테라폼 상태 관리하기

1. 테라폼 상태란?

테라폼을 실행할 때마다 테라폼은 생성한 인프라에 대한 정보를 테라폼 상태 파일에 기록한다.

기본적으로 /foo/bar 폴더에서 테라폼을 실행하면 테라폼은 /foo/bar/terraform.tfstate 파일을 생성한다. 이 파일에는 구성 파일(.tf)의 테라폼 리소스가 실제 리소스의 표현으로 매핑되는 내용을 기록하는 사용자 정의 JSON 형식이 포함되어 있다. 예를 들어 Terraform 구성에 다음이 포함되어 있다고 가정하자.

```
resource "aws_instance" "example" {
  ami           = "ami-06eea3cd85e2db8ce"
  instance_type = "t2.micro"
}
```

terraform.tfstate

```
{
  "version": 4,
  "terraform_version": "1.3.4",
  "serial": 175,
  "lineage": "49353501-2336-2d25-c23f-6c969e9bce2c",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-06eea3cd85e2db8ce",
             ...
            "id": "i-0f4b4f268104dcad3",
            "instance_type": "t2.micro",
        }
      ]
    }
  ],
  "check_results": null
}
```

테라폼은 이 JSON 형식을 통해 타입이 aws_instance고 이름이 example인 리소스가

i-0f4b4f268104dcad3라는 ID를 사용하는 AWS 계정의 EC2 인스턴스와 일치하는 것을 알고 있다. 테라폼을 실행할 때마다 AWS에서 이 EC2인스턴스의 최신 상태를 가져와서 테라폼의 구성과 비교하여 어느 변경 사항을 적용해야 하는지 결정할 수 있다. 즉, plan 명령의 출력은 상태 파일의 ID를 통해 발견된 컴퓨터의 코드와 실제 세계에 배포된 인프라 간의 차이이다.

참고: 상태 파일은 프라이빗 API이다.

상태 파일은 배포할 때마다 변경되는 프라이빗 API로 오직 테라폼 내부에서 사용하기 위한 것이다. 테라폼 상태 파일을 직접 편집하거나 직접 읽는 코드를 작성해서는 안된다.

상태 파일을 조작해야 하는 경우 terraform import 또는 terraform state 명령을 사용해야 한다.

테라폼을 팀 단위로 사용하고자 할 때는 다음과 같은 몇 가지 문제에 직면한다.

- **상태 파일을 저장하는 공유 스토리지**

  태라폼을 사용하여 인프라를 업데이트하려면 각 팀원이 동일한 테라폼 상태 파일에 액세스해야 한다. 즉, 상태 파일을 공유 위치에 저장해야 한다.

- **상태 파일 잠금**

  상태 데이터가 공유되자마자 '잠금(locking)'이라는 새로운 문제가 발생한다. 잠금 기능없이 두 팀원이 동시에 테라폼을 실행하는 경우 여러 테라폼 프로세스가 상태 파일을 동시에 업데이트하여 충돌을 일으킬 수 있다. 이러한 경쟁 상태에 처하면 데이터가 손실되거나 상태 파일이 손상될 수 있다.

- **상태 파일 격리**

  인프라를 변경할 때는 다른 환경을 격리하는 것이 가장 좋다. 예를 들어 테스트 또는 스테이징 환경을 변경할 때 실수로 프로덕션 환경이 중단되는 경우는 없는지 확인해야 한다. 그러나 모든 인프라가 동일한 테라폼 상태 파일에 정의되어 있다면 변경사항을 어떻게 격리할 수 있을까?

2. 상태 파일 공유

여러 명의 팀원이 파일에 공통으로 액세스할 수 있게 하는 가장 일반적인 방법은 파일을 깃과 같은 버전 관리 시스템에 두는 것이다. 그러나 테라폼 상태 파일을 버전 관리 시스템에 저장하는 것은 다음과 같은 이유 때문에 부적절하다.

- **수동 오류**

테라폼을 실행하기 전에 최신 변경 사항을 가져오거나 실행하고 나서 푸시하는 것을 잊기 쉽다. 팀의 누군가가 이전 버전의 상태 파일로 테라폼을 실행하고, 그 결과 실수로 이전 버전으로 롤백하거나 이전에 배포된 인프라를 복제하는 문제가 발생할 수 있다.

- **잠금**

대부분의 버전 관리 시스템은 여러 명의 팀 구성원이 동시에 하나의 상태 파일에 terraform apply 명령을 실행하지 못하게 하는 잠금 기능을 제공하지 않는다.

- **시크릿**

테라폼 상태 파일의 모든 데이터는 평문으로 저장되는데 특정 테라폼 리소스에 중요한 데이터를 저장해야 할 때 문제가 발생한다. 예를 들어 aws_db_instance 리소스를 사용하여 데이터베이스를 만드는 경우 테라폼은 데이터베이스의 사용자 이름과 비밀번호를 상태 파일에 평문으로 저장한다. 사용자 이름과 비밀번호 같은 중요한 데이터를 평문으로 저장하는 것은 버전 관리나 보안 측면에서도 적절하지 않다.

상태 파일 공유 스토리지를 관리하는 가장 좋은 방법은 버전 관리 시스템을 사용하는 대신 테라폼에 내장된 원격 백엔드 기능을 사용하는 것이다. 테라폼 백엔드는 테라폼이 상태를 로드하고 저장하는 방법을 결정한다. 기본 백엔드는 로컬 백엔드로써 로컬 디스크에 상태 파일을 저장한다. 원격 백엔드를 사용하면 상태 파일을 원격 공유 저장소에 저장할 수 있다. 아마존 S3와 애저 스토리지, 구글 클라우드 스토리지, 해시코프의 테라폼 클라우드, 테라폼 프로, 테라폼 엔터프라이즈 등 다양한 원격 백엔드가 지원된다.

- **수동 오류**

원격 백엔드를 구성하면 테라폼은 plan이나 apply 명령을 실행할 때마다 해당 백엔드에서 상태 파일을 자동으로 로드한다. 또한 apply 명령을 실행한 후에는 상태 파일을 백엔드에 자동 저장하므로 수동 오류가 발생하지 않는다.

- **잠금**

대부분의 원격 백엔드는 기본적으로 잠금 기능을 지원한다. terraform apply 명령을 실행하면 테라폼은 자동으로 잠금을 활성화한다. 다른 사람이 apply 명령을 실행 중인 경우 이미 잠금이 활성화되어 있어 잠금이 해제될 때까지 기다려야 한다. -lock-timeout=<TIME> 매개변수를 사용하면 apply 명령을 실행할 때 잠금이 해제되기까지 테라폼이 얼마 동안 대기하도록 할지 설정할 수 있다.

예를 들어 -lock-timeout=10m 으로 설정하면 10분 동안 대기한다.

- **시크릿**

대부분의 원격 백엔드는 기본적으로 데이터를 보내거나 상태 파일을 저장할 때 암호화하는 기능을 지원한다. 또한 이러한 백엔드는 아마존 S3 버킷과 IAM 정책을 사용하는 것 같이 일반적으로 액세스 제한을 구성하는 방법을 노출하므로 상태 파일에 액세스할 수 있는 사용자와 파일에 포함될 수 있는 시크릿을 제어할 수 있다. 테라폼이 상태 파일 내에서 시크릿 암호화를 기본적으로 지원하면 더 좋겠지만 최소한 상태 파일이 디스크 어디에도 평문으로 저장되지는 않으므로 이러한 백엔드로 대부분의 보안 문제를 해결할 수 있다.

원격 백엔드를 S3로 사용하면 다음과 같은 장점이 있다.

- 관리형 서비스이므로 추가 인프라를 배포하고 관리할 필요가 없다.
- 99.999999999%의 내구성과 99.99%의 가용성을 제공하도록 설계되었으므로 데이터 손실 또는 서비스 중단을 걱정할 필요가 없다.
- 암호화를 지원하므로 상태 파일에 민감한 데이터를 저장할 때 안전성을 높인다. S3 버킷에 액세스할 수 있는 팀원은 상태 파일을 암호화되지 않은 형태로 볼 수 있으므로 여전히 부분적인 솔루션이지만 최소한 전송 중인 데이터와 저장되는 데이터는 암호화할 수 있다.
- 아마존 다이나모DB를 통한 잠금 기능을 지원한다.
- 버전 관리를 지원하므로 상태 파일의 수정 사항이 모두 저장된다. 따라서 문제가 발생하면 이전 버전으로 롤백할 수 있다.
- 대부분의 테라폼 기능을 프리 티어로 쉽게 사용할 수 있으므로 비용이 저렴하다.

새로운 디렉토리에 main.tf 파일을 생성한다.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

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

다음으로 aws_s3_bucket 리소스를 사용하여 S3 버킷을 생성한다.

```
resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-state"

  # 실수로 S3 버킷을 삭제하는 것을 방지한다.
  # lifecycle {
```

```
  #  prevent_destroy = true
  # }

  # S3 버킷 삭제 허용
  lifecycle {
    prevent_destroy = false
  }
  force_destroy = true

  # 코드 이력을 관리하기 위해 상태 파일의 버전 관리를 활성화한다.
  versioning {
    enabled = true
  }

  # 서버측 암호화를 활성화한다.
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

- **bucket**

S3 버킷의 이름이다. S3 버킷 이름은 AWS의 다른 모든 고객들과 겹치지 않게 고유해야 한다.

- **prevent_destroy**

prevent_destroy는 수명주기 설정이다. 리소스에서 prevent_destroy를 true로 설정하면 terraform destroy를 실행하는 것 같이 리소스를 삭제하려고 시도하는 경우 테라폼이 오류와 함께 종료된다. prevent_destroy 설정은 S3 버킷과 같은 중요한 리소스를 실수로 삭제하지 않게 해 준다. 삭제하려는 경우 prevent_destroy 설정을 주석 처리하면 된다.

- **versioning**

S3 버킷에 버전 관리를 활성화하여 버킷의 파일이 업데이트될 때마다 새 버전을 만들도록 한다. 이를 통해 언제든지 이전 버전으로 되돌릴 수 있다.

- **server_side_encryption_configuration**

이 부분은 S3 버킷에 기록된 모든 데이터에 서버 측 암호화를 설정한다. 이렇게 하면 S3에 저장 될 때 상태 파일이나 그 파일에 포함된 어떤 시크릿이라도 암호화된다.

다음으로 잠금에 사용한 다이나모DB 테이블을 생성해야 한다. 다이나모DB는 아마존의 분산형 키-값 저장소이다.

테라폼에서 다이나모DB를 잠금에 사용하려면 LockID(대소문자 구분)라는 기본키가 있는 다이나모 DB 테이블을 생성해야 한다.

aws_dynamodb_table 리소스를 사용하여 테이블을 생성할 수 있다.

```
resource "aws_dynamodb_table" "terraform_locks" {
  name         = "terraform-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

모든 리소스가 배포된 후에는 S3 버킷과 다이나모DB 테이블이 생성되지만 테라폼 상태는 여전히 로컬 디스크에 저장된다. S3 버킷에 상태를 저장하도록 테라폼을 구성하려면 테라폼 코드 내에 backend 구성을 설정해야 한다.

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

BACKEND_NAME : s3와 같은 사용하려는 백엔드의 이름

CONFIG : 사용할 S3 버킷의 이름과 같이 해당 백엔드에 고유한 하나 이상의 인수

S3 버킷에 대한 backend 구성은 다음과 같다.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }

  backend "s3" {

    # 이전에 생성한 버킷 이름으로 변경
    bucket         = "terraform-state"
    key            = "globla/s3/terraform.tfstate"
    region         = "ap-northeast-2"

    # 이전에 생성한 다이나모DB 테이블 이름으로 변경
    dynamodb_table = "terraform-locks"
    encrypt        = true

  }
}
```

- **bucket** : 사용할 S3 버킷의 이름이다. 이전에 생성한 S3 버킷의 이름으로 입력한다.
- **key** : 테라폼 상태 파일을 저장할 S3 버킷 내의 파일 경로이다.
- **region** : S3 버킷이 있는 AWS의 리전이다.
- **dynamodb_table** : 잠금에 사용할 다이나모DB 테이블이다.
- **encrypt** : 이 부분을 true로 설정하면 테라폼 상태가 S3 디스크에 저장될 때 암호화된다.

테라폼이 S3 버킷에 상태 파일을 저장하도록 지시하려면 terraform init 명령을 다시 실행해야 한다.

백엔드가 활성화되면 테라폼을 명령을 실행하기 전에 이 S3 버킷에서 최신 상태를 자동으로 가져온다. 그리고 명령을 실행한 후에는 최신 상태를 S3 버킷으로 자동 푸시한다.

output.tf

```
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"
}
```

variables.tf

```
variable "bucket_name" {
  description = "The name of the S3 bucket. Must be globally unique."
  type        = string
  default     = "terraform-state"
}

variable "table_name" {
  description = "The name of the DynamoDB table. Must be unique in this AWS account."
  type        = string
  default     = "terraform-locks"
}
```

이 변수들은 S3 버킷의 ARN(Amazon Resource name, 아마존 리소스 이름)과 다이나모DB 테이블의 이름을 출력한다.

완성된 global/S3/main.tf

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

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

# S3 버킷 생성
resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-state"

  # 실수로 S3 버킷을 삭제하는 것을 방지한다.
  # lifecycle {
  #  prevent_destroy = true
  # }

  # S3 버킷 삭제 허용
  lifecycle {
    prevent_destroy = false
  }
  force_destroy = true

  # 코드 이력을 관리하기 위해 상태 파일의 버전 관리를 활성화한다.
  versioning {
    enabled = true
  }

  # 서버측 암호화를 활성화한다.
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}

# 다이나모DB 테이블 생성
resource "aws_dynamodb_table" "terraform_locks" {
  name         = "terraform-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
```

```
  }
}
```

수정된 global/S3/main.tf

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

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

resource "aws_s3_bucket" "terraform_state" {

  bucket = var.bucket_name

  // 버킷을 삭제할 수 있게 한다.
  force_destroy = true

}

# 상태 파일의 전체 리비전 기록을 볼 수 있도록 버전 지정
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}

# 기본적으로 서버측 암호화 사용
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"
    }
  }
}

# S3 버킷에 대한 모든 공용 액세스를 명시적으로 차단
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         = var.table_name
  billing_mode = "PAY_PER_REQUEST"
```

```
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

S3 버킷 삭제

```
# S3 버킷 생성
resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-state"

  # 실수로 S3 버킷을 삭제하는 것을 방지한다.
  lifecycle {
    prevent_destroy = false # 변경
  }
  force_destroy = true # 추가
}
...
```

이 후 terraform init, terraform apply 실행 후 terraform destroy 실행

3. 테라폼 백엔드의 단점

1. 테라폼 코드를 작성하여 S3 버킷 및 다이나모DB 테이블을 생성하고 해당 코드를 로컬 백엔드와 함께 배포한다.
2. 테라폼 코드로 돌아가서 원격 backend 구성을 추가합니다. 새로 생성된 S3 버킷과 다이나모DB 테이블을 사용하고, terraform init 명령을 실행하여 로컬 상태를 S3에 복사한다.

S3 버킷과 다이나모DB 테이블을 삭제하려면 이 단계를 반대로 수행한다.

1. 테라폼 코드로 이동하여 backend 구성을 제거한 다음 terraform init 명령을 재실행하여 테라폼 상태를 로컬 디스크에 다시 복사한다.
2. terraform destroy 명령을 실행하여 S3 버킷 및 다이나모DB 테이블을 삭제한다.
3. 이 때 S3 버킷이 비어있지 않고 lifecycle이 삭제 불가로 되어있어 삭제가 되지 않으면 lifecycle 블록의 prevent_destroy를 false로 변경하고 aws_s3_bucket 리소스에 force_destroy = true 를 추가한 후 terraform init, terraform apply를 실행하고 terraform destroy 명령을 실행한다.

테라폼의 두 번째 단점을 backend 블록에서는 변수나 참조를 사용할 수 없다. 다음 코드는 동작하지 않는다.

```
terraform {
  backend "s3" {
    backut         = var.bucket
    region         = var.region
    dynamodb_table = var.dynamodb_table
    key            = "example/terraform.tfstate"
    encrypt        = true
  }
}
```

즉 S3 버킷 이름, 리전, 다이나모DB 테이블 이름 등을 모두 테라폼 모듈에 수동으로 복사해서 붙여 넣어야 한다. 심지어 key 값을 복사해서 붙여넣어도 안되고 배포하는 모든 테라폼 모듈마다 고유한 key를 확보해서 실수로 다른 모듈의 상태를 덮어쓰지 않도록 해야 한다. 복사해서 붙여넣기와 수동 변경 작업을 자주하면 에러가 발생하기 쉽다. 특히 다중 환경에서 여러 테라폼 모듈을 배포하고 관리해야 하는 경우 에러가 발생하기 쉽다.

테라폼 코드의 backend 구성에서 특정 매개 변수를 생략하고 대신 terraform init를 호출할 때

-backend-config 인수를 통해 매개 변수를 전달하는 것이다. 예를 들어 backet 및 region 같은 반복되는 백엔드 인스를 backend.hcl이라는 별도의 파일로 추출할 수 있다.

```
# backend.hcl
bucket         = "terraform-state"
region         = "ap-northeast-2"
dynamodb_table = "terraform-locks"
encrypt        = true
```

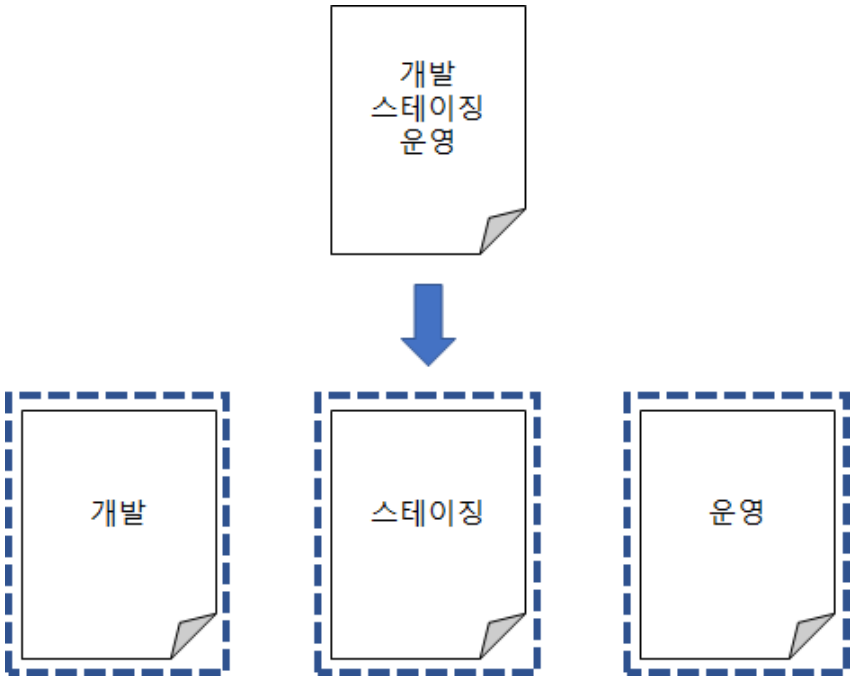
모듈마다 서로 다른 key 값을 설정해야 하기 때문에 key 매개 변수만 테라폼 코드에 남아있다.

```
# 부분 구성, 버킷, 리전 같은 다른 설정은 파일에서 -backend-config 인수를 통해
# terraform init 으로 전달된다.

terraform {
  backend "s3" {
    key = "example/terraform.tfstate"
  }
}

# 실행 시 -backend-config 인수와 함께 terraform init 를 실행한다.
$ terraform init -backend-config=backend.hcl
```

테라폼은 backend.hcl의 구성을 테라폼 코드의 구성과 병합하여 모듈에서 사용하는 전체 구성을 생성한다.

4. 상태 파일 격리

테라폼을 처음 사용하기 시작하면 모든 인프라를 단 하나의 테라폼 파일, 또는 같은 폴더에 들어 있는 단 하나의 테라폼 파일 세트로 정의하고 싶을 수 있다. 그러나 이렇게 하면 모든 테라폼 상태가 하나의 파일에 저장되므로 실수로 전체를 날려버릴 수 있다.



위의 그림처럼 모든 환경을 단 하나의 테라폼 구성 세트로 정의하는 대신 하나의 환경에서 문제가 생기더라도 다른 환경에 영향을 주지 않도록 각 환경을 별도의 구성 세트로 정의하려고 한다. 상태 파일을 격리하는 2가지 방법이 있다.

- 작업 공간을 통한 격리

  동일한 구성에 빠르고 격리된 테스트 환경에 유용하다.

- 파일 레이아웃을 이용한 격리

  보다 강력하게 분리해야 하는 운영 환경에 적합하다.

4.1 작업 공간을 통한 격리

테라폼 작업 공간(Terraform workspace)을 통해 테라폼 상태를 별도의 이름을 가진 여러 개의 작업 공간에 저장할 수 있다. 테라폼은 ‘default’라는 기본 작업 공간에서 시작하며 작업공간을 따로 지정하지 않으면 기본 작업 공간을 사용한다.

새 작업 공간을 만들거나 작업 공간을 전환하려면 terraform workspace 명령을 사용한다.

새로운 디렉토리로 이동해서 main.tf 생성 후 앞에서 생성한 S3 버킷 및 다이나모DB 테이블을 사용하여 백엔드 설정을 구성한다.

key 값은 workspace-example/terraform.tfstate로 설정한다.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }

  backend "s3" {
    # 이전에 생성한 버킷 이름으로 변경
    bucket         = "terraform-state"
    key            = "workspace-example/terraform.tfstate"
    region         = "ap-northeast-2"

    # 이전에 생성한 다이나모DB 테이블 이름으로 변경
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}

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

resource "aws_instance" "example" {
  ami           = "ami-06eea3cd85e2db8ce"
  instance_type = "t2.micro"

  # tags 추가
  tags = {
    Name = "example"
  }
}
```

terraform init과 terraform apply 명령을 사용하여 코드를 배포한다.

이 배포 작업의 상태 정보는 기본 작업 공간에 저장된다. Terraform workspace show 명령을 실행하여 현재 작업 공간을 확인할 수 있다.

```
$ terraform workspace show
* default
```

기본 작업 공간은 key 구성을 통해 지정한 위치에 상태를 저장한다. S3 버킷을 살펴보면 workspace-example 폴더에 terraform.tfstate 파일이 있다.

terraform worksapce new 명령을 사용하여 ‘example1’이라는 새 작업 공간을 만들어 보자.

```
$ terraform workspace new example1                                                    [9:41:22]
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 실행하면 다음과 같이 된다.

```
$ terraform plan                                                                      [9:42:00]

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                                  = "ami-06eea3cd85e2db8ce"
      + instance_type                        = "t2.micro"

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

테라폼은 완전히 새로운 EC2 인스턴스를 처음부터 만들려고 한다. 이는 기본 작업 공간과 example1 작업 공간의 상태 파일이 서로 분리되었고 우리는 ‘example1’ 작업 공간에 있기 때문이다. 이제 테라폼은 기본 작업 공간의 상태 파일을 사용하지 않으므로 EC2 인스턴스가 이미 생성되어 있다는 메시지가 출력되지 않는다.

terraform apply 명령을 실행하여 두 번째 EC2 인스턴스를 example1 작업 공간에 배포한다.

한 번 더 반복하여 ‘example2’라는 또 다른 작업 공간을 만들어 보자

```
$ terraform workspace new example2
```

terraform apply 명령을 다시 실행하여 세 번째 EC2 인스턴스를 배포한다.

terraform workspace list 명령을 사용하여 3개의 작업 공간이 생성된 것을 확인할 수 있다.

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

또한 terraform workspace select 명령을 사용하여 언제든지 작업 공간을 전환할 수 있다.

```
$ terraform workspace select example1
```

S3 버킷에 env: 라는 폴더가 생성되어 있고 env: 폴더 안에는 각 작업 공간마다 하나의 폴더가 있다.

각 작업 공간 내에서 테라폼은 backend 구성에서 지정한 key를 사용한다. 작업 공간마다 별도의 상태 파일이 있으므로 example1과 example2 작업 공간을 사용하면 테라폼은 example1/workspace-example/terraform.tfstate 과 example2/workspace-example/terraform.tfstate를 찾아야 한다. 즉 다른 작업 공간으로 전환하는 것은 상태 파일이 저장된 경로를 변경하는 것과 같다.

이처럼 작업 공간을 나누는 기능은 코드 리팩터링을 시도하는 것 같이 이미 배포되어 있는 인프라에 영향을 주지 않고 테라폼 모듈을 테스트할 때 유용하다. terraform workspace new 명령으로 새로운 작업 공간을 생성하여 완전히 동일한 인프라의 복사본을 배포할 수 있지만 상태 정보는 별도의 파일에 저장된다.

terraform.workspace 표현식을 사용하여 작업 공간 이름을 읽으면 현재 작업 공간을 기준으로 해당 모듈의 동작 방식을 변경할 수도 있다. 예를 들어 테스트 비용을 절감하기 위해 기본 작업 공간에서 인스턴스 유형을 t2.medium으로 설정하고 다른 작업 공간에서 t2.micro로 설정하는 방법은 다음과 같다.

```
resource "aws_instance" "example" {
  ami           = "ami-06eea3cd85e2db8ce"
  instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"
}
```

테라폼 작업 공간을 사용하면 코드의 다른 버전을 빠르게 가동하고 분해할 수 있지만 몇 가지 단점이 있다.

- 모든 작업 공간의 상태 파일은 동일한 백엔드(예를 들어, 동알한 S3 버킷)에 저장된다.
- terraform workspace 명령을 실행하지 않으면 코드나 터미널에 작업 공간에 대한 정보가 표시되지 않는다.

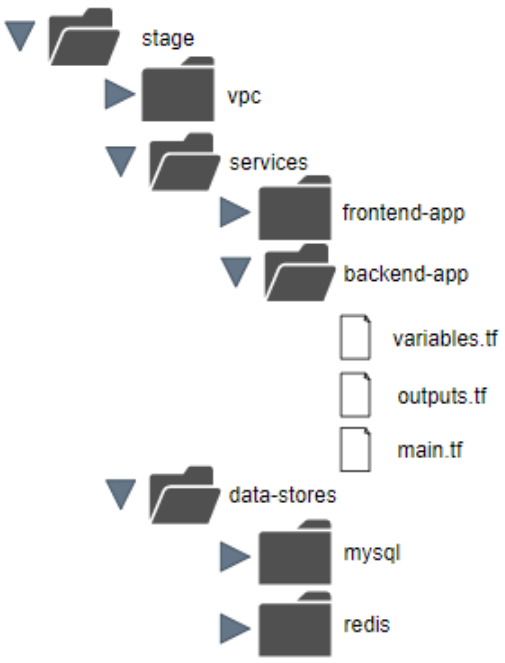
- 이전 항목 2개를 결합하면 작업 공간에 오류가 발생할 수 있다.

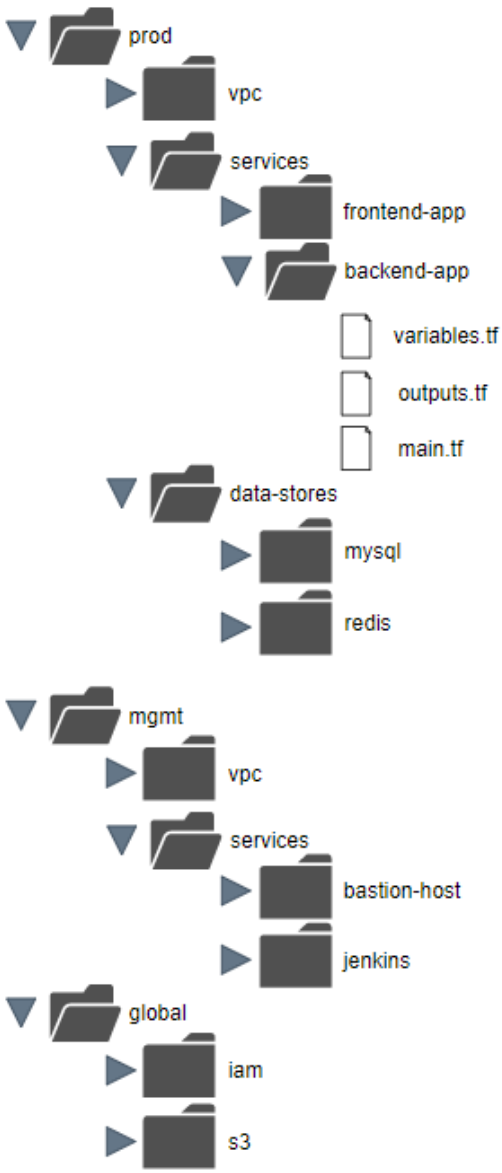
4.2 파일 레이아웃을 이용한 격리

환경을 완전히 격리하려면 다음 작업을 수행해야 한다.

- 각 테라폼 구성 파일을 분리된 폴더에 넣는다. 예를 들어 스테이징 환경에 대한 모든 구성은 stage 폴더에, 프로덕션 환경의 모든 구성은 prod 폴더에 넣는다.
- 서로 다른 인증 매커니즘과 액세스 제어를 사용하여 각 환경에 서로 다른 백엔드를 구성한다. 예를 들어 각 환경은 각각 분리된 S3 버킷을 백엔드로 사용하는 별도의 AWS 계정에 있을 수 있다.

일반적인 테라폼 프로젝트의 파일 레이아웃





최상위에는 각 환경마다 별도의 폴더가 있다. 프로젝트마다 환경이 다르지만 일반적인 환경은 다음과 같다.

- **stage**

  테스트 환경과 같은 사전 프로덕션 워크로드 환경

- **prod**

  사용자용 맵 같은 프로덕션 워크로드 환경

- **mgmt**

  베스천 호스트, 젠킨스와 같은 데브옵스 도구 환경

- **global**

  S3, IAM과 같이 모든 환경에서 사용되는 리소스를 배치할 수 있는 장소

각 환경에는 구성 요소마다 별도의 폴더가 있다. 프로젝트마다 구성 요소가 다르지만 일반적인 구성 요소는 다음과 같다.

- **vpc**

  해당 환경을 위한 네트워크 토폴로지

- **services**

  루비 온 레일즈 프런트앤드 또는 스칼라 백엔드 같이 해당 환경에서 서비스되는 애플리케이션 또는 마이크로서비스. 각각의 앱은 자체 폴더에 위치하여 다른 모든 앱과 분리할 수 있다.

- **data-storage**

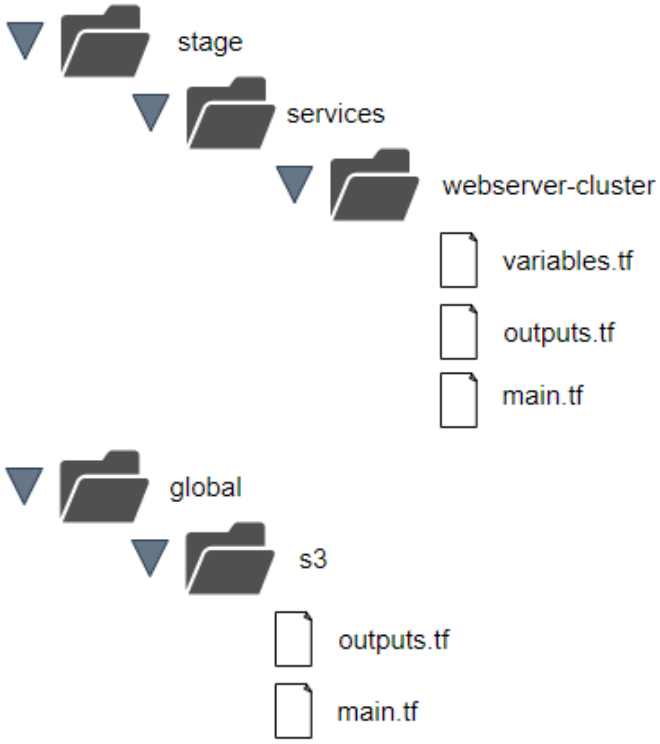
  MySQL 또는 레디스와 같은 해당 환경에서 실행할 데이터 저장소. 각 데이터 저장소는 자체 폴더에 상주하여 다른 모든 데이터 저장소와 분리할 수 있다.

각 구성 요소에는 다음과 같은 명명 규칙에 따라 구성되는 실제 테라폼 구성 파일이 있다.

- **variables.tf** : 입력 변수
- **output.tf** : 출력 변수
- **main.tf** : 리소스

테라폼을 실행할 때 확장명이 .tf이면 원하는 파일 이름을 사용할 수 있다

앞에서 작성한 웹 서버 클러스터 코드와 S3 및 다이나모DB 코드를 가져와서 폴더 구조를 재구성합니다.




```
terraform {
...
  backend "s3" {
    # 이전에 생성한 버킷 이름으로 변경
    bucket         = "terraform-state"
    key            = "stage/services/webserver-cluster/terraform.tfstate"
    region         = "ap-northeast-2"

    # 이전에 생성한 다이나모DB 테이블 이름으로 변경
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}
```

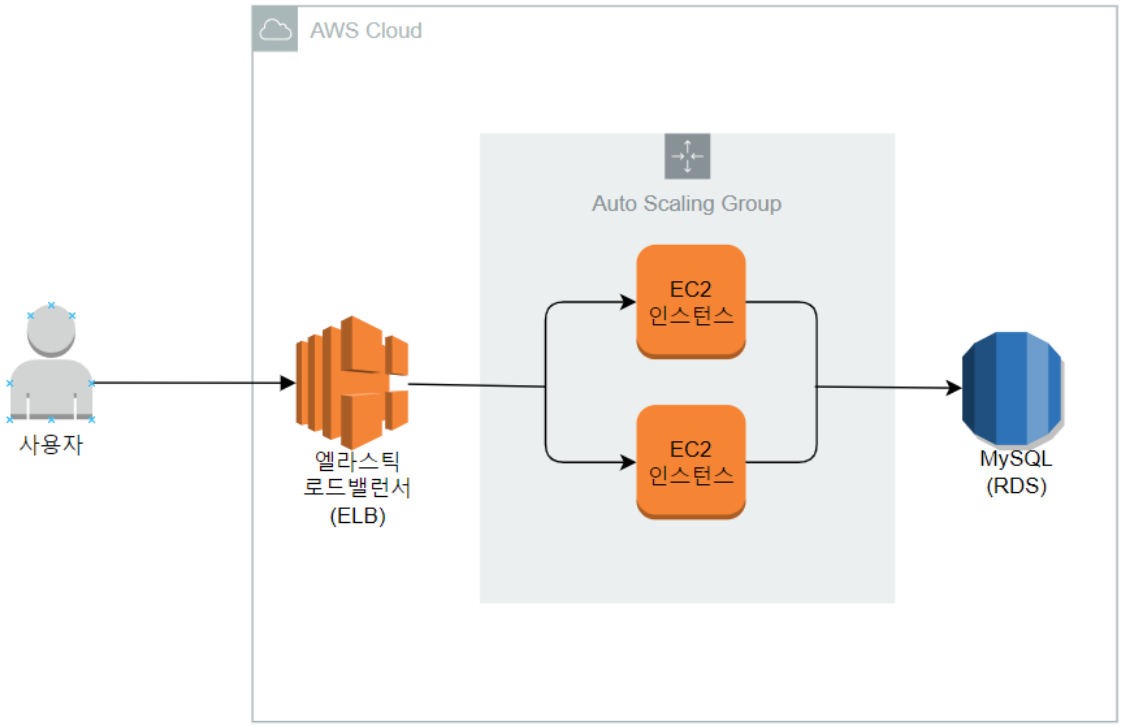
5. terraform_remote_state 데이터 소스

앞에서는 AWS에서 VPC의 서브넷 목록을 반환하는 aws_subnet_ids 데이터 소스 같은 읽기 전용 정보를 가져오기 위해 데이터 소스를 사용했었다. 그런데 상태 작업 시 특히 유용한 terraform_remote_state라는 다른 데이터 소스가 있다. 이 데이터 소스를 사용하면 다른 테라폼 구성 세트에 완전한 읽기 전용 방식으로 저장된 테라폼 상태 파일을 가져올 수 있다.

예를 들어 웹 서버 클러스터가 MySQL 데이터베이스와 통신해야 한다고 가정해 보자. 확장 가능하고 안전하며 내구성이 뛰어나고 가용성이 높은 데이터베이스를 실행하는데는 많은 작업이 필요하다.

아마존 RDS를 사용하여 AWS가 자동으로 처리하도록 할 수 있다. RDS는 MySQL, PostgreSQL, SQL 서버 및 오라클을 포함한 다양한 데이터베이스를 지원한다.

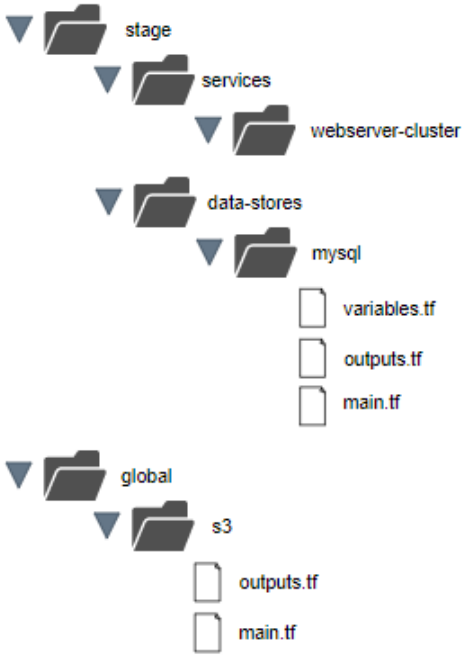
웹 서버 클러스터는 MySQL 데이터베이스보다 훨씬 자주 배포할 것이므로 그 과정에서 실수로 데이터베이스를 손상시키고 싶지 않다면 웹 서버 클러스터와 MySQL 데이터베이스 설정은 같은 위치에 정의하지 않아야 한다. 따라서 첫 번째로 stage/data-stores/mysql에 새 폴더를 만들고 그 안에 기본적인 테라폼 파일들을 만들어야 한다.



두 번째로 stage/data-stores/mysql/main.tf 파일에 데이터베이스 리소스를 정의할 것이다.

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

# RDS에 데이터베이스를 생성한다.
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-example"
  engine              = "mysql"
  allocated_storage   = 10                  # 스토리지는 10GB
  instance_class      = "db.t2.micro"       # vCPU 1개, 1GB 메모리
  name                = "example_database"
```

```
  username            = "admin"

  password = "???"
}
```



패스워드를 테라폼 리소스로 전달할 수 있는 2가지 방법

시크릿을 테라폼 리소스로 전달하는 첫 번째 방법은 테라폼 데이터 소스를 사용하여 시크릿 저장소에서 정보를 읽는 것이다. 예를 들어 데이터베이스 패스워드 같은 시크릿은 AWS 시크릿 매니저에 저장할 수 있습니다.

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

# RDS에 데이터베이스를 생성한다.
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-example"
  engine              = "mysql"
  allocated_storage   = 10                  # 스토리지는 10GB
  instance_class      = "db.t2.micro"       # vCPU 1개, 1GB 메모리
  name                = "example_database"
  username            = "admin"
```

```
  password = data.aws_secretsmanager_secret_version.db_password.secret_string
}

data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = "mysql-master-password-stage"
}
```

다음은 다양한 공급자가 지원하는 시크릿 저장소와 데이터 소스입니다.

- AWS 시크릿 매니저와 aws_secretsmanager_secret_version 데이터 소스(바로 앞 코드에서 사용)
- AWS 시스템 관리자 매개 변수 저장소(System Manager Parameter Store)와 aws_ssm_parameter 데이터 소스
- AWS KMS(Key Management Service, 키 관리 서비스)와 aws_kms_secrets 데이터 소스
- 구글 클라우드 KMS와 google_kms_secrets 데이터 소스
- 마이크로소프트 애저 키 볼트(Azure key Vault) 와 azurem_key_vault_secret 데이터 소스
- 해시코프 볼트(hashiCorp Vault) 및 vault_generic_secrets 데이터 소스

시크릿을 테라폼 리소스로 전달하는 두 번째 방법은 시크릿 값을 원패스워드, 라스트패스 또는 macOS의 키체인 접근과 같은 테라폼 외부에서 관리하고 환경 변수를 통해 시크릿 값을 테라폼에 전달하는 것이다. 이를 위해 stage/data-stores/mysql/variables.tf에서 db_password라는 변수를 선언한다.

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

이 변수에는 default 값이 없는데 이는 의도적인 것이다. 데이터베이스 패스워드 또는 민감한 정보는 평문으로 저장해서는 안된다. 대신 환경 변수를 사용하여 이 변수를 설정한다.

테라폼 구성에 정의된 각 입력 변수 foo에 환경 변수 TF_VAR_foo를 사용하여 이 변수의 값을 테라폼에 제공할 수 있다. db_password 입력 변수의 경우 리눅스/유닉스/macOS 시스템에서 TF_VAR_db_password 환경 변수를 설정하는 방법은 다음과 같다.

```
$  export TF_VAR_db_password=("password")
$  export TF_VAR_db_username=("admin")
$ terraform apply
...
```

export 명령 앞에는 시크릿 값이 배시 히스토리의 디스크에 저장되지 않도록 하기 위한 공간이 있다. 시크릿 값이 실수로 디스크에 평문으로 저장되는 것을 방지하기 위한 더 좋은 방법은 pass와 같은 명령으로 서브셸을 사용하여 안전항 환경에서 시크릿 값을 읽는 것이다.

```
$ export TF_VAR_db_password=$(pass database-password)
$ terraform apply
...
```

시크릿 값은 항상 테라폼 상태에 저장된다.

시크릿 저장소 또는 환경 변수에서 시크릿 값을 읽는 것은 시크릿 값이 코드에 평문으로 저장되지 않도록 하는 적절한 방법이다. 하지만 시크릿 값을 어떤 방법으로 읽든 간에 aws_db_instance와 같은 테라폼 리소스에 시크릿 값을 인수로 전달하면 해당 시크릿 값은 테라폼 상태 파일에 평문으로 저장된다.

그러므로 항상 암호화를사용하는 등 상태 파일을 저장할 때 특히 주의해야 한다. 또한 IAM 권한을 사용해 S3 버킷에 대한 액세스를 제한하는 등 상태 파일에 액세스할 수 있는 사용자를 세심하게 파악하고 관리해야 한다.

패스워드를 구성했으니 다음 단계는 stage/data-stores/mysql/terraform.tfstate 경로에서 생성한 S3 버킷에 상태를 저장하도록 이 모듈을 구성하는 것이다.

stage/data-stores/mysql/main.tf

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }

  backend "s3" {
    bucket         = "terraform-state"
    key            = "stage/data-stores/mysql/terraform.tfstate"
    region         = "ap-northeast-2"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}

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

resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform_example"
```

```
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true

  db_name             = var.db_name

  username = var.db_username
  password = var.db_password
}
```

stage/data-stores/mysql/variables.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
}

variable "db_name" {
  description = "The name to use for the database"
  type        = string
  default     = "example_database_stage"
}
```

terraform init과 terraform apply를 실행하여 데이터베이스를 생성한다.

데이터베이스가 준비되었으면 웹 서버 클러스터에 해당 주소와 포트를 제공해야 한다. 주소와 포트를 제공하는 첫 번째 단계는 stage/data-store/mysql/outputs.rf 에 2개의 출력 변수를 추가하는 것이다.

stage/data-stores/mysql/outputs.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"
}
```

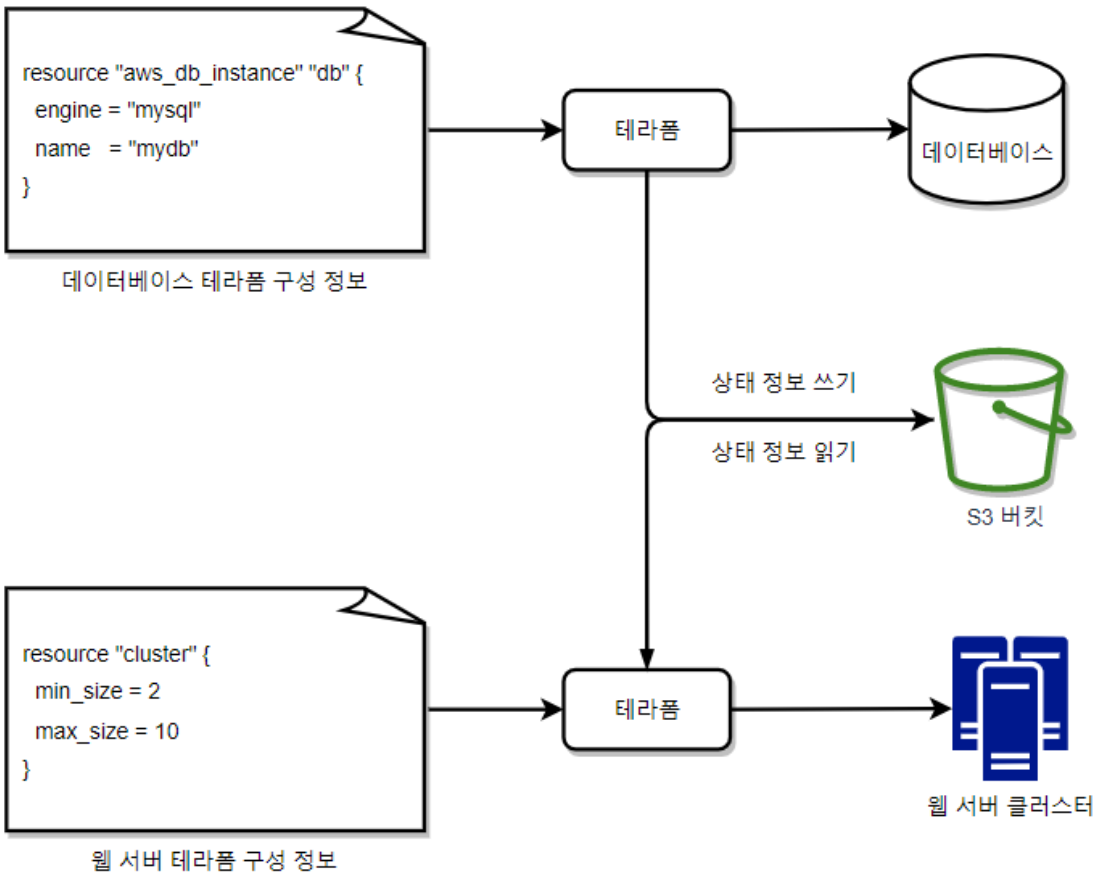
terraform apply 명령을 한 번 더 실행하면 터미널에 다음과 같이 출력된다.

```
$ terraform apply
Terraform will perform the following actions:

  # aws_db_instance.example will be created
  + resource "aws_db_instance" "example" {
      + address                             = (known after apply)
      + allocated_storage                   = 10
...
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

address = "terraform-example.cw3czwdcv6oy.ap-northeast-2.rds.amazonaws.com"
port = 3306
```

데이터베이스의 주소와 출력값은 stage/data-store/mysql/terraform.tfstate 경로에 있는 S3 버킷의 테라폼 상태 파일에 저장된다. stage/services/webserver-cluster/main.tf에 terraform_remote_state 데이터 소스를 추가하여 웹 서버 클러스터 코드가 이 상태 파일에서 데이터를 읽도록 할 수 있다.

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = "terragorm-state"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "ap-northeast-2"
  }
}
```

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



모든 테라폼 데이터 소스와 마찬가지로 terraform_remote_state에 의해 반환된 데이터는 읽기 전용이라는 점을 이해해야 한다. 웹 서버 클러스터에서 수행하는 테라폼 코드는 데이터베이스의 상태를 수정할 수 없으므로 데이터베이스 자체에 문제가 발생할 위험 없이 상태 정보를 가져올 수 있다.

모든 데이터베이스의 출력 변수는 상태 파일에 저장되며 아래와 같은 형식의 속성 참조를 이용해 terraform_remote_state 데이터 소스에서 읽을 수 있다.

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

예를 들어 다음은 웹 서버 클러스터 인스턴스의 사용자 데이터를 업데이트하여 terraform_remote_state 데이터 소스에서 데이터베이스 주소 및 포트를 가져와서 HTTP 응답에 해당 정보를 노출시키는 방법이다.

```
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
```

template_file 데이터 소스를 이용해 외부 파일을 읽을 수 있다.

```
data "template_file" "user_data" {
  template = file("user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

앞 코드에서 template 매개 변수는 user-data.sh 스크립트의 내용으로 설정되어 있다. 그리고 vars 매개 변수는 사용자 데이터 스크립트에 필요한 3가지 변수, 즉 서버 포트, 데이터베이스 주소 및 데이터베이스 포트에 대한 변수이다. 이러한 변수를 사용하려면 다음과 같이 stage/serviers/webserver-cluster/user-data.sh 스크립트를 업데이트해야 한다.

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} &
```

완성된 code

global/s3/main.tf

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

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

resource "aws_s3_bucket" "terraform_state" {
  bucket = "<본인의 버킷 이름>"

  # 실수로 S3 버킷을 삭제하는 것을 방지한다.
  # lifecycle {
  #  prevent_destroy = true
  # }

  # S3를 삭제할 수 있게 한다.
  lifecycle {
    prevent_destroy = false
  }
  force_destroy = true
}

# 서버측 암호화를 활성화한다.
resource "aws_s3_bucket_server_side_encryption_configuration" "terraform_encryption" {
  bucket = aws_s3_bucket.terraform_state.bucket

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}

# 코드 이력을 관리하기 위해 상태 파일의 버전 관리를 활성화한다.
resource "aws_s3_bucket_versioning" "terraform_versioning" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}

# 다이나모DB 테이블 생성
resource "aws_dynamodb_table" "terraform_locks" {
  name         = "<본인의 테이블 이름>"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

global/s3/outputs.tf

```
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"
}
```

stage/data-stores/mysql/main.tf

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }

  backend "s3" {
    bucket         = "<본인의 버킷 이름>"
    key            = "stage/data-stores/mysql/terraform.tfstate"
    region         = "ap-northeast-2"
    dynamodb_table = "<본인의 테이블 이름>"
    encrypt        = true
  }
}

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

resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-example"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true

  db_name             = var.db_name

  username = var.db_username
  password = var.db_password
}
```

stage/data-stores/mysql/variables.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
```

```
}

variable "db_name" {
  description = "The name to use for the database"
  type        = string
  default     = "aws00_example_database_stage"
}
```

stage/data-stores/mysql/outputs.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"
}
```

stage/services/webserver-cluster/main.tf

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

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

resource "aws_launch_template" "example" {
  name                   = "example"
  image_id               = "ami-0ab04b3ccbadfae1f"
  instance_type          = "t2.micro"
  key_name               = "<본인의 key name>"
  vpc_security_group_ids = [aws_security_group.instance.id]

  user_data = base64encode(data.template_file.web_output.rendered)

  # Required when using a launch configuration with an auto scaling group.
  lifecycle {
    create_before_destroy = true
  }
}

# 오토스케일링 그룹
```

```
resource "aws_autoscaling_group" "example" {
  availability_zones = ["ap-northeast-2a", "ap-northeast-2c"]

  name             = "terraform-asg-example"
  desired_capacity = 1
  min_size         = 1
  max_size         = 2

  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"

  launch_template {
    id      = aws_launch_template.example.id
    version = "$Latest"
  }
  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
    propagate_at_launch = true
  }
}

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

# 로드밸런서 리스너
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"

  # 기본값으로 단순한 404 페이지 오류를 반환한다.
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}

# 로드 밸런서 리스너 룰 구성
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
  }
}
```

```
# 로드밸런서 대상그룹
resource "aws_lb_target_group" "asg" {
  name     = "aws00-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
  }
}

# 보안 그룹 - instance
resource "aws_security_group" "instance" {
  name = var.security_group_name

  ingress {
    from_port   = var.server_port
    to_port     = var.server_port
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# 보안 그룹 - ALB
resource "aws_security_group" "alb" {
  name = "terraform-example-alb"

  # 인바운드 HTTP 트래픽 허용
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # 모든 아웃바운드 트래픽 허용
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

data "terraform_remote_state" "db" {
        backend = "s3"

        config = {
                bucket = "terraform-state"
                key    = "stage/data-stores/mysql/terraform.tfstate"
                region = "ap-northeast-2"
        }
}

data "aws_vpc" "default" {
```

```
  default = true
}

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

data "template_file" "web_output" {
  template = file("${path.module}/web.sh")
  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

stage/services/webserver-cluster/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} &
```

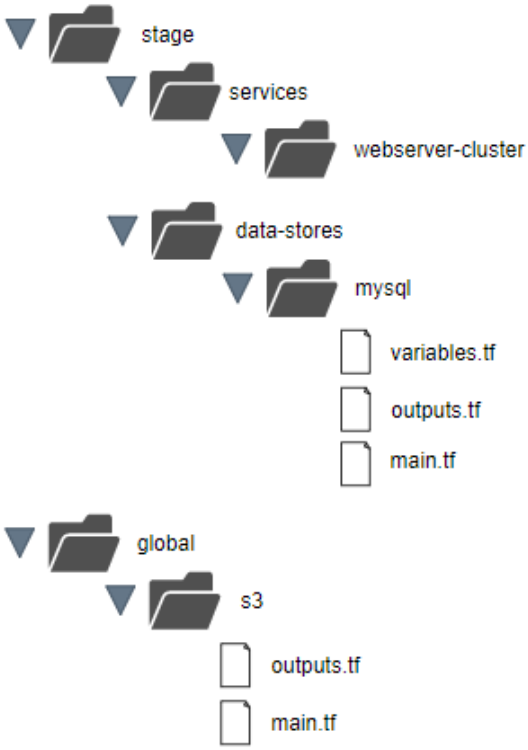
stage/services/webserver-cluster/variables.tf

```
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket used for the database's remote state storage"
  type        = string
  default     = "terraform-state"
}

variable "db_remote_state_key" {
  description = "The name of the key in the S3 bucket used for the database's remote state storage"
  type        = string
  default     = "stage/data-stores/mysql/terraform.tfstate"
}

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

variable "alb_name" {
  description = "The name of the ALB"
```

```
  type        = string
  default     = "terraform-asg-example"
}

variable "instance_security_group_name" {
  description = "The name of the security group for the EC2 Instances"
  type        = string
  default     = "terraform-example-instance"
}

variable "alb_security_group_name" {
  description = "The name of the security group for the ALB"
  type        = string
  default     = "terraform-example-alb"
}
```

stage/services/webserver-cluster/outputs.tf

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


stage
services
webserver-cluster
data-stores
mysql
variables.tf
outputs.tf
main.tf
global
s3
outputs.tf
main.tf