8 / 5

리마인드

자동화

자동화는 시나리오 세우고 검증하는 것이 중요하다 어렵지는 않다.

kubetnetes playbook

- 커널 모듈 설정(br_netfilter, overlay)
- 패키지 구성(containered, kubeadm, kubelet)
  container 기본 패키지가 없기에 직접 구성
- 보안 설정 (방화벽, selinux)
- cluster 구성(init + join)
  kubeadm init과 join
  worker node들이 join하도록
- kubectl cli 구성 (자격증명 +binary)
  대쉬보드 api 사용하다 해킹당한 사례가 있다

  ssh 키 + 자격증명 구성

git

- 파일 상태 관리(modify, staging, commit)
  add 명령으로 stage 상태를 만든다.

  > 💡 새로 만든 파일은 add 명령으로 수동으로 넣어줘야 커밋 반영

- 브랜치 관리(git branch, git checkout, git merge)

- 원격 저장소 관리(git remote add / remove , git pull / push)

Terraform

- .tf 파일 작성
- terraform init, terraform plan, terraform apply

  - init

provider 체크 aws인지 azure인지 등등

- plan
  어떻게 동작할 것인지
- apply
  실제로 적용

- 특징
  - 순서가 상관이 없다 앤서블은 네트워크 구성을 순서의 상관이 있다.

GitOps 보안

- SASD
  Static Application Security Test

- DAST
  Dynamic Application Security Test

- IAST
  Interactive Application Security Test

실습(테라폼 기본 사용)

단일 웹 서버 배포

다음 단계는 이 인스턴스에서 웹 서버를 실행하는 것입니다. 목표는 가능한 가장 간단한 웹 아키텍처, 즉 그림에 표시된 것처럼 HTTP 요청에 응답할 수 있는 단일 웹 서버를 배포하는 것입니다.

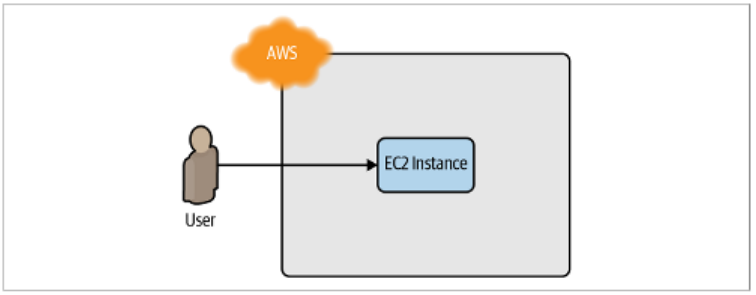


Figure 2-6. Start with a simple architecture: a single web server running in AWS that responds to HTTP requests.

실제 사용 사례에서는 아마도 Ruby on Rails 또는 Django와 같은 웹 프레임워크를 사용하여 웹 서버를 구축하겠지만, 이 예를 단순하게 "Hello, World" 를 항상 출력하는 웹서버를 실행합니다.

```
#!/bin/bash
echo "Hello, World" > index.html
nohub busybox httpd -f -p 8080 &
```

위의 스크립트는 index.html에 "Hello, World" 텍스트를 쓰고 busybox(Ubuntu에 기본적으로 설치됨)라는 도구를 실행하여 해당 파일을 제공하기 위해 포트 8080에서 웹 서버를 시작하는 Bash 스크립트입니다. 웹 서버가 백그라운드에서 영구적으로 실행되는 반면 Bash 스크립트 자체는 종료될 수 있도록 busybox 명령을 nohup 및 앰퍼샌드(&)로 래핑했습니다.

> 💡 포트 번호 이 예에서 기본 HTTP 포트 80이 아닌 포트 8080을 사용하는 이유는 1024보다 작은 포트에서 수신 대기하려면 루트 사용자 권한이 필요하기 때문입니다. 서버를 손상시키는 공격자는 루트 권한도 얻게 되므로 이는 보안상 위험합니다. 따라서 제한된 권한을 가진 루트가 아닌 사용자로 웹 서버를 실행하는 것이 가장 좋습니다. 이는 더 높은 번호의 포트에서 수신해야 한다는 의미이지만, 이 장의 뒷부분에서 볼 수 있듯이 로드 밸런서를 구성하여 포트 80에서 수신하고 서버의 높은 번호가 있는 포트로 트래픽을 라우팅할 수 있습니다.

이 스크립트를 실행하기 위해 EC2 인스턴스를 어떻게 얻습니까? 일반적으로 Packer와 같은 도구를 사용하여 웹 서버가 설치된 사용자 지정 AMI를 생성합니다. 이 예의 더미 웹 서버는 busybox 를 사용하는 단일 라이너이므로 일반 Ubuntu 20.04 AMI를 사용하고 EC2 인스턴스 사용자 데이터 구성의 일부로 "Hello, World" 스크립트를 실행할 수 있습니다. EC2 인스턴스를 시작할 때 쉘 스크립트나 cloud-init 지시어를 사용자 데이터에 전달할 수 있는 옵션이 있으며, EC2 인스턴스는 처음 부팅할 때 이를 실행합니다. 다음과 같이 Terraform 코드에서 user_data 인수를 설정하여 사용자 데이터에 셸 스크립트를 전달합니다.

```
resource  "aws_instance" "example"  {
  ami                    =  "ami-0fb653ca2d3203ac1"
  instance_type          =  "t2.micro"

  user_data  =  <<-EOF
                #!/bin/bash
                echo  "Hello, World"  >  index.html
                nohup  busybox  httpd  -f  -p  8080  &
                EOF
  user_data_replace_on_change  =  true

  tags  =  {
    Name  =  "terraform-example"
  }
}
```

앞의 코드에서 주목해야 할 두 가지 사항은 다음과 같습니다.

- <←EOF 및 EOF는 Terraform의 heredoc 구문으로, 여기저기에 \n 문자를 삽입하지 않고도 여러 줄 문자열을 생성할 수 있습니다.
- user_data_replace_on_change 매개변수가 true로 설정되어 user_data 매개변수를 변경하고 apply 를 실행하면 Terraform이 원래 인스턴스를 종료하고 완전히 새로운 인스턴스를 시작합니다. Terraform의 기본 동작은 원래 인스턴스를 제자리에 업데이트하는 것입니다. 그러나 사용자 데이터는 첫 번째 부팅 시에만 실행되고 원래 인스턴스는 이미 해당 부팅 프로세스를 거쳤으므로 새 인스턴스를 강제로 생성하여 데이터 스크립트가 실행되도록 해야합니다.

이 웹 서버가 작동하기 전에 한 가지 작업을 더 수행해야 합니다. 기본적으로 AWS는 EC2 인스턴스에서 들어오거나 나가는 트래픽을 허용하지 않습니다. EC2 인스턴스가 포트 8080에서 트래픽을 수신하도록 허용하려면 보안 그룹을 생성해야 합니다.

```
resource  "aws_security_group" "instance"  {
  name  =  "terraform-example-instance"

  ingress  {
    from_port    =  8080
    to_port      =  8080
    protocol     =  "tcp"
    cidr_blocks  =  [ "0.0.0.0/0" ]
  }
}
```

이 코드는 aws_security_group이라는 새 리소스를 생성하고 이 그룹이 CIDR 블록 0.0.0.0/0에서 포트 8080으로 들어오는 TCP 요청을 허용하도록 지정합니다. CIDR 블록은 IP 주소 범위를 지정하는 간결한 방법입니다. 예를 들어 CIDR 블록 10.0.0.0/24는 10.0.0.0에서 10.0.0.255 사이의 모든 IP 주소를 나타냅니다. CIDR 블록 0.0.0.0/0은 가능한 모든 IP 주소를 포함하는 IP 주소 범위이므로 이 보안 그룹은 모든 IP에서 포트 8080으로 들어오는 요청을 허용합니다.

단순히 보안 그룹을 생성하는 것 만으로는 충분하지 않습니다. 보안 그룹의 ID를 aws_instance 리소스의 vpc_security_group_ids 인수에 전달하여 EC2 인스턴스에 실제로 사용하도록 지시해야 합니다. 그러기 위해서는 먼저 Terraform 표현식에 대해 배워야 합니다. Terraform의 표현식은 값을 반환하는 모든 것입니다. 문자열(예: "ami-0fb653ca2d3203ac1") 및 숫자(예: 5)와 같은 가장 간단한 유형의 표현식인 리터럴을 이미 살펴보았습니다. Terraform은 책 전반에 걸쳐 볼 수 있는 다양한 유형의 표현식을 지원합니다. 특히 유용한 표현식 유형 중 하나는 참조로, 이를 통해 코드의 다른 부분에서 값에 액세스할 수 있습니다. 보안 그룹 리소스의 ID에 액세스하려면 다음 구문을 사용하는 리소스 속성 참조를 사용해야 합니다.

```
<PROVIDER>_<TYPE> . <NAME> . <ATTRIBUTE>
```

여기서 PROVIDER는 공급자의 이름(예: aws)이고, TYPE은 리소스 유형(예: security_group)이며, NAME은 해당 리소스의 이름(예: 보안 그룹 이름은 "instance")이며, ATTRIBUTE는 다음 중 하나입니다. 해당 리소스의 인수 중 하나(예: name) 또는 리소스에서 내보낸 속성 중 하나(각 리소스에 대한 문서에서 사용 가능한 속성 목록을 찾을 수 있음) 보안 그룹은 id 라는 속성을 내보내므로 이를 참조하는 표현식은 다음과 같습니다.

```
aws_security_group.instance.id
```

다음과 같이 aws_instance의 vpc_security_group_ids 인수에서 이 보안 그룹 ID를 사용할 수 있습니다.

```
resource "aws_instance" "example" {
  ami                    = "ami-0fb653ca2d3203ac1"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]

  user_data = <<-EOF
                          #!/bin/bash
                          echo "Hello, World" > index.html
                          nohup busybox httpd -f -p 8080 &
                         EOF

  user_data_replace_on_change = true
```

```
  tags = {
    Name = "terraform-example"
  }
}
```

한 리소스에서 다른 리소스로 참조를 추가하면 암시적 종속성이 생성됩니다. Terraform은 이러한 종속성을 구문 분석하고, 종속성 그래프를 작성하며, 이를 사용하여 리소스를 생성해야 하는 순서를 자동으로 결정합니다. 예를 들어 이 코드를 처음부터 배포하는 경우 EC2 인스턴스가 보안 그룹의 ID를 참조하기 때문에 Terraform은 EC2 인스턴스보다 먼저 보안 그룹을 생성해야 한다는 것을 알게 됩니다. 그래프 명령을 실행하면 Terraform에서 종속성 그래프를 표시할 수도 있습니다.

```
$ terraform graph

digraph {
                compound = "true"
                newrank = "true"
                subgraph "root" {
                "[root] aws_instance.example"
                   [label = "aws_instance.example", shape = "box"]
                "[root] aws_security_group.instance"
                 [label = "aws_security_group.instance", shape = "box"]
                "[root] provider.aws"
                 [label = "provider.aws", shape = "diamond"]
                "[root] aws_instance.example" ->
                 "[root] aws_security_group.instance"
                "[root] aws_security_group.instance" ->
                "[root] provider.aws"
                "[root] meta.count-boundary (EachMode fixup)" ->
                "[root] aws_instance.example"
                "[root] provider.aws (close)" ->
                  "[root] aws_instance.example"
                "[root] root" ->
                  "[root] meta.count-boundary (EachMode fixup)"
                "[root] root" ->
                  "[root] provider.aws (close)"
                }
}
```

출력은 DOT라는 그래프 설명 언어로 되어 있으며 Graphviz와 같은 데스크톱 앱이나 GraphvizOnline과 같은 웹 앱을 사용하여 그림에 표시된 종속성 그래프와 유사한 이미지로 변환할 수 있습니다.

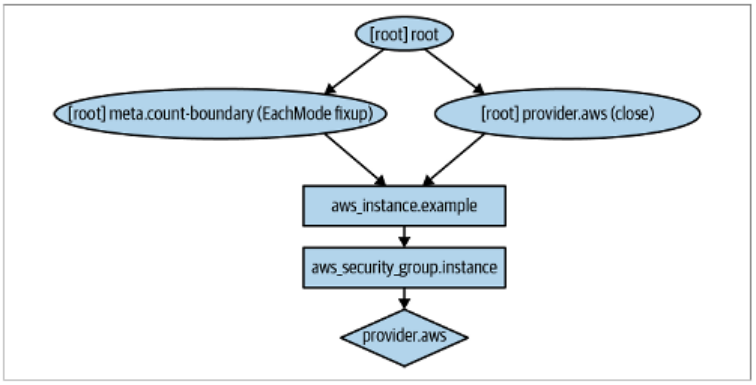


Figure 2-7. This is what the dependency graph for the EC2 Instance and its security group looks like when rendered with Graphviz.

Terraform은 종속성 트리를 탐색할 때 가능한 한 많은 리소스를 병렬로 생성하므로 변경 사항을 상당히 효율적으로 적용할 수 있습니다. 이것이 바로 선언적 언어의 장점입니다. 원하는 것을 지정하기만 하면 Terraform이 이를 실현하는 가장 효율적인 방법을 결정합니다. ***apply*** 명령을 실행하면 Terraform이 보안 그룹을 생성하고 EC2 인스턴스를 새 사용자 데이터가 있는 새 인스턴스로 교체하려고 한다는 것을 알 수 있습니다.

```
$ terraform apply
(...)
Terraform will perform the following actions:
 # aws_instance.example must be replaced
-/+ resource "aws_instance" "example" {
 ami = "ami-0fb653ca2d3203ac1"
 ~ availability_zone = "us-east-2c" -> (known after apply)
 ~ instance_state = "running" -> (known after apply)
 instance_type = "t2.micro"
 (...)
 + user_data = "c765373..." # forces replacement
 ~ volume_tags = {} -> (known after apply)
 ~ vpc_security_group_ids = [
 - "sg-871fa9ec",
 ] -> (known after apply)
 (...)
 }
 # aws_security_group.instance will be created
 + resource "aws_security_group" "instance" {
 + arn = (known after apply)
 + description = "Managed by Terraform"
 + egress = (known after apply)
 + id = (known after apply)
 + ingress = [
 + {
 + cidr_blocks = [
 + "0.0.0.0/0",
 ]
 + description = ""
 + from_port = 8080
 + ipv6_cidr_blocks = []
 + prefix_list_ids = []
 + protocol = "tcp"
```

```
  + security_groups = []
  + self = false
  + to_port = 8080
  },
  ]
  + name = "terraform-example-instance"
  + owner_id = (known after apply)
  + revoke_rules_on_delete = false
  + vpc_id = (known after apply)
  }
Plan: 2 to add, 0 to change, 1 to destroy.
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
  Enter a value:
```

계획 출력의 -/ +는 "교체"를 의미합니다. Terraform이 교체를 수행하도록 강제하는 원인을 파악하려면 계획 출력에서 ***forces replacement***라는 텍스트를 찾아보세요. user_data_replace_on_change를 true로 설정하고 user_data 파라미터를 변경했기 때문에 교체가 강제됩니다. 즉, 원래 EC2 인스턴스가 종료되고 완전히 새로운 인스턴스가 생성됩니다. 이는 이전에 설명한 불변 인프라 패러다임의 예입니다. 웹 서버가 교체되는 동안 해당 웹 서버의 모든 사용자는 다운타임이 생깁니다.

이제 apply 의 출력을 검토하고 yes를 입력하면 그림과 같이 새 EC2 인스턴스가 배포되는 것을 볼 수 있습니다.

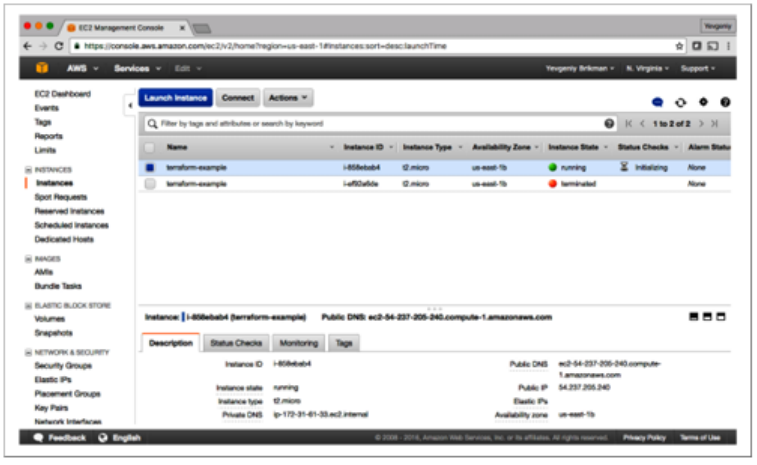


Figure 2-8. The new EC2 Instance with the web server code replaces the old Instance.

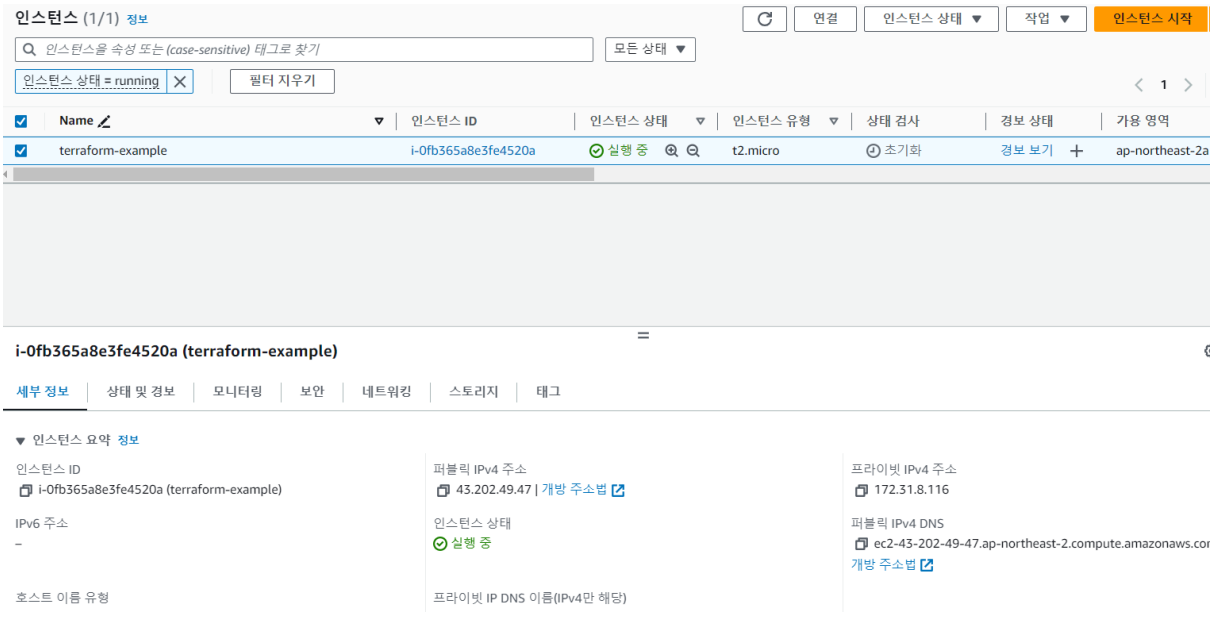
새 인스턴스를 클릭하면 화면 하단의 설명 패널에서 퍼블릭 IP 주소를 확인할 수 있습니다. 인스턴스가 부팅될 때까지 1~2분 정도 기다린 후 웹 브라우저나 컬과 같은 도구를 사용하여 포트 8080에서 이 IP 주소에 대한 HTTP 요청을 만듭니다.

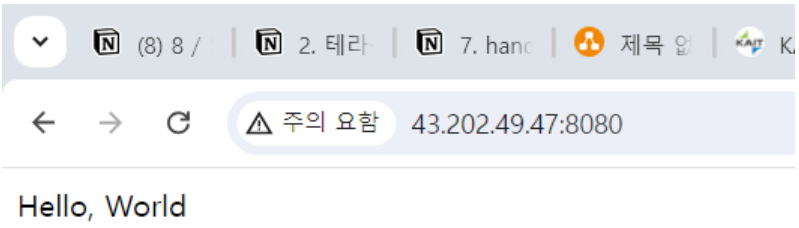
```
$ curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
Hello, World
```

네트워크 보안
이 책의 모든 예제를 단순하게 유지하기 위해 앞서 언급한 대로 기본 VPC뿐만 아니라 해당 VPC의 기본 서브넷에도 배포합니다. VPC는 각각 자체 IP 주소를 갖는 하나 이상의 서브넷으로 분할됩니다. 기본 VPC의 서브넷은 모두 퍼블릭 서브넷입니다. 즉, 퍼블릭 인터넷에서 액세스할 수 있는 IP 주소를 얻습니다. 이것이 바로 가정용 컴퓨터에서 EC2 인스턴스를 테스트할 수 있는 이유입니다. 빠른 실험에는 퍼블릭 서브넷에서 서버를 실행하는 것이 좋지만 실제 사용에서는 보안 위험이 있습니다. 전 세계의 해커들은 약점이 있는지 무작위로 IP 주소를 끊임없이 스캔하고 있습니다. 서버가 공개적으로 노출된 경우 실수로 단일 포트를 보호하지 않은 상태로 두거나 알려진 취약점이 있는 오래된 코드를 실행하면 누군가 침입할 수 있습니다. 따라서 프로덕션 시스템의 경우 모든 서버를 배포해야 합니다. 그리고 확실히 모든 데이터 저장소는 프라이빗 서브넷에 있으며 퍼블릭 인터넷이 아닌 VPC 내에서만 액세스할 수 있는 IP 주소를 가지고 있습니다. 퍼블릭 서브넷에서 실행해야 하는 유일한 서버는 가능한 한 많이 허용하지 않는 소수의 역방향 프록시와 로드 밸런서입니다

결과

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

resource "aws_instance" "example"{
  ami = "ami-062cf18d655c0b1e8"
  vpc_security_group_ids = [aws_security_group.instance.id]
  instance_type = "t2.micro"
  user_data  =  <<-EOF
                #!/bin/bash
                echo  "Hello, World"  >  index.html
                nohup  busybox  httpd  -f  -p  8080  &
                EOF
  user_data_replace_on_change  =  true
  tags = {
    Name = "terraform-example"
  }
}

resource  "aws_security_group" "instance"  {
  name  =  "terraform-example-instance"

  ingress  {
    from_port    =  8080
    to_port      =  8080
    protocol     =  "tcp"
    cidr_blocks  =  [ "0.0.0.0/0" ]
  }
}
```





구성 가능한 웹 서버 배포

웹 서버 코드의 보안 그룹과 사용자 데이터 구성 모두에서 포트 8080이 중복되어 있음을 알 수 있습니다. 이는 ***DRY***(Don't Repeat Yourself- 반복하지 않는다) 원칙에 위배됩니다. 즉, 모든 지식은 시스템 내에서 단일하고 명확하며 권위 있는 표현을 가져야 합니다. 포트 번호가 두 곳에 있는 경우 한 곳에서는 쉽게 업데이트할 수 있지만 다른 곳에서는 동일하게 변경하는 것을 잊어버립니다. 코드를 더욱 건조하고 구성 가능하게 만들기 위해 Terraform을 사용하면 입력 변수를 정의할 수 있습니다. 변수를 선언하는 구문은 다음과 같습니다.

```
variable "NAME" {
  [CONFIG ...]
}
```

변수 선언의 본문에는 다음과 같은 선택적 매개변수가 포함될 수 있습니다.

- 설명(description)
  변수가 어떻게 사용되는지 문서화하기 위해 이 매개변수를 사용하는 것이 좋습니다. 팀원은 코드를 읽는 동안 뿐만 아니라 계획을 실행하거나 명령을 적용할 때도 이 설명을 볼 수 있습니다
- 기본(default)
  명령줄(-var 옵션 사용), 파일(-varfile 옵션 사용) 또는 환경 변수(TF_VAR_<variable_name> 이름의 환경 변수에서 검색)를 통해 변수 값을 전달하는 등 변수 값을 제공하는 방법에는 여러 가지가 있습니다. 하지만 값이 전달되지 않

으면 변수는 이 기본값으로 대체됩니다. 기본값이 없으면 Terraform은 사용자에게 기본값을 묻는 메시지를 대화형으로 표시합니다.

- 유형(type)
  이를 통해 사용자가 전달하는 변수에 유형 제약 조건을 적용할 수 있습니다. Terraform은 string , number , bool , list , map , set , object , tuple 및 any 를 포함한 다양한 유형 제약 조건을 지원합니다. 간단한 오류를 포착하기 위해 유형 제약 조건을 정의하는 것은 항상 좋은 생각입니다. 유형을 지정하지 않으면 Terraform은 유형이 any 라고 가정합니다.
- 검증(validation)
  이를 통해 숫자에 최소값 또는 최대값을 적용하는 등 기본 유형 확인 이상의 입력 변수에 대한 사용자 정의 유효성 검사 규칙을 정의할 수 있습니다.
- 중요(sensitive)
  입력 변수에서 이 매개변수를 true로 설정하면 Terraform은 plan 또는 apply 를 실행할 때 이를 기록하지 않습니다. 변수(예: 비밀번호, API 키 등)를 통해 Terraform 코드에 전달하는 모든 비밀에 대해 이를 사용해야 합니다.

다음은 전달한 값이 숫자인지 확인하는 입력 변수의 예입니다.

```
variable "number_example" {
  description = "An example of a number variable in Terraform"
  type = number
  default = 42
}
```

다음은 값이 목록인지 확인하는 변수의 예입니다.

```
variable "list_example" {
 description = "An example of a list in Terraform"
 type = list
 default = ["a", "b", "c"]
}
```

유형 제약 조건을 결합할 수도 있습니다. 예를 들어, 목록의 모든 항목이 숫자여야 하는 목록 입력 변수는 다음과 같습니다.

```
variable "list_numeric_example" {
 description = "An example of a numeric list in Terraform"
 type = list(number)
 default = [1, 2, 3]
}
```

다음은 모든 값이 문자열이어야 하는 맵입니다.

```
variable "map_example" {
 description = "An example of a map in Terraform"
 type = map(string)
 default = {
   key1 = "value1"
   key2 = "value2"
   key3 = "value3"
 }
}
```

객체 유형 제약 조건을 사용하여 더 복잡한 구조 유형을 만들 수도 있습니다.

```
variable "object_example" {
 description = "An example of a structural type in Terraform"
 type = object({
   name = string
   age = number
   tags = list(string)
   enabled = bool
 })
 default = {
   name = "value1"
   age = 42
   tags = ["a", "b", "c"]
   enabled = true
 }
}
```

앞의 예에서는 값이 키 이름(문자열이어야 함), 나이(숫자여야 함), 태그(문자열 목록이어야 함) 및 활성화된 객체(부울이어야 함) 여야 하는 입력 변수를 생성합니다. 이 변수를 이 유형과 일치하지 않는 값으로 설정하려고 하면 Terraform에서 즉시 유형 오류를 표시합니다. 다음 예에서는 부울 대신 문자열로 활성화 설정을 시도하는 방법을 보여줍니다.

```
variable "object_example_with_error" {
 description = "An example of a structural type in Terraform with an error"
 type = object({
   name = string
   age = number
   tags = list(string)
   enabled = bool
 })
 default = {
   name = "value1"
   age = 42
   tags = ["a", "b", "c"]
   enabled = "invalid"
 }
}
```

다음과 같은 오류가 발생합니다.

```
$ terraform apply
Error: Invalid default value for variable
 on variables.tf line 78, in variable "object_example_with_error":
 78: default = {
 79: name = "value1"
 80: age = 42
 81: tags = ["a", "b", "c"]
 82: enabled = "invalid"
 83: }
This default value is not compatible with the variable's type constraint: a
bool is required.
```

웹 서버 예제로 돌아가서 필요한 것은 포트 번호를 저장하는 변수입니다.

```
variable "server_port" {
 description = "The port. the server will use for HTTP requests"
 type = number
}
```

server_port 입력 변수에는 기본값이 없으므로 지금 Apply 명령을 실행하면 Terraform은 대화형으로 server_port 값을 입력하라는 메시지를 표시하고 변수에 대한 설명을 표시합니다.

```
$ terraform apply

var.server_port
 The port the server will use for HTTP requests

 Enter a value:
```

대화형 프롬프트를 처리하지 않으려면 -var 명령줄 옵션을 통해 변수 값을 제공할 수 있습니다.

```
$ terraform plan -var "server_port=8080"
```

TF_VAR_<name>이라는 환경 변수를 통해 변수를 설정할 수도 있습니다. 여기서 <name>은 설정하려는 변수의 이름입니다.

```
$ export TF_VAR_server_port=8080
$ terraform plan
```

plan이나 apply를 실행할 때마다 추가 명령줄 인수를 기억하고 싶지 않다면 기본값을 지정할 수 있습니다.

```
variable "server_port" {
 description = "The port the server will use for HTTP requests"
 type = number
 default = 8080
}
```

Terraform 코드에서 입력 변수의 값을 사용하려면 다음 구문을 포함하는 변수 참조라는 새로운 유형의 표현식을 사용할 수 있습니다.

```
var.<VARIABLE_NAME>
```

예를 들어 보안 그룹의 from_port 및 to_port 매개변수를 server_port 변수의 값으로 설정하는 방법은 다음과 같습니다.

```
resource "aws_security_group" "instance" {
 name = "terraform-example-instance"
 ingress {
 from_port = var.server_port
 to_port = var.server_port
 protocol = "tcp"
 cidr_blocks = ["0.0.0.0/0"]
 }
}
```

사용자 데이터 스크립트에서 포트를 설정할 때 동일한 변수를 사용하는 것도 좋은 생각입니다. 문자열 리터럴 내부에서 참조를 사용하려면 다음 구문을 사용하는 ***interpolation*** 이라는 새로운 유형의 표현식을 사용해야 합니다.

```
"${...}"
```

중괄호 안에 유효한 참조를 넣으면 Terraform이 이를 문자열로 변환합니다. 예를 들어, 사용자 데이터 문자열 내에서 var.server_port를 사용하는 방법은 다음과 같습니다.

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

입력 변수 외에도 Terraform에서는 다음 구문을 사용하여 출력 변수를 정의할 수 있습니다.

```
output "<NAME>" {
 value = <VALUE>
 [CONFIG ...]
}
```

NAME은 출력 변수의 이름이고 VALUE는 출력하려는 Terraform 표현식 입니다. CONFIG에는 다음과 같은 선택적 매개변수가 포함될 수 있습니다.

- 설명 (description)
  이 매개변수를 사용하여 출력 변수에 포함된 데이터 유형을 문서화하는 것은 항상 좋은 생각입니다.
- 중요 (sensitive)
  계획 또는 적용 종료 시 이 출력을 기록하지 않도록 Terraform에 지시하려면 이 매개변수를 true로 설정합니다. 이는 출력 변수에 비밀번호나 개인 키와 같은 비밀이 포함된 경우 유용합니다. sensitive = true로 표시된 입력 변수 또는 리소스 속성을 참조하는 경우 비밀을 출력하고 있음을 나타내기 위해 출력 변수를 sensitive = true로 표시해야 합니다.
- 의존성
  일반적으로 Terraform은 코드 내의 참조를 기반으로 종속성 그래프를 자동으로 파악하지만, 드문 경우에는 추가 참고 정보를 제공해야 합니다. 예를 들어 서버의 IP 주소를 반환하는 출력 변수가 있지만 해당 서버에 대해 보안 그룹(방화벽)이 올바르게 구성될 때까지 해당 IP에 액세스할 수 없습니다. 이 경우, dependency_on 을 사용하여 Terraform에 IP 주소 출력 변수와 보안 그룹 리소스 사이에 종속성이 있음을 명시적으로 알릴 수 있습니다.

예를 들어 서버의 IP 주소를 찾기 위해 EC2 콘솔을 수동으로 탐색할 필요 없이 IP 주소를 출력 변수로 제공할 수 있습니다.

```
output "public_ip" {
 value = aws_instance.example.public_ip
 description = "The public IP address of the web server"
}
```

이 코드는 속성을 다시 참조하는데, 이번에는 aws_instance 리소스의 public_ip 속성을 참조합니다. 적용 명령을 다시 실행하면 Terraform은 리소스를 변경하지 않았기 때문에 새롭게 적용하지 않지만 맨 끝에 새 출력이 표시됩니다.

```
$ terraform apply

(...)

aws_security_group.instanceｽﾛｽ /: Refreshing state... [id=sg-078ccb4f9533d2c1a]
```

```
aws_instance.example: Refreshing state... [id=i-028cad2d4e6bddec6]

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

Outputs:

public_ip = "54.174.13.5"
```

보시다시피, terraform apply 를 실행한 후 출력 변수가 콘솔에 표시되며, Terraform 코드 사용자가 유용하다고 생각할 수 있습니다(예: 이제 웹 서버가 배포된 후 테스트할 IP를 알 수 있습니다). terraform 출력 명령을 사용하여 변경 사항을 적용하지 않고 모든 출력을 나열할 수도 있습니다.

```
$ terraform output
public_ip = "54.174.13.5"
```

그리고 terraform 출력 <OUTPUT_NAME> 을 실행하여 <OUTPUT_NAME>이라는 특정 출력의 값을 확인할 수 있습니다.

```
$ terraform output public_ip
"54.174.13.5"
```

이는 스크립팅에 특히 편리합니다. 예를 들어, terraform Apply를 실행하여 웹 서버를 배포하고, terraform 출력 public_ip를 사용하여 공용 IP를 확보하고, curl을 IP에서 빠른 스모크 테스트로 실행하여 배포가 작동했는지 확인하는 배포 스크립트를 생성할 수 있습니다. 입력 및 출력 변수는 구성 및 재사용이 가능한 인프라 코드를 생성하는 데 필수적인 요소이기도 합니다.

- 결과

```
[root@controller terraform-demo]# terraform apply
var.server_port
  The port. the server will use for HTTP requests

  Enter a value: 8080

aws_security_group.instance: Refreshing state... [id=sg-03fc72b2e24f09f3e]
aws_instance.example: Refreshing state... [id=i-0fb365a8e3fe4520a]

Changes to Outputs:
  + public_ip = "43.202.49.47"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes


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

Outputs:

public_ip = "43.202.49.47"
[root@controller terraform-demo]# curl 43.202.49.47:8080
Hello, World
```

웹 서버 클러스터 배포

단일 서버를 실행하는 것은 좋은 시작이지만 실제로는 단일 서버는 단일 실패 지점입니다. 해당 서버가 충돌하거나 너무 많은 트래픽으로 인해 과부하가 발생하는 경우 사용자는 귀하의 사이트에 액세스할 수 없습니다. 해결책은 서버 클러스터를

실행하고, 다운되는 서버를 중심으로 라우팅하고 트래픽에 따라 클러스터 크기를 늘리거나 줄이는 것입니다. 이러한 클러스터를 수동으로 관리하는 것은 많은 작업입니다. 다행히도 그림과 같이 ASG(Auto Scaling Group)를 사용하면 AWS가 이를 처리하도록 할 수 있습니다. ASG는 EC2 인스턴스 클러스터 시ls

작, 각 인스턴스 상태 모니터링, 실패한 인스턴스 교체, 로드에 따른 클러스터 크기 조정 등 많은 작업을 자동으로 처리합니다.

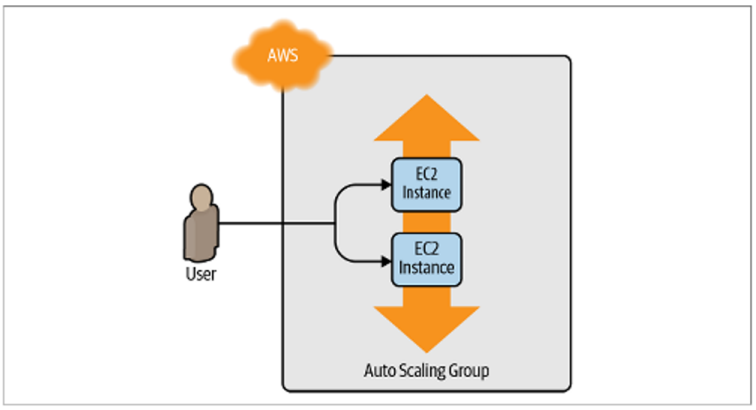


Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group.

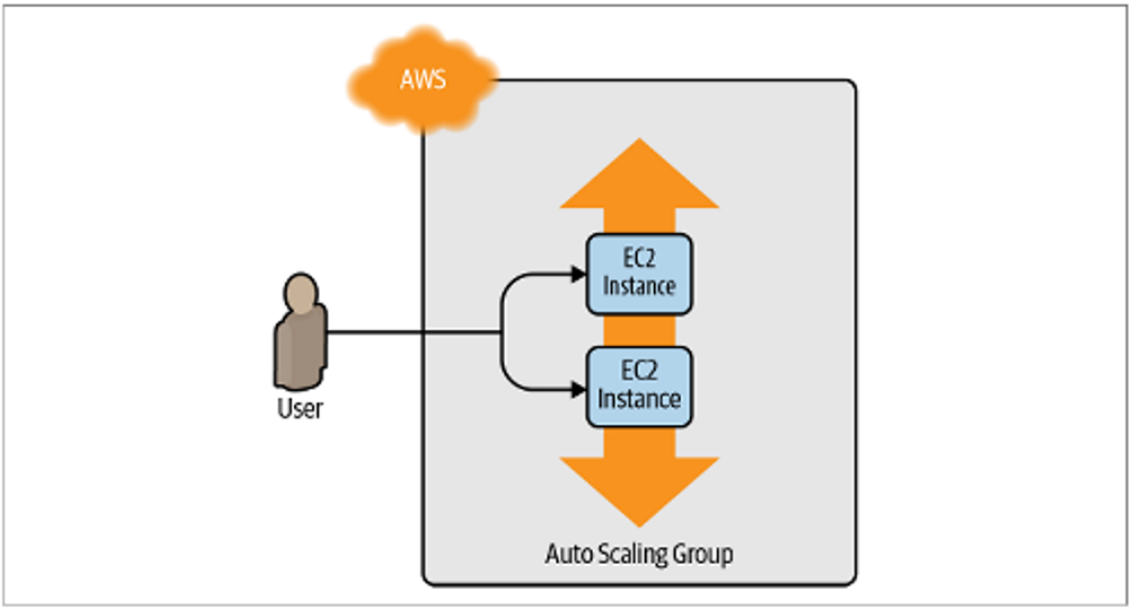


Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group.

ASG 생성의 첫 번째 단계는 ASG에서 각 EC2 인스턴스를 구성하는 방법을 지정하는 시작 구성을 생성하는 것입니다. aws_launch_configuration 리소스는 aws_instance 리소스와 거의 동일한 매개 변수를 사용하지만 이름 태그 또는 user_data_replace_on_change 매개 변수 (ASG는 기본적으로 새 인스턴스를 시작하므로 지원하지 않음)를 지원하지 않기 때문에 필요하지 않음. 매개변수 중 2개는 서로 다른 이름(ami는 이제 image_id 이고 vpc_security_group_ids는 이제 security_groups 으로 표시)을 가지며 다음과 같이 aws_instance를 aws_launch_configuration으로 바꿉니다.

```
resource "aws_launch_configuration" "example" {
 image_id = "ami-0fb653ca2d3203ac1"
 instance_type = "t3.micro"
 security_groups = [aws_security_group.instance.id]
 user_data = <<-EOF
 #!/bin/bash
 echo "Hello, World" > index.html
 nohup busybox httpd -f -p ${var.server_port} &
 EOF
}
```

이제 aws_autoscaling_group 리소스를 사용하여 ASG 자체를 생성할 수 있습니다.

```
resource "aws_autoscaling_group" "example" {
 launch_configuration = aws_launch_configuration.example.name
 min_size = 2
 max_size = 10
 tag {
 key = "Name"
 value = "terraform-asg-example"
 propagate_at_launch = true
 }
}
```

이 ASG는 2~10개의 EC2 인스턴스(초기 시작 시 기본값은 2개)에서 실행되며 각각 terraform-asg-example이라는 이름으로 태그가 지정됩니다. ASG는 리소스 참조를 참조하는 방식으로 시작 구성 이름을 입력하는데 이로 인해 문제가 발생합니다. 시작 구성은 변경할 수 없으므로 시작 구성의 매개변수를 변경하면 Terraform이 이를 교체하려고 시도합니다. 일반적으로 리소스를 교체할 때 Terraform은 이전 리소스를 먼저 삭제한 다음 대체 리소스를 생성하지만 이제 ASG에 이전 리소스에 대한 참조가 있으므로 Terraform은 이를 삭제할 수 없습니다.

이 문제를 해결하려면 수명 주기 설정을 사용할 수 있습니다. 모든 Terraform 리소스는 해당 리소스가 생성, 업데이트 및/또는 삭제되는 방식을 구성하는 여러 수명 주기 설정을 지원합니다. 특히 유용한 수명 주기 설정은 create_before_destroy 입니다. create_before_destroy를 true 로 설정하면 Terraform은 리소스를 교체하는 순서를 반전하여 먼저 대체 리소스를 생성한 다음(이전 리소스를 가리키는 모든 참조를 대체 리소스를 가리키도록 업데이트하는 것을 포함하여) 이전 리소스를 삭제합니다. 다음과 같이 aws_launch _configuration에 수명 주기 블록을 추가합니다.

```
resource "aws_launch_configuration" "example" {
 image_id = "ami-0fb653ca2d3203ac1"
 instance_type = "t3.micro"
 security_groups = [aws_security_group.instance.id]
 user_data = <<-EOF
 #!/bin/bash
 echo "Hello, World" > index.html
 nohup busybox httpd -f -p ${var.server_port} &
 EOF
 # Required when using a launch configuration with an auto scaling group.
```

```
  lifecycle {
    create_before_destroy = true
  }
}
```

ASG가 작동하도록 하려면 ASG에 추가해야 하는 또 다른 매개변수인 subnet_ids도 있습니다. 이 매개변수는 EC2 인스턴스가 배포되어야 하는 VPC 서브넷의 ASG를 지정합니다. 각 서브넷은 격리된 AWS AZ(즉, 격리된 데이터 센터)에 있으므로 여러 서브넷에 인스턴스를 배포하면 일부 데이터 센터가 중단되더라도 서비스가 계속 실행될 수 있습니다. 서브넷 목록을 하드코딩할 수 있지만 유지 관리나 이식이 불가능하므로 데이터 원본을 사용하여 AWS 계정의 서브넷 목록을 가져오는 것이 더 나은 옵션입니다.

데이터 소스는 Terraform을 실행할 때마다 공급자(이 경우 AWS)에서 가져오는 읽기 전용 정보를 나타냅니다. Terraform 구성에 데이터 소스를 추가해도 새로운 것은 생성되지 않습니다. 이는 공급자의 API에서 데이터를 쿼리하고 해당 데이터를 나머지 Terraform 코드에서 사용할 수 있도록 하는 방법일 뿐입니다. 각 Terraform 제공자는 다양한 데이터 소스를 노출합니다. 예를 들어, AWS 공급자에는 VPC 데이터, 서브넷 데이터, AMI ID, IP 주소 범위, 현재 사용자의 ID 등을 조회할 수 있는 데이터 소스가 포함되어 있습니다. 데이터 소스를 사용하는 구문은 리소스(variable output) 구문과 매우 유사합니다.

```
data "<PROVIDER>_<TYPE>" "<NAME>" {
 [CONFIG ...]
}
```

여기서 PROVIDER는 공급자 이름(예: aws)이고, TYPE은 사용하려는 데이터 소스 유형(예: vpc)이며, NAME은 Terraform 코드 전체에서 이 데이터 소스를 참조하는 데 사용할 수 있는 식별자입니다. CONFIG는 해당 데이터 소스와 관련된 하나 이상의 인수로 구성됩니다. 예를 들어 aws_vpc 데이터 소스를 사용하여 기본 VPC에 대한 데이터를 조회하는 방법은 다음과 같습니다

```
data "aws_vpc" "default" {
 default = true
}
```

데이터 원본의 경우 전달하는 인수는 일반적으로 찾고 있는 정보가 무엇인지 데이터 원본에 나타내는 검색 필터입니다. aws_vpc 데이터 소스에서 필요한 필터는 default = true 이며, 이는 Terraform이 AWS 계정에서 기본 VPC를 조회하도록 지시합니다. 데이터 소스에서 데이터를 가져오려면 다음 속성 참조 구문을 사용합니다.

```
data.<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

예를 들어 aws_vpc 데이터 원본에서 VPC의 ID를 얻으려면 다음을 사용합니다.

```
data.aws_vpc.default.id
```

이를 다른 데이터 소스인 aws_subnets 와 결합하여 해당 VPC 내의 서브넷을 조회할 수 있습니다.

```
data "aws_subnets" "default" {
 filter {
 name = "vpc-id"
 values = [data.aws_vpc.default.id]
 }
}
```

마지막으로 aws_subnets 데이터 소스에서 서브넷 ID를 가져오고 ASG에 (다소 이상한 이름의) vpc_zone_identifier 인수를 통해 해당 서브넷을 사용하도록 지시할 수 있습니다.

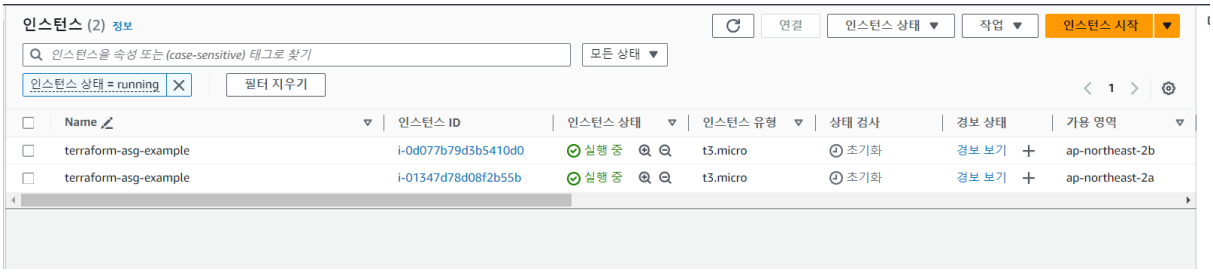
```
resource "aws_autoscaling_group" "example" {
 launch_configuration = aws_launch_configuration.example.name
 vpc_zone_identifier = data.aws_subnets.default.ids
 min_size = 2
 max_size = 10
 tag {
 key = "Name"
 value = "terraform-asg-example"
 propagate_at_launch = true
 }
}
```

결과

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

resource "aws_launch_configuration" "example" {
  image_id = "ami-062cf18d655c0b1e8"
  instance_type = "t3.micro"
  security_groups = [aws_security_group.instance.id]
  user_data = <<-EOF
  #!/bin/bash
  echo "Hello, World" > index.html
  nohup busybox httpd -f -p ${var.server_port} &
  EOF
  # Required when using a launch configuration with an auto scaling group.
  lifecycle {
    create_before_destroy = true
  }
}
```

```
resource  "aws_security_group" "instance"  {
  name  =  "terraform-example-instance"
  ingress  {
    from_port    =  var.server_port
    to_port      =  var.server_port
    protocol     =  "tcp"
    cidr_blocks  =  [ "0.0.0.0/0" ]
  }
}

resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier = data.aws_subnets.default.ids
  min_size = 2
  max_size = 10
  tag {
    key = "Name"
    value = "terraform-asg-example"
    propagate_at_launch = true
  }
}

data "aws_vpc" "default" {
  default = true
}

data "aws_subnets" "default" {
  filter {
  name = "vpc-id"
  values = [data.aws_vpc.default.id]
  }
}


variable "server_port" {
  description = "The port. the server will use for HTTP requests"
  type = number
  default = 8080
}

#output "public_ip" {
#  value = aws_instance.example.public_ip
#  description = "apache web server ip"
#}
```

```
Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  - public_ip = "43.202.49.47" -> null

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_launch_configuration.example: Creating...
aws_launch_configuration.example: Creation complete after 0s [id=terraform-20240805034705571700000001]
aws_autoscaling_group.example: Creating...
aws_autoscaling_group.example: Still creating... [10s elapsed]
aws_autoscaling_group.example: Creation complete after 15s [id=terraform-20240805034706269700000002]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```



로드밸런서 배포

이제 ASG를 배포할 수 있지만 작은 문제가 있습니다. 각각 고유한 IP 주소를 가진 여러 서버가 있지만 일반적으로 최종 사용자에게 사용할 단일 IP만 제공하려고 합니다. 이 문제를 해결하는 한 가지 방법은 로드 밸런서를 배포하여 서버 전체에 트래픽을 분산하고 모든 사용자에게 로드 밸런서의 IP(실제로는 DNS 이름)를 제공하는 것입니다. 가용성과 확장성이 뛰어난 로드 밸런서를 만드는 것은 많은 작업입니다. 이번에도 그림과 같이 Amazon의 ELB(Elastic Load Balancer) 서비스를 사용하여 AWS가 이를 처리하도록 할 수 있습니다.

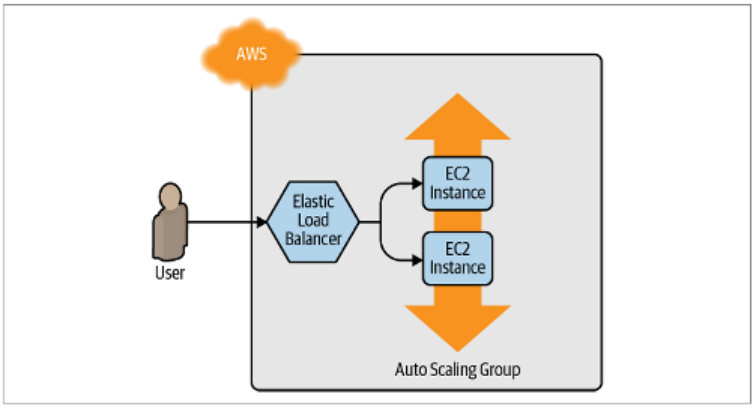


Figure 2-10. Use Amazon ELB to distribute traffic across the Auto Scaling Group.

AWS는 세 가지 유형의 로드 밸런서를 제공합니다.

- 애플리케이션 로드 밸런서(ALB)
  HTTP 및 HTTPS 트래픽의 로드 밸런싱에 가장 적합합니다. OSI(개방형 시스템 상호 연결) 모델의 애플리케이션 계층(계층 7)에서 작동합니다.
- 네트워크 로드 밸런서(NLB)
  TCP, UDP, TLS 트래픽의 로드 밸런싱에 가장 적합합니다. ALB보다 더 빠르게 로드에 응답하여 확장 및 축소할 수 있

습니다(NLB는 초당 수천만 요청으로 확장되도록 설계되었습니다). OSI 모델의 전송 계층(계층 4)에서 작동합니다.

- 클래식 로드 밸런서(CLB)
  이는 ALB와 NLB보다 앞서는 "레거시" 로드 밸런서입니다. HTTP, HTTPS, TCP 및 TLS 트래픽을 처리할 수 있지만 ALB 또는 NLB보다 기능이 훨씬 적습니다. OSI 모델의 애플리케이션 계층(L7)과 전송 계층(L4) 모두에서 작동합니다.

요즘 대부분의 애플리케이션은 ALB 또는 NLB를 사용해야 합니다. 작업 중인 간단한 웹 서버 예제는 극단적인 성능 요구 사항이 없는 HTTP 앱이므로 ALB가 가장 적합할 것입니다. ALB는 그림에 표시된 대로 여러 부분으로 구성됩니다.

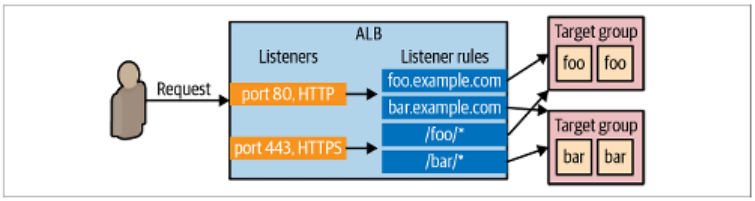


Figure 2-11. An ALB consists of listeners, listener rules, and target groups.

- 리스너
  특정 포트(예: 80) 및 프로토콜(예: HTTP)을 수신합니다.
- 리스너 규칙
  리스너로 들어오는 요청을 받아 특정 경로(예: /foo 및 /bar) 또는 호스트 이름(예: foo.example.com 및 bar.example.com)과 일치하는 요청을 특정 대상 그룹에 보냅니다.
- 대상 그룹
  로드 밸런서로부터 요청을 수신하는 하나 이상의 서버입니다. 또한 대상 그룹은 이러한 서버에서 상태 확인을 수행하고 건강한 노드에만 요청을 보냅니다.

첫 번째 단계는 aws_lb 리소스를 사용하여 ALB 자체를 생성하는 것입니다.

```
resource "aws_lb" "example" {
 name = "terraform-asg-example"
 load_balancer_type = "application"
 subnets = data.aws_subnets.default.ids
}
```

subnets 파라미터는 aws_subnets 데이터 원본을 사용하여 기본 VPC의 모든 서브넷을 사용하도록 로드 밸런서를 구성합니다. AWS 로드 밸런서는 단일 서버로 구성되지 않고 별도의 서브넷(따라서 별도의 데이터 센터)에서 실행될 수 있는 여러 서버로 구성됩니다. AWS는 트래픽을 기준으로 로드 밸런서 서버 수를 자동으로 확장 및 축소하고 해당 서버 중 하나가 다운되면 장애 조치를 처리하므로 즉시 확장성과 고가용성을 얻을 수 있습니다. 다음 단계는 aws_lb_listener 리소스를 사용하여 이 ALB에 대한 리스너를 정의하는 것입니다.

```
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
 }
```

```
 }
}
```

이 리스너는 기본 HTTP 포트인 포트 80에서 수신 대기하고, HTTP를 프로토콜로 사용하고, 리스너 규칙과 일치하지 않는 요청에 대한 기본 응답으로 간단한 404 페이지를 보내도록 ALB를 구성합니다. 기본적으로 ALB를 포함한 모든 AWS 리소스는 들어오거나 나가는 트래픽을 허용하지 않으므로 ALB 전용으로 새 보안 그룹을 생성해야 합니다. 이 보안 그룹은 HTTP를 통해 로드 밸런서에 액세스할 수 있도록 포트 80에서 들어오는 요청을 허용해야 하며, 로드 밸런서가 상태 확인을 수행할 수 있도록 모든 포트에서 나가는 요청을 허용해야 합니다.

```
resource "aws_security_group" "alb" {
 name = "terraform-example-alb"
 # Allow inbound HTTP requests
 ingress {
 from_port = 80
 to_port = 80
 protocol = "tcp"
 cidr_blocks = ["0.0.0.0/0"]
 }
 # Allow all outbound requests
 egress {
 from_port = 0
 to_port = 0
 protocol = "-1"
 cidr_blocks = ["0.0.0.0/0"]
 }
}
```

security_groups 인수를 통해 이 보안 그룹을 사용하도록 aws_lb 리소스에 알려야 합니다.

```
resource "aws_lb" "example" {
 name = "terraform-asg-example"
 load_balancer_type = "application"
 subnets = data.aws_subnets.default.ids
 security_groups = [aws_security_group.alb.id]
}
```

다음으로 aws_lb_target_group 리소스를 사용하여 ASG에 대한 대상 그룹을 생성해야 합니다.

```
resource "aws_lb_target_group" "asg" {
 name = "terraform-asg-example"
 port = var.server_port
 protocol = "HTTP"
 vpc_id = data.aws_vpc.default.id
 health_check {
 path = "/"
 protocol = "HTTP"
 matcher = "200"
 interval = 15
 timeout = 3
 healthy_threshold = 2
```

```
 unhealthy_threshold = 2
 }
}
```

이 대상 그룹은 정기적으로 각 인스턴스에 HTTP 요청을 보내 인스턴스의 상태를 확인하고, 인스턴스가 구성된 ***matcher*** 와 일치하는 응답을 반환하는 경우에만 인스턴스를 "정상"으로 간주합니다(예: 200을 찾도록 ***matcher***를 구성할 수 있음). 인스턴스가 다운되거나 과부하되어 인스턴스가 응답하지 못하는 경우 해당 인스턴스는 "비정상"으로 표시되며 대상 그룹은 사용자의 혼란을 최소화하기 위해 자동으로 트래픽 전송을 중지합니다. 대상 그룹은 요청을 보낼 EC2 인스턴스를 어떻게 확인할까요? aws_lb_tar get_group_attachment 리소스를 사용하여 EC2 인스턴스의 정적 목록을 대상 그룹에 연결할 수 있지만 ASG를 사용하면 인스턴스가 언제든지 시작되거나 종료될 수 있으므로 정적 목록은 작동하지 않습니다. 대신 ASG와 ALB 간의 최고 수준의 통합을 활용할 수 있습니다. aws_autoscal ing_group 리소스로 돌아가서 target_group_arns 인수가 새 대상 그룹을 가리키도록 설정합니다.

```
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 = 2
 max_size = 10
 tag {
  key = "Name"
  value = "terraform-asg-example"
  propagate_at_launch = true
 }
}
```

또한 health_check_type을 "ELB"로 업데이트해야 합니다. 기본 health_check_type은 "EC2" 입니다. 이는 AWS 하이퍼바이저가 VM이 완전히 다운되었거나 접근할 수 없다고 말하는 경우에만 인스턴스를 비정상으로 간주하는 최소 상태 확인입니다. "ELB" 상태 확인은 ASG에 대상 그룹의 상태 확인을 사용하여 인스턴스가 정상인지 확인하고 대상 그룹이 비정상으로 보고하는 경우 인스턴스를 자동으로 교체하도록 지시하므로 더욱 강력합니다. 이렇게 하면 인스턴스가 완전히 다운된 경우 뿐만 아니라 메모리가 부족하거나 중요한 프로세스가 중단되어 요청 처리가 중단된 경우에도 인스턴스가 교체됩니다. 마지막으로 aws_lb_listener_rule 리소스를 사용하여 리스너 규칙을 생성하여 이러한 모든 부분을 하나로 묶을 시간입니다.

```
resource "aws_lb_listener_rule" "asg" {
 listener_arn = aws_lb_listener.http.arn
 priority = 100
 condition {
  path_pattern {
   values = ["*"]
  }
 }
 action {
  type = "forward"
  target_group_arn = aws_lb_target_group.asg.arn
 }
}
```

앞의 코드는 ASG가 포함된 대상 그룹에 대한 경로와 일치하는 요청을 보내는 리스너 규칙을 추가합니다.

로드 밸런서를 배포하기 전에 마지막으로 수행할 작업이 하나 있습니다. 이전에 보유했던 단일 EC2 인스턴스의 이전 public_ip 출력을 ALB의 DNS 이름을 표시하는 출력으로 바꾸는 것입니다.

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

terraform apply 를 실행하고 계획 출력을 읽어보세요. 원래 단일 EC2 인스턴스가 제거되고 그 자리에 Terraform이 시작 구성, ASG, ALB 및 보안 그룹을 생성하는 것을 확인해야 합니다. 모두 준비가 되었다면 yes를 입력하고 Enter 키를 누르세요. 적용이 완료되면 alb_dns_name 출력이 표시됩니다.

```
Outputs:
alb_dns_name = "terraform-asg-example-123.us-east-2.elb.amazonaws.com"
```

이 URL을 복사해 두었다가 나중에 웹 페이지를 확인 할때 입력합니다. 인스턴스가 부팅되고 ALB에 정상으로 표시되는 데 몇 분 정도 걸립니다. 그동안 배포한 내용을 검사할 수 있습니다. EC2 콘솔의 ASG 섹션을 열면 그림과 같이 ASG가 생성된 것을 볼 수 있습니다.

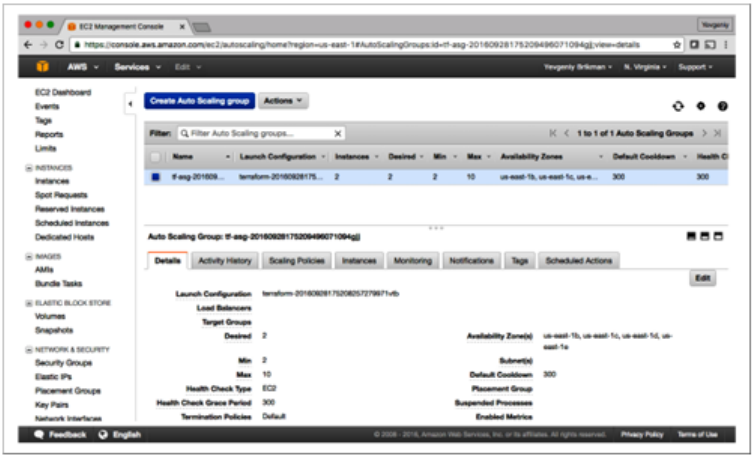


Figure 2-12. The AWS Console shows all the ASGs you've created.

인스턴스 탭으로 전환하면 그림과 같이 두 개의 EC2 인스턴스가 시작되는 것을 볼 수 있습니다.

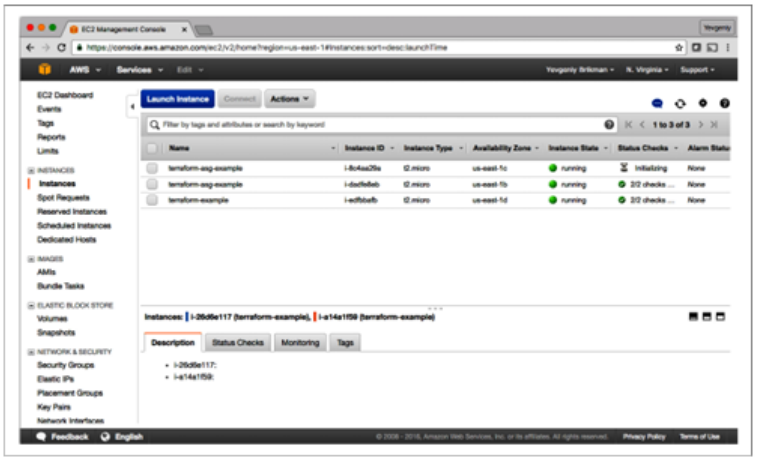


Figure 2-13. The EC2 Instances in the ASG are launching.

Load Balancers 탭을 클릭하면 그림 2-14와 같이 ALB가 표시됩니다.

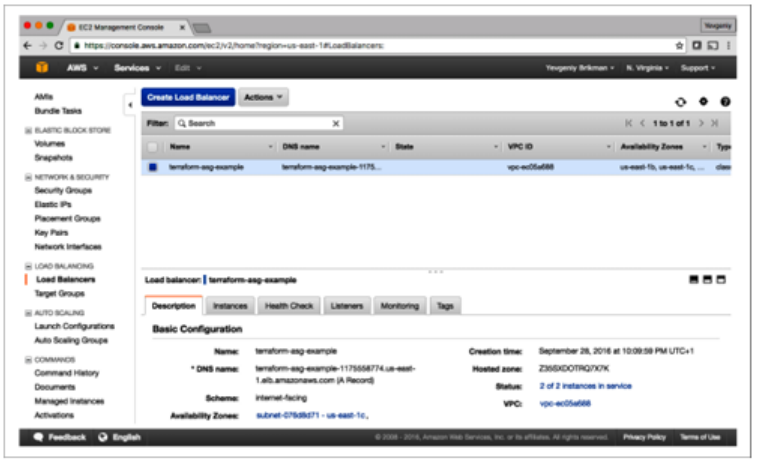


Figure 2-14. The AWS Console shows all the ALBs you've created.

마지막으로 Target Groups 탭을 클릭하면 그림 2-15와 같이 대상 그룹을 찾을 수 있습니다.

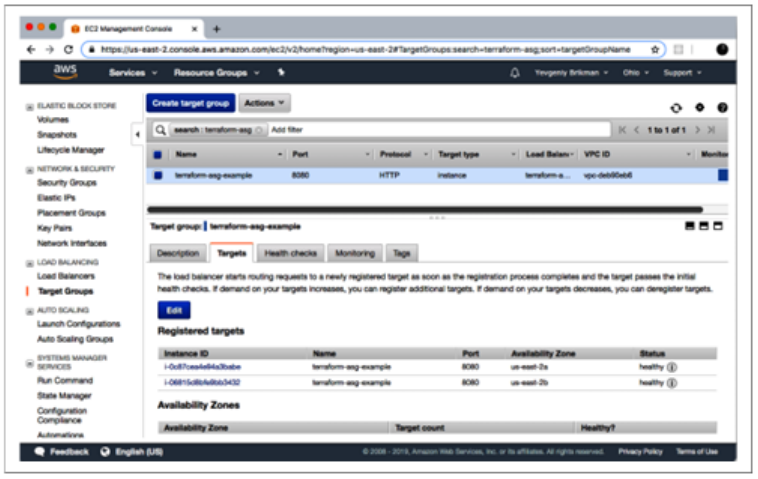


Figure 2-15. The AWS Console shows all the target groups you've created.

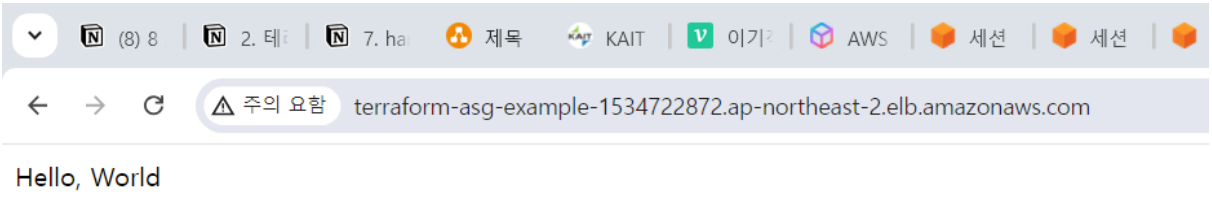
대상 그룹을 클릭하고 화면 하단에서 대상 탭을 찾으면 인스턴스가 대상 그룹에 등록되고 상태 확인을 거치는 것을 볼 수 있습니다. 상태 표시기가 둘 다 "정상"으로 표시될 때까지 기다립니다. 이 작업은 일반적으로 1~2분 정도 소요됩니다. 표시되면 이전에 복사한 alb_dns_name 출력을 테스트하세요.

```
$ curl http://<alb_dns_name>
Hello, World
```

ALB는 EC2 인스턴스로 트래픽을 라우팅하고 있습니다. URL에 액세스할 때마다 요청을 처리하기 위해 다른 인스턴스를 선택합니다. 이제 완벽하게 작동하는 웹 서버 클러스터가 생겼습니다!

결과

```
b503107e57ab/7f244b64ed9a1d36/379d14cc1b8137fd]

Apply complete! Resources: 5 added, 1 changed, 0 destroyed.

Outputs:

alb_dns_name = "terraform-asg-example-1534722872.ap-northeast-2.elb.amazonaws.com"
[root@controller terraform-demo]#
```



자원반환

이제 클러스터가 새 인스턴스를 시작하거나 이전 인스턴스를 종료하는 데 어떻게 반응하는지 확인할 수 있습니다. 예를 들어 인스턴스 탭으로 이동하여 확인란을 선택하고 상단에 있는 작업 버튼을 클릭한 다음 인스턴스 상태를 종료로 설정하여 인스턴스 중 하나를 종료합니다. ALB URL을 계속 테스트하면 인스턴스를 종료하는 동안에도 각 요청에 대해 200 OK를 받아야 합니다. ALB는 인스턴스가 다운되었음을 자동으로 감지하고 해당 인스턴스로의 라우팅을 중지하기 때문입니다. 더욱 흥미로운 점은 인스턴스가 종료된 후 짧은 시간 내에 ASG가 두 개 미만의 인스턴스가 실행 중임을 감지하고 이를 교체하기 위해 자동으로 새 인스턴스를 시작한다는 것입니다. Terraform 코드에 Desired_capacity 매개변수를 추가하고 Apply 를 다시 실행하여 ASG의 크기가 어떻게 자체적으로 조정되는지 확인할 수도 있습니다.

이 장의 끝이나 다음 장의 끝에서 Terraform 실험을 마친 후에는 AWS에서 비용을 청구하지 않도록 생성한 모든 리소스를 제거하는 것이 좋습니다. Terraform은 생성한 리소스를 추적하므로 정리가 간단합니다. 여러분이 해야 할 일은 destroy 명령을 실행하는 것뿐입니다:

```
$ terraform destroy
(...)
Terraform will perform the following actions:
 # aws_autoscaling_group.example will be destroyed
 - resource "aws_autoscaling_group" "example" {
 (...)
 }
 # aws_launch_configuration.example will be destroyed
 - resource "aws_launch_configuration" "example" {
 (...)
 }
 # aws_lb.example will be destroyed
 - resource "aws_lb" "example" {
 (...)
 }
 (...)
Plan: 0 to add, 0 to change, 8 to destroy.
Do you really want to destroy all resources?
 Terraform will destroy all your managed infrastructure, as shown above.
 There is no undo. Only 'yes' will be accepted to confirm.
 Enter a value:
```

프로덕션 환경에서는 destroy를 거의 실행할 필요는 없습니다. destroy 명령에는 "실행 취소"가 없으므로 Terraform은 수행 중인 작업을 검토할 수 있는 마지막 기회를 제공하고 삭제하려는 모든 리소스 목록을 표시하며 삭제 확인 메시지를 표시합니다. 모든 것이 괜찮아 보이면 yes를 입력하고 Enter를 누르십시오. Terraform은 가능한 한 많은 병렬 처리를 사용하여 종속성 그래프를 작성하고 모든 리소스를 올바른 순서로 삭제합니다. 결국, 코드형 인프라의 장점은 해당 리소스에 대한 모든 정보가 코드에 캡처되므로 언제든지 단일 명령인 terraform apply를 사용하여 모든 정보를 다시 생성할 수 있다는 것입니다. 실제로 인프라 기록을 추적할 수 있도록 최신 변경 사항을 Git에 커밋할 수도 있습니다.

결론

이제 Terraform 사용 방법에 대한 기본적인 내용을 이해했습니다. 선언적 언어를 사용하면 생성하려는 인프라를 정확하게 설명하기가 쉽습니다. plan 명령을 사용하면 변경 사항을 배포하기 전에 변경 사항을 확인하고 버그를 잡을 수 있습니다. 변수, 참조 및 종속성을 사용하면 코드에서 중복을 제거하고 고도로 구성 가능하게 만들 수 있습니다.

실습 (상태 관리)

이전 장에서는 리소스를 생성하고 업데이트하면서 terraform plan 또는 terraform Apply를 실행할 때마다 Terraform이 이전에 생성한 리소스를 찾아 그에 따라 업데이트할 수 있다는 것을 알아차렸을 것입니다. 하지만 Terraform은 관리해야 할 리소스가 무엇인지 어떻게 알았습니까? AWS 계정에는 다양한 메커니즘(일부는 수동으로, 일부는 Terraform을 통해, 일부는 CLI를 통해)을 통해 배포된 모든 종류의 인프라가 있을 수 있는데, Terraform은 자신이 담당하는 인프라가 무엇인지 어떻게 알 수 있습니까? 이 장에서는 Terraform이 인프라 상태를 추적하는 방법과 Terraform 프로젝트의 파일 레이아웃, 격리 및 잠금에 미치는 영향을 살펴보겠습니다. 이제부터 다룰 주요 주제는 다음과 같습니다.

- Terraform 상태(state)란 무엇입니까?
- 상태 파일용 공유 저장소
- Terraform 백엔드의 제한사항
- 상태 파일 격리
  - 작업 공간을 통한 격리
  - 파일 레이아웃을 통한 격리
- terraform_remote_state 데이터 소스

테라폼 상태(State)란

Terraform 상태란 무엇일까요? Terraform을 실행할 때마다 Terraform 상태 파일에 생성된 인프라에 대한 정보가 기록됩니다. 기본적으로 /foo/bar 폴더에서 Terraform을 실행하면 Terraform은 /foo/bar/terraform.tfstate 파일을 생성합니다. 이 파일에는 구성 파일의 Terraform 리소스에서 실제 리소스 표현으로의 매핑을 기록하는 사용자 지정 JSON 형식이 포함되어 있습니다. 예를 들어 Terraform 구성에 다음이 포함되어 있다고 가정해 보겠습니다.

```
resource "aws_instance" "example" {
 ami = "ami-0fb653ca2d3203ac1"
 instance_type = "t2.micro"
}
```

그러고 다음은 terraform apply 를 실행한 후 terraform.tfstate 파일 내용의 작은 조각입니다.

```
{
 "version": 4,
 "terraform_version": "1.2.3",
 "serial": 1,
 "lineage": "86545604-7463-4aa5-e9e8-a2a221de98d2",
```

```
  "outputs": {},
  "resources": [
  {
   "mode": "managed",
   "type": "aws_instance",
   "name": "example",
   "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
   "instances": [
    {
     "schema_version": 1,
     "attributes": {
     "ami": "ami-0fb653ca2d3203ac1",
     "availability_zone": "us-east-2b",
     "id": "i-0bc4bbe5b84387543",
     "instance_state": "running",
     "instance_type": "t2.micro",
     "(...)": "(truncated)"
    }
   }
   ]
  }
  ]
}
```

이 JSON 형식을 사용하여 Terraform은 aws_instance 유형과 이름 example의 리소스가 ID가 i-0bc4bbe5b84387543 인 AWS 계정의 EC2 인스턴스에 해당한다는 것을 알고 있습니다. Terraform을 실행할 때마다 AWS에서 이 EC2 인스턴스의 최신 상태를 가져오고 이를 Terraform 구성의 내용과 비교하여 적용해야 할 변경 사항을 결정할 수 있습니다. 즉, plan 명령의 출력은 컴퓨터의 코드와 상태 파일의 ID를 통해 발견된 실제 AWS에 배포된 인프라 간의 차이입니다.

상태 파일은 비공개 API입니다.
상태 파일 형식은 Terraform 내에서 내부 용도로만 사용되는 비공개 API입니다. Terraform 상태 파일을 직접 편집하거나 이를 직접 읽는 코드를 작성해서는 안 됩니다. 어떤 이유로 상태 파일을 조작해야 하는 경우(비교적 드물게 발생함) terraform import 또는 terraform state 명령을 사용하십시오

개인 프로젝트에 Terraform을 사용하는 경우 컴퓨터에 로컬로 존재하는 단일 terraform.tfstate 파일에 상태를 저장하면 문제가 없습니다. 그러나 실제 제품에서 팀으로 Terraform을 사용하려는 경우 몇 가지 문제에 직면하게 됩니다.

- 상태 파일을 위한 공유 저장소
  Terraform을 사용하여 인프라를 업데이트하려면 각 팀 구성원이 동일한 Terraform 상태 파일에 액세스해야 합니다. 이는 해당 파일을 공유 위치에 저장해야 함을 의미합니다.
- 상태 파일 잠금(lock)
  데이터가 공유되자마자 잠금이라는 새로운 문제에 직면하게 됩니다. 잠금 없이 두 팀 구성원이 동시에 Terraform을 실행하는 경우 여러 Terraform 프로세스가 상태 파일을 동시에 업데이트하여 충돌, 데이터 손실 및 상태 파일 손상을 초래하므로 경합 상태가 발생할 수 있습니다.
- 상태 파일 격리
  인프라를 변경할 때는 다양한 환경을 격리하는 것이 모범 사례입니다. 예를 들어, test 또는 stage 환경을 변경할 때 실수로 production이 중단될 수 있는 점을 해결할 방법이 없는지 확인하고 싶을 것입니다. 하지만 모든 인프라가 동일한 Terraform 상태 파일에 정의되어 있는 경우 변경 사항을 격리 할 수 있을까요?

다음 섹션에서는 이러한 각 문제를 자세히 살펴보고 해결 방법을 보여 드리겠습니다.

상태 파일을 위한 공유 저장소

여러 팀 구성원이 공통 파일 세트에 액세스할 수 있도록 허용하는 가장 일반적인 기술은 해당 구성원을 버전 제어(예: Git)에 두는 것입니다. 버전 제어에 Terraform 코드를 저장해야 하지만 버전 제어에 Terraform 상태를 저장하는 것은 다음과 같은 이유로 좋지 않습니다.

- 수동 오류
  Terraform을 실행하기 전에 버전 제어에서 최신 변경 사항을 풀다운하거나 Terraform을 실행한 후 버전 제어에 최신 변경 사항을 푸시하는 것을 잊어버리기 쉽습니다. 팀의 누군가가 오래된 상태 파일로 Terraform을 실행하여 결과적으로 실수로 이전 배포를 롤백하거나 복제하는 것은 시간 문제입니다.
- 잠금
  대부분의 버전 제어 시스템은 두 팀 구성원이 동시에 동일한 상태 파일에 대해 Terraform Apply를 실행하는 것을 방지하는 잠금 형식을 제공하지 않습니다.
- 비밀(secret)
  Terraform 상태 파일의 모든 데이터는 일반 텍스트로 저장됩니다. 이는 특정 Terraform 리소스가 민감한 데이터를 저장해야 하기 때문에 문제가 됩니다. 예를 들어 aws_db_instance 리소스를 사용하여 데이터베이스를 생성하는 경우 Terraform은 데이터베이스의 사용자 이름과 비밀번호를 상태 파일에 일반 텍스트로 저장하므로 버전 제어에 일반 텍스트 암호를 저장하면 안 됩니다.

버전 제어를 사용하는 대신 상태 파일의 공유 저장소를 관리하는 가장 좋은 방법은 Terraform의 기본 기능인 원격 백엔드 지원을 사용하는 것입니다. Terraform 백엔드는 Terraform이 상태를 로드하고 저장하는 방법을 결정합니다. 지금까지 사용해 온 기본 백엔드는 로컬 디스크에 상태 파일을 저장하는 로컬 백엔드입니다. 원격 백엔드를 사용하면 상태 파일을 원격 공유 저장소에 저장할 수 있습니다. Amazon S3, Azure Storage, Google Cloud Storage, HashiCorp의 Terraform Cloud 및 Terraform Enterprise를 포함하여 다양한 원격 백엔드가 지원됩니다. 원격 백엔드는 방금 나열된 세 가지 문제를 해결합니다.

- 수동 오류
  원격 백엔드를 구성한 후 Terraform은 plan 또는 apply 를 실행할 때마다 해당 백엔드에서 상태 파일을 자동으로 로드하고 각 적용 후 해당 백엔드에 상태 파일을 자동으로 저장하므로 수동 오류가 발생할 가능성이 없습니다.
- 잠금
  대부분의 원격 백엔드는 기본적으로 잠금을 지원합니다. terraform apply 를 실행하기 위해 Terraform은 자동으로 잠금이 됩니다. 다른 사람이 이미 apply 를 실행하고 있다면 이미 잠금 되어 있기 때문에 기다려야 합니다. -lock-timeout=<TIME> 매개변수와 함께 적용을 실행하여 잠금이 해제될 때까지 최대 TIME까지 기다리도록 Terraform에 지시할 수 있습니다(예: -lock-timeout=10m은 10분 동안 대기합니다).
- 비밀
  대부분의 원격 백엔드는 기본적으로 전송 중 암호화와 나머지 상태 파일 암호화를 지원합니다. 또한 이러한 백엔드는 일반적으로 액세스 권한을 구성하는 방법(예: Amazon S3 버킷과 함께 IAM 정책 사용)을 제공하므로 상태 파일과 여기에 포함될 수 있는 비밀에 액세스할 수 있는 사람을 제어할 수 있습니다. Terraform이 기본적으로 상태 파일 내에서 암호화 비밀을 지원한다면 더 좋을 것입니다. 그리고 이러한 원격 백엔드는 최소한 상태 파일이 디스크의 어느 곳에도 일반 텍스트로 저장되지 않는다는 점을 고려하면 대부분의 보안 문제를 줄여줍니다.

AWS에서 Terraform을 사용하는 경우 Amazon의 관리형 파일 저장소인 Amazon S3(Simple Storage Service)는 일반적으로 다음과 같은 이유로 원격 백엔드로 가장 적합합니다.

- 관리형 서비스이므로 사용하기 위해 추가 인프라를 배포하고 관리할 필요가 없습니다.
- 99.999999999%의 내구성과 99.99%의 가용성을 제공하도록 설계되었으므로 데이터 손실이나 중단에 대해 너무 걱정할 필요가 없습니다.
- 암호화를 지원하므로 상태 파일에 민감한 데이터를 저장하는 것에 대한 걱정이 줄어듭니다. 데이터는 유휴 상태(AES-256을 사용하여 서버 측 암호화 지원) 및 전송 중(Terraform과 통신할 때 TLS를 사용함) 암호화됩니다. 하지만 팀원 중 S3 버킷에 액세스할 수 있는 사람이 누구인지 매우 주의해야 합니다.

- DynamoDB를 통한 잠금을 지원합니다.
- 버전 관리를 지원하므로 상태 파일의 모든 개정판이 저장되며 문제가 발생할 경우 이전 버전으로 롤백할 수 있습니다.
- 가격이 저렴하며 대부분의 Terraform 사용량이 AWS 프리 티어에 쉽게 들어맞습니다.

Amazon S3를 사용하여 원격 상태 스토리지를 활성화하려면 첫 번째 단계는 S3 버킷을 생성하는 것입니다. 이전에 실습한 폴더와 다른 새 폴더에 main.tf 파일을 생성하고 파일 상단에서 AWS를 공급자로 지정합니다.

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

다음으로 aws_s3_bucket 리소스를 사용하여 S3 버킷을 생성합니다.

```
resource "aws_s3_bucket" "terraform_state" {
 bucket = "terraform-state-alias"
 # Prevent accidental deletion of this S3 bucket
 lifecycle {
 prevent_destroy = true
 }
}
```

이 코드는 다음 인수를 설정합니다.

- 버킷(bucket)
  이는 S3 버킷의 이름입니다. S3 버킷 이름은 모든 AWS 고객 간에 전역적으로 고유해야 합니다. 따라서 버킷 매개변수를 "terraform-up-and-running-state"에서 사용자 이름으로 변경해야 합니다. 나중에 두 가지 정보가 모두 필요하므로 이 이름을 기억하고 사용 중인 AWS 리전을 기록해 두십시오.
- 삭제 방지(prevent_destroy)
  리소스에 대해 Prevent_destroy를 true로 설정하면 해당 리소스를 삭제하려고 하면(예: terraform destroy 실행) 오류와 함께 Terraform이 종료됩니다. 이는 모든 Terraform 상태를 저장하는 S3 버킷과 같은 중요한 리소스가 실수로 삭제되는 것을 방지하는 좋은 방법입니다.

이제 이 S3 버킷에 몇 가지 추가 보호 계층을 추가해 보겠습니다. 먼저 aws_s3_bucket_versioning 리소스를 사용하여 S3 버킷의 버전 관리를 활성화하면 버킷의 파일을 업데이트할 때마다 실제로 해당 파일의 새 버전이 생성됩니다. 이를 통해 언제든지 파일의 이전 버전을 확인하고 이전 버전으로 되돌릴 수 있습니다. 이는 문제가 발생할 경우 유용한 대체 메커니즘이 될 수 있습니다.

```
# Enable versioning so you can see the full revision history of your
# state files
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

둘째, aws_s3_bucket_server_side_encryption_configuration 리소스를 사용하여 이 S3 버킷에 기록된 모든 데이터에 대해 기본적으로 서버 측 암호화를 활성화합니다. 이렇게 하면 상태 파일과 여기에 포함될 수 있는 모든 비밀이 S3에 저장될 때 항상 디스크에서 암호화됩니다.

```
#Enable server-side encryption by default
resource "aws_s3_bucket_server_side_encryption_configuration" "default" {
```

```
  bucket = aws_s3_bucket.terraform_state.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
```

셋째, aws_s3_bucket_public_access_block 리소스를 사용하여 S3 버킷에 대한 모든 퍼블릭 액세스를 차단합니다. S3 버킷은 기본적으로 비공개이지만 정적 콘텐츠(예: 이미지, 글꼴, CSS, JS, HTML)를 제공하는 데 자주 사용되므로 버킷을 공개로 만드는 것이 가능하고 쉽습니다. Terraform 상태 파일에는 민감한 데이터와 비밀이 포함될 수 있으므로 팀 구성원 중 누구도 실수로 이 S3 버킷을 공개하지 않도록 보호 계층을 추가하는 것이 좋습니다.

```
#Explicitly block all public access to the S3 bucket
resource "aws_s3_bucket_public_access_block" "public_access" {
  bucket = aws_s3_bucket.terraform_state.id
  block_public_acls = true
  block_public_policy = true
  ignore_public_acls = true
  restrict_public_buckets = true
}
```

다음으로 잠금에 사용할 DynamoDB 테이블을 생성해야 합니다. DynamoDB는 Amazon의 분산 키-값 저장소입니다. 이는 분산 잠금 시스템에 필요한 모든 요소인 강력하게 일관된 읽기 및 조건부 쓰기를 지원합니다. 게다가 완벽하게 관리되므로 직접 실행할 인프라가 없으며, 대부분의 Terraform 사용량이 AWS 프리 티어에 쉽게 들어맞을 정도로 저렴합니다. Terraform을 통한 잠금에 DynamoDB를 사용하려면 LockID(정확한 철자와 대소문자 사용)라는 기본 키가 있는 DynamoDB 테이블을 생성해야 합니다. aws_dynamodb_table 리소스를 사용하여 이러한 테이블을 생성할 수 있습니다.

```
resource "aws_dynamodb_table" "terraform_locks" {
  name = "terraform-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

terraform init를 실행하여 공급자 코드를 다운로드한 다음 terraform Apply를 실행하여 배포합니다.

```
$ terraform init
$ terraform apply
```

모든 것이 배포되면 S3 버킷과 DynamoDB 테이블이 생성되지만 Terraform 상태는 여전히 로컬에 저장됩니다. S3 버킷에 상태를 저장하도록 Terraform을 구성하려면(암호화 및 잠금 포함) Terraform 코드에 백엔드 구성을 추가해야 합니다. 이는 Terraform 자체에 대한 구성이므로 Terraform 블록 내에 상주하며 다음 구문을 갖습니다.

```
terraform {
 backend "<BACKEND_NAME>" {
 [CONFIG...]
```

```
 }
}
```

여기서 BACKEND_NAME은 사용하려는 백엔드의 이름(예: "s3")이고 CONFIG는 해당 백엔드와 관련된 하나 이상의 인수(예: 사용할 S3 버킷의 이름)로 구성됩니다. S3 버킷의 백엔드 구성은 다음과 같습니다.

```
terraform {
 backend "s3" {
   bucket = "terraform-state"
   key = "global/s3/terraform.tfstate"
   region = "ap-northeast-2"
   dynamodb_table = "terraform-locks"
   encrypt = true
 }
}
```

이러한 설정을 한 번에 하나씩 살펴보겠습니다.

- 버킷
  사용할 S3 버킷의 이름입니다. 이를 이전에 생성한 S3 버킷의 이름으로 바꿔야 합니다.
- 키
  Terraform 상태 파일을 작성해야 하는 S3 버킷 내의 파일 경로입니다. 앞의 예제 코드가 이를 global/s3/terraform.tfstate로 설정한 이유는 잠시 후에 살펴보실 수 있습니다.
- 지역
  S3 버킷이 있는 AWS 리전입니다. 이를 이전에 생성한 S3 버킷의 리전으로 바꿔야 합니다.
- dynamodb_table
  잠금에 사용할 DynamoDB 테이블입니다. 이를 이전에 생성한 DynamoDB 테이블의 이름으로 바꿔야 합니다.
- 암호화
  이를 true로 설정하면 S3에 저장될 때 Terraform 상태가 디스크에서 암호화됩니다. 우리는 이미 S3 버킷 자체에서 기본 암호화를 활성화했으므로 이는 데이터가 항상 암호화되도록 보장하는 두 번째 계층입니다.

이 S3 버킷에 상태 파일을 저장하도록 Terraform에 지시하려면 terraform init 명령을 다시 사용합니다. 이 명령은 공급자 코드를 다운로드할 수 있을 뿐만 아니라 Terraform 백엔드를 구성할 수도 있습니다(나중에 또 다른 용도도 볼 수 있습니다). 게다가 init 명령은 멱등성이 있으므로 여러 번 실행해도 안전합니다.

```
$ terraform init
Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
 Pre-existing state was found while migrating the previous "local" backend
 to the newly configured "s3" backend. No existing state was found in the
 newly configured "s3" backend. Do you want to copy this state to the new
 "s3" backend? Enter "yes" to copy and "no" to start with an empty state.
 Enter a value:
```

Terraform은 이미 로컬에 상태 파일이 있음을 자동으로 감지하고 이를 새 S3 백엔드로 복사하라는 메시지를 표시합니다. yes 를 입력하면 다음이 표시됩니다.

```
Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
```

이 명령을 실행하면 Terraform 상태가 S3 버킷에 저장됩니다. 브라우저에서 S3 관리 콘솔로 이동하여 버킷을 클릭하면 이를 확인할 수 있습니다. 그림과 비슷한 내용이 표시되어야 합니다.

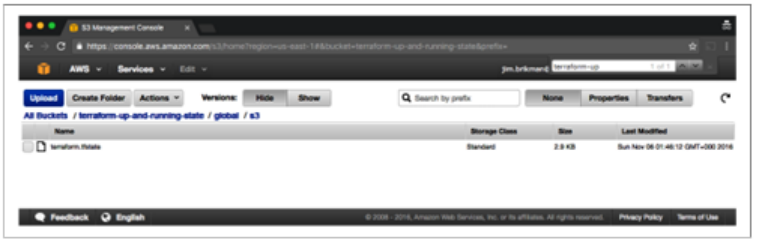


Figure 3-1. You can use the AWS Console to see how your state file is stored in an S3 bucket.

이 백엔드가 활성화되면 Terraform은 명령을 실행하기 전에 이 S3 버킷에서 최신 상태를 자동으로 가져오고 명령을 실행한 후에 최신 상태를 S3 버킷에 자동으로 푸시합니다. 실제로 이를 확인하려면 다음 출력 변수를 추가하세요.

```
output "s3_bucket_arn" {
 value = aws_s3_bucket.terraform_state.arn
 description = "The ARN of the S3 bucket"
}
output "dynamodb_table_name" {
 value = aws_dynamodb_table.terraform_locks.name
 description = "The name of the DynamoDB table"
}
```

이러한 변수는 S3 버킷의 Amazon 리소스 이름(ARN)과 DynamoDB 테이블의 이름을 출력합니다. Terraform Apply를 실행하여 확인하세요.

```
$ terraform apply
(...)
Acquiring state lock. This may take a few moments...
aws_dynamodb_table.terraform_locks: Refreshing state...
aws_s3_bucket.terraform_state: Refreshing state...
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...
Outputs:
dynamodb_table_name = "terraform-locks"
s3_bucket_arn = "arn:aws:s3:::terraform-up-and-running-state"
```

이제 Terraform이 적용을 실행하기 전에 잠금을 획득하고 적용 후 잠금을 해제하는 방법에 주목하세요! 이제 다시 S3 콘솔로 가서 페이지를 새로 고치고 버전 옆에 있는 회색 표시 버튼을 클릭하세요. 이제 그림과 같이 S3 버킷에 terraform.tfstate 파일의 여러 버전이 표시됩니다.

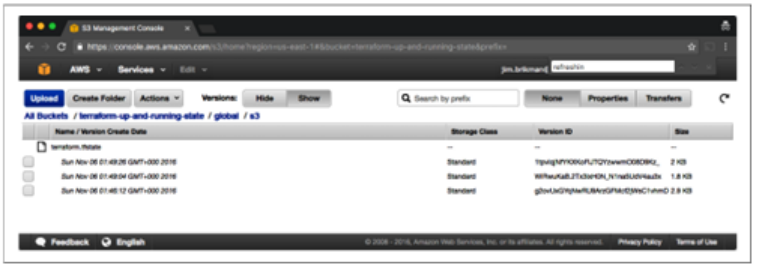


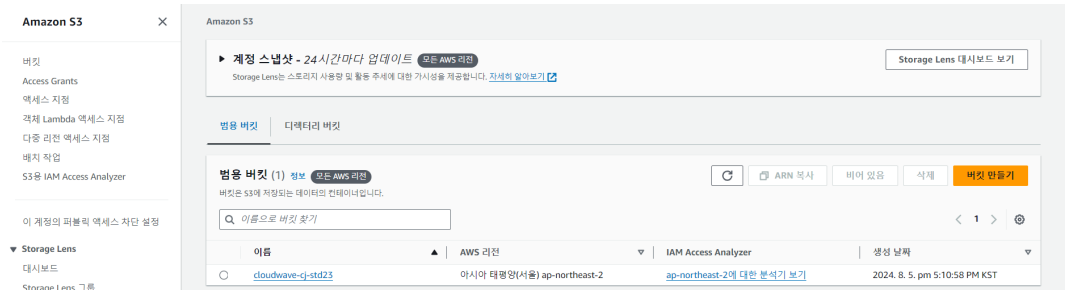
Figure 3-2. If you enable versioning for your S3 bucket, every change to the state file will be stored as a separate version.

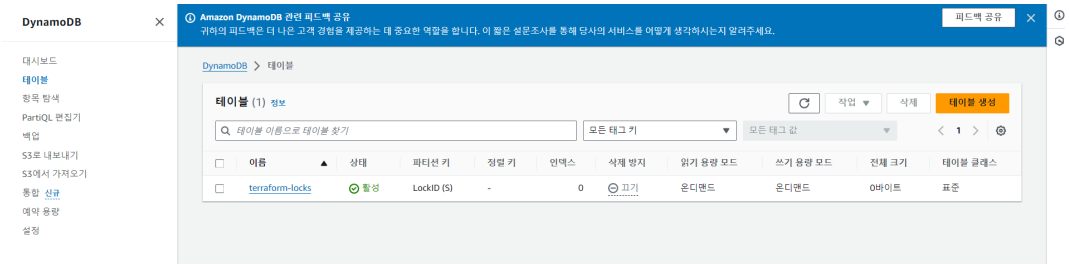
즉, Terraform은 상태 데이터를 S3로 자동으로 밀고 당기고 S3는 상태 파일의 모든 개정판을 저장하므로 문제가 발생할 경우 디버깅하고 이전 버전으로 롤백하는 데 유용할 수 있습니다.

결과

```
[root@controller terraform-demo2]# terraform init
Initializing the backend...
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.
```

```
Plan: 5 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_s3_bucket.terraform_state: Creating...
aws_dynamodb_table.terraform_locks: Creating...
aws_s3_bucket.terraform_state: Creation complete after 1s [id=cloudwave-cj-std23]
aws_s3_bucket_versioning.enabled: Creating...
aws_s3_bucket_server_side_encryption_configuration.default: Creating...
aws_s3_bucket_public_access_block.public_access: Creating...
aws_s3_bucket_server_side_encryption_configuration.default: Creation complete after 1s [id=cloudwave-cj-std23]
aws_s3_bucket_public_access_block.public_access: Creation complete after 1s [id=cloudwave-cj-std23]
aws_s3_bucket_versioning.enabled: Creation complete after 2s [id=cloudwave-cj-std23]
aws_dynamodb_table.terraform_locks: Creation complete after 7s [id=terraform-locks]

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

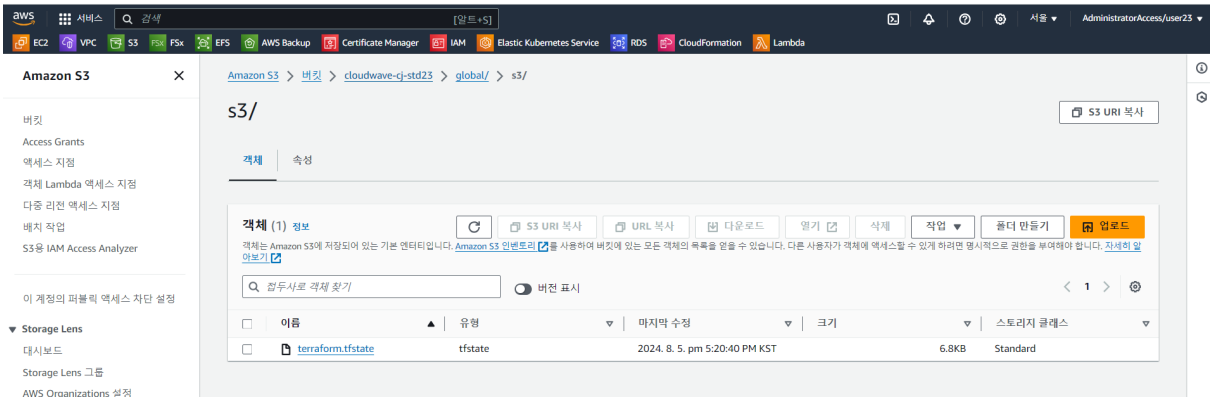





```
terraform {
 backend "s3" {
   bucket = "cloudwave-cj-std23"
   key = "global/s3/terraform.tfstate"
   region = "ap-northeast-2"
   dynamodb_table = "terraform-locks"
   encrypt = true
 }
}

provider "aws" {
 region = "ap-northeast-2"
}


resource "aws_s3_bucket" "terraform_state" {
 bucket = "cloudwave-cj-std23"
 # Prevent accidental deletion of this S3 bucket
 lifecycle {
 prevent_destroy = true
 }
}


# Enable versioning so you can see the full revision history of your
# state files
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
```

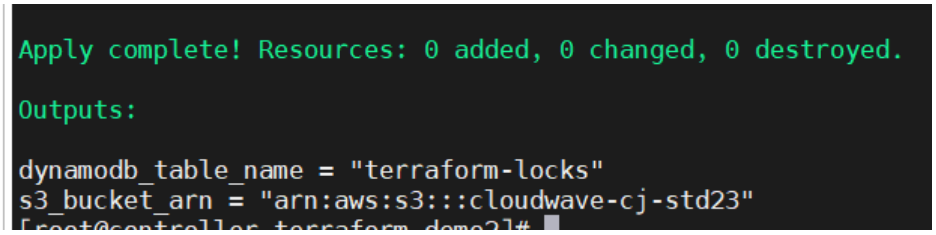
```
    status = "Enabled"
  }
}

#Enable server-side encryption by default
resource "aws_s3_bucket_server_side_encryption_configuration" "default" {
  bucket = aws_s3_bucket.terraform_state.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}

#Explicitly block all public access to the S3 bucket
resource "aws_s3_bucket_public_access_block" "public_access" {
  bucket = aws_s3_bucket.terraform_state.id
  block_public_acls = true
  block_public_policy = true
  ignore_public_acls = true
  restrict_public_buckets = true
}

resource "aws_dynamodb_table" "terraform_locks" {
  name = "terraform-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

```
[root@controller terraform-demo2]# terraform init
Initializing the backend...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "local" backend to the
  newly configured "s3" backend. No existing state was found in the newly
  configured "s3" backend. Do you want to copy this state to the new "s3"
  backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value: yes


Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v5.61.0

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.
```




```
output "s3_bucket_arn" {
 value = aws_s3_bucket.terraform_state.arn
 description = "The ARN of the S3 bucket"
}
output "dynamodb_table_name" {
 value = aws_dynamodb_table.terraform_locks.name
 description = "The name of the DynamoDB table"
}
```

코드 추가 후 재 apply



테라폼 제한 사항

Terraform의 백엔드에는 알아야 할 몇 가지 제한 사항과 문제점이 있습니다. 첫 번째 제한 사항은 Terraform을 사용하여 Terraform 상태를 저장할 S3 버킷을 생성할 때 닭과 달걀이 결정되는 상황입니다. 이 작업을 수행하려면 다음 두 단계 프로세스를 사용해야 했습니다.

1. Terraform 코드를 작성하여 S3 버킷과 DynamoDB 테이블을 생성하고 해당 코드를 로컬 백엔드에 배포합니다.
2. Terraform 코드로 돌아가서 새로 생성된 S3 버킷 및 DynamoDB 테이블을 사용하도록 원격 백엔드 구성을 추가하고 terraform init를 실행하여 로컬 상태를 S3에 복사합니다.

S3 버킷과 DynamoDB 테이블을 삭제하려면 이 2단계 프로세스를 반대로 수행해야 합니다.

1. Terraform 코드로 이동하여 백엔드 구성을 제거하고 terraform init를 다시 실행하여 Terraform 상태를 로컬 디스크에 다시 복사합니다.
2. terraform destroy를 실행하여 S3 버킷과 DynamoDB 테이블을 삭제합니다.

이 2단계 프로세스는 약간 어색하지만 대신 모든 Terraform 코드에서 단일 S3 버킷과 DynamoDB 테이블을 공유할 수 있으므로 이 작업을 한 번(또는 계정이 여러 개인 경우 AWS 계정당 한 번)만 수행하면 됩니다. S3 버킷이 존재하면 나머지 Terraform 코드에서 추가 단계 없이 처음부터 바로 백엔드 구성을 지정할 수 있습니다.

두 번째 제한은 더 복잡합니다. Terraform의 **백엔드 블록에서는 변수나 참조를 사용할 수 없습니다.** 다음 코드는 작동하지 않습니다.

```
# This will NOT work. Variables aren't allowed in a backend configuration.
terraform {
 backend "s3" {
 bucket = var.bucket
 region = var.region
 dynamodb_table = var.dynamodb_table
 key = "example/terraform.tfstate"
 encrypt = true
 }
}
```

즉, S3 버킷 이름, 지역, DynamoDB 테이블 이름 등을 모든 Terraform 모듈에 수동으로 복사하여 붙여넣어야 합니다. 모듈은 Terraform 코드를 구성하고 재사용하는 방법이며 실제 Terraform 코드는 일반적으로 많은 작은 모듈로 구성됩니다.

키 값을 매우 조심스럽게 복사하여 붙여넣지 말고 배포하는 모든 Terraform 모듈에 대해 고유한 키를 보장하여 실수로 다른 모듈의 상태를 덮어쓰지 않도록 해야 합니다! 많은 복사 및 붙여넣기와 수동 변경을 수행해야 하면 오류가 발생하기 쉽습니다. 특히 여러 환경에서 많은 Terraform 모듈을 배포하고 관리해야 하는 경우 더욱 그렇습니다.

복사하여 붙여넣기를 줄이는 한 가지 옵션은 부분 구성을 사용하는 것입니다. 여기서는 Terraform 코드의 백엔드 구성에서 특정 매개변수를 생략하고 대신 terraform init를 호출할 때 -backend-config 명령줄 인수를 통해 해당 매개변수를 전달합니다. 예를 들어, bucket 및 지역과 같은 반복되는 백엔드 인수를 backend.hcl이라는 별도의 파일로 추출할 수 있습니다.

```
# backend.hcl
bucket = "terraform-state-ID"
region = "us-east-2"
dynamodb_table = "terraform-locks"
encrypt = true
```

각 모듈마다 다른 키 값을 설정해야 하므로 Terraform 코드에는 키 매개변수만 남습니다.

```
# Partial configuration. The other settings (e.g., bucket, region) will be
# passed in from a file via -backend-config arguments to 'terraform init'
terraform {
 backend "s3" {
 key = "example/terraform.tfstate"
 }
}
```

모든 부분 구성을 함께 사용하려면 -backend-config 인수를 사용하여 terraform init를 실행합니다.

```
$ terraform init -backend-config=backend.hcl
```

Terraform은 backend.hcl의 부분 구성을 Terraform 코드의 부분 구성과 병합하여 모듈에서 사용되는 전체 구성을 생성합니다. 모든 모듈에 동일한 backend.hcl 파일을 사용할 수 있으므로 중복이 상당히 줄어듭니다. 그러나 여전히 모든 모듈에서 고유 키 값을 수동으로 설정해야 합니다.

복사하여 붙여넣기를 줄이는 또 다른 옵션은 Terraform의 몇 가지 공백을 메우려는 오픈 소스 도구인 ***Terragrunt***를 사용하는 것입니다. Terragrunt는 모든 기본 백엔드 설정(버킷 이름, 리전, DynamoDB 테이블 이름)을 하나의 파일에 정의하고 키 인수를 해당 파일의 상대 폴더 경로로 자동 설정하여 전체 백엔드 구성을 DRY(Don't Repeat Yourself - 반복하지 마세요)로 유지하는 데 도움이 됩니다.

상태 파일 환경 격리

원격 백엔드 및 잠금 기능을 사용하면 협업이 더 이상 문제가 되지 않습니다. 그러나 아직 한 가지 문제가 더 남아 있습니다. 바로 격리입니다. Terraform을 처음 사용하기 시작하면 단일 Terraform 파일 또는 하나의 폴더에 있는 단일 Terraform 파일 세트에 모든 인프라를 정의하고 싶은 유혹을 느낄 수 있습니다. 이 접근 방식의 문제점은 이제 모든 Terraform 상태가 단일 파일에 저장되며 어디에서든 실수가 발생하면 모든 것이 중단될 수 있다는 것입니다. 예를 들어, 스테이징 단계에서 앱의 새 버전을 배포하려고 시도하는 동안 프로덕션 단계에서 앱이 중단될 수 있습니다. 또는 더 나쁜 것은 잠금을 사용하지 않았거나 드문 Terraform 버그로 인해 전체 상태 파일이 손상되어 이제 모든 환경의 모든 인프라가 손상될 수 있다는 것입니다.

별도의 환경을 갖는 요점은 서로 격리되어 있다는 것입니다. 따라서 단일 Terraform 구성 세트에서 모든 환경을 관리하는 경우 해당 격리가 깨집니다. 선박에 선박의 한 부분에서 누출이 발생하여 다른 모든 부분이 즉시 침수되는 것을 방지하는 장벽 역할을 하는 격벽이 있는 것처럼, Terraform 설계에 다음과 같이 "격벽"이 내장되어 있어야 합니다.

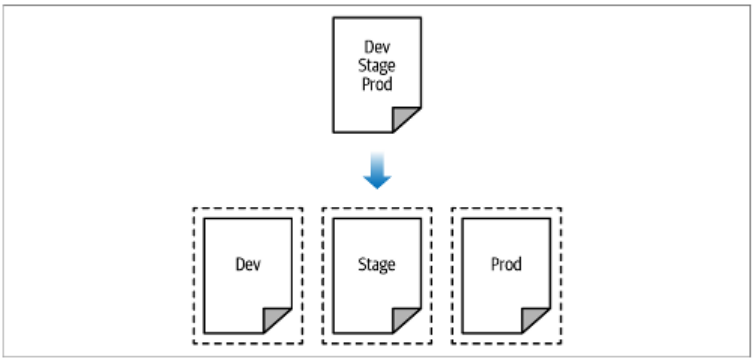


Figure 3-3. Create isolation ("bulkheads") between your environments by defining each environment in a separate Terraform configuration.

위 그림 에서 볼 수 있듯이 단일 Terraform 구성 세트(상단)에서 모든 환경을 정의하는 대신 별도의 구성 세트(하단)에서 각 환경을 정의하여 한 환경의 문제가 완전히 격리되도록 하려고 합니다. 다른 사람. 상태 파일을 격리하는 방법에는 두 가지가 있습니다.

- 작업 공간을 통한 격리
  동일한 구성에 대한 빠르고 격리된 테스트에 유용합니다.
- 파일 레이아웃을 통한 격리
  환경 간의 강력한 분리가 필요한 프로덕션 사용 사례에 유용합니다.

다음 두 섹션에서 이들 각각에 대해 자세히 살펴보겠습니다.

작업공간(Workspace)을 통한 격리

Terraform 작업공간을 사용하면 Terraform 상태를 별도의 이름이 지정된 여러 작업공간에 저장할 수 있습니다. Terraform은 "default"이라는 단일 작업 공간으로 시작하며, 작업 공간을 명시적으로 지정하지 않으면 기본 작업 공간이 전체 시간 동안 사용됩니다. 새 작업공간을 만들거나 작업공간 간에 전환하려면 terraform 작업공간 명령을 사용합니다. 단일 EC2 인스턴스를 배포하는 일부 Terraform 코드에서 작업공간을 실험해 보겠습니다.

```
resource "aws_instance" "example" {
 ami = "ami-0fb653ca2d3203ac1"
 instance_type = "t2.micro"
}
```

이 장의 앞부분에서 생성한 S3 버킷과 DynamoDB 테이블을 사용하고 키는 Workspaces-example/terraform.tfstate로 설정하여 이 인스턴스에 대한 백엔드를 구성합니다.

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

terraform init 및 terraform Apply를 실행하여 이 코드를 배포합니다.

```
$ terraform init
Initializing the backend...
Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
Initializing provider plugins...
(...)
Terraform has been successfully initialized!
$ terraform apply
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

이 배포의 상태는 기본 작업 영역에 저장됩니다. 현재 어떤 작업공간에 있는지 식별하는 terraform 작업공간 표시 명령을 실행하여 이를 확인할 수 있습니다.

```
$ terraform workspace show
default
```

기본 작업공간은 키 구성을 통해 지정한 위치에 정확하게 상태를 저장합니다. 그림과 같이 S3 버킷을 살펴보면 작업 공간-예제 폴더에 terraform.tfstate 파일이 있습니다.

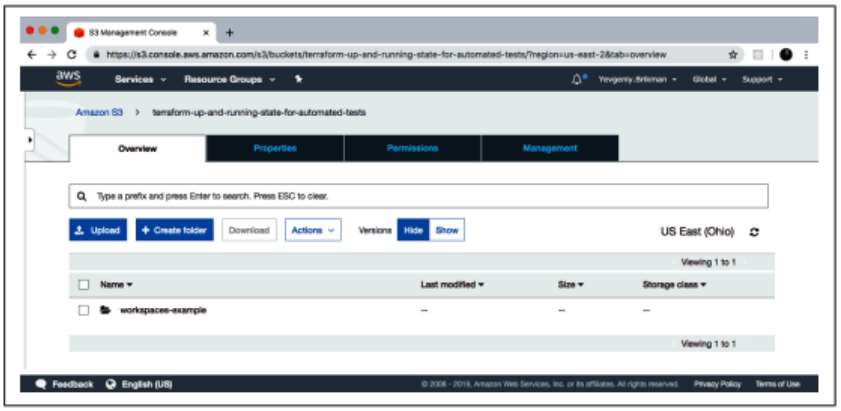


Figure 3-4. When using the default workspace, the S3 bucket will have just a single folder and state file in it.

terraform Workspace new 명령을 사용하여 "example1"이라는 새 작업 공간을 만들어 보겠습니다.

```
$ terraform workspace new example1
Created and switched to workspace "example1"!
You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

이제 terraform plan을 실행하려고 하면 어떤 일이 발생하는지 살펴보세요.

```
$ terraform plan
Terraform will perform the following actions:
 # aws_instance.example will be created
 + resource "aws_instance" "example" {
 + ami = "ami-0fb653ca2d3203ac1"
 + instance_type = "t2.micro"
 (...)
```

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

Terraform은 완전히 새로운 EC2 인스턴스를 처음부터 생성하려고 합니다! 이는 각 작업공간의 상태 파일이 서로 격리되어 있고 현재 example1 작업공간에 있기 때문에 Terraform은 기본 작업공간의 상태 파일을 사용하지 않으므로 해당 위치에 이미 생성된 EC2 인스턴스가 표시되지 않습니다. . terraform Apply를 실행하여 새 작업 공간에 두 번째 EC2 인스턴스를 배포해 보세요.

```
$ terraform apply
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

한 번 더 연습을 반복하고 "example2"라는 또 다른 작업 공간을 만듭니다.

```
$ terraform workspace new example2
Created and switched to workspace "example2"!
You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Terraform Apply를 다시 실행하여 세 번째 EC2 인스턴스를 배포합니다.

```
$ terraform apply
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

이제 terraform 작업공간 목록 명령을 사용하여 볼 수 있는 세 가지 작업공간을 사용할 수 있습니다.

```
$ terraform workspace list
 default
 example1
* example2
```

그리고 Terraform 작업 공간 선택 명령을 사용하여 언제든지 둘 사이를 전환할 수 있습니다.

```
$ terraform workspace select example1
Switched to workspace "example1".
```

이것이 내부적으로 어떻게 작동하는지 이해하려면 S3 버킷을 다시 살펴보세요. 이제 그림과 같이 env: 라는 새 폴더가 표시됩니다.

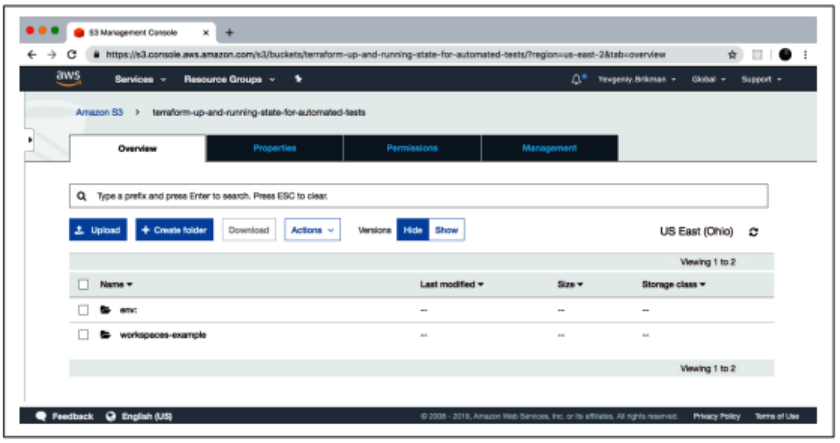


Figure 3-5. When using custom workspaces, the S3 bucket will have multiple folders and state files in it.

env: 폴더 내에는 그림과 같이 각 작업 공간에 대해 하나의 폴더가 있습니다.

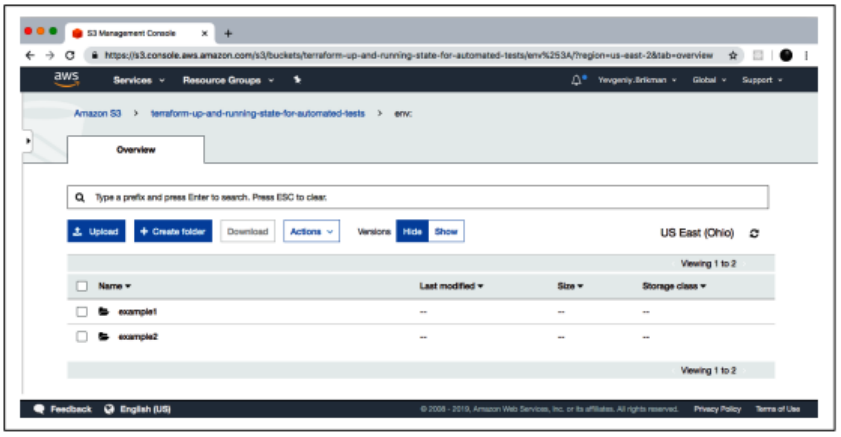


Figure 3-6. Terraform creates one folder per workspace.

각 작업공간 내에서 Terraform은 백엔드 구성에 지정된 키를 사용하므로 example1/workspaces-example/terraform.tfstate 및 example2/workspaces-example/terraform.tfstate 를 찾아야 합니다. 즉, 다른 작업 공간으로 전환하는 것은 상태 파일이 저장된 경로를 변경하는 것과 같습니다.
이는 Terraform 모듈이 이미 배포되어 있고 이를 사용하여 몇 가지 실험을 수행하고 싶지만(예: 코드 리팩터링 시도) 실험이 이미 배포된 인프라 상태에 영향을 미치는 것을 원하지 않을 때 유용합니다.

Terraform 작업공간을 사용하면 정확히 동일한 인프라의 새 복사본을 배포할 수 있지만 상태는 별도의 파일에 저장할 수 있습니다. 실제로 terraform.workspace 표현식을 사용하여 작업 공간 이름을 읽으면 현재 작업 공간에 따라 해당 모듈이 작동하는 방식을 변경할 수도 있습니다. 예를 들어 기본 작업 공간에서는 인스턴스 유형을 t2.medium으로 설정하고 다른 모든 작업 공간에서는 t2.micro로 설정하는 방법은 다음과 같습니다(예: 실험 시 비용을 절약하기 위해).

```
resource "aws_instance" "example" {
 ami = "ami-0fb653ca2d3203ac1"
 instance_type = 9393 ? "t2.medium" : "t2.micro"
}
```

앞의 코드는 삼항 구문을 사용하여 terraform.workspace 값에 따라 조건부로 인스턴스 유형을 t2.medium 또는 t2.micro 로 설정합니다.

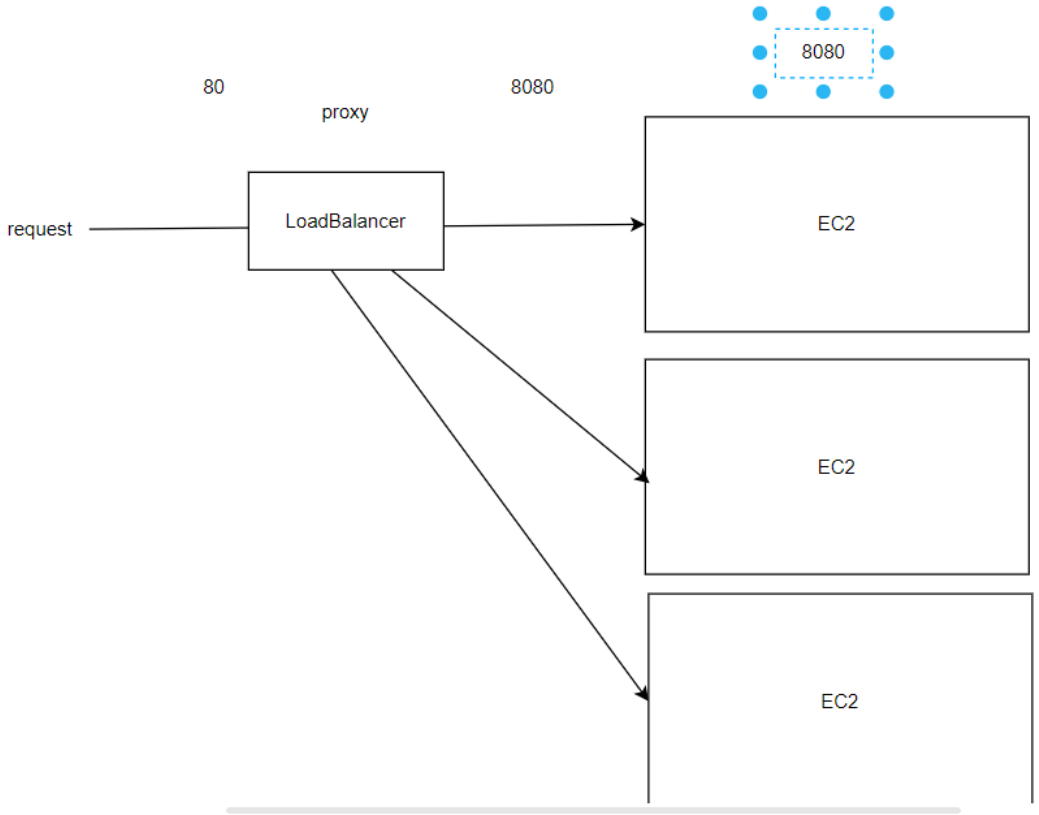
Terraform 작업공간은 다양한 버전의 코드를 신속하게 가동하고 해체할 수 있는 좋은 방법이 될 수 있지만 몇 가지 단점이 있습니다.

- 모든 작업 영역의 상태 파일은 동일한 백엔드(예: 동일한 S3 버킷)에 저장됩니다. 즉, 모든 작업 영역에 대해 동일한 인증 및 액세스 제어를 사용한다는 의미이며, 이는 작업 영역이 환경을 격리하는 데 부적합한 메커니즘(예: 프로덕션에서 스테이징 격리)에 적합하지 않은 주요 이유 중 하나입니다.
- terraform 작업 공간 명령을 실행하지 않는 한 작업 공간은 코드나 터미널에 표시되지 않습니다. 코드를 찾아보면 하나의 작업 공간에 배포된 모듈은 10개의 작업 공간에 배포된 모듈과 완전히 똑같아 보입니다. 인프라에 대한 좋은 그림이 없기 때문에 유지 관리가 더 어려워집니다.
- 앞의 두 항목을 합치면 작업 공간에서 오류가 발생할 가능성이 상당히 높습니다. 가시성이 부족하면 현재 작업공간이 무엇인지 잊어버리고 실수로 잘못된 작업공간에 변경사항을 배포하기 쉽습니다(예: "스테이징" 작업공간이 아닌 "프로덕션" 작업공간에서 실수로 terraform destroy를 실행함). 모든 작업 공간에 동일한 인증 메커니즘이 적용되므로 이러한 오류로부터 보호할 수 있는 다른 방어 계층이 없습니다.

이러한 단점으로 인해 작업 공간은 프로덕션과 스테이징을 격리하는 것과 같이 한 환경을 다른 환경과 격리하는 데 적합한 메커니즘이 아닙니다. 환경 간 적절한 격리를 위해서는 작업 공간 대신 다음 섹션의 주제인 파일 레이아웃을 사용하는 것이 가장 좋습니다. 계속 진행하기 전에 3개의 작업공간 각각에서 terraform Workspace select <name> 및 terraform destroy를 실행하여 방금 배포한 3개의 EC2 인스턴스를 정리해야 합니다.

필기

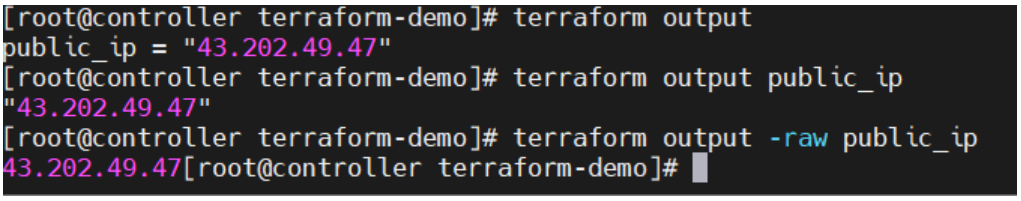
- load balancer 구조



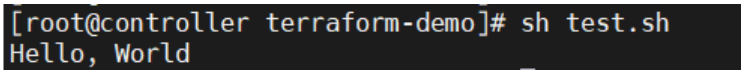
💡 반복적인 코드의 사용은 유지보수를 힘들게 한다. 변수화를 해야 한다.

- terraform output

state를 저장하였다가 output을 출력 가능하다.




```
#!/bin/bash

IP=$(terraform output -raw public_ip)
curl $IP:8080
```



다음과 같이 스크립트 파일로 만들어서 IP를 따로 보유하거나 즉시 활용 가능하다.

- 단일실패지점 SPOF(Single Point Of Failure)

이를 지양해야 한다. 포인트 지점을 여러 개 만들어서 하나가 실패할 경우 옮겨갈 수 있게 해야 한다.

- create_before_destroy = true

auto scaling group에서는 거의 필수적으로 들어가는 옵션

- CLB / NLB / ALB

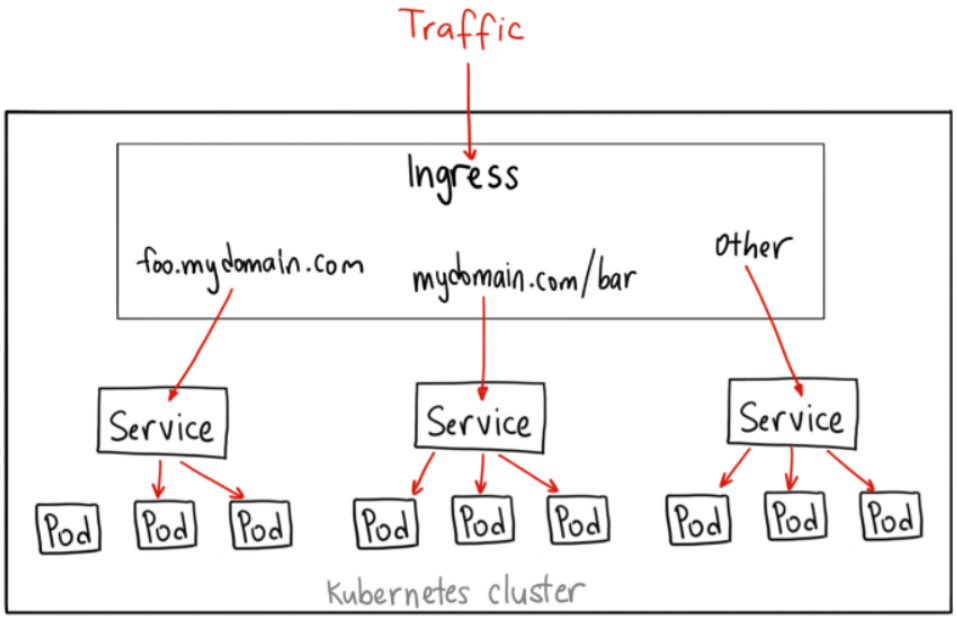
면접에서 NLB와 ALB의 차이점을 묻기도 한다.

ALB는 특정한 HOST, 도메인이 왔을 때 캐스팅해서 주기도한다. 7계층

PATH기반은 DOMAIN.COM/INDEX
DOMAIN 기반 INDEX.DOMAIN.COM

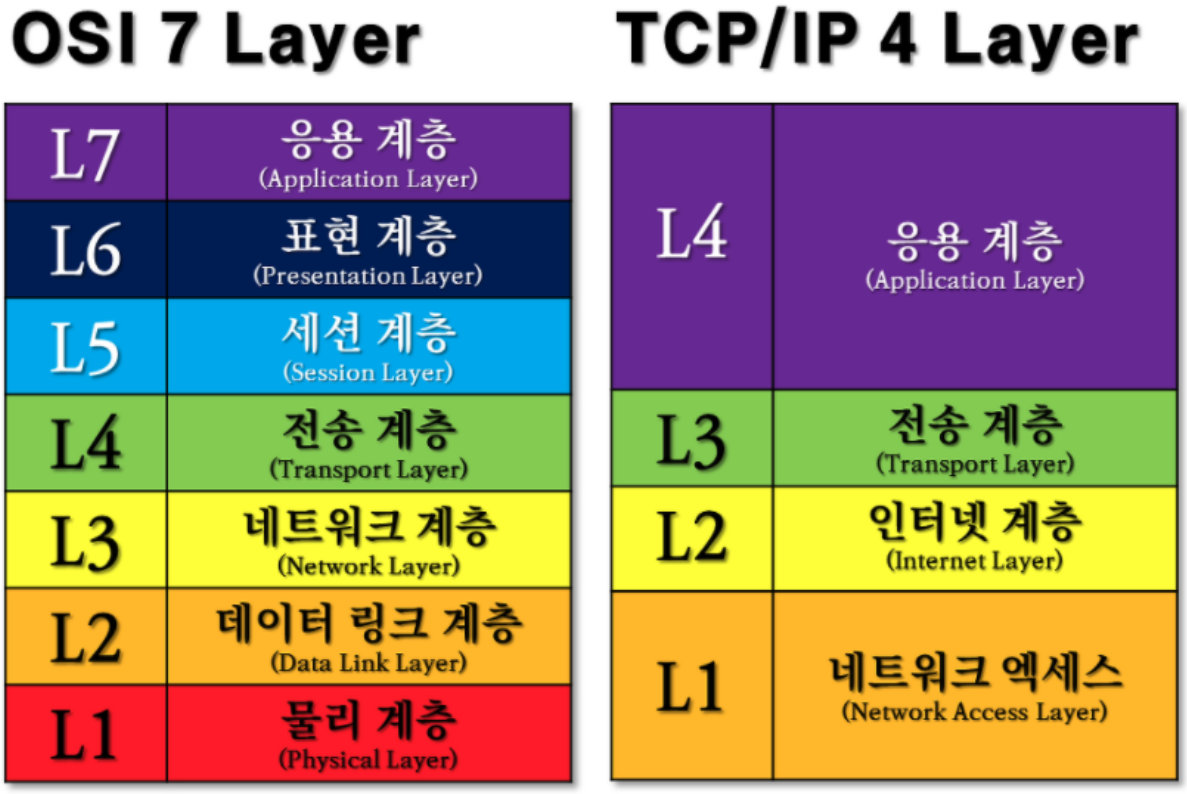
NLB는 특정한 IP가 왔을 때 매핑 4계층

비슷하게 SERVICE와 INGRESS의 차이점을 묻는데
SERVICE는 4계층이고 INGRESS는 7계층이다.



인그레스-서비스 네트워크 구조
출처: https://medium.com/google-cloud

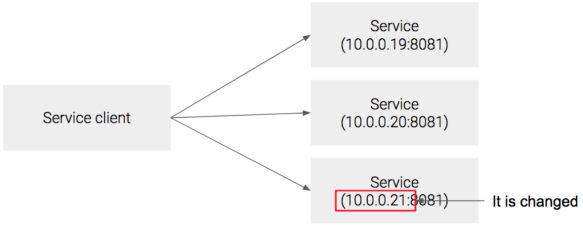
최근 IT에는 L7기능을 중요시 여기고 있다. 속도는 느릴 지라도 도메인 기반으로 접근했을 때
뒷단에 있는 서비스 여러 개로 트래픽을 분산 시킬 수 있기 때문 L4는 IP 기반으로 PORT로 접근



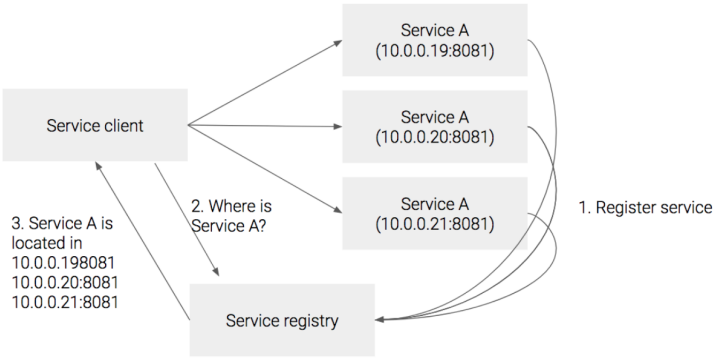
- MSA(micro service architecture)

service discovery

MSA와 같은 분산 환경은 서비스 간의 원격 호출로 구성이 된다. 원격 서비스 호출은 IP 주소와 포트를 이용하는 방식이 된다. 클라우드 환경이 되면서 서비스가 오토 스케일링 등에 의해서 동적으로 생성되거나 컨테이너 기반의 배포로 인해서, 서비스의 IP가 동적으로 변경되는 일이 잦아졌다.그래서 서비스 클라이언트가 서비스를 호출할 때 서비스의 위치 (즉 IP주소와 포트)를 알아낼 수 있는 기능이 필요한데, 이것을 바로 서비스 디스커버리 (Service discovery)라고 한다.

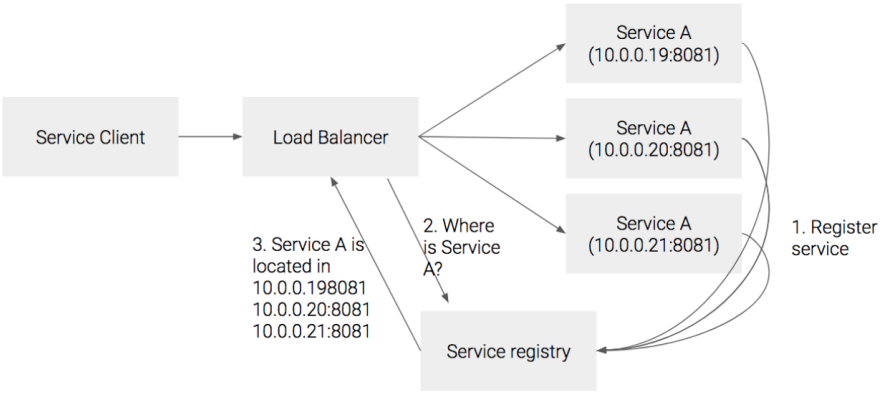


다음 그림을 보자 Service A의 인스턴스들이 생성이 될때, Service A에 대한 주소를 Service registry (서비스 등록 서버) 에 등록해놓는다. Service A를 호출하고자 하는 클라이언트는 Service registry에 Service A의 주소를 물어보고 등록된 주소를 받아서 그 주소로 서비스를 호출한다.



Client side discovery vs server side discovery

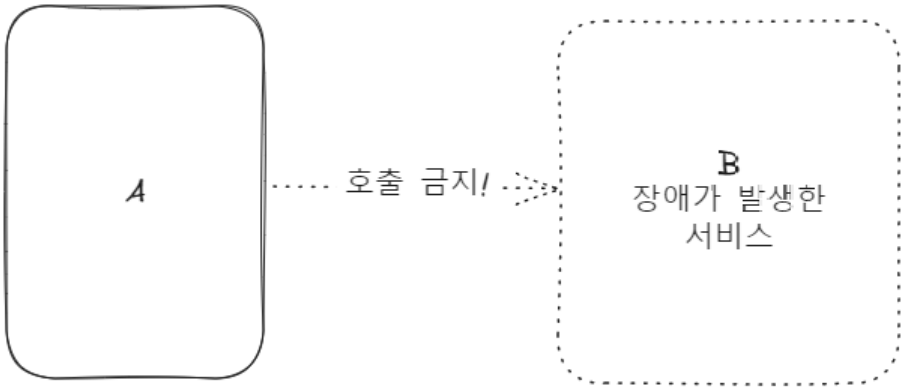
이러한 Service discovery 기능을 구현하는 방법으로는 크게 client discovery 방식과 server side discovery 방식이 있다.앞에서 설명한 service client가 service registry에서 서비스의 위치를 찾아서 호출 하는 방식을 client side discovery 라고 한다.다른 접근 방법으로는 호출이 되는 서비스 앞에 일종의 proxy 서버 (로드밸런서)를 넣는 방식인데, 서비스 클라이언트는 이 로드밸런서를 호출하면 로드밸런서가 Service registry로 부터 등록된 서비스의 위치를 리턴하고, 이를 기반으로 라우팅을 하는 방식이다.가장 흔한 예제로는 클라우드에서 사용하는 로드밸런서를 생각하면 된다. AWS의 ELB나 구글 클라우드의 로드 밸런서가 대표적인 Server side discovery 방식에 해당 한다.



circuit breaker

Circuit Breaker 패턴은 장애가 발생했을 때 장애가 전파되는 것을 막기 위한 패턴입니다.

원래라면, 서비스 A가 서비스 B를 호출하고, 서비스 B가 장애가 발생하면 서비스 A도 장애가 발생합니다.



Circuit Breaker 패턴을 적용하면, 서비스 A가 서비스 B를 호출할 때 더 이상 호출하지 않도록 차단합니다.

차단하는 것을 통해서 서비스 A는 정상적으로 작동할 수 있도록 하는 목적을 가지고 있습니다.

- acl (access control list)

- 작업공간

현업에서는 디렉토리를 이용해서 분리하지 작업공간(work space)를 분리해서 나누지는 않는다

복잡하기 때문

파일 레이아웃을 통한 격리

환경을 완전히 격리하려면 다음을 수행해야 합니다.

- 각 환경의 Terraform 구성 파일을 별도의 폴더에 넣습니다. 예를 들어, 스테이징 환경에 대한 모든 구성은 stage라는 폴더에 있을 수 있고 프로덕션 환경에 대한 모든 구성은 prod라는 폴더에 있을 수 있습니다.
- 서로 다른 인증 메커니즘과 액세스 제어를 사용하여 각 환경에 대해 서로 다른 백엔드를 구성합니다. 예를 들어 각 환경은 별도의 S3 버킷을 백엔드로 사용하는 별도의 AWS 계정에 있을 수 있습니다.

이 접근 방식을 사용하면 별도의 폴더를 사용하면 배포할 환경이 훨씬 더 명확해지고, 별도의 인증 메커니즘과 함께 별도의 상태 파일을 사용하면 한 환경에서 문제가 발생할 가능성이 훨씬 줄어듭니다.

실제로 격리 개념을 환경을 넘어 "구성 요소" 수준까지 가져갈 수 있습니다. 여기서 구성 요소는 일반적으로 함께 배포하는 일관된 리소스 집합입니다. 예를 들어, AWS 용어, Virtual Private Cloud(VPC) 및 모든 관련 서브넷, 라우팅 규칙, VPN 및 네트워크 ACL 등 인프라에 대한 기본 네트워크 토폴로지를 설정한 후에는 한 번만 변경하게 됩니다. 많아야 몇 달에 한 번씩. 반면에 새 버전의 웹 서버를 하루에 여러 번 배포할 수도 있습니다. 동일한 Terraform 구성 세트에서 VPC 구성 요소와 웹 서버 구성 요소 모두에 대한 인프라를 관리하는 경우 불필요하게 전체 네트워크 토폴로지가 손상될 위험에 놓이게 됩니다(예: 코드의 단순한 오타 또는 누군가 실수로 잘못된 명령)

따라서 각 환경(스테이징, 프로덕션 등)과 해당 환경 내의 각 구성 요소(VPC, 서비스, 데이터베이스)에 대해 별도의 Terraform 폴더(따라서 별도의 상태 파일)를 사용하는 것이 좋습니다. 이것이 실제로 어떻게 보이는지 확인하기 위해

Terraform 프로젝트에 권장되는 파일 레이아웃을 살펴보겠습니다.

Figure 3-7. The typical file layout for a Terraform project uses separate folders for each environment and for each component within that environment.

최상위 수준에는 각 "환경"에 대한 별도의 폴더가 있습니다. 정확한 환경은 프로젝트마다 다르지만 일반적인 환경은 다음과 같습니다.

- stage
  사전 프로덕션 워크로드를 위한 환경(예: 테스트)
- prod
  프로덕션 워크로드를 위한 환경(예: 사용자 대상 앱)
- mgmt
  DevOps 도구 환경(예: 배스천 호스트, CI 서버)
- global
  모든 환경(예: S3, IAM)에서 사용되는 리소스를 보관하는 장소

각 환경 내에는 각 "구성 요소"에 대한 별도의 폴더가 있습니다. 구성 요소는 프로젝트마다 다르지만 일반적인 구성 요소는 다음과 같습니다.

- vpc
  이 환경의 네트워크 토폴로지입니다.
- services
  Ruby on Rails 프런트엔드 또는 Scala 백엔드와 같이 이 환경에서 실행할 앱 또는 마이크로서비스입니다. 각 앱은 자체 폴더에 위치하여 다른 모든 앱과 격리될 수도 있습니다.

- data-storage
  MySQL 또는 Redis와 같은 이 환경에서 실행할 데이터 저장소입니다. 각 데이터 저장소는 자체 폴더에 상주하여 다른 모든 데이터 저장소와 격리될 수도 있습니다.

각 구성요소 내에는 다음 명명 규칙에 따라 구성된 실제 Terraform 구성 파일이 있습니다.

- variables.tf : 입력변수
- outputs.tf : 출력 변수
- main.tf : 리소스 및 데이터 소스

Terraform을 실행하면 현재 디렉터리에서 확장자가 .tf인 파일을 찾으므로 원하는 파일 이름을 사용할 수 있습니다. 일관되고 예측 가능한 명명 규칙을 사용하면 코드를 더 쉽게 탐색할 수 있습니다. 예를 들어 변수, 출력 또는 리소스를 찾으려면 어디를 찾아야 하는지 항상 알 수 있습니다.

이전 규칙은 따라야 하는 최소 규칙입니다. 사실상 모든 Terraform 사용에서 입력 변수, 출력 변수 및 리소스로 매우 빠르게 이동할 수 있는 것이 유용하지만 이 규칙을 넘어서는 것이 필요할 수도 있습니다. 다음은 몇 가지 예입니다.

- dependencies.tf
  코드가 의존하는 외부 항목을 더 쉽게 확인할 수 있도록 모든 데이터 소스를 dependency.tf 파일에 저장하는 것이 일반적입니다.
- providers.tf
  공급자 블록을 providers.tf 파일에 넣어 코드가 어떤 공급자와 통신하는지, 어떤 인증을 제공해야 하는지 한눈에 확인할 수 있습니다.
- main-xxx.tf
  main.tf 파일에 많은 리소스가 포함되어 너무 길어지면 파일을 논리적인 방식으로 리소스를 그룹화하는 작은 파일로 나눌 수 있습니다. 예를 들어 main-iam.tf에는 모든 IAM 리소스가 포함될 수 있습니다. mains3.tf에는 모든 S3 리소스 등이 포함될 수 있습니다. main- 접두사를 사용하면 모든 리소스가 함께 그룹화되므로 알파벳순으로 정리된 폴더의 파일 목록을 더 쉽게 검색할 수 있습니다. 매우 많은 수의 리소스를 관리하고 이를 여러 파일로 분류하는 데 어려움을 겪고 있다면 이는 코드를 더 작은 모듈로 나누어야 한다는 신호일 수 있습니다.

이전에 작성한 웹 서버 클러스터 코드와 이 장에서 작성한 Amazon S3 및 DynamoDB 코드를 가져와 다음 그림의 폴더 구조를 사용하여 재정렬해 보겠습니다.

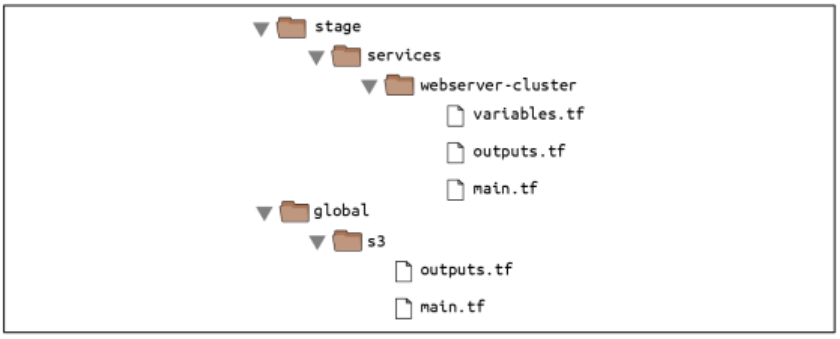


Figure 3-8. Move the web server cluster code into a stage/services/webserver-cluster folder to indicate that this is a testing or staging version of the web server.

```
mkdir -p stage/services/webserver-cluster
mkdir -p global/s3
```

이 장에서 생성한 S3 버킷을 global/s3 폴더로 이동해야 합니다. 출력 변수(s3_bucket_arn 및 dynamodb_table_name)를 outputs.tf 로 이동합니다. 폴더를 이동할 때 파일을 새 위치로 복사할 때 .terraform 폴더를 놓치지 않도록 하여 모든 것을 다시 초기화할 필요가 없도록 하세요.

이전에 생성한 웹 서버 클러스터는 stage/services/ webserver-cluster로 이동해야 합니다. 다시 한 번 .terraform 폴더를 복사하고, 입력 변수를 Variable.tf 로 이동하고, 출력 변수를 Outputs.tf 로 이동하세요. 또한 S3를 백엔드로 사용하려면 웹 서버 클러스터를 업데이트해야 합니다. global/s3/main.tf에서 백엔드 구성을 거의 그대로 복사하여 붙여넣을 수 있지만 tfstate키를 웹 서버 Terraform 코드(stage/services/webserver-cluster/terraform)와 동일한 폴더 경로로 변경해야 합니다. 이는 버전 제어의 Terraform 코드 레이아웃과 S3의 Terraform 상태 파일 간의 1:1 매핑을 제공하므로 둘이 어떻게 연결되어 있는지 분명합니다. s3 모듈은 이미 이 규칙을 사용하여 키를 설정합니다.

이 파일 레이아웃에는 다음과 같은 여러 가지 장점이 있습니다.

- 명확한 코드/환경 레이아웃
  코드를 탐색하고 각 환경에 배포된 구성 요소를 정확하게 이해하는 것은 쉽습니다.
- 격리
  이 레이아웃은 환경 간 및 환경 내의 구성 요소 간에 상당한 수준의 격리를 제공하여 문제가 발생할 경우 전체 인프라의 작은 부분 하나만으로 피해가 최대한 제한되도록 합니다.

어떤 면에서는 이러한 장점이 단점이기도 합니다.

- 여러 폴더 작업
  구성 요소를 별도의 폴더로 분할하면 하나의 명령으로 전체 인프라를 실수로 폭파하는 것을 방지할 수 있을 뿐만 아니라 하나의 명령으로 전체 인프라를 생성하는 것도 방지할 수 있습니다. 단일 환경의 모든 구성 요소가 단일 Terraform 구성에 정의된 경우 terraform apply 에 대한 단일 호출로 전체 환경을 스핀업할 수 있습니다. 그러나 모든 구성 요소가 별도의 폴더에 있는 경우 각 구성 요소에서 별도로 terraform Apply를 실행해야 합니다.
  참고:
  Terragrunt를 사용하는 경우 run-all 명령을 사용하여 여러 폴더에서 동시에 명령을 실행할 수 있습니다.
- 복사/붙여넣기
  이 섹션에서 설명하는 파일 레이아웃은 중복되는 부분이 많습니다. 예를 들어 동일한 frontend-app과 backend-app이 스테이지 폴더와 prod 폴더 모두에 있습니다. 모듈을 사용하지 않을 경우 재사용할때 매번 복사해서 붙여 사용합니다.
- 자원 의존성
  코드를 여러 폴더로 나누면 리소스 종속성을 사용하기가 더 어려워집니다. 앱 코드가 데이터베이스 코드와 동일한 Terraform 구성 파일에 정의된 경우 해당 앱 코드는 속성 참조를 사용하여 데이터베이스의 속성에 직접 액세스할 수 있습니다(예: aws_db_instance.foo.address 를 통해 데이터베이스 주소에 액세스). 하지만 권장한 대로 앱 코드와 데이터베이스 코드가 다른 폴더에 있는 경우에는 더 이상 그렇게 할 수 없습니다.

결과