은 CloudWatch에 대한 전체(읽기 및 쓰기) 액세스를 허용하는 IAM 정책입니다.

```
resource "aws_iam_policy" "cloudwatch_full_access" {
 name = "cloudwatch-full-access"
 policy = data.aws_iam_policy_document.cloudwatch_full_access.json
}
data "aws_iam_policy_document" "cloudwatch_full_access" {
 statement {
  effect = "Allow"
  actions = ["cloudwatch:*"]
  resources = ["*"]
 }
}
```

목표는 give_neo_cloudwatch_full_access 라는 새 입력 변수의 값을 기반으로 이러한 IAM 정책 중 하나를 "neo" 에 연결하는 것입니다.

```
variable "give_neo_cloudwatch_full_access" {
 description = "If true, neo gets full access to CloudWatch"
 type = bool
}
```

범용 프로그래밍 언어를 사용하고 있다면 다음과 같은 if-else-문을 작성할 수 있습니다.

```
# This is just pseudo code. It won't actually work in Terraform.
if var.give_neo_cloudwatch_full_access {
 resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
  user = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_full_access.arn
 }
} else {
 resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only" {
  user = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_read_only.arn
 }
}
```

Terraform에서 이를 수행하려면 각 리소스에 대해 count 매개변수와 조건식을 사용할 수 있습니다.

```
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
 count = var.give_neo_cloudwatch_full_access ? 1 : 0
 user = aws_iam_user.example[0].name
 policy_arn = aws_iam_policy.cloudwatch_full_access.arn
}
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only" {
 count = var.give_neo_cloudwatch_full_access ? 0 : 1
 user = aws_iam_user.example[0].name
 policy_arn = aws_iam_policy.cloudwatch_read_only.arn
}
```

이 코드에는 두 개의 aws_iam_user_policy_attachment 리소스가 포함되어 있습니다. CloudWatch 전체 액세스 권한을 연결하는 첫 번째 항목에는 var.give_neo_cloudwatch_full_access가 true이면 1로 평가되고 그렇지 않으면 0으로 평가되는 조건식이 있습니다(if 절). CloudWatch 읽기 전용 권한을 연결하는 두 번째 권한에는 정반대를 수행하는 조건식이 있습니다. 즉, var.give_neo_cloudwatch_full_access가 true이면 0으로 평가되고 그렇지 않으면 1로 평가됩니다(else 절). 그리고 여기까지 왔습니다. 이제 if-else 문을 수행하는 방법을 알게 되었습니다!
이제 if/else 조건을 기반으로 하나의 리소스 또는 다른 리소스를 생성할 수 있으므로 실제로 생성된 리소스의 속성에 액세

스할 수 있습니다. 예를 들어 실제로 연결한 정책의 ARN이 포함된 neo_cloudwatch_policy_arn이라는 출력 변수를 추가합니다.

가장 간단한 옵션은 삼항 구문을 사용하는 것입니다.

```
output "neo_cloudwatch_policy_arn" {
 value = (
  var.give_neo_cloudwatch_full_access
  ? aws_iam_user_policy_attachment.neo_cloudwatch_full_access[0].policy_arn
  : aws_iam_user_policy_attachment.neo_cloudwatch_read_only[0].policy_arn
 )
}
```

지금은 잘 작동하지만 이 코드는 약간 불안정합니다. 나중에는 var.give_neo_cloudwatch_full_access에만 의존하는 것이 아니라 여러 변수에 의존하게 될 것입니다. 이 출력 변수의 조건을 업데이트하는 것을 잊어버릴 위험이 있으며, 그 결과 존재하지 않을 수 있는 배열 요소에 액세스하려고 할 때 매우 혼란스러운 오류가 발생하게 됩니다.
더 안전한 접근 방식은 연결과 단일 목록 기능을 활용하는 것입니다. concat 함수는 두 개 이상의 목록을 입력으로 사용하여 단일 목록으로 결합합니다. 하나의 함수는 목록을 입력으로 사용하고 목록에 요소가 0개 있으면 null 을 반환합니다. 목록에 요소가 1개 있으면 해당 요소를 반환합니다. 목록에 요소가 2개 이상 있으면 오류가 표시됩니다. 이 두 가지를 합치고 표현식으로 결합하면 다음과 같은 결과를 얻을 수 있습니다.

```
output "neo_cloudwatch_policy_arn" {
 value = one(concat(
  aws_iam_user_policy_attachment.neo_cloudwatch_full_access[*].policy_arn,
  aws_iam_user_policy_attachment.neo_cloudwatch_read_only[*].policy_arn
 ))
}
```

if/else 조건의 결과에 따라 neo_cloud watch_full_access가 비어 있고 neo_cloudwatch_read_only에 하나의 요소가 포함되거나 그 반대가 됩니다. 따라서 이들을 연결하면 하나의 요소가 포함된 목록이 생성되고 하나의 함수가 반환됩니다. if/else 조건을 어떻게 변경하더라도 계속해서 올바르게 작동합니다.

if-else 문을 시뮬레이션하기 위해 카운트 및 내장 함수를 사용하는 것은 약간의 꼼수와 같지만 상당히 잘 작동하며 코드에서 볼 수 있듯이 사용자에게 많은 복잡성을 숨길 수 있습니다. 깨끗하고 간단한 API로 작업할 수 있다는 점입니다.

for_each 및 for 표현식을 사용한 조건부

이제 count 매개변수를 사용하여 리소스에 대해 조건부 논리를 수행하는 방법을 이해했으므로 비슷한 전략을 사용하여 for_each 표현식을 사용하여 조건부 논리를 수행할 수 있습니다.

for_each 표현식에 빈 컬렉션을 전달하면 결과는 for_each가 있는 리소스, 인라인 블록 또는 모듈의 복사본이 0개가 됩니다. 비어 있지 않은 컬렉션을 전달하면 리소스, 인라인 블록 또는 모듈의 복사본이 하나 이상 생성됩니다. 이제 고민해야 할 것은 컬렉션이 비어 있어야 하는지 여부를 조건부로 어떻게 결정하는가 입니다. 해결 방안은 for_each 표현식과 for 표현식을 결합하는 것입니다. 예를 들어, module/services/webserver-cluster/main.tf의 webserver-cluster 모듈이 태그를 설정하는 방식을 떠올려 보세요.

```
dynamic "tag" {
 for_each = var.custom_tags
 content {
  key = tag.key
  value = tag.value
  propagate_at_launch = true
 }
}
```

var.custom_tags가 비어 있으면 for_each 표현식에는 반복할 항목이 없으므로 태그가 설정되지 않습니다. 즉, 여기에는 이미 몇 가지 조건부 논리가 있습니다. 그리고 다음과 같이 for_each 표현식과 for 표현식을 결합하면 더 나아갈 수 있습니다.

```
dynamic "tag" {
 for_each = {
  for key, value in var.custom_tags:
  key => upper(value)
  if key != "Name"
 }
 content {
  key = tag.key
  value = tag.value
  propagate_at_launch = true
 }
}
```

중첩된 for 표현식은 var.custom_tags를 반복하고, 각 값을 대문자로 변환하고(일관성을 위해) for 표현식의 조건을 사용하여 모듈이 이미 자체 Name 태그를 설정했기 때문에 Name으로 설정된 모든 키를 필터링합니다. for 표현식의 값을 필터링하면 임의의 조건부 논리를 구현할 수 있습니다.

리소스 또는 모듈의 여러 복사본을 생성할 때 거의 항상 count보다 for_each를 선호해야 하지만 조건부 논리의 경우 count를 0 또는 1로 설정하는 것이 for_each를 비어 있거나 비어 있지 않은 컬렉션으로 설정하는 것보다 더 간단한 경향이 있습니다. 따라서 일반적으로 count를 사용하여 조건부로 리소스와 모듈을 생성하고 다른 모든 유형의 루프 및 조건에 대해 for_each를 사용하는 것이 좋습니다.

if 문자열 지시문을 사용한 조건문

이제 다음 구문을 갖는 if 문자열 지시문을 살펴보겠습니다.

```
%{ if <CONDITION> }<TRUEVAL>%{ endif }
```

여기서 CONDITION은 부울로 평가되는 표현식이고 TRUEVAL은 CONDITION이 true로 평가되는 경우 렌더링할 표현식입니다.

이 장의 앞부분에서는 for string 지시어를 사용하여 문자열 내에서 루프를 수행하여 여러 개의 쉼표로 구분된 이름을 출력했습니다. 문제는 문자열 끝에 여분의 쉼표와 공백이 있다는 것이었습니다. if string 지시문을 사용하여 다음과 같이 이 문

제를 해결할 수 있습니다.

```
output "for_directive_index_if" {
 value = <<EOF
%{ for i, name in var.names }
 ${name}%{ if i < length(var.names) - 1 }, %{ endif }
%{ endfor }
EOF
}
```

원본 버전에서 몇 가지 변경 사항이 있습니다.

여러 줄의 문자열을 정의하는 방법인 HEREDOC에 코드를 넣었습니다. 이를 통해 코드를 여러 줄에 걸쳐 분산시켜 더 읽기 쉽게 만들 수 있습니다.

목록의 마지막 항목에 대해 쉼표와 공백을 출력하지 않기 위해 if string 지시문을 사용했습니다.

terraform apply 를 실행하면 다음과 같은 출력이 표시됩니다.

```
$ terraform apply
(...)
Outputs:
for_directive_index_if = <<EOT
 neo,
 trinity,
 morpheus
```

마지막 쉼표는 사라졌지만 공백(공백 및 줄바꿈)이 추가되었습니다. HEREDOC에 넣은 모든 공백은 최종 문자열로 끝납니다. 문자열 지시문에 스트립 마커(~)를 추가하면 이 문제를 해결할 수 있습니다. 이렇게 하면 스트립 마커 앞이나 뒤에 추가 공백이 소모됩니다.

```
output "for_directive_index_if_strip" {
 value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ endif }
%{~ endfor ~}
EOF
}
```

이 버전을 사용해 보겠습니다.

```
$ terraform apply
(...)
```

```
Outputs:
for_directive_index_if_strip = "neo, trinity, morpheus"
```

아주 보기 좋게 개선되었습니다. 추가 공백이나 쉼표가 없습니다. 다음 구문을 사용하는 문자열 지시문에 else를 추가하면 이 출력을 더욱 보기 좋게 만들 수 있습니다.

```
%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif }
```

여기서 FALSEVAL은 CONDITION이 false로 평가되는 경우 렌더링할 표현식입니다. 다음은 else 절을 사용하여 끝에 마침표를 추가하는 방법의 예입니다.

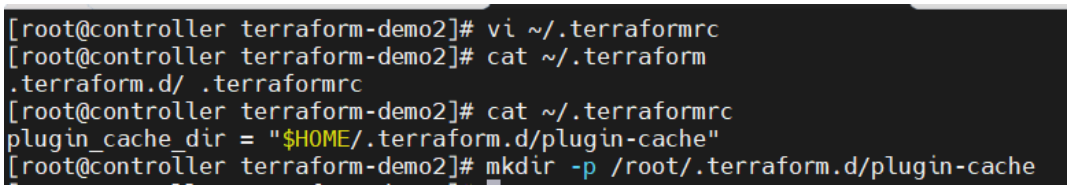
```
output "for_directive_index_if_else_strip" {
 value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ else }.%{ endif }
%{~ endfor ~}
EOF
}
```

terraform apply 를 실행하면 다음과 같은 출력이 표시됩니다.

```
$ terraform apply
(...)
Outputs:
for_directive_index_if_else_strip = "neo, trinity, morpheus."
```

필기

테라폼 플러그인 캐시



desired capacity

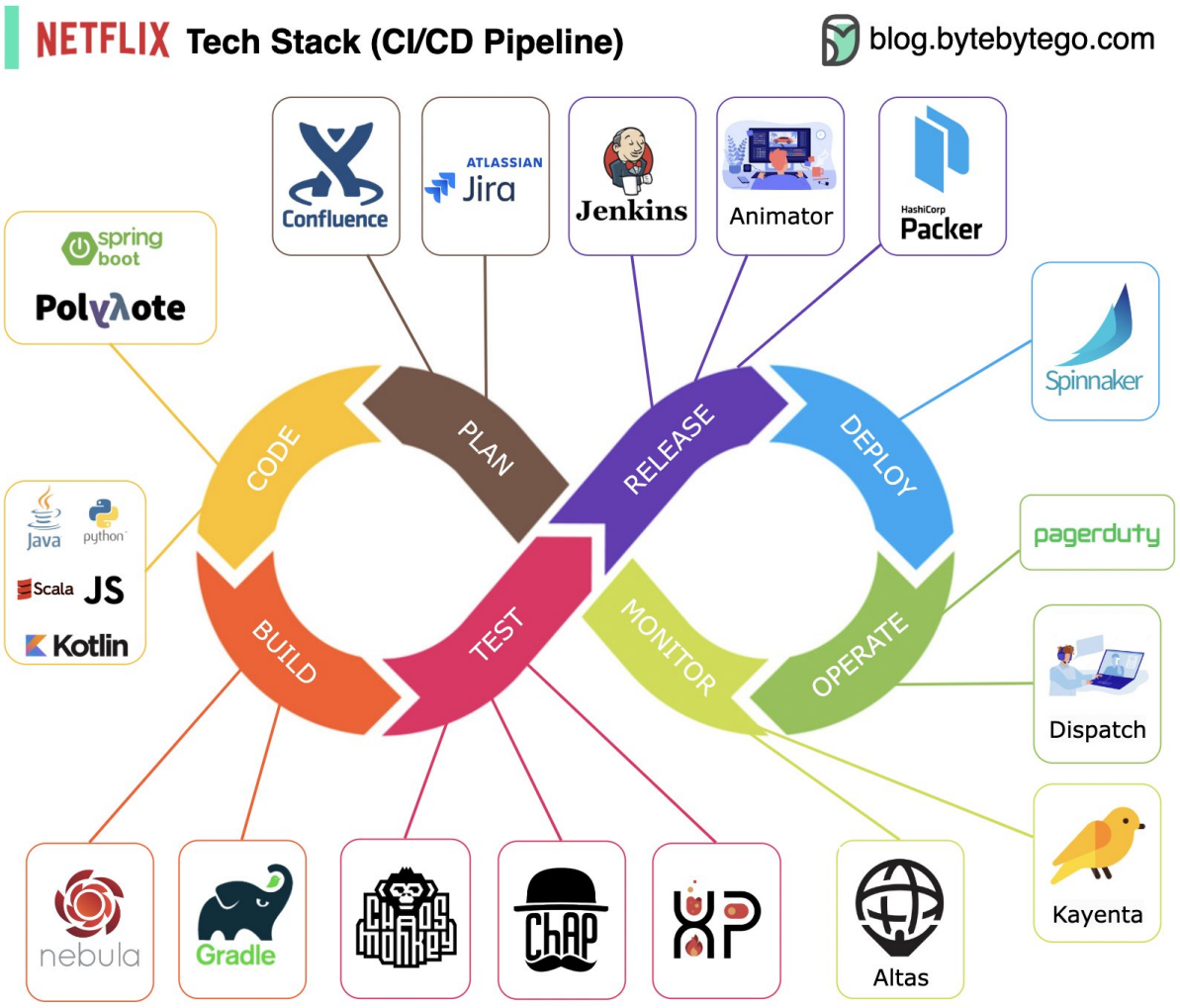
```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
scheduled_action_name = "scale-out-during-business-hours"
min_size = 2
max_size = 10
desired_capacity = 10
recurrence = "0 9 * * *"
autoscaling_group_name = module.webserver_cluster.asg_name
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
scheduled_action_name = "scale-in-at-night"
min_size = 2
max_size = 10
desired_capacity = 2
recurrence = "0 17 * * *"
autoscaling_group_name = module.webserver_cluster.asg_name
}
```

해당 cron 시간대에는 항상 정해진 숫자만큼 유지해야

- DEV SEC OPS

trivy open source가 굉장히 hot하다. 보안 측에서

- 넷플릭스 기술 스택



- jira 현업에서 많이 사용
- chaos monkey팀이 chaos testing 진행

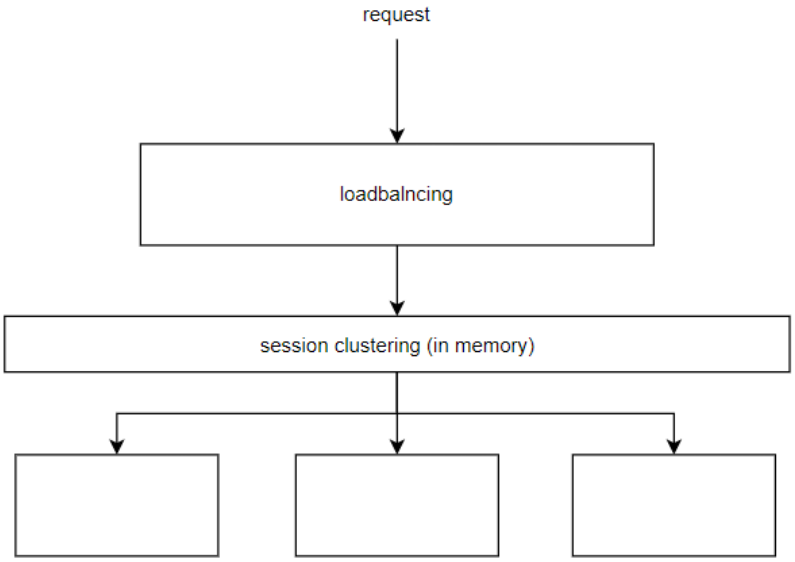
이렇게 다양한 도구들을 회사에선 많이 사용 회사에 가서 재교육을 받는다
패턴 학습 후 프로젝트 투입

- 무중단 배포

테라폼에서 무중단 배포를 하는 경우는 거의 없다. 가상 머신에서 굳이해야하나? 싶다는 의문

컨테이너의 경우 쿠버네티스가 담당하고 rolling update 하는 경우 많음

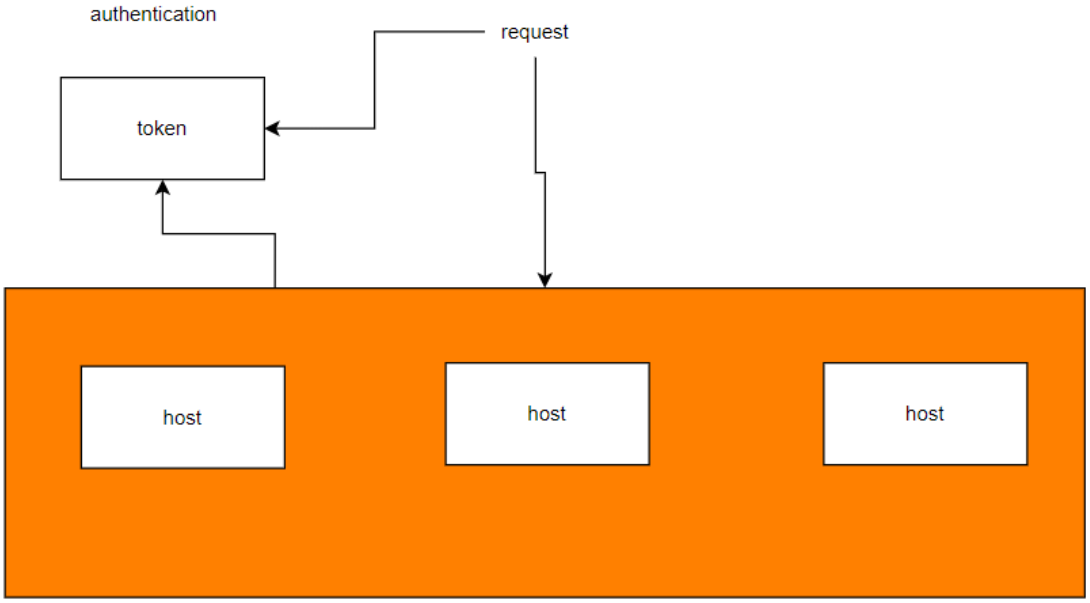
- session clustring



처음에 세션 사용했는데 부하가 너무 많았다. 쿠키로 넘어감

항상 염두해둬야 할 것은 세션은 끊어진다는 것

- token



토큰은 노출 돼도 큰 문제가 없다. 컨테이너 기반에 어울리고 확장성이 좋다.

끊어지더라도 네트워크에 retry 설정을 통해 토큰이 유지가 되어있다면 새로 로그인 필요 x

- cookie

세션에서 쿠키로 넘어왔는데 보안 문제가 있었다. 토큰으로 넘어감
노출되면 보안 위험 보안에선 쿠키 사용하는거 지양

- keycloak

authz

여러가지 보안 패턴 존재