8/2

테라폼 소개

HashiCorp Terraform은 버전 관리, 재사용 및 공유가 가능한 사람이 읽을 수 있는 구성 파일로 클라우드 및 온프레미스 리소스를 모두 정의할 수 있는 코드 도구로서의 인프라입니다. 그런 다음 일관된 워크플로우를 사용하여 수명주기 전반에 걸쳐 모든 인프라를 프로비저닝하고 관리할 수 있습니다

Terraform의 세부 사항을 살펴보기 전에 다음 주제를 자세히 살펴보겠습니다.

- DevOps란 무엇입니까?
- 코드형 인프라란 무엇입니까?
- 코드형 인프라의 이점은 무엇입니까?
- Terraform은 어떻게 작동하나요?
- Terraform은 다른 코드형 인프라 도구와 어떻게 비교됩니까?

DevOps란?

그리 멀지 않은 과거에는 소프트웨어 회사를 세우려면 많은 하드웨어도 관리해야 했습니다. 랙을 설치하고, 서버를 로드하고, 배선을 연결하고, 냉각 장치를 설치하고, 중복 전원 시스템을 구축하는 등의 작업을 수행했습니다. 소프트웨어 개발을 전담하는 일반적으로 개발자(“Devs”)라고 하는 하나의 팀과 이 하드웨어 관리를 전담하는 운영(“Ops”)이라는 별도의 팀을 갖는 것이 합리적이었습니다.

일반적인 개발 팀은 애플리케이션을 구축하고 이를 운영 팀에 "경계 너머로 던집니다". 그런 다음 해당 애플리케이션을 배포하고 실행하는 방법을 파악하는 것은 Ops의 몫이었습니다. 대부분 이 작업은 수동으로 수행되었습니다. 왜냐하면 하드웨어를 물리적으로 연결하는 작업(예: 서버 랙 설치, 네트워크 케이블 연결)과 관련이 있었기 때문입니다. 그러나 애플리케이션 및 해당 종속성 설치와 같이 Ops가 소프트웨어에서 수행한 작업도 종종 서버에서 명령을 수동으로 실행하여 수행되었습니다.

이것은 한동안 잘 작동하지만 회사가 성장함에 따라 결국 문제에 직면하게 됩니다. 일반적으로 다음과 같이 진행됩니다. 릴리스는 수동으로 수행되기 때문에 서버 수가 증가함에 따라 릴리스는 느리고 고통스럽고 예측할 수 없게 됩니다. Ops 팀은 때때로 실수를 하기 때문에 눈송이 서버(snowflake server)로 끝나게 됩니다. 각 서버는 다른 서버와 미묘하게 다른 구성을 갖습니다(구성 드리프트라고 알려진 문제). 결과적으로 버그 수가 증가합니다. 개발자들은 어깨를 으쓱하며 “내 컴퓨터에서는 작동해요!”라고 말하지만 가동 중단 및 다운타임이 더 자주 발생합니다. 매 릴리스 후 오전 3시에 울리는 호출기에 지친 Ops 팀은 릴리스 주기를 일주일에 한 번으로 줄입니다. 그런 다음 한 달에 한 번. 그러면 6개월에 한 번씩. 2년마다 출시되기 몇 주 전에 팀은 모든 프로젝트를 하나로 병합하려고 시도하기 시작하여 병합 충돌이 엄청나게 발생합니다. 누구도 릴리스 브랜치를 안정화할 수 없습니다. 사일로 형태의 팀은 서로를 비난하기 시작합니다. 회사는 망해 가는 것입니다.

요즘에는 근본적인 변화가 일어나고 있습니다. 자체 데이터 센터를 관리하는 대신 많은 기업이 Amazon Web Services(AWS), Microsoft Azure, Google Cloud Platform(GCP)과 같은 서비스를 활용하여 클라우드로 전환하고 있습니다. 많은 Ops 팀은 하드웨어에 막대한 투자를 하는 대신 Chef, Puppet, Terraform, Docker, Kubernetes와 같은 도구를 사용하여 소프트웨어 작업에 모든 시간을 쏟고 있습니다. 서버를 랙에 설치하고 네트워크 케이블을 연결하는 대신 많은 시스템 관리자가 코드를 작성하고 있습니다. 결과적으로 Dev와 Ops 모두 대부분의 시간을 소프트웨어 작업에 소비하며 두 팀 간의 구분이 모호해집니다. 애플리케이션 코드를 담당하는 별도의 개발팀과 운영 코드를 담당하는 운영팀을 갖는 것이 여전히 타당할 수 있지만, 개발팀과 운영팀이 더욱 긴밀하게 협력해야 한다는 점은 분명합니다. DevOps 운동이 시작된 것입니다. DevOps는 팀 이름이나 직위 또는 특정 기술이라기 보다는 일련의 프로세스, 아이디어 및 기술입니다. DevOps에 대한 정의는 사람마다 조금씩 다르지만 이 책에서는 다음과 같이 설명하겠습니다.

DevOps의 목표는 소프트웨어 제공을 훨씬 더 효율적으로 만드는 것입니다.

며칠간의 병합 악몽 대신 코드를 지속적으로 통합하고 항상 배포 가능한 상태로 유지합니다. 한 달에 한 번 코드를 배포하는 대신 하루에 수십 번 또는 커밋이 완료될 때마다 코드를 배포할 수 있습니다. 그리고 지속적인 중단과 가동 중지 시간 대신 탄력적인 자가 치유 시스템을 구축하고 모니터링 및 경고를 사용하여 자동으로 해결할 수 없는 문제를 포착합니다. DevOps 혁신을 거친 기업의 결과는 놀랍습니다. 예를 들어, Nordstrom은 DevOps 방식을 조직에 적용한 후 매월 제공하는 기능 수를 100% 늘리고, 결함을 50% 줄이고, 리드 타임(아이디어가 떠오르는 데 걸리는 시간)을 줄일 수 있다는 사실을 발견했습니다. 프로덕션에서 코드 실행을 60% 줄이고, 프로덕션 사고 건수를 60%에서 90%로 줄입니다. HP의 LaserJet 펌웨어 사업부가 DevOps 방식을 사용하기 시작한 후 개발자가 새로운 기능을 개발하는 데 소요하는 시간은 5%에서 40%로 늘어났고 전체 개발 비용은 40% 감소했습니다. Etsy는 DevOps 방식을 사용하여 수많은 중단을 초래하는 스트레스가 많고 빈번하지 않은 배포에서 중단을 훨씬 적게 하면서 하루에 25~50회 배포하는 방식으로 전환했습니다.

DevOps 운동에는 문화, 자동화, 측정, 공유(약어로 CAMS라고도 함)라는 네 가지 핵심 가치가 있습니다. 이 책은 DevOps에 대한 포괄적인 개요를 다루려는 것이 아니므로, 자동화라는 가치 중 하나에만 초점을 맞추겠습니다. 목표는 가능한 많은 소프트웨어 제공 프로세스를 자동화하는 것입니다. 이는 웹 페이지를 클릭하거나 수동으로 셸 명령을 실행하는 것이 아니라 코드를 통해 인프라를 관리한다는 의미입니다. 이는 일반적으로 코드형 인프라(Infrastructure as Code)라고 불리는 개념입니다.

코드형 인프라란?

IaC(Infrastructure as Code)의 기본 개념은 코드를 작성하고 실행하여 인프라를 정의, 배포, 업데이트 및 제거하는 것입니다. 이는 하드웨어를 나타내는 측면(예: 물리적 서버 설정)까지 포함하여 운영의 모든 측면을 소프트웨어로 취급하는 사고방식의 중요한 변화를 나타냅니다. 실제로 DevOps의 주요 통찰력은 서버, 데이터베이스, 네트워크, 로그 파일, 애플리케이션 구성, 문서, 자동화된 테스트, 배포 프로세스 등을 포함하여 코드로 거의 모든 것을 관리할 수 있다는 것입니다.

IaC 도구에는 다섯 가지 광범위한 범주가 있습니다.

- 애드훅 스크립트
- 관리형 도구의 구성파일
- 템플릿 도구 지원 서버
- 오케스트레이션 도구
- 프로비저닝 도구

이제 이것들을 하나씩 살펴보겠습니다.

애드훅 (Ad hoc) 스크립트

모든 것을 자동화하는 가장 간단한 접근 방식은 임시 스크립트를 작성하는 것입니다. 수동으로 수행하던 작업을 개별 단계로 나누고, 선호하는 스크립트 언어(예: Bash, Ruby, Python)를 사용하여 각 단계를 코드로 정의하고, 다음과 같이 서버에서 해당 스크립트를 실행합니다.

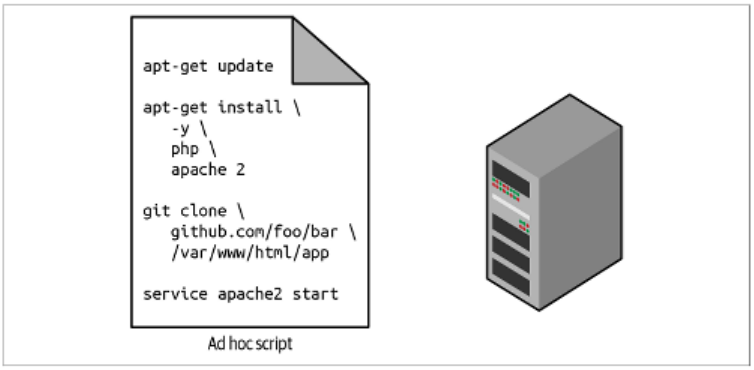


Figure 1-1. The most straightforward way to automate things is to create an ad hoc script that you run on your servers.

예를 들어 다음은 종속성을 설치하고, Git 저장소에서 일부 코드를 확인하고, Apache 웹 서버를 실행하여 웹 서버를 구성하는 setup-webserver.sh라는 Bash 스크립트입니다.

```
#!/bin/bash

# Update the apt-get cache
sudo apt-get update
# Install PHP and Apache
sudo apt-get install -y php apache2
# Copy the code from the repository
sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app
# Start Apache
sudo service apache2 start
```

임시 스크립트의 가장 큰 장점이자 단점은 널리 사용되는 범용 프로그래밍 언어를 사용할 수 있고 원하는 대로 코드를 작성할 수 있다는 것입니다. IaC용으로 특별히 제작된 도구는 복잡한 작업을 수행하기 위한 간결한 API를 제공하는 반면, 범용 프로그래밍 언어를 사용하는 경우 모든 작업에 대해 완전히 사용자 정의 코드를 작성해야 합니다. 또한 IaC용으로 설계된 도구는 일반적으로 코드에 특정 구조를 적용하는 반면, 범용 프로그래밍 언어를 사용하면 각 개발자가 자신의 스타일을 사용하고 다른 작업을 수행합니다. 이러한 문제는 Apache를 설치하는 4줄 스크립트에는 큰 문제가 되지 않지만 임시 스크립트를 사용하여 수십 개의 서버, 데이터베이스, 로드 밸런서, 네트워크 구성 등을 관리하려고 하면 문제가 발생합니다. Bash 스크립트의 대규모 저장소를 유지해야 했던 적이 있다면 거의 항상 유지 관리할 수 없는 스파게티 코드가 엉망이 된다는 것을 알고 계실 것입니다. 임시 스크립트는 소규모의 일회성 작업에 적합하지만 모든 인프라를 코드로 관리하려면 해당 작업을 위해 특별히 제작된 IaC 도구를 사용해야 합니다.

관리형 도구의 구성파일

Chef, Puppet 및 Ansible은 모두 구성 관리 도구입니다. 즉, 기존 서버에 소프트웨어를 설치하고 관리하도록 설계되었습니다. 예를 들어 다음은 setup-webserver.sh 스크립트와 동일한 Apache 웹 서버를 구성하는 web-server.yml이라는 Ansible 역할입니다.

```
- name: Update the apt-get cache
  apt:
     update_cache :  yes
- name: Install PHP
  apt:
     name: php
- name: Install Apache
```

```
    apt:
      name: apache2
  - name: Copy the code from the repository
    git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app
  - name: Start Apache
    service: name=apache2 state=started enabled=yes
```

코드는 Bash 스크립트와 유사해 보이지만 Ansible과 같은 도구를 사용하면 다음과 같은 여러 가지 이점을 얻을 수 있습니다.

- 코딩 규칙
  Ansible은 문서, 파일 레이아웃, 명확한 이름이 지정된 매개변수, 비밀 관리 등을 포함하여 일관되고 예측 가능한 구조를 적용합니다. 모든 개발자는 임시 스크립트를 다른 방식으로 구성하지만 대부분의 구성 관리 도구에는 코드를 더 쉽게 탐색할 수 있는 일련의 규칙이 함께 제공됩니다.
- 멱등성
  한 번 작동하는 임시 스크립트를 작성하는 것은 그리 어렵지 않지만 반복해서 실행하더라도 올바르게 작동하는 임시 스크립트를 작성하는 것은 훨씬 어렵습니다. 스크립트에 폴더를 만들 때마다 해당 폴더가 이미 존재하는지 확인해야 하고 파일에 구성 줄을 추가할 때마다 해당 줄이 이미 존재하지 않는지 확인해야 하며 앱을 실행하고 싶을 때마다 해당 앱이 아직 실행되고 있지 않은지 확인해야 합니다. 이런 불편함 없이 몇 번이나 실행해도 올바르게 작동하는 코드를 멱등성 코드라고 합니다. 이전 섹션의 Bash 스크립트를 멱등성으로 만들려면 많은 if 문을 포함하여 많은 코드 줄을 추가해야 합니다. 반면에 대부분의 Ansible 함수는 기본적으로 멱등성을 갖습니다. 예를 들어, web-server.yml 은 Apache가 아직 설치되지 않은 경우에만 설치하고 Apache 웹 서버가 아직 실행되고 있지 않은 경우에만 시작을 시도합니다.
- 분산
  임시 스크립트는 단일 로컬 시스템에서 실행되도록 설계되었습니다. Ansible 및 기타 구성 관리 도구는 다수의 원격 서버를 관리하기 위해 특별히 설계되었습니다.

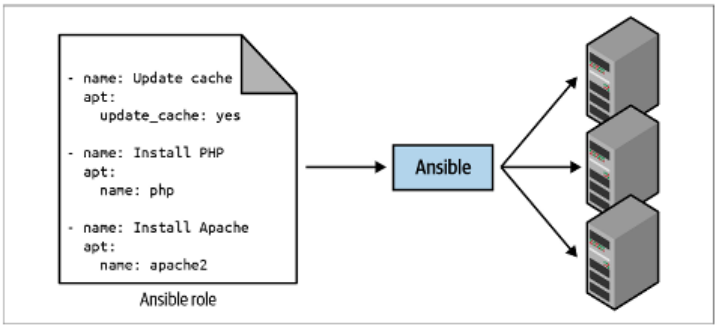


Figure 1-2. A configuration management tool like Ansible can execute your code across a large number of servers.

예를 들어 web-server.yml 역할을 5개의 서버에 적용하려면 먼저 해당 서버의 IP 주소가 포함된 호스트라는 파일을 생성합니다.

```
[webservers]
11.11.11.11
11.11.11.12
11.11.11.13
11.11.11.14
11.11.11.15
```

다음으로 다음 Ansible 플레이북을 정의합니다.

```
- hosts: webservers
  roles:
```

```
  - webserver
```

마지막으로 다음과 같이 플레이북을 실행합니다.

```
ansible-playbook playbook.yaml
```

이는 Ansible에 5개의 서버를 모두 병렬로 구성하도록 지시합니다. 또는 플레이북에서 serial이라는 매개변수를 설정하여 서버를 일괄적으로 업데이트하는 롤링 배포를 수행할 수 있습니다. 예를 들어, serial을 2로 설정하면 Ansible은 5개가 모두 완료될 때까지 한 번에 2개의 서버를 업데이트하도록 지시합니다. 임시 스크립트에서 이 논리를 복제하려면 수십 또는 수백 줄의 코드가 필요합니다.

템플릿 도구 지원 서버

최근 인기가 높아지고 있는 구성 관리의 대안으로는 Docker, Packer, Vagrant와 같은 서버 템플릿 도구가 있습니다. 여러 대의 서버를 시작하고 각 서버에서 동일한 코드를 실행하여 구성하는 대신, 서버 템플릿 도구의 기본 개념은 운영 체제(OS)의 완전히 독립적인 "스냅샷"을 캡처하는 서버 이미지를 생성하는 것입니다. , 소프트웨어, 파일 및 기타 모든 관련 세부 정보. 그런 다음 그림과 같이 다른 IaC 도구를 사용하여 모든 서버에 해당 이미지를 설치할 수 있습니다.

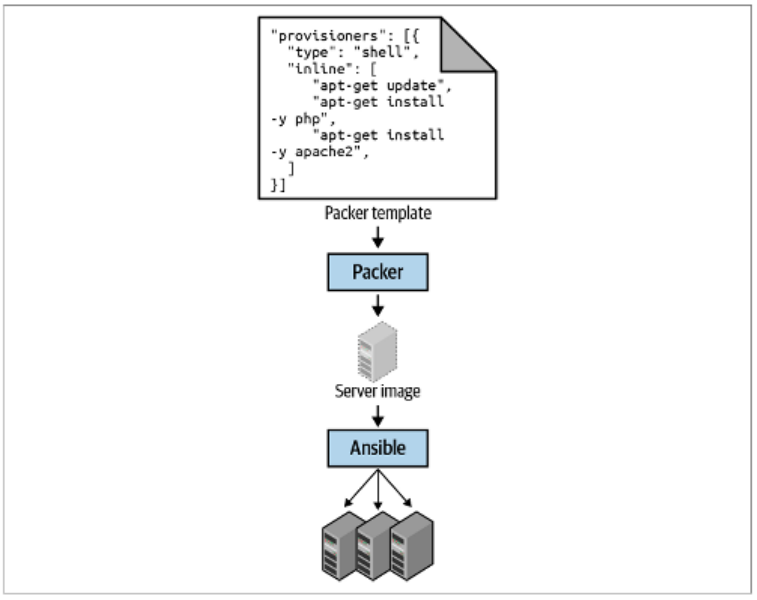


Figure 1-3. You can use a server templating tool like Packer to create a self-contained image of a server. You can then use other tools, such as Ansible, to install that image across all of your servers.

이미지 작업을 위한 도구에는 크게 두 가지 범주가 있습니다

- 가상머신
  VM(가상 머신)은
  하드웨어를 포함한 전체 컴퓨터 시스템을 에뮬레이트합니다. VMware, VirtualBox 또는 Parallels와 같은 하이퍼바이저를 실행하여 기본 CPU, 메모리, 하드 드라이브 및 네트워킹을 가상화(즉, 시뮬레이션)합니다. 이것의 이점은 하이퍼바이저 위에서 실행하는 모든 VM 이미지가 가상화된 하드웨어만 볼 수 있으므로 호스트 시스템 및 다른 VM 이미지와 완전히 격리되고 모든 환경에서 정확히 동일한 방식으로 실행된다는 것입니다(예: 컴퓨터, QA 서버, 프로덕션 서버). 하지만 단점은 이 모든 하드웨어를 가상화하고 각 VM에 대해 완전히 별도의 OS를 실행하면 CPU 사용량, 메모리 사용량 및 시작 시간 측면에서 많은 오버헤드가 발생한다는 것입니다. Packer 및 Vagrant와 같은 도구를 사용하여 VM 이미지를 코드로 정의할 수 있습니다.

- 컨테이너
  컨테이너는
  OS의 사용자 공간을 에뮬레이트합니다. Docker, CoreOS rkt 또는 cri-o와 같은 컨테이너 엔진을 실행하여 격리된 프로세스, 메모리, 마운트지점 및 네트워킹을 생성합니다. 이것의 이점은 컨테이너 엔진 위에서 실행되는 모든 컨테이너가 자체 사용자 공간만 볼 수 있으므로 호스트 머신 및 기타 컨테이너로부터 격리되어 모든 환경(컴퓨터, QA)에서 정확히 동일한 방식으로 실행된다는 것입니다. 서버, 프로덕션 서버 등). **단점은 단일 서버에서 실행되는 모든 컨테이너가 해당 서버의 OS 커널과 하드웨어를 공유하므로 VM에서 얻는 격리 및 보안 수준을 달성하기가 훨씬 더 어렵다는 것**입니다. 그러나 **커널과 하드웨어가 공유되기 때문에 컨테이너는 밀리초 안에 부팅될 수 있으며 CPU 또는 메모리 오버헤드가 거의 없습니다**. Docker 및 CoreOS rkt와 같은 도구를 사용하여 컨테이너 이미지를 코드로 정의할 수 있습니다.

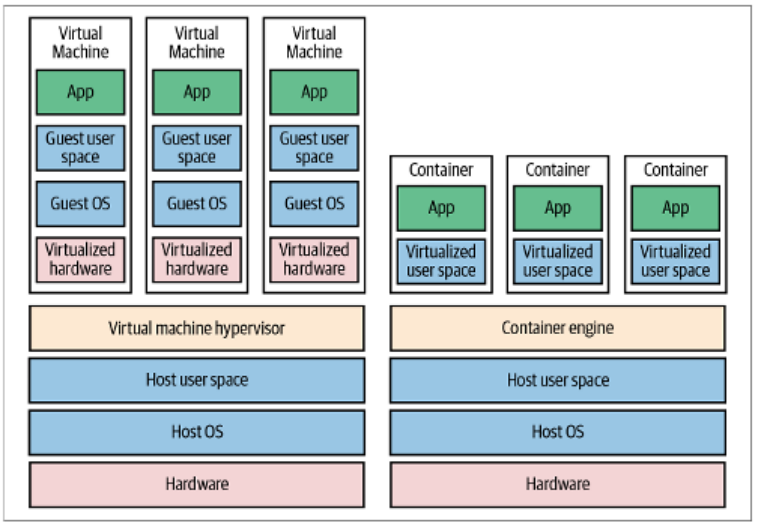


Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers virtualize only user space.

예를 들어 다음은 AWS에서 실행할 수 있는 VM 이미지인 Amazon Machine Image(AMI)를 생성하는 web-server.json이라는 Packer 템플릿입니다.

```
{
  "builders": [{
    "ami_name": "packer-example-",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0fb653ca2d3203ac1",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php apache2",
      "sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app"
    ],
    "environment_vars": [
      "DEBIAN_FRONTEND=noninteractive"
    ],
    "pause_before": "60s"
```

```
  }]
}
```

이 Packer 템플릿은 동일한 Bash 코드를 사용한 webserver.sh에서 본 것과 동일한 Apache 웹 서버를 구성합니다. Packer 템플릿의 코드와 이전 예제의 유일한 차이점은 이 Packer 템플릿이 Apache 웹 서버를 시작하지 않는다는 것입니다(sudo service apache2 start 명령어를 통해). 그 이유는 서버 템플릿이 일반적으로 이미지에 소프트웨어를 설치하는 데 사용되지만 실제로 해당 소프트웨어를 실행해야 하는 것은 이미지를 실행할 때만(예: 서버에 이미지 배포) 때문입니다. 이 템플릿에서 AMI를 빌드하려면 packer build webserver.json 을 실행합니다. 빌드가 완료된 후 모든 AWS 서버에 해당 AMI를 설치하고 서버가 부팅될 때 Apache를 실행하도록 각 서버를 구성할 수 있습니다

다양한 서버 템플릿 도구의 목적은 약간 다릅니다. Packer는 일반적으로 프로덕션 AWS 계정에서 실행하는 AMI와 같이 프로덕션 서버 위에서 직접 실행하는 이미지를 생성하는 데 사용됩니다. Vagrant는 일반적으로 Mac 또는 Windows 노트북에서 실행하는 VirtualBox 이미지와 같이 개발 컴퓨터에서 실행하는 이미지를 만드는 데 사용됩니다. Docker는 일반적으로 개별 애플리케이션의 이미지를 만드는 데 사용됩니다. 다른 도구에서 해당 컴퓨터에 Docker 엔진을 구성했다면 프로덕션 또는 개발 컴퓨터에서 Docker 이미지를 실행할 수 있습니다. 예를 들어 일반적인 패턴은 Packer를 사용하여 Docker 엔진이 설치된 AMI를 생성하고 AWS 계정의 서버 클러스터에 해당 AMI를 배포한 다음 해당 클러스터 전체에 개별 Docker 컨테이너를 배포하여 애플리케이션을 실행하는 것입니다.

서버 템플릿은 불변(immutable) 인프라로의 전환을 위한 핵심 구성 요소입니다. 이 아이디어는 변수가 불변이므로 변수를 값으로 설정한 후에는 해당 변수를 다시 변경할 수 없는 함수형 프로그래밍에서 영감을 받았습니다. 업데이트해야 할 사항이 있으면 새 변수를 만듭니다. 변수는 절대 변하지 않기 때문에 코드에 대해 추론하는 것이 훨씬 쉽습니다. 불변 인프라의 기본 개념은 비슷합니다. 즉, 서버를 배포한 후에는 다시 변경할 필요가 없습니다. 새 버전의 코드 배포와 같이 무언가를 업데이트해야 하는 경우 서버 템플릿에서 새 이미지를 생성하고 이를 새 서버에 배포합니다. 서버는 절대 변하지 않기 때문에 배포된 내용을 추론하기가 훨씬 쉽습니다.

오케스트레이션 도구

서버 템플릿 도구는 VM 및 컨테이너를 만드는 데 적합하지만 실제로 이를 어떻게 관리합니까? 대부분의 실제 사용 사례에서는 다음을 수행할 수 있는 방법이 필요합니다.

- VM과 컨테이너를 배포하여 하드웨어를 효율적으로 활용
- 롤링 배포, 블루-그린 배포, 카나리아 배포와 같은 전략을 사용하여 기존 VM 및 컨테이너 집합에 대한 업데이트를 롤아웃합니다.
- VM 및 컨테이너의 상태를 모니터링하고 비정상인 항목을 자동으로 교체합니다(자동 복구).
- 로드에 따라 VM 및 컨테이너 수를 늘리거나 줄입니다(자동 크기 조정).
- VM과 컨테이너 전체에 트래픽을 분산합니다(로드 밸런싱).
- VM과 컨테이너가 네트워크를 통해 서로 찾고 통신할 수 있도록 허용합니다(서비스 검색).

서비스 검색 : 가장 핵심적인 기능 중 하나이다.

kubernetes distribution

- eks, gke, aks

클라우드에 배포한 배포판

- Tanzu, Openshift

리눅스 배포판

이러한 작업을 처리하는 것은 Kubernetes, Marathon/Mesos, Amazon Elastic Container Service(Amazon ECS), Docker Swarm 및 Nomad와 같은 오케스트레이션 도구의 영역입니다. 예를 들어 Kubernetes를 사용하면 컨테이너를 코드로 관리하는 방법을 정의할 수 있습니다. 먼저 Kubernetes가 컨테이너를 실행하기 위해 관리하고 사용할 서버 그룹인 Kubernetes 클러스터를 배포합니다. 대부분의 주요 클라우드 제공업체는 Amazon Elastic Kubernetes Service(EKS), Google Kubernetes Engine(GKE) 및 Azure Kubernetes Service(AKS)와 같은 관리형 Kubernetes 클러스터 배포를 기본적으로 지원합니다.

작동 중인 클러스터가 있으면 Docker 컨테이너를 YAML 파일의 코드로 실행하는 방법을 정의할 수 있습니다.

```
apiVersion: apps/v1
# Use a Deployment to deploy multiple replicas of your Docker
# container(s) and to declaratively roll out updates to them
kind: Deployment
# Metadata about this Deployment, including its name
metadata:
  name: example-app
# The specification that configures this Deployment
spec:
  # This tells the Deployment how to find your container(s)
  selector:
    matchLabels:
      app: example-app
  # This tells the Deployment to run three replicas of your
  # Docker container(s)
  replicas: 3
  # Specifies how to update the Deployment. Here, we
  # configure a rolling update.
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate
  # This is the template for what container(s) to deploy
  template:
    # The metadata for these container(s), including labels
    metadata:
      labels:
        app: example-app
    # The specification for your container(s)
    spec:
      containers:

      # Run Apache listening on port 80
      - name: example-app
        image: httpd:2.4.39
        ports:
          - containerPort: 80
```

이 파일은 Kubernetes에 다음을 정의하는 선언적 방법인 배포를 생성하도록 지시합니다.

- 함께 실행할 하나 이상의 Docker 컨테이너. 이 컨테이너 그룹을 Pod 라고 합니다. 이전 코드에 정의된 Pod에는 Apache를 실행하는 단일 Docker 컨테이너가 포함되어 있습니다.

- 포드의 각 Docker 컨테이너에 대한 설정입니다. 이전 코드의 포드는 포트 80에서 수신 대기하도록 Apache를 구성합니다.
- 클러스터에서 실행할 포드의 복사본(복제본이라고도 함) 수. 앞의 코드는 세 개의 복제본을 구성합니다. Kubernetes는 고가용성 측면에서 최적의 서버를 선택하는 예약 알고리즘을 사용하여 클러스터에서 각 Pod를 배포할 위치를 자동으로 파악합니다. 예를 들어 단일 서버 충돌로 인해 Pod가 중단되지 않도록 별도의 서버에서 각 Pod를 실행합니다. 앱), 리소스(예: 컨테이너에 필요한 포트, CPU, 메모리 및 기타 리소스를 사용할 수 있는 서버 선택), 성능(예: 로드가 가장 적고 컨테이너가 가장 적은 서버를 선택) 등 . 또한 Kubernetes는 클러스터를 지속적으로 모니터링하여 항상 3개의 복제본이 실행되고 있는지 확인하고 충돌하거나 응답을 중지하는 Pod를 자동으로 교체합니다.
- 업데이트 배포 방법. Docker 컨테이너의 새 버전을 배포할 때 이전 코드는 세 개의 새 복제본을 롤아웃하고 정상 상태가 될 때까지 기다린 다음 세 개의 이전 복제본을 배포 취소합니다.

단 몇 줄의 YAML만으로도 엄청난 성능을 발휘합니다! kubectl apply -f example-app.yml을 실행하여 Kubernetes에 앱을 배포하도록 지시합니다. 그런 다음 YAML 파일을 변경하고 kubectl Apply를 다시 실행하여 업데이트를 롤아웃할 수 있습니다. Terraform을 사용하여 Kubernetes 클러스터와 클러스터 내의 앱을 모두 관리할 수도 있습니다.

프로비저닝 도구

구성 관리, 서버 템플릿 및 오케스트레이션 도구는 각 서버에서 실행되는 코드를 정의하는 반면 Terraform, CloudFormation, OpenStack Heat 및 Pulumi와 같은 프로비저닝 도구는 서버 자체를 생성합니다. 실제로 프로비저닝 도구를 사용하면 서버뿐만 아니라 데이터베이스, 캐시, 로드 밸런서, 대기열, 모니터링, 서브넷 구성, 방화벽 설정, 라우팅 규칙, SSL(Secure Sockets Layer) 인증서 및 기타 거의 모든 측면을 생성할 수 있습니다. 예를 들어 다음 코드는 Terraform을 사용하여 웹 서버를 배포합니다.

```
resource  "aws_instance" "app" {
  instance_type      =  "t2.micro"
  availability_zone  =  "us-east-2a"
  ami                =  "ami-0fb653ca2d3203ac1"
  user_data  =  <<-EOF
                #!/bin/bash
                sudo  service  apache 2  start
                EOF
}
```

일부 구문에 아직 익숙하지 않더라도 걱정하지 말고 지금은 두 가지 매개변수에만 집중하세요.

- ami
  이 파라미터는 서버에 배포할 AMI의 ID를 지정합니다. 이 매개변수를 PHP, Apache 및 애플리케이션 소스 코드가 포함된 이전 섹션의 web-server.json Packer 템플릿에서 구축된 AMI의 ID로 설정할 수 있습니다.
- user_data
  이것은 웹 서버가 부팅될 때 실행되는 Bash 스크립트입니다. 앞의 코드는 이 스크립트를 사용하여 Apache를 부팅합니다.

즉, 이 코드는 프로비저닝과 서버 템플릿이 함께 작동하는 것을 보여줍니다. 이는 불변 인프라의 일반적인 패턴입니다.

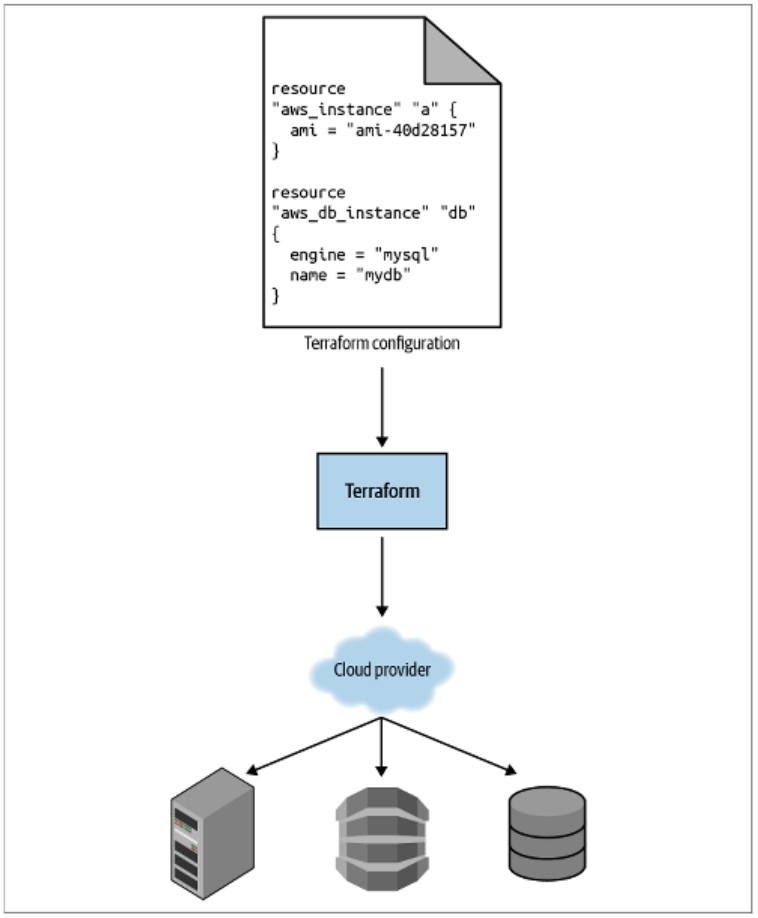


Figure 1-5. You can use provisioning tools with your cloud provider to create servers, databases, load balancers, and all other parts of your infrastructure.

인프라형 코드의 장점

이제 IaC의 다양한 특징을 모두 살펴보았으므로 물어볼 좋은 질문은 '왜 굳이 귀찮게 해야 합니까?'입니다. 왜 수많은 새로운 언어와 도구를 배우고 더 많은 코드를 관리해야 합니까? 대답은 코드가 강력하다는 것입니다. 수동 작업을 코드로 전환하는 선행 투자의 대가로 소프트웨어 제공 능력이 크게 향상됩니다. 2016년 DevOps 현황 보고서에 따르면 IaC와 같은 DevOps 방식을 사용하는 조직은 200배 더 자주 배포하고, 24배 더 빠르게 오류를 복구하며, 리드 타임이 2,555배 더 짧습니다. 인프라가 코드로 정의되면 다양한 소프트웨어 엔지니어링 방법을 사용하여 다음을 포함한 소프트웨어 제공 프로세스를 획기적으로 개선할 수 있습니다.

- 셀프서비스
  코드를 수동으로 배포하는 대부분의 팀에는 배포 작업을 수행하는 데 필요한 모든 정보를 알고 프로덕션에 액세스할 수 있는 유일한 시스템 관리자가 소수 있습니다. 이는 회사가 성장함에 따라 주요 병목 현상이 됩니다. 인프라가 코드로 정의된 경우 전체 배포 프로세스를 자동화할 수 있으며 개발자는 필요할 때마다 자체 배포를 시작할 수 있습니다.
- 속도와 안전성
  배포 프로세스가 자동화되면 자동화된 프로세스가 더 일관되고 반복 가능하며 수동 오류가 발생하지 않는다는 점을 고려하면 컴퓨터가 사람보다 훨씬 빠르고 안전하게 배포 단계를 수행할 수 있으므로 훨씬 더 빨라질 것입니다.
- 공유 정신
  인프라 상태가 단일 시스템 관리자의 머리에 잠겨 있고 해당 시스템 관리자가 휴가를 가거나 회사를 떠나거나 버스에 치인 경우 갑자기 자신의 인프라를 더 이상 관리할 수 없다는 것을 깨닫게 될 수 있습니다. 반면, 인프라가 코드로 정의된 경우 인프라 상태는 누구나 읽을 수 있는 소스 파일에 있습니다. 즉, IaC는 문서 역할을 하여 시스템 관리자가 휴가를 떠나더라도 조직의 모든 사람이 작업 방식을 이해할 수 있도록 해줍니다.

- 버전 관리
  IaC 소스 파일을 버전 제어에 저장할 수 있습니다. 즉, 인프라의 전체 기록이 이제 커밋 로그에 캡처됩니다. 이는 문제 디버깅을 위한 강력한 도구가 됩니다. 문제가 발생할 때마다 첫 번째 단계는 커밋 로그를 확인하고 인프라에서 변경된 내용을 찾는 것이고, 두 번째 단계는 간단히 되돌려 문제를 해결하는 것일 수 있기 때문입니다. 이전의 정상적으로 동작한 버전의 IaC 코드로 복원할 수 있습니다.
- 검증
  인프라 상태가 코드에 정의된 경우 모든 단일 변경 사항에 대해 코드 검토를 수행하고, 자동화된 테스트 모음을 실행하고, 정적 분석 도구를 통해 코드를 전달할 수 있습니다. 이 모든 방법은 결함의 위험을 크게 줄이는 것으로 알려져 있습니다.
- 재사용
  인프라를 재사용 가능한 모듈로 패키징하여 모든 환경의 모든 제품에 대한 모든 배포를 처음부터 수행하는 대신 알려지고 문서화되었으며 실전 테스트를 거친 부분을 기반으로 구축할 수 있습니다.
- 행복
  IaC를 사용해야 하는 또 다른 매우 중요하면서도 종종 간과되는 이유가 있습니다. 바로 행복입니다. 코드를 배포하고 인프라를 수동으로 관리하는 것은 반복적이고 지루합니다. 개발자와 시스템 관리자는 이러한 유형의 작업에 창의성, 도전, 인정이 전혀 포함되지 않기 때문에 분개합니다. 몇 달 동안 코드를 완벽하게 배포할 수 있으며, 어느 날 코드를 엉망으로 만들기 전까지는 아무도 눈치 채지 못할 것입니다. 이는 스트레스가 많고 불쾌한 환경을 조성합니다. IaC는 컴퓨터가 가장 잘하는 일(자동화)을 수행하고 개발자가 가장 잘하는 일(코딩)을 수행할 수 있도록 하는 더 나은 대안을 제공합니다.

이제 IaC가 왜 중요한지 이해했으므로 다음 질문은 Terraform이 최고의 IaC 도구인지 여부입니다. 이에 대한 답을 얻기 위해 먼저 Terraform의 작동 방식에 대한 간단한 입문서를 살펴본 다음 이를 Chef, Puppet 및 Ansible과 같은 다른 인기 있는 IaC 옵션과 비교해 보겠습니다.

테라폼 동작 방법

다음은 Terraform의 작동 방식에 대한 높은 수준의 다소 단순화된 설명입니다. Terraform은 HashiCorp에서 만들고 Go 프로그래밍 언어로 작성된 오픈 소스 도구입니다. Go 코드는 당연히 terraform이라는 단일 바이너리(또는 지원되는 운영 체제 각각에 대해 하나의 바이너리)로 컴파일됩니다. 이 바이너리를 사용하여 랩탑이나 빌드 서버 또는 거의 모든 다른 컴퓨터에서 인프라를 배포할 수 있으며 이를 위해 추가 인프라를 실행할 필요가 없습니다. 그 이유는 내부적으로 Terraform 바이너리가 사용자를 대신하여 AWS, Azure, Google Cloud, DigitalOcean, OpenStack 등과 같은 하나 이상의 공급자에게 API 호출을 하기 때문입니다. 이는 Terraform이 해당 공급자가 이미 API 서버를 위해 실행하고 있는 인프라와 해당 공급자와 함께 이미 사용하고 있는 인증 메커니즘(예: AWS에 대해 이미 보유하고 있는 API 키)을 활용하게 된다는 것을 의미합니다.

Terraform은 어떤 API 호출을 해야 하는지 어떻게 알 수 있나요? 대답은 생성하려는 인프라를 지정하는 텍스트 파일인 Terraform 구성을 생성하는 것입니다. 이러한 구성은 "코드로서의 인프라"의 "코드"입니다. Terraform 구성의 예는 다음과 같습니다.

```
resource  "aws_instance" "example"  {
  ami           =  "ami-0fb653ca2d3203ac1"
  instance_type  =  "t2.micro"
}
resource  "google_dns_record_set" "a"  {
  name          =  "demo.google-example.com"
  managed_zone  =  "example-zone"
  type          =  "A"
  ttl           =  300
```

```
  rrdatas        =  [ aws_instance.example.public_ip ]
}
```

HCL (Hashicorp Configuration Launguage) 는 Terraform 을 실행하기 위해 미리 선언하는 리소스들의 목록들을 나열한 코드입니다. 리소스의 정의를 보다 유연하고 편리하게 만들기 위해 개발한 언어 이며 확장자는 .tf 파일 이거나 혹은 Json 기반의 .tf.json 파일도 있습니다. 블록과 인수 그리고 표현식을 작성해서 구성하려는 리소스들의 상세한 속성을 작성할 수 있습니다.

이전에 Terraform 코드를 본 적이 없더라도 읽는 데 큰 어려움이 없을 것입니다. 이 스니펫은 Terraform에 AWS에 대한 API 호출을 수행하여 서버를 배포한 다음 Google Cloud에 대한 API 호출을 수행하여 AWS 서버의 IP 주소를 가리키는 DNS(Domain Name System) 항목을 생성하도록 지시합니다. 단 하나의 간단한 구문으로 Terraform을 사용하면 여러 클라우드 제공자 간에 상호 연결된 리소스를 배포할 수 있습니다. Terraform 구성 파일에서 전체 인프라(서버, 데이터베이스, 로드 밸런서, 네트워크 토폴로지 등)를 정의하고 해당 파일을 버전 제어에 커밋할 수 있습니다. 그런 다음 terraform apply 와 같은 특정 Terraform 명령을 실행하여 해당 인프라를 배포합니다. Terraform 바이너리는 코드를 구문 분석하고 이를 코드에 지정된 클라우드 공급자에 대한 일련의 API 호출로 변환하며 그림과 같이 사용자를 대신하여 해당 API 호출을 최대한 효율적으로 수행합니다.

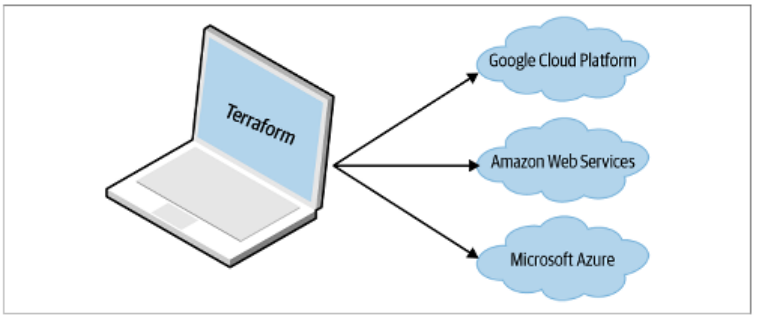


Figure 1-6. Terraform is a binary that translates the contents of your configurations into API calls to cloud providers.

팀의 누군가가 인프라를 변경해야 하는 경우 서버에서 인프라를 수동 및 직접 업데이트하는 대신 Terraform 구성 파일을 변경하고 자동화된 테스트 및 코드 검토를 통해 해당 변경 사항을 검증하고 업데이트된 코드를 커밋합니다. 버전 제어를 수행한 다음 terraform apply 명령을 실행하여 Terraform이 변경 사항을 배포하는 데 필요한 API 호출을 수행하도록 합니다.

클라우드 공급자 간 투명한 이식성 Terraform은 다양한 클라우드 공급자를 지원하기 때문에 발생하는 일반적인 질문은 둘 사이의 투명한 이식성을 지원하는지 여부입니다. 예를 들어 Terraform을 사용하여 AWS에서 다수의 서버, 데이터베이스, 로드 밸런서 및 기타 인프라를 정의한 경우 단 한 번에 Azure 또는 Google Cloud와 같은 다른 클라우드 제공업체에 정확히 동일한 인프라를 배포하도록 Terraform에 지시할 수 있습니까? 몇 가지 명령? 현실은 클라우드 공급자가 동일한 유형의 인프라를 제공하지 않기 때문에 다른 클라우드 공급자에 "완전히 동일한 인프라"를 배포할 수 없다는 것입니다! AWS가 제공하는 서버, 로드 밸런서, 데이터베이스는 기능, 구성, 관리, 보안, 확장성, 가용성, 관찰 가능성 등의 측면에서 Azure 및 Google Cloud의 서버, 로드 밸런서 및 데이터베이스와 매우 다릅니다. 특히 한 클라우드 제공업체의 기능이 다른 클라우드 제공업체에는 전혀 존재하지 않는 경우가 많기 때문에 이러한 차이점을 "투명하게" 설명할 수 있는 쉬운 방법은 없습니다. Terraform의 접근 방식은 해당 공급자의 고유한 기능을 활용하여 각 공급자에 특정한 코드를 작성할 수 있도록 하는 동시에 모든 공급자에 대해 동일한 언어, 도구 세트 및 IaC 방식을 사용할 수 있도록 하는 것입니다.

테라폼 실습

- 테라폼 설치

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashico
sudo yum -y install terraform
```

- AWS 연결

```
AWS CONFIGURE
```

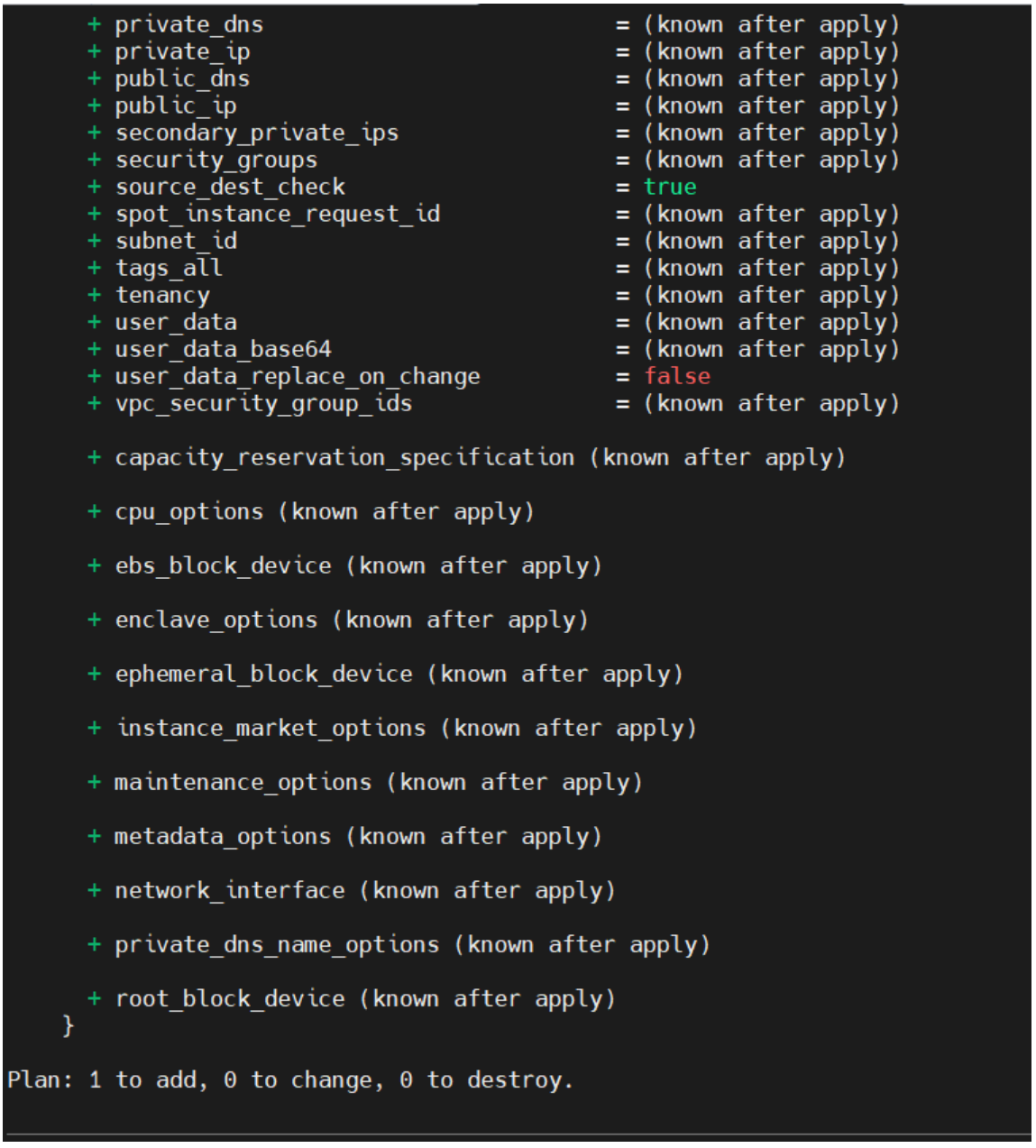
iam-user access key 생성하고 연결해준다.

- main.tf

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

resource "aws_instance" "example"{
  ami = "ami-062cf18d655c0b1e8"
  instance_type = "t2.micro"
}
```

후에 terraform init을 해준다.

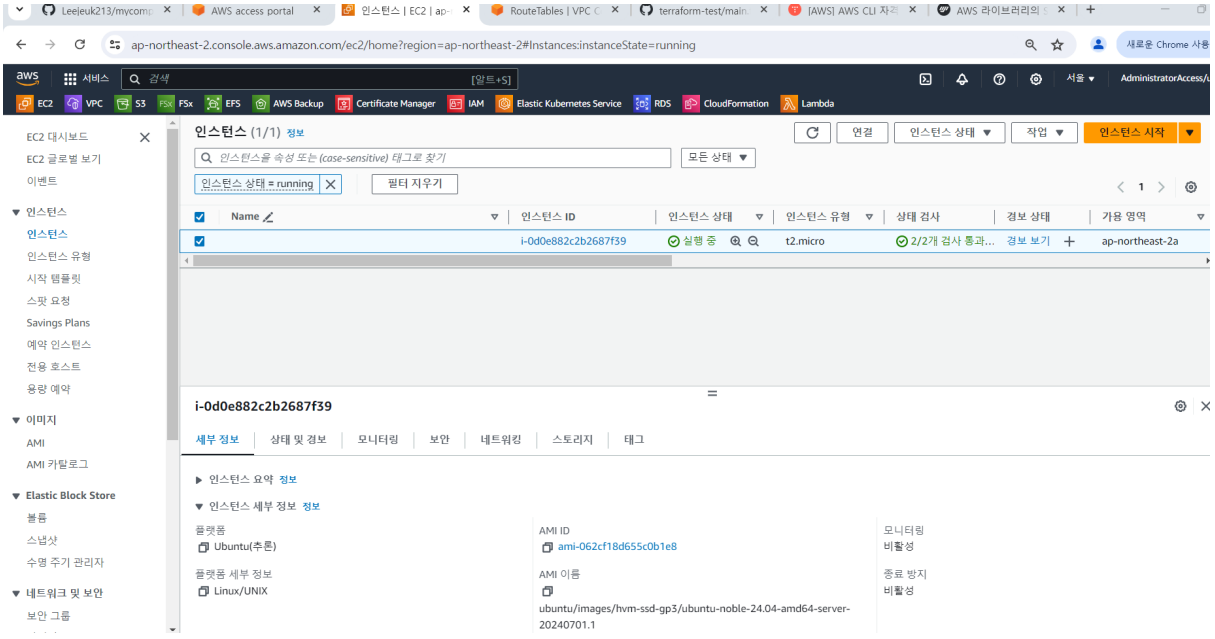
- terraform plan



- terraform apply

```
Plan: 1 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_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Still creating... [30s elapsed]
aws_instance.example: Creation complete after 32s [id=i-0d0e882c2b2687f39]

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

- 인스턴스 생성 확인



> 시간 동기화가 잘 안되면 오류가 발생할 수 있음

- tag 정보 추가 후 재 apply

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

```
resource "aws_instance" "example"{
  ami = "ami-062cf18d655c0b1e8"
  instance_type = "t2.micro"
  tags = {
    Name = "terraform-example"
 }
}
```

- 결과

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # aws_instance.example will be updated in-place
  ~ resource "aws_instance" "example" {
        id                                   = "i-0d0e882c2b2687f39"
      ~ tags                                 = {
          + "Name" = "terraform-example"
        }
      ~ tags_all                             = {
          + "Name" = "terraform-example"
        }
        # (38 unchanged attributes hidden)

        # (8 unchanged blocks hidden)
    }

Plan: 0 to add, 1 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
```

```
Plan: 0 to add, 1 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_instance.example: Modifying... [id=i-0d0e882c2b2687f39]
aws_instance.example: Modifications complete after 1s [id=i-0d0e882c2b2687f39]

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

- terraform destroy

```
Plan: 0 to add, 0 to change, 1 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: yes

aws_instance.example: Destroying... [id=i-0d0e882c2b2687f39]
aws_instance.example: Still destroying... [id=i-0d0e882c2b2687f39, 10s elapsed]
aws_instance.example: Still destroying... [id=i-0d0e882c2b2687f39, 20s elapsed]
aws_instance.example: Still destroying... [id=i-0d0e882c2b2687f39, 30s elapsed]
aws_instance.example: Still destroying... [id=i-0d0e882c2b2687f39, 40s elapsed]
aws_instance.example: Destruction complete after 40s

Destroy complete! Resources: 1 destroyed.
```