ECR & ECS

Container Service

❖ 가상화

AWS Container Service

- Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ 프로젝트 생성
    - 가상 환경 생성
      - python3 -m venv ./{your venv name}
      - 가상 환경 활성화
        - Linux, Mac: Source {your venv name}/bin/activate
        - Windows: {your venv name}/Scripts/activate
    - Django 설치 및 실행
      - (venv)$ pip install django djangorestframework
      - (venv)$ django-admin startproject apiserver
      - (venv)$ cd apiserver
      - (venv)$ python manage.py runserver
      - 브라우저에 접속해서 localhost:8000 으로 접속
    - 라이브러리의 의존성을 파일에 내보내기
      - pip freeze > requirements.txt

AWS Container Service

- Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ 프로젝트 설정
    - settings.py 파일 수정

```
ALLOWED_HOSTS = ['*']


# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "rest_framework",
    "apiserver",]
```

AWS Container Service

- Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ 프로젝트 설정
    - settings.py 파일 수정

```
# Internationalization
# https://docs.djangoproject.com/en/4.1/topics/i18n/

LANGUAGE_CODE = "en-us"
TIME_ZONE = "Asia/Seoul"
USE_I18N = True
USE_TZ = True
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.1/howto/static-files/

STATIC_URL = "static/"

# Default primary key field type
# https://docs.djangoproject.com/en/4.1/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"
```

AWS Container Service

- Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ 요청 처리
    - views.py 파일을 생성하고 요청 처리 메서드 작성

```
from rest_framework.response import Response
from rest_framework.decorators import api_view
from rest_framework import status

@api_view(['GET'])
def index(request):
    data = {"result":"success", "data": [{"id":"itstudy", "name":"군계"},
                                         {"id":"ggangpae1", "name":"아담"}]}
    return Response(data, status=status.HTTP_200_OK)
```

AWS Container Service

- Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ 요청 처리
    - urls.py 파일 수정

```
from django.contrib import admin
from django.urls import path
from .views import index
urlpatterns = [
    path("admin/", admin.site.urls),
    path("", index )
]
```

AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ 프로젝트 실행 및 확인
    - 프로젝트 실행: python manage.py runserver
    - 확인

AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Docker Image 생성 및 실행
    - ● Dockerfile 작성

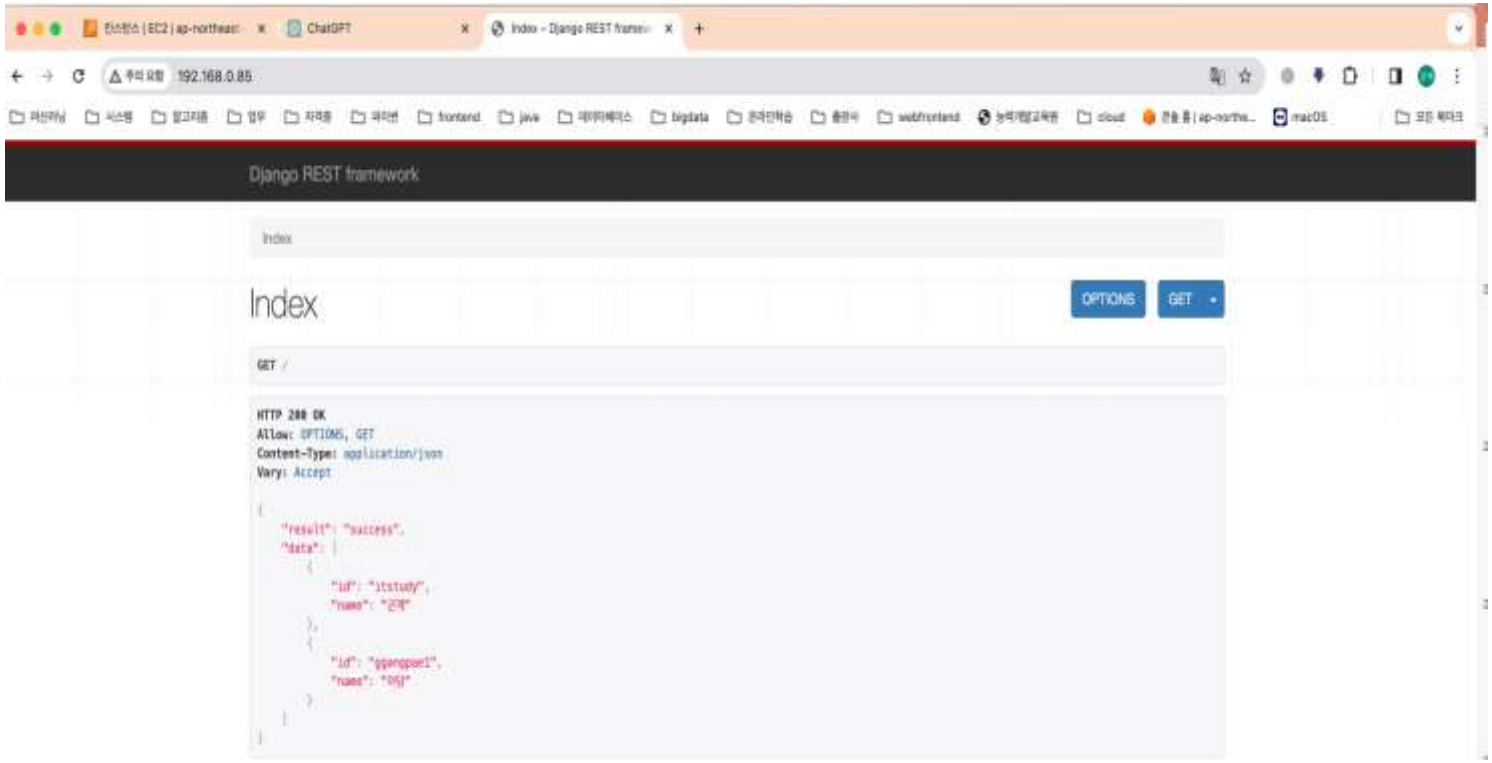
```
FROM --platform=linux/amd64 python:3.8-slim-buster as build

RUN apt-get update ₩
    && apt-get install -y --no-install-recommends ₩
            postgresql-client ₩
               && rm -rf /var/lib/apt/lists/*
WORKDIR /usr/src/app
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . .

EXPOSE 80
CMD ["python", "manage.py", "runserver", "0.0.0.0:80"]
```

AWS Container Service

- Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Docker Image 생성 및 실행
    - 이미지 빌드 후 실행
      - docker build -t apiserver .
      - docker run --name apiserver -p 80:80 -d apiserver:latest

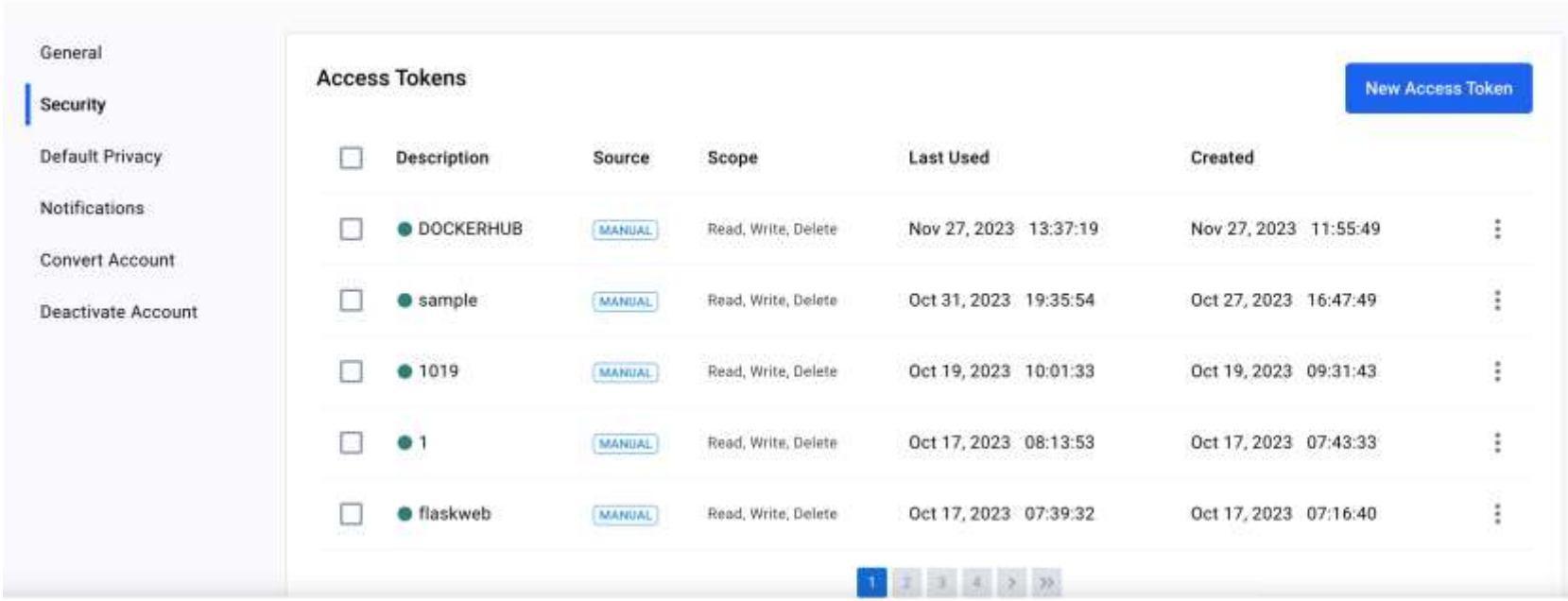
AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Docker Image 생성 및 실행
    - ● 확인



AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - Git Hub에 업로드
    - Docker Hub에 Repository 생성
    - Docker Hub에서 Access Token 생성 – 오른쪽 상단의 아이디 옆을 클릭한 후 [Account Settings] – [Security] – [Access Token] – [New Access Token]



AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - GIT HUB에서 Secret Key 등록: [Settings] – [Secrets and variable] – [Actions] – [New repository secret]
      - ◆ DOCKERHUB_USERNAME
      - ◆ DOCKERHUB_TOKEN

AWS Container Service

- Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - .github/workflows/ 디렉토리에 yaml 파일을 생성하고 push

```
name: django

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - .github/workflows/ 디렉토리에 yaml 파일을 생성하고 push

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Set up Python 3.10
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: |
            python -m pip install --upgrade pip
            pip install -r requirements.txt
```

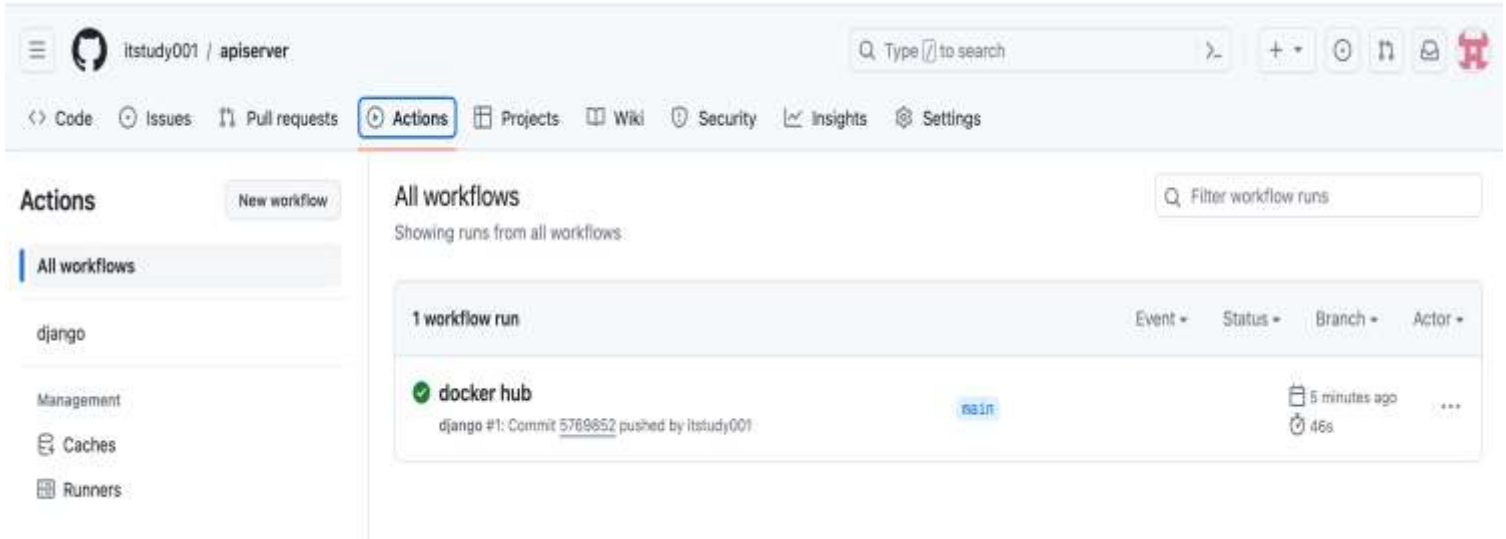
AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - ● .github/workflows/ 디렉토리에 yaml 파일을 생성하고 push

```
- name: Login to DockerHub
  uses: docker/login-action@v1
  with:
    username: ${{secrets.DOCKERHUB_USERNAME}}
    password: ${{secrets.DOCKERHUB_TOKEN}}

- name: build and release to DockerHub
  env:
    NAME: ${{secrets.DOCKERHUB_USERNAME}}
    REPO: apiserver
  run: |
    docker build -t $REPO .
    docker tag $REPO:latest $NAME/$REPO:latest
    docker push $NAME/$REPO:latest
```

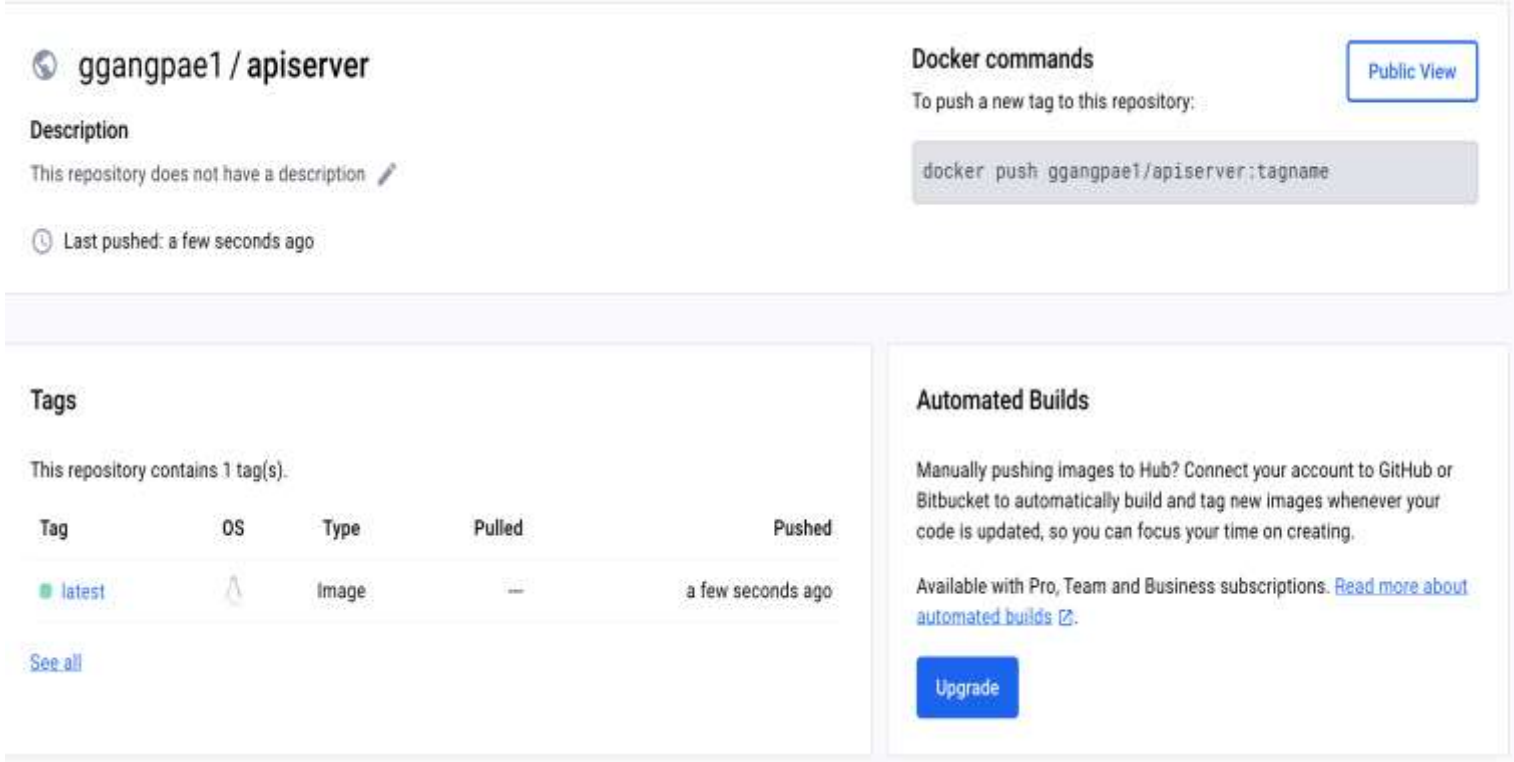
AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - 확인



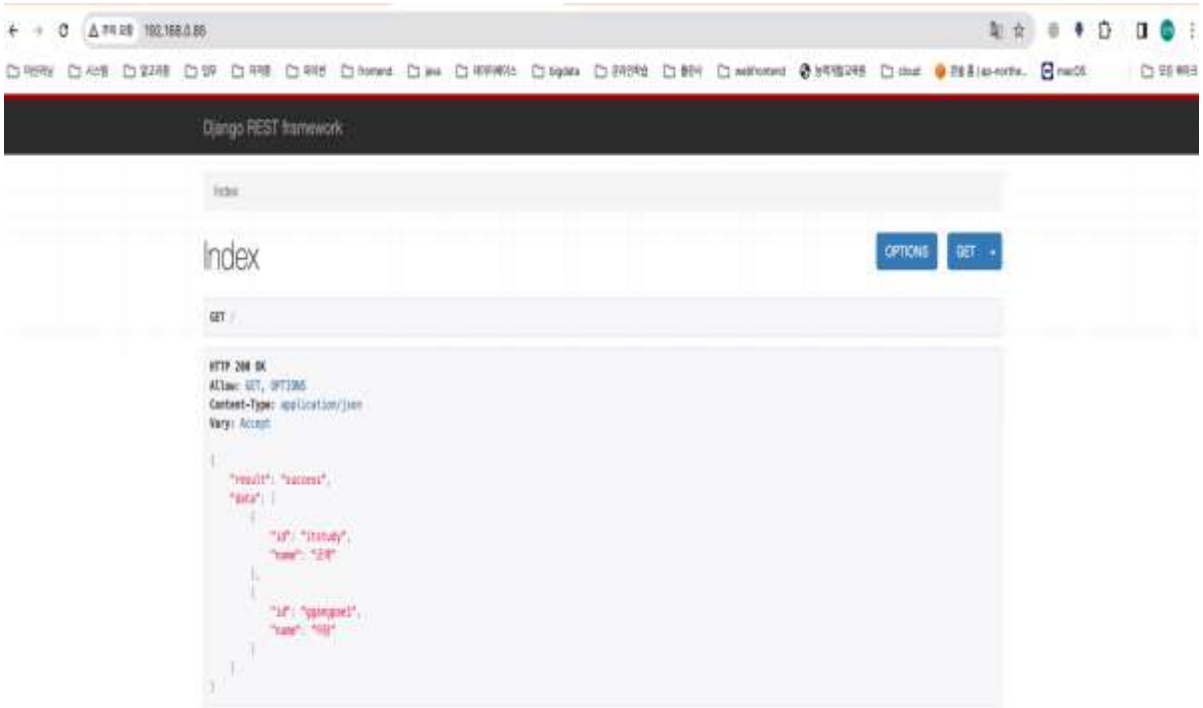
AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - 확인



AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - ● 확인
      - ◆ docker run -dit -p 80:80 --name apiserver ggangpae1/apiserver



AWS Container Service

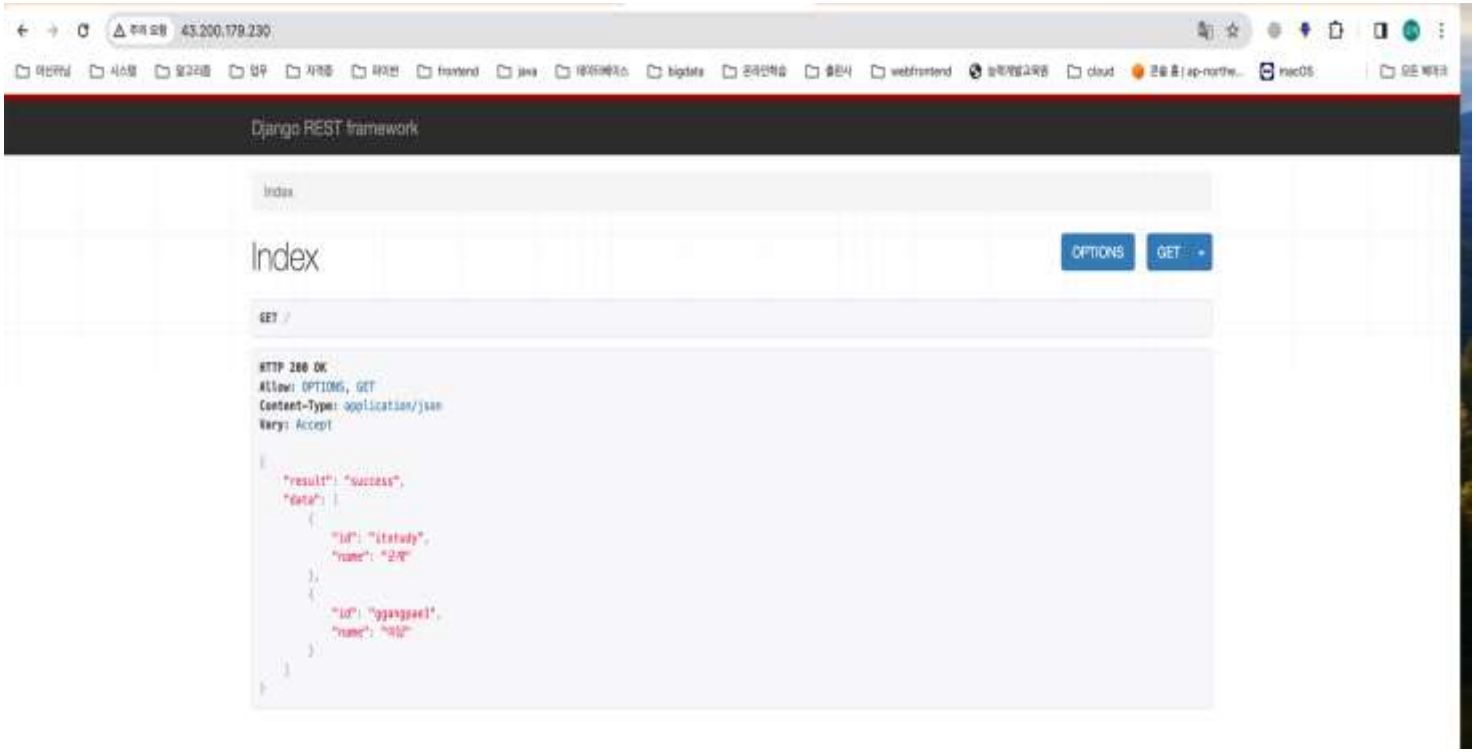
- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ EC2 인스턴스 작업
    - ● Docker 설치
      - ◆ 패키지 업데이트: sudo apt-get update
      - ◆ 패키지 설치: sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-properties-common
      - ◆ Docker의 공식 GPG키를 추가
        - ▪ 경고 발생: curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –
        - ▪ 경고 발생하지 않음: curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
      - ◆ Docker의 공식 apt 저장소를 추가
        - ▪ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
        - ▪ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 인스턴스에 배포
  - ✓ EC2 인스턴스 작업
    - ● Docker 설치
      - ◆ 시스템 패키지 업데이트: sudo apt-get update
      - ◆ Docker 설치: sudo apt-get install docker-ce docker-ce-cli containerd.io
      - ◆ Docker 버전 확인: sudo docker version
      - ◆ Docker 서비스 등록 및 실행
        - ▪ sudo systemctl enable docker
        - ▪ sudo systemctl start docker

AWS Container Service

- ❖ Django 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ 이미지 다운로드 및 실행
    - sudo docker run -p 80:80 -d --rm ggangpae1/apiserver
  - ✓ 브라우저에서 퍼블릭 IP로 확인



AWS Container Service

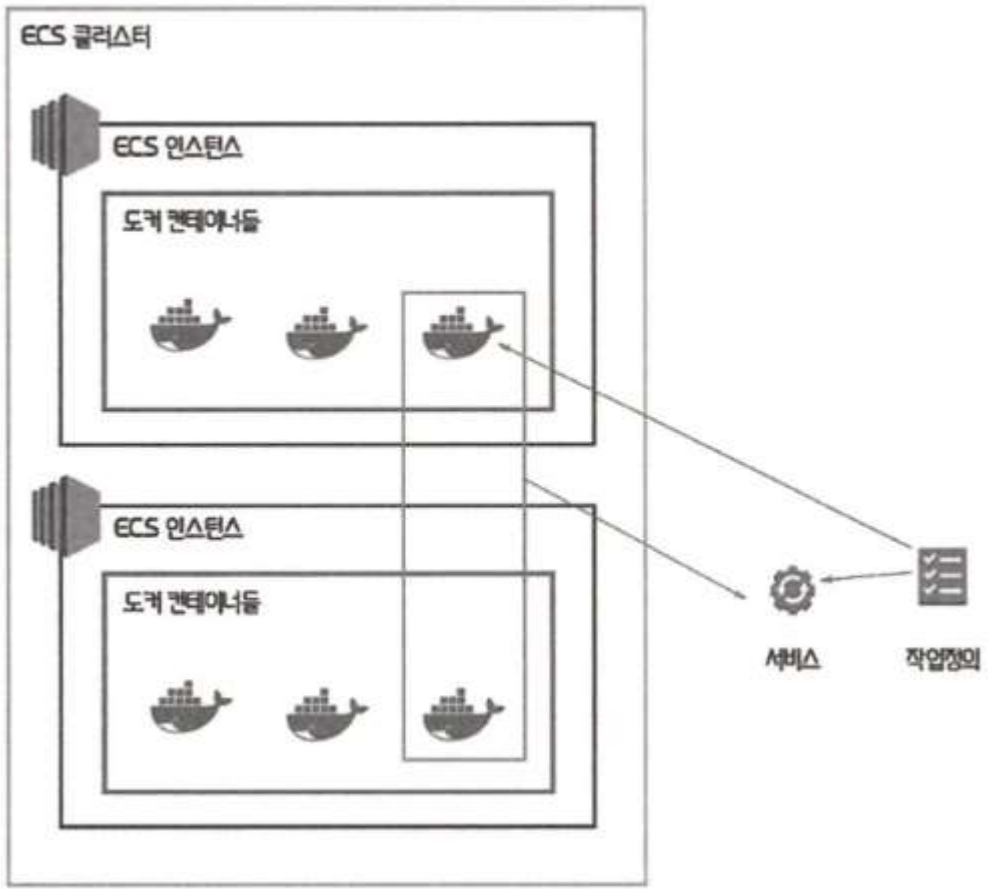
- ECR
  - ✓ Docker Hub와 비슷한 개념으로 Amazon Elastic Container Registry의 약자로 안전하고 확장 가능하고 신뢰할 수 있는 AWS 관리형 컨테이너 이미지 레지스트리 서비스
  - ✓ Docker Hub와 동일하다고 볼 수 있지만 장점으로는 S3로 Docker Image를 관리하므로 고가용성을 보장하고 AWS IAM 인증을 통해 이미지 push/pull에 대한 권한 관리가 가능
  - ✓ Private 과 Public Repository 모두 가능

AWS Container Service

- ECS
  - ✓ 개요
    - Docker와 같은 컨테이너 가상화 도구를 처음 사용했을 때 컨테이너 기술의 장점 과 배포의 어려움을 동시에 느낌
    - 운영 중인 Docker 컨테이너나 인스턴스가 몇 개 정도라면 오케스트레이션 도구까지는 필요 없겠지만 수십 개에서 수백 개가 필요하다면 어디에 어떤 컨테이너가 실행되고 있고 어떤 문제점이 있는지 모니터링하고 관리할 수 있는 도구가 필요한데 이러한 도구가 컨테이너 오케스트레이션(Container Orchestration)
    - Docker에서 만든 스웜(Swarm), 구글의 노하우가 담긴 Kubernetes, 하시코프의 Nomad, AWS에서 제공하는 ECS, EKS 등 다양한 오케스트레이션 도구들이 있음
    - AWS ECS(Elastic Container Service)는 AWS 컴퓨팅 서비스를 제공하는 오케스트레이션 서비스로 AWS 환경에 최적화되어 있음
    - AWS Cloud 환경에서 사용하는 것을 전제로 하기 때문에 Docker의 모든 기능을 사용하기에는 일부 제약이 있으며 세세한 컨트롤이 어려움
    - 구글에서 만든 Kubernetes는 오픈 소스로 개발되었기 때문에 사용자가 원하는 대로 변형하고 수정할 수 있지만 ECS는 Auto Scaling Task 기반 IAM 지원, Application Load Balancer 지원, Service Discovery, AWS Batch Release, Fargate 지원 등 AWS에 맞춤화된 기능 등을 추가하며 최근 활용도가 더욱 높아지고 있음

AWS Container Service

- ECS
  - ✓ 구조

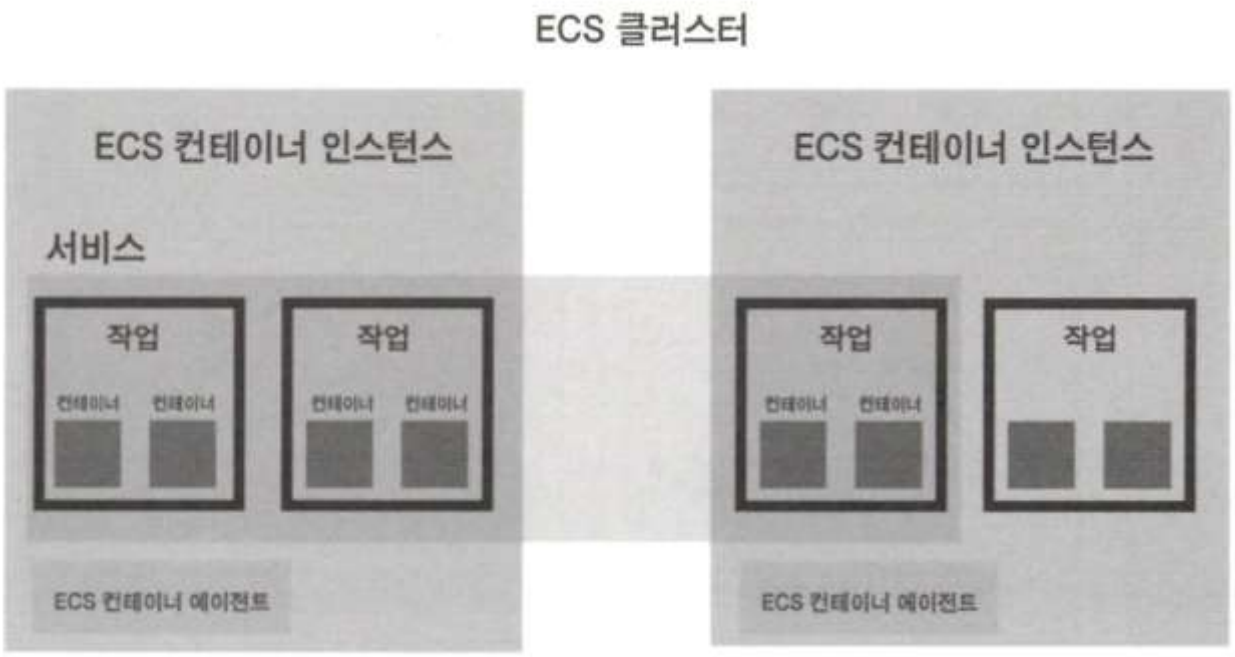


AWS Container Service

- ECS
  - ✓ 구성 요소
    - ● 클러스터
      - ◆ ECS의 가장 기본적인 단위는 클러스터
      - ◆ 클러스터는 Docker 컨테이너를 실행할 수 있는 가상의 공간
      - ◆ 클러스터는 프로젝트나 컨테이너의 성격에 따라서 나눠질 수 있음
      - ◆ ECS 클러스터는 기본적으로 EC2와 같은 컴퓨팅 자원을 기본적으로 포함하지 않은 논리적인 단위이므로 컴퓨팅 자원이 없는 빈 클러스터를 만드는 것도 가능
      - ◆ EC2에 ECS-Client라는 서비스를 실행해서 특정 클러스터에 연결할 수 있는데 이렇게 클러스터에 연결된 EC2 인스턴스를 컨테이너 인스턴스라고 부름
      - ◆ ECS-Client는 컨테이너 인스턴스의 자원을 모니터링 및 관리하고 클러스터로 요청된 컨테이너들을 적절하게 실행하는 역할을 수행

AWS Container Service

- ECS
  - ✓ 구성 요소
    - ● 클러스터
      - ◆ ECS 클러스터 와 클러스터 인스턴스



AWS Container Service

- ECS
  - ✓ 구성 요소
    - 작업: pod
      - ECS에서 컨테이너를 실행하는 최소 단위는 작업
      - 클러스터 상에서 동작하는 1개 혹은 1개 이상의 컨테이너들을 지칭
      - 노드 서버를 위한 Docker 컨테이너와 데이터베이스를 위한 Docker 컨테이너가 필요하다면 각각 개별 작업으로 정의할 수 도 있고 하나의 작업으로 정의할 수도 있음
      - 같은 하나의 작업으로 묶여 있다면 서로 네트워크 통신을 할 수 있음
    - 작업 정의: docker-compose.yml
      - Docker 컨테이너를 실행하기 위해 정의한 도면
      - 컨테이너의 이미지, CPU/메모리 리소스 할당 설정, 포트 매핑, 볼륨 설정 같은 것들이 포함되며 기존 Docker 명령에서 가능했던 대부분 옵션이 설정 가능
      - 작업 정의를 바탕으로 작업 즉 Docker 컨테이너들을 실행할 수 있음

AWS Container Service

- ECS
  - ✓ 구성 요소
    - 서비스: deployment
      - 작업을 하나의 오케스트레이션 단위로 묶은 것이 서비스
      - 작업은 일반적으로 작업 개수로 정의하지만 서비스는 로드 밸런서 와 오토 스케일링을 연결하여 애플리케이션 단위로 조절
    - 컨테이너 인스턴스
      - ECS는 컨테이너 배포를 EC2 인스턴스 기반에 올리도록 설계되어 있음
      - 컨테이너를 동작시킬 컴퓨터를 지칭하지만 파게이트 서비스는 컨테이너 인스턴스를 사용자가 직접 설정하거나 만들어줄 필요없이 AWS에서 관리 해주기 때문에 컨테이너 인스턴스를 필요로 하지 않음

AWS Container Service

- Fargate
  - ✓ AWS 파게이트(Fargate)는 AWS에서 2017년에 발표한 서비스 – 서버리스 서비스
  - ✓ 파게이트는 AWS의 매니지드 컨테이너 오케스트레이션 서비스인 ECS와 EKS를 기반으로 작동하는 서비스
  - ✓ 특징은 Docker 컨테이너를 EC2 인스턴스 없이 독립적으로 실행할 수 있게 함
  - ✓ EC2보다 컴퓨팅 성능을 더 세세하게 선택할 수 있으며 태스크 단위에서 IAM 롤이나 네트워크 인터페이스를 부여하는 것도 가능
  - ✓ 람다 서비스보다는 저렴하지만 EC2보다는 비용이 비싸기 때문에 AWS 람다 와 EC2 기반의 ECS 컨테이너의 중간에 위치한 서비스라고 할 수 있음
  - ✓ 컨테이너 오케스트레이션은 개발자 입장에서 애플리케이션 배포에 새로운 시대를 염
  - ✓ AWS ECS, AWS EKS, Kubernetes, Docker Swarm, Apache Methos 와 같은 오케스트레이션을 통한 컨테이너 애플리케이션 배포는 빠르게 자리를 잡아가고 있음
  - ✓ AWS에서 오케스트레이션 툴을 사용한다면 AWS의 EC2와 같은 컴퓨팅 자원을 묶어서 클러스터로 관리
  - ✓ 컴퓨팅 자원을 관리하는 작업은 클러스터를 운영하는 데 있어서 가장 까다로운 작업 중 하나
  - ✓ 파게이트가 비록 AWS 람다를 대체하기 위한 서비스는 아니지만 파게이트를 사용하면 AWS 람다와 같이 컴퓨팅 자원에 대한 추가적인 관리 없이 애플리케이션 실행이 가능
  - ✓ 컨테이너를 기반으로 하기 때문에 람다에서 지원하는 언어들보다 훨씬 더 다양한 환경을 그대로 이식하는 것이 가능

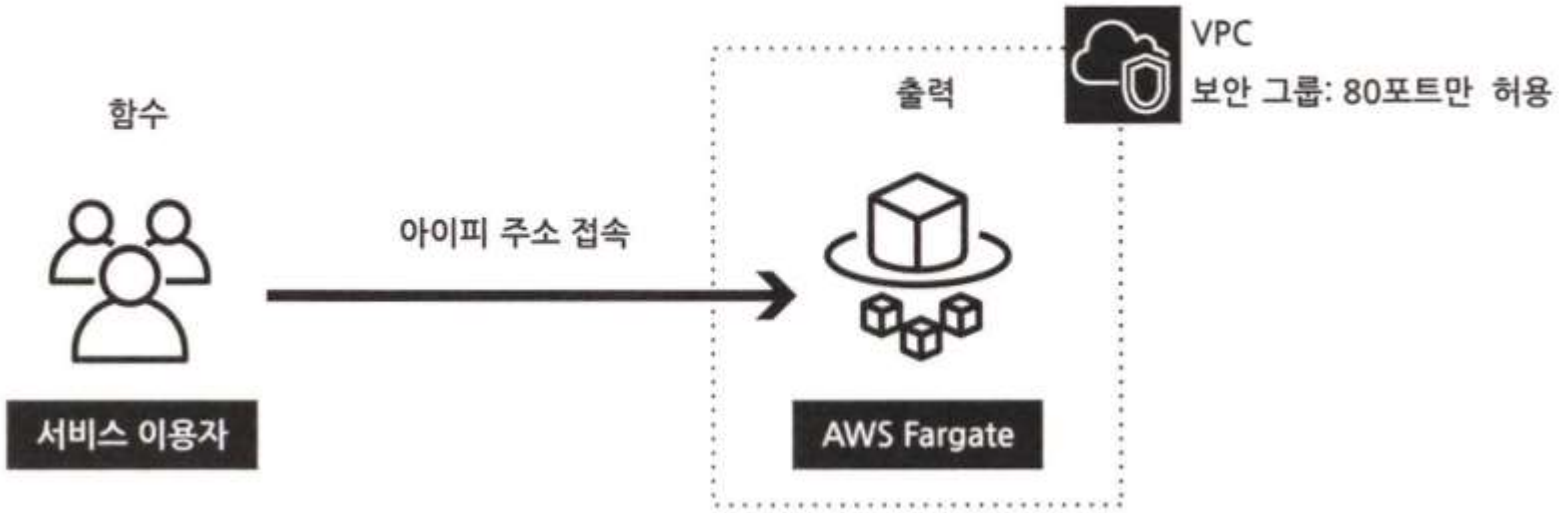
AWS Container Service

- Fargate
  - ✓ EC2 인스턴스보다 vCPU와 RAM을 세세하게 설정할 수 있으며 컨테이너에 IAM 롤을 지정하거나 ENI를 설정할 수도 있음
  - ✓ 현재는 서울 리전에서도 파게이트 서비스를 사용할 수 있음
  - ✓ ECS를 기반으로만 사용할 수 있음
  - ✓ AWS 컨테이너 서비스

AWS Container Service

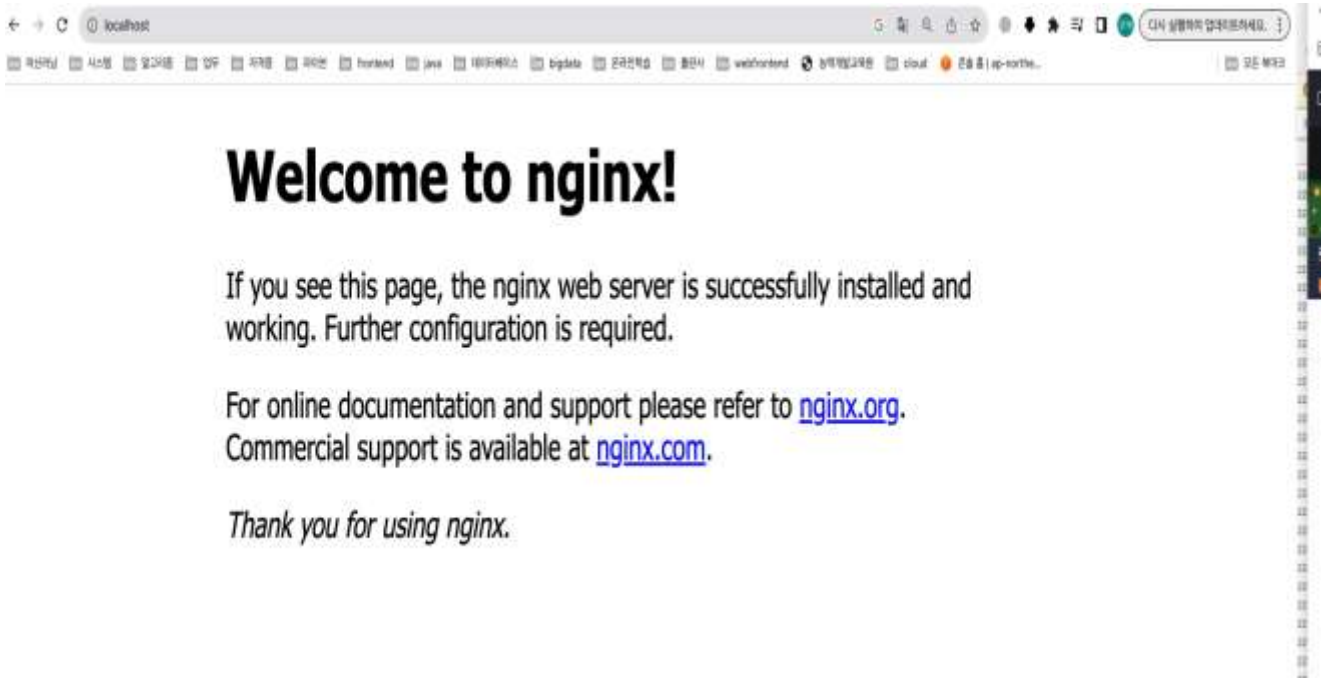
❖ ECS 컨테이너 서비스 구축

✓ 아키텍쳐



AWS Container Service

- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ nginx 이미지 확인
    - 컨테이너 생성: docker run -it -p 80:80 nginx
    - 브라우저에서 확인: localhost

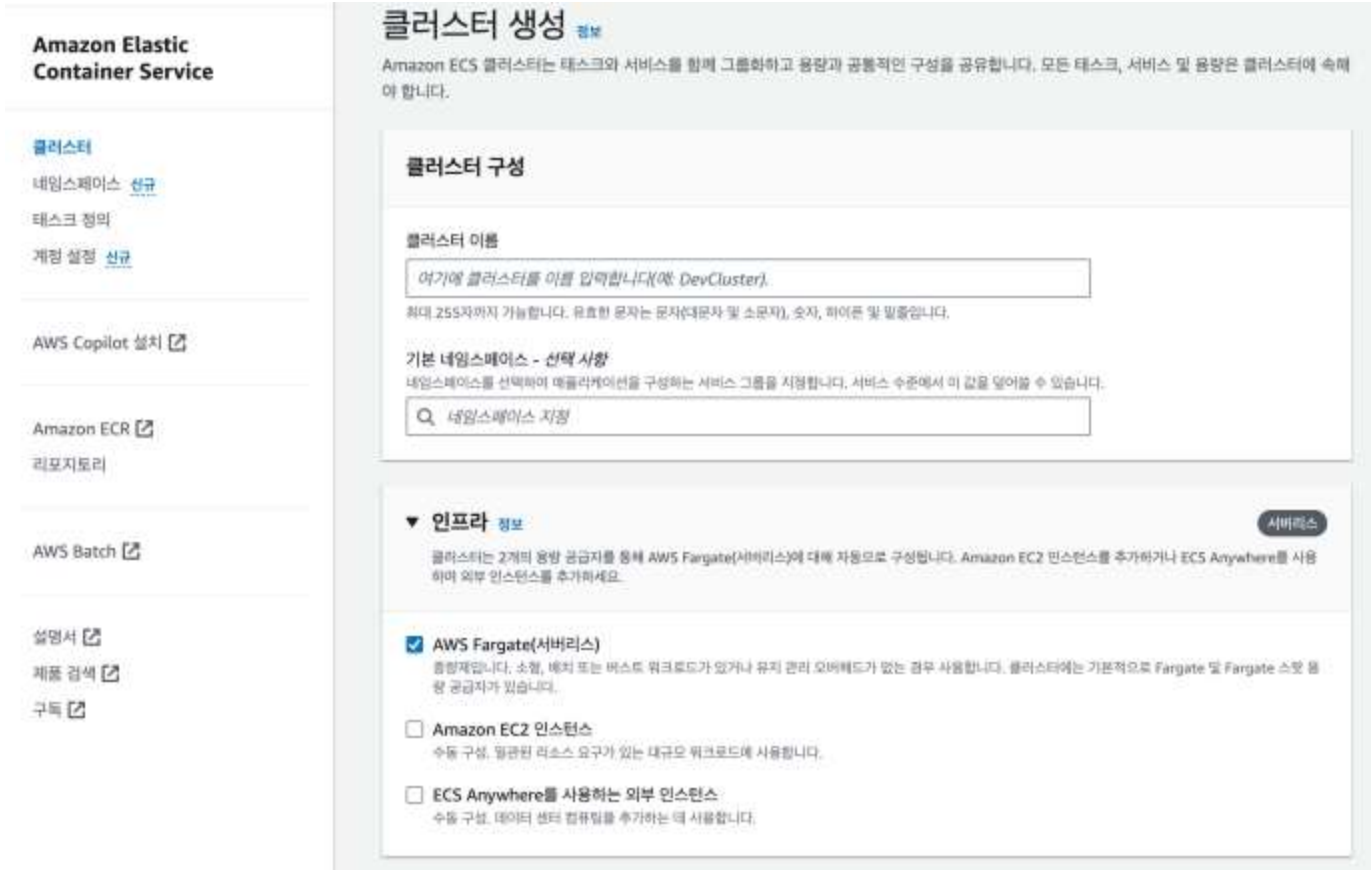


AWS Container Service

- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 클러스터 생성

AWS Container Service

- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 클러스터 생성

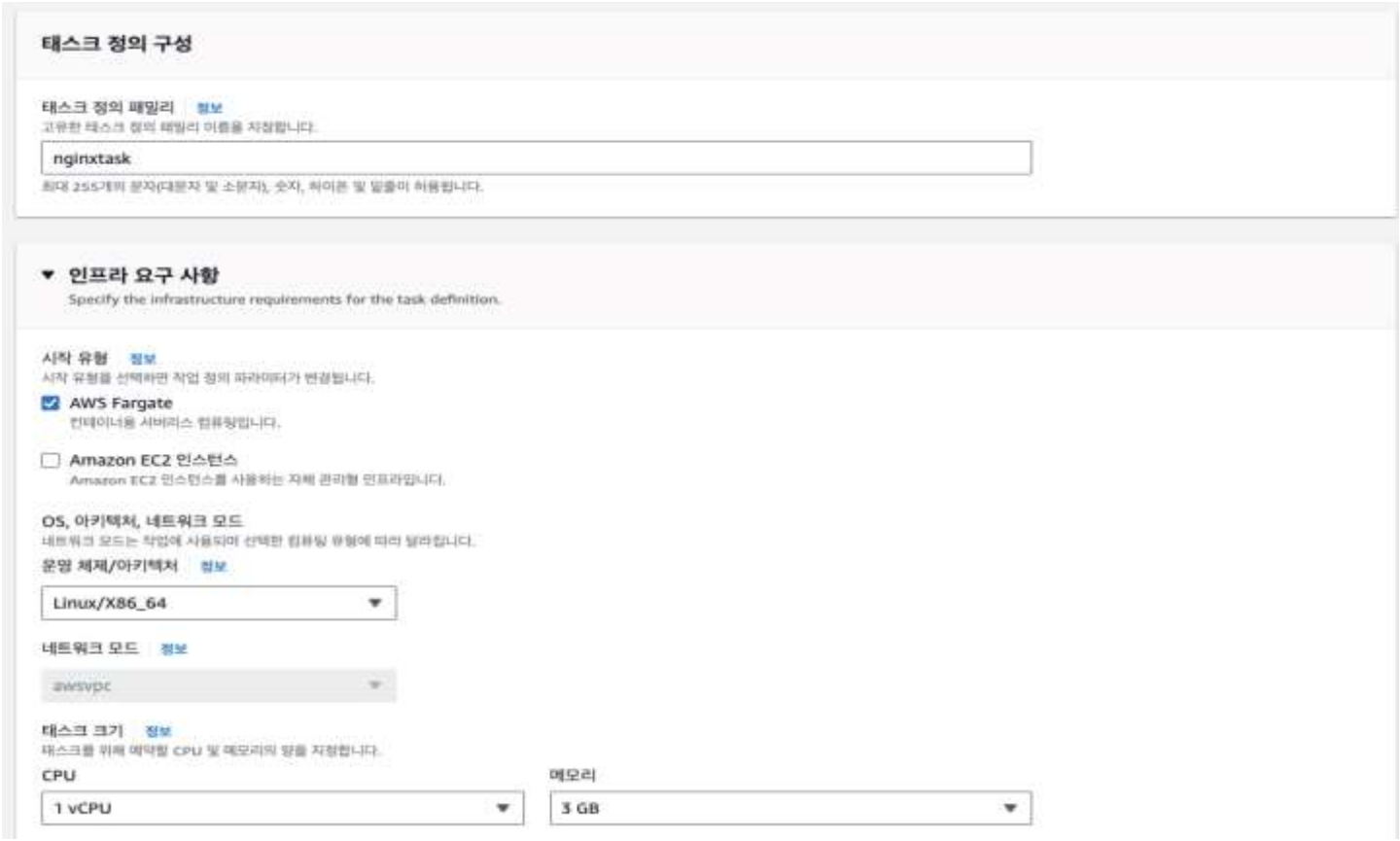


AWS Container Service

- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 태스크 생성: 태스크 정의 클릭

AWS Container Service

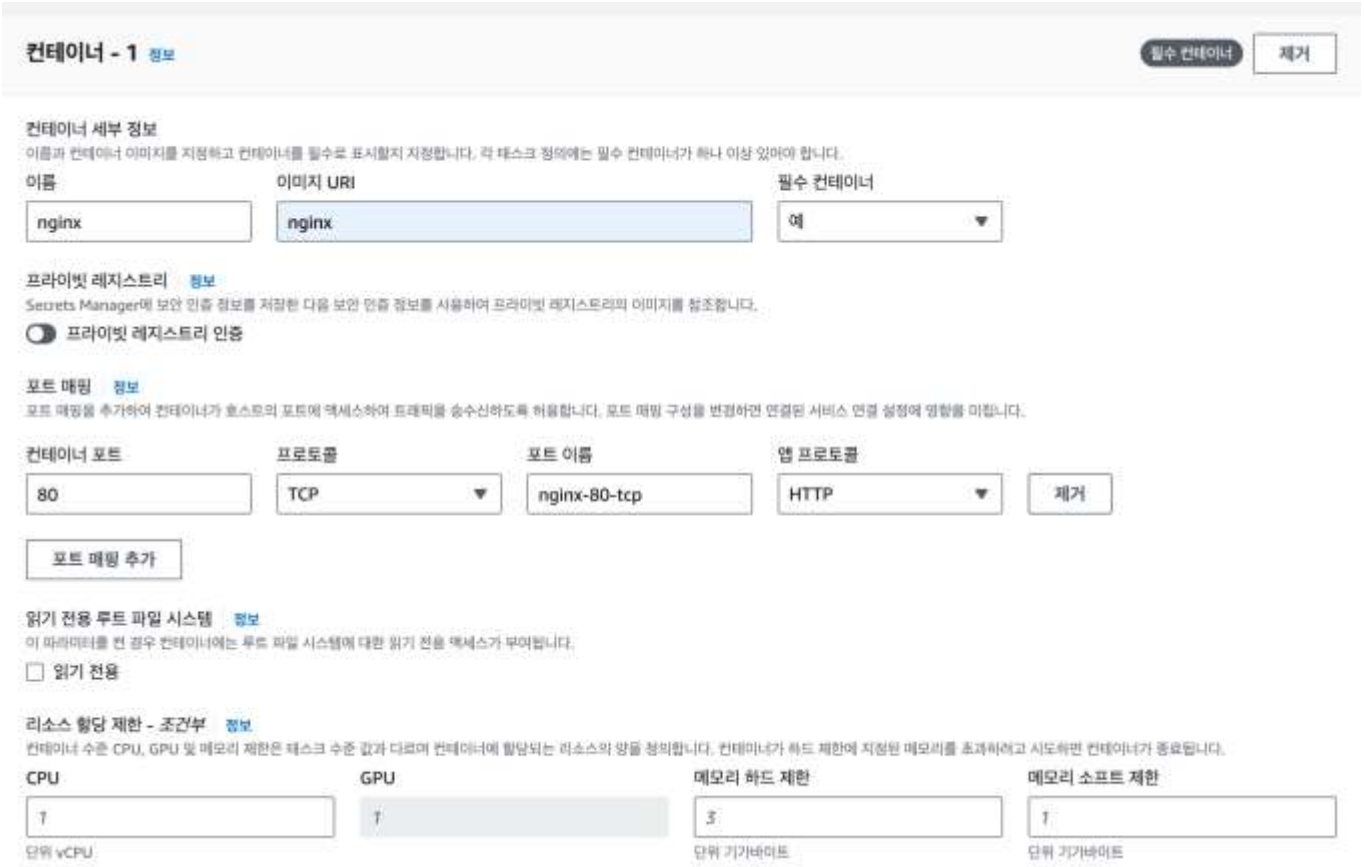
- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 태스크 생성



AWS Container Service

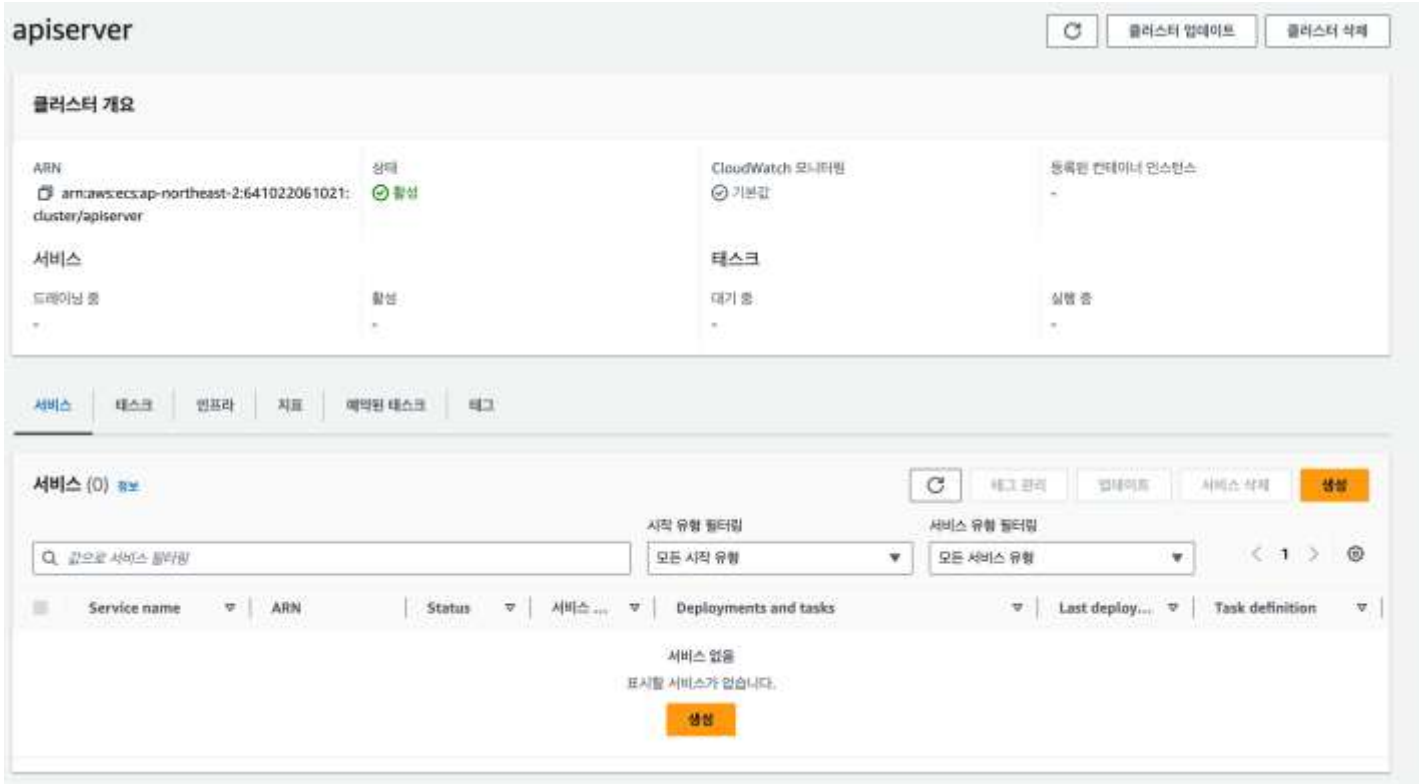
❖ Nginx ECS 배포 – 태스크를 이용한 배포

✓ 태스크 생성



AWS Container Service

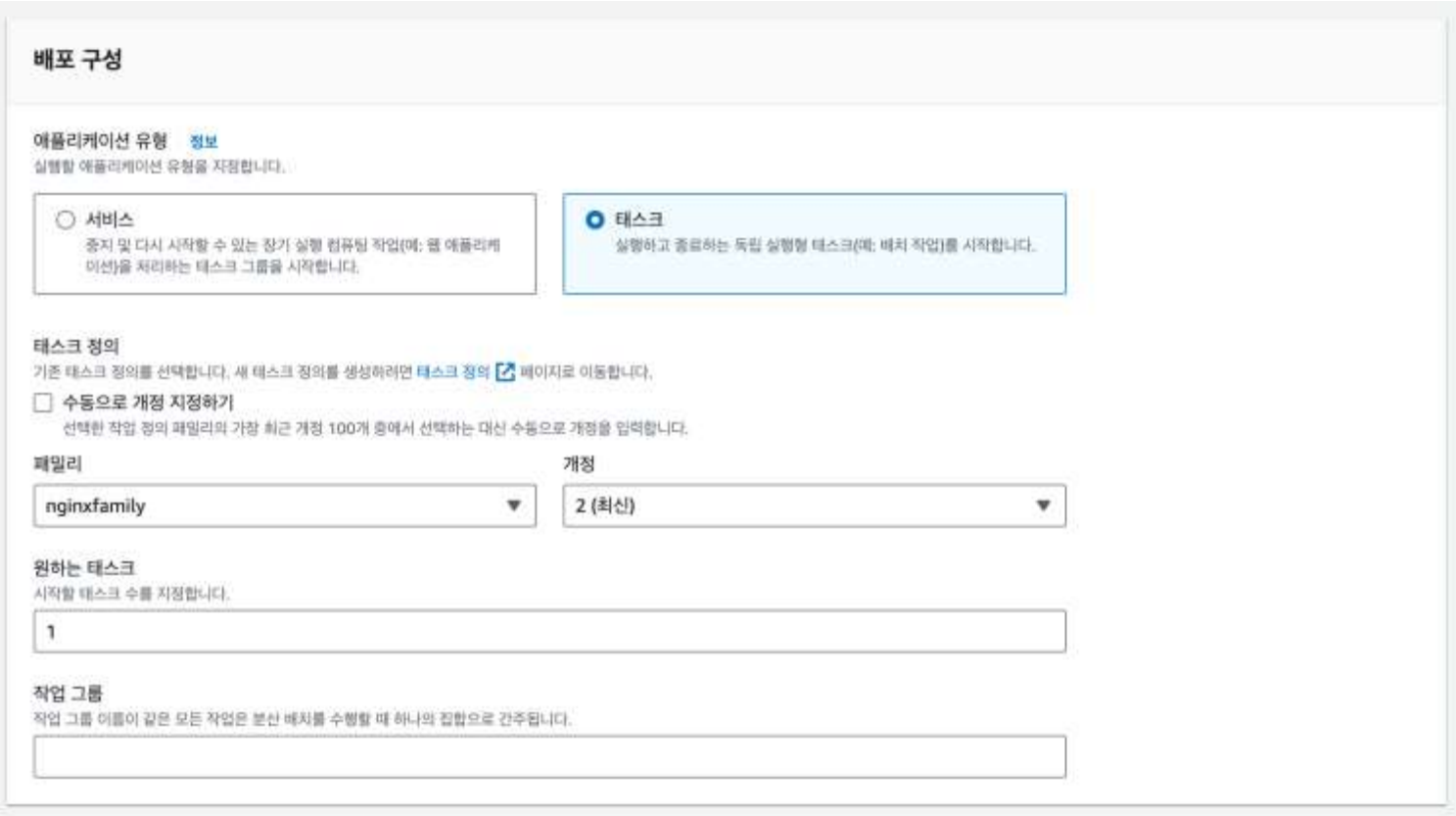
- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 클러스터에 태스크를 연결해서 배포



AWS Container Service

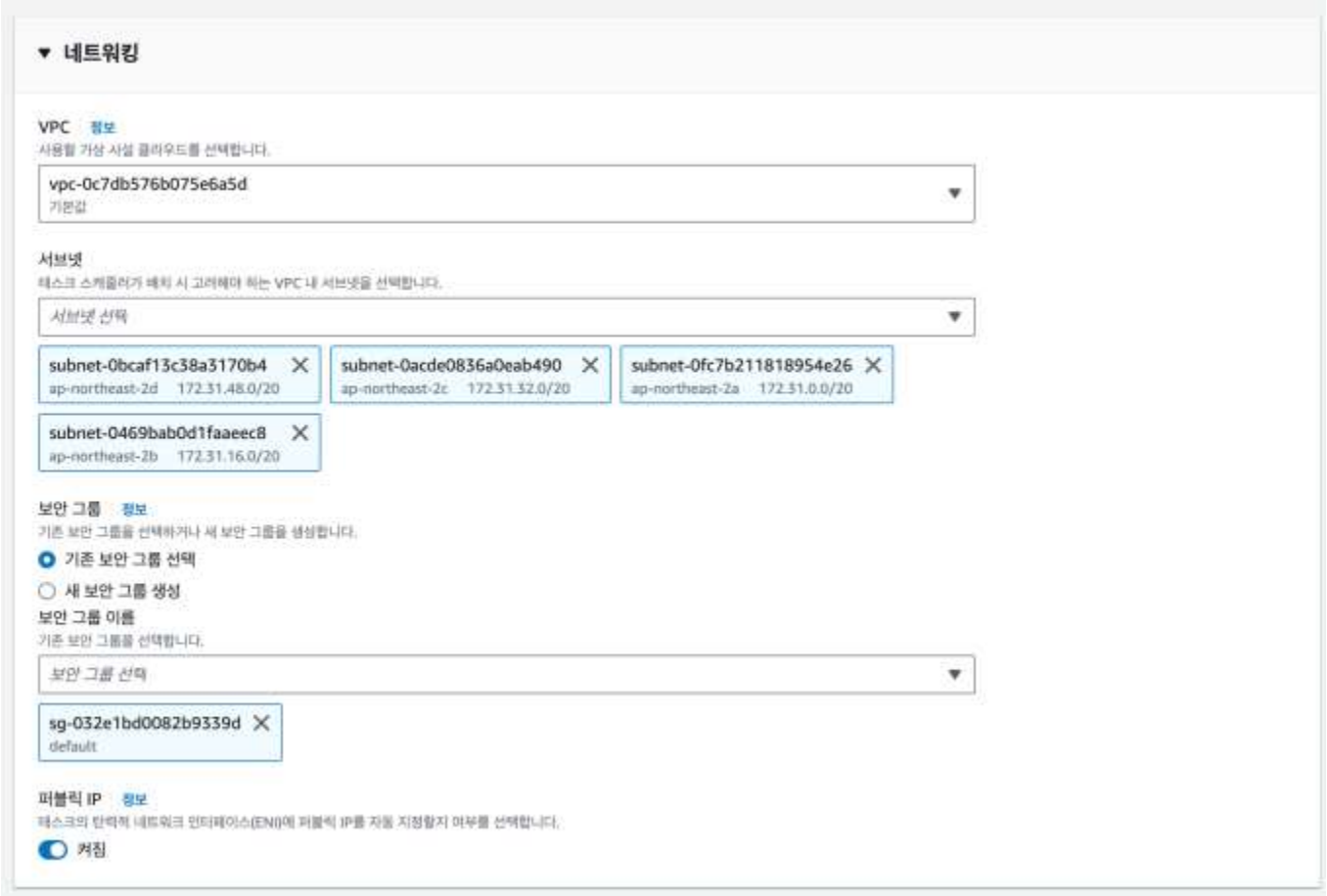
❖ Nginx ECS 배포 – 태스크를 이용한 배포

✓ 클러스터에 태스크를 연결해서 배포



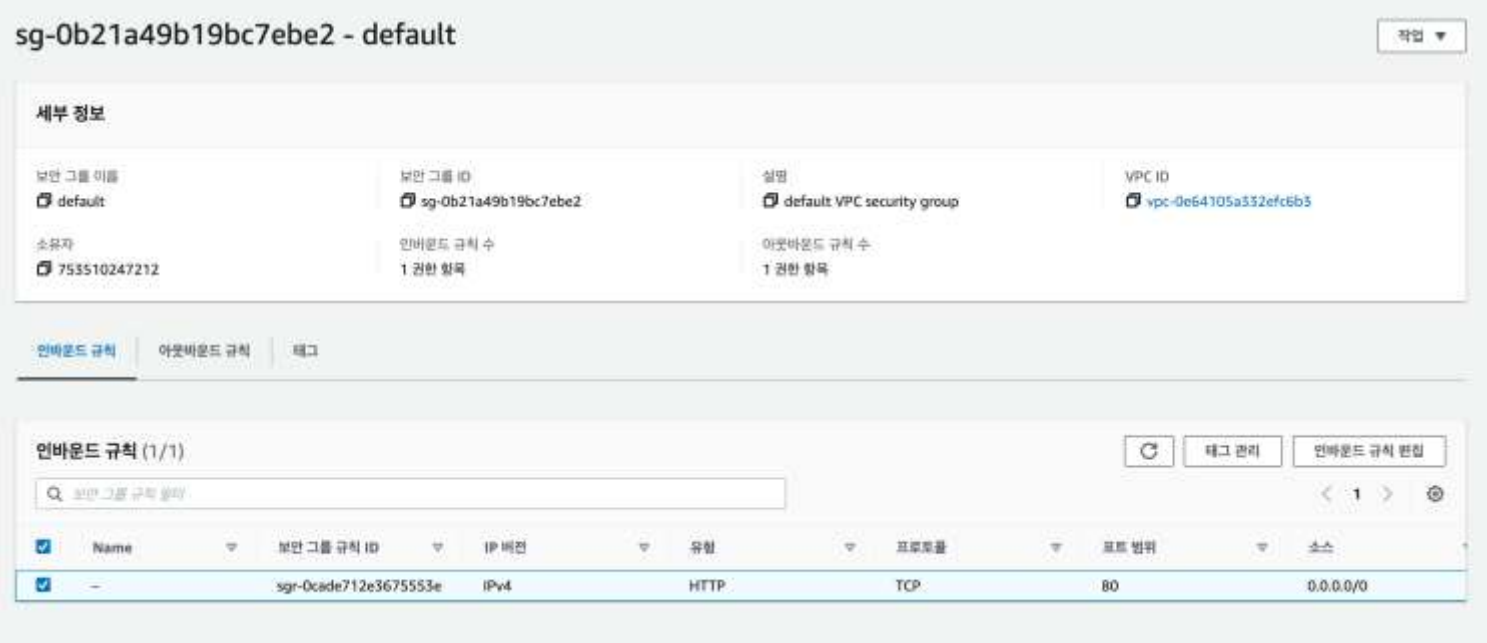
AWS Container Service

- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 클러스터에 태스크를 연결해서 배포



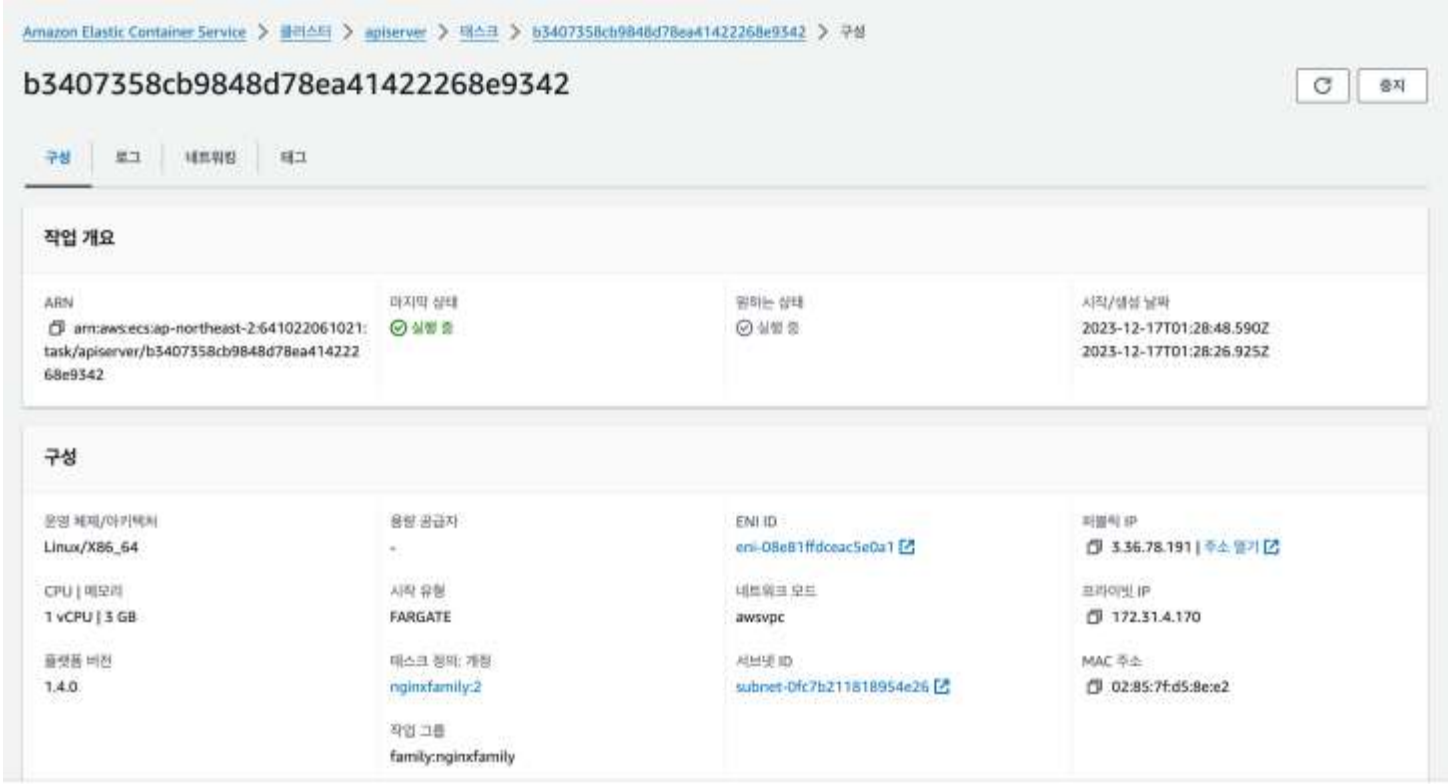
AWS Container Service

- ❖ Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 클러스터에 태스크를 연결해서 배포
    - ● 보안 그룹의 인바운드 규칙에서 80 번 포트를 모든 IP에 개방



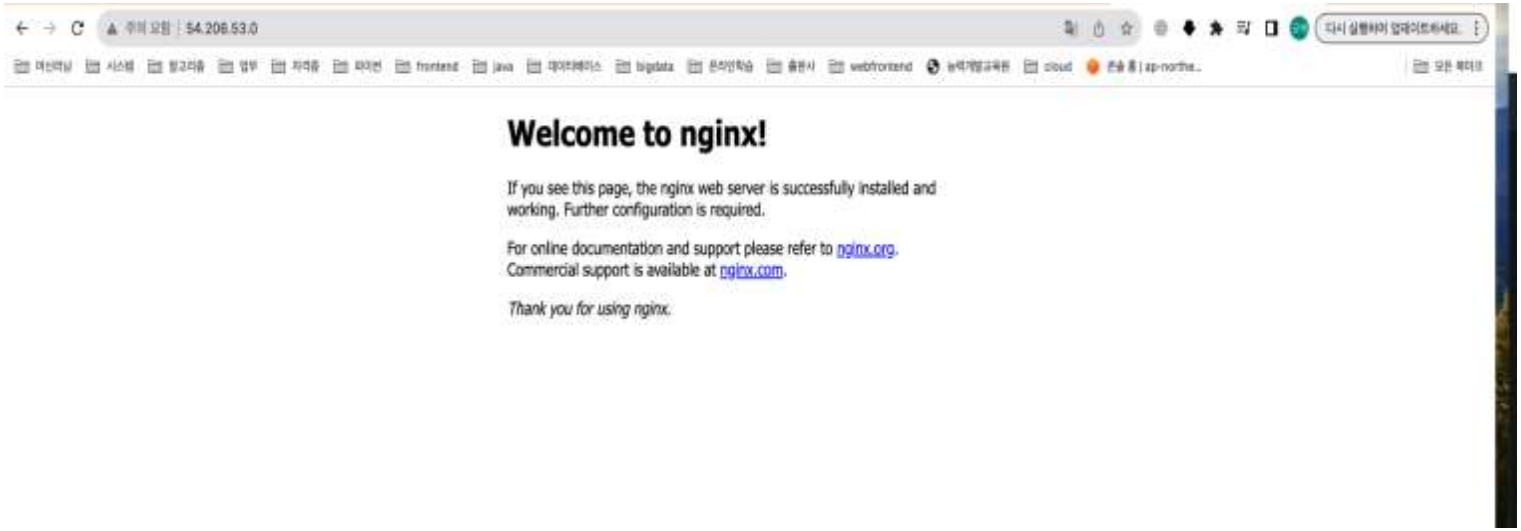
AWS Container Service

- ❖ Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 브라우저에서 태스크의 퍼블릭 IP로 접속



AWS Container Service

- Nginx ECS 배포 – 태스크를 이용한 배포
  - ✓ 브라우저에서 태스크의 퍼블릭 IP로 접속



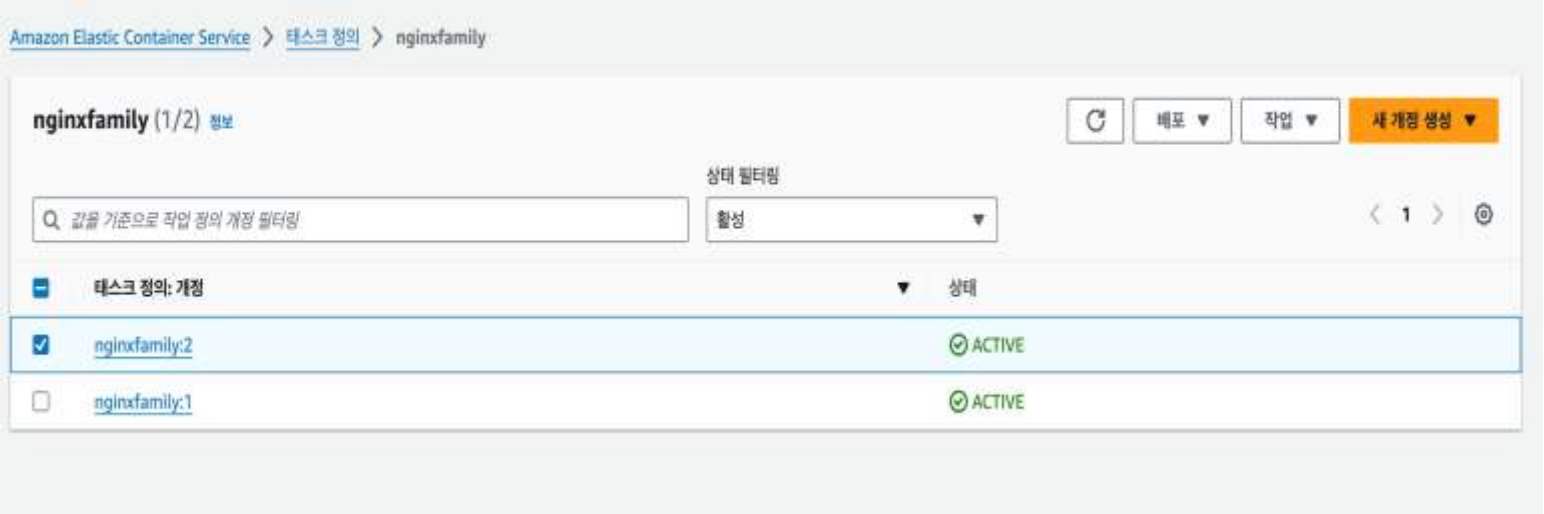
AWS Container Service

- ❖ Nginx ECS 배포 – 서비스를 이용한 배포
  - ✓ Service 사용
    - Task 대신 Service를 이용해서 배포를 하면 Load Balancer를 설정할 수 있음
    - Task 의 Public IP 대신 Load Balancer 의 Domain을 사용할 수 있음

AWS Container Service

❖ Nginx ECS 배포 – 서비스를 이용한 배포

✓ 태스크를 새로 생성하거나 태스크 내에서 새 개정 생성



AWS Container Service

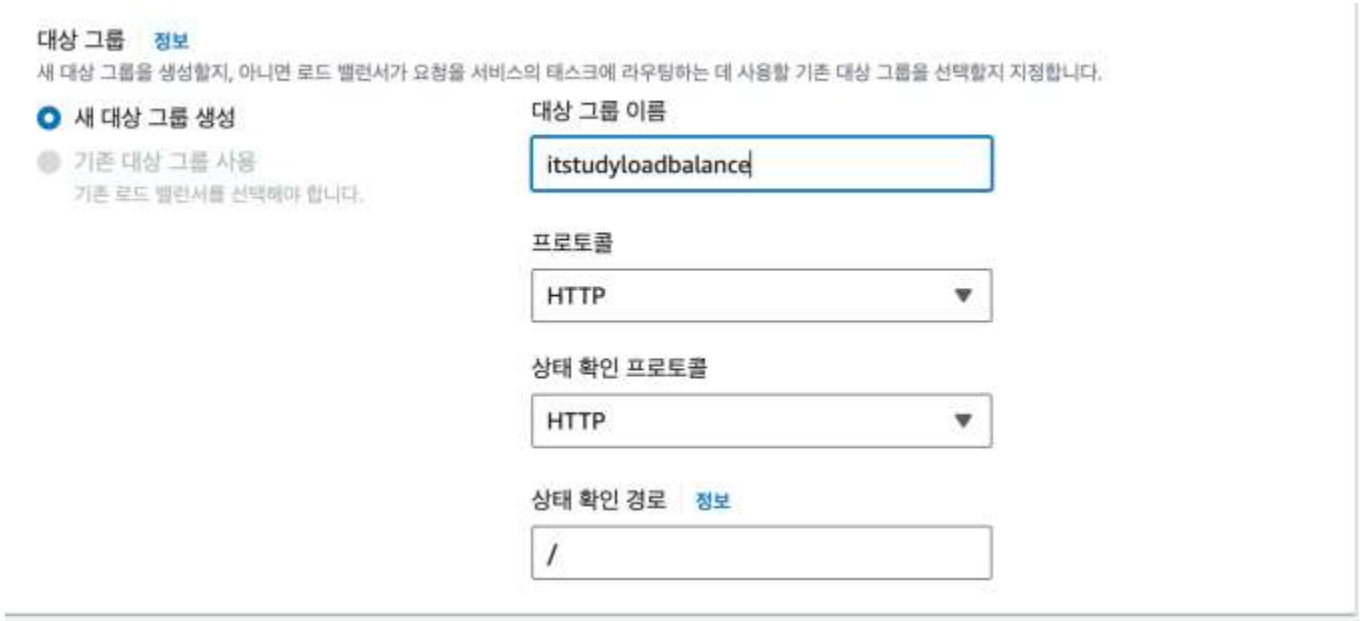
- Nginx ECS 배포 – 서비스를 이용한 배포
  - ✓ 클러스터에서 서비스 생성
    - 서비스 생성

AWS Container Service

- ❖ Nginx ECS 배포 – 서비스를 이용한 배포
  - ✓ 클러스터에서 서비스 생성
    - ● 서비스 생성

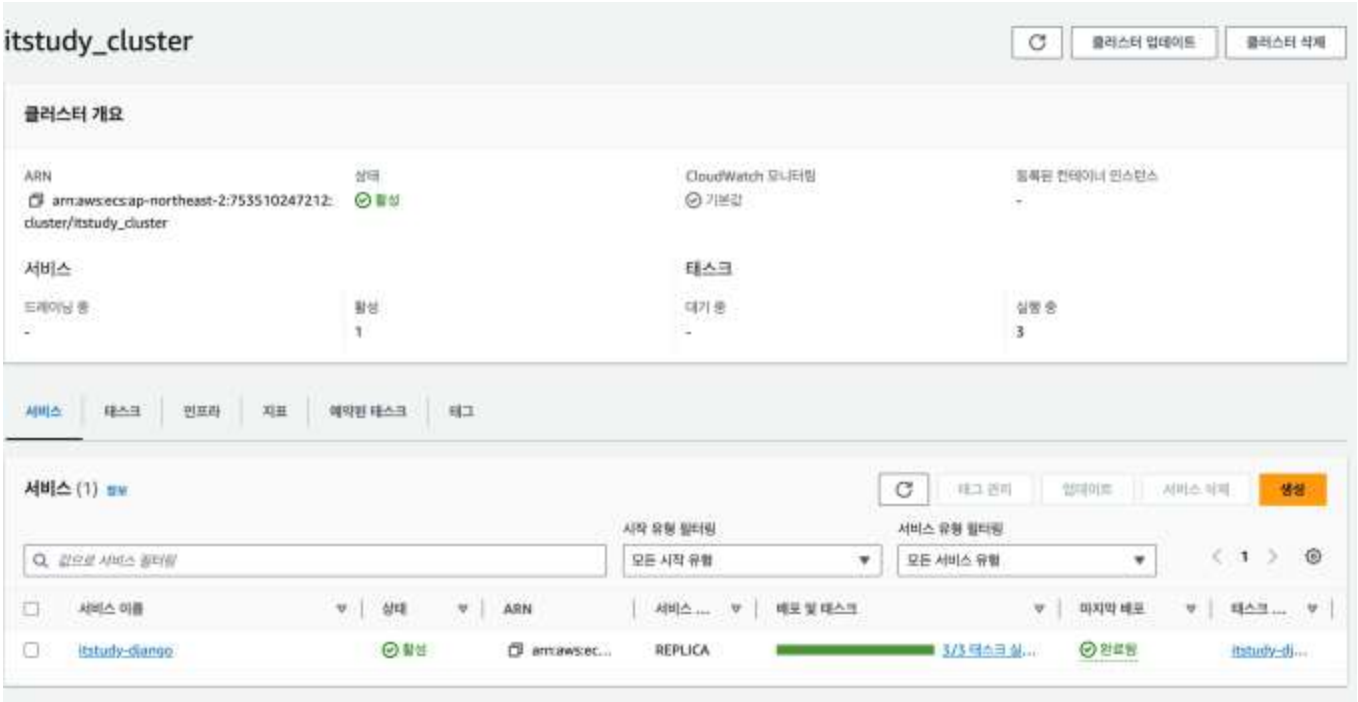
AWS Container Service

- ❖ Nginx ECS 배포 – 서비스를 이용한 배포
  - ✓ 클러스터에서 서비스 생성
    - 서비스 생성



AWS Container Service

- ❖ Nginx ECS 배포 – 서비스를 이용한 배포
  - ✓ 클러스터에서 서비스 생성
    - 서비스 생성

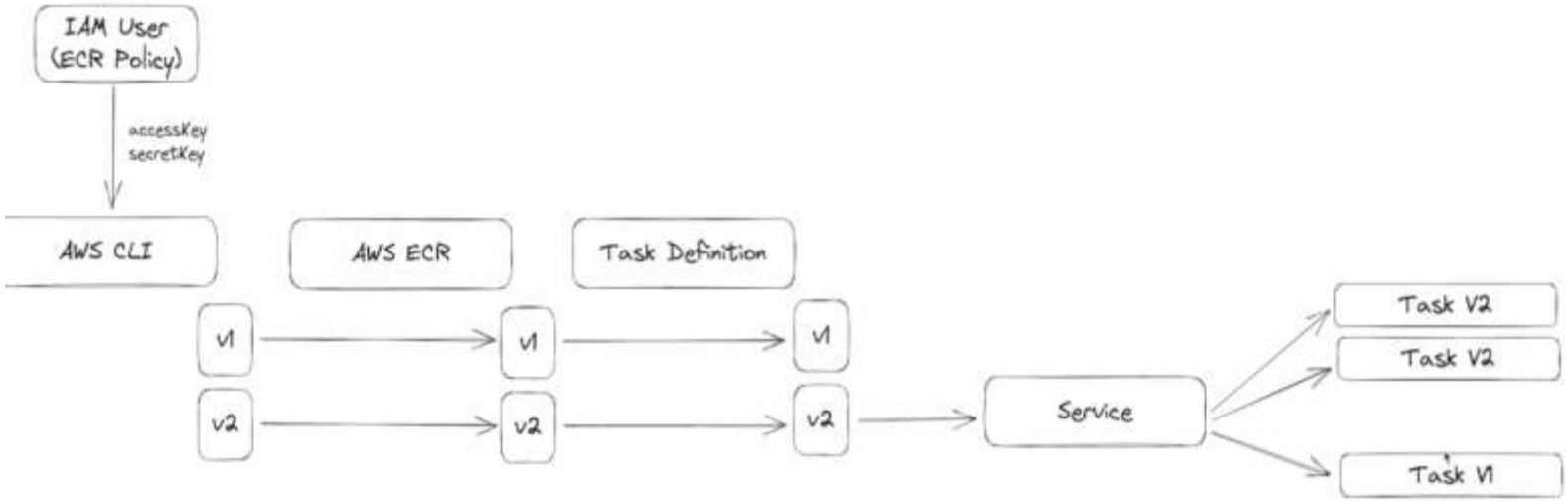


AWS Container Service

- ❖ Nginx ECS 배포 – 서비스를 이용한 배포
  - ✓ 클러스터에서 서비스 생성
    - Load Balancer 도메인으로 접속

AWS Container Service

❖ ECS 배포 전략



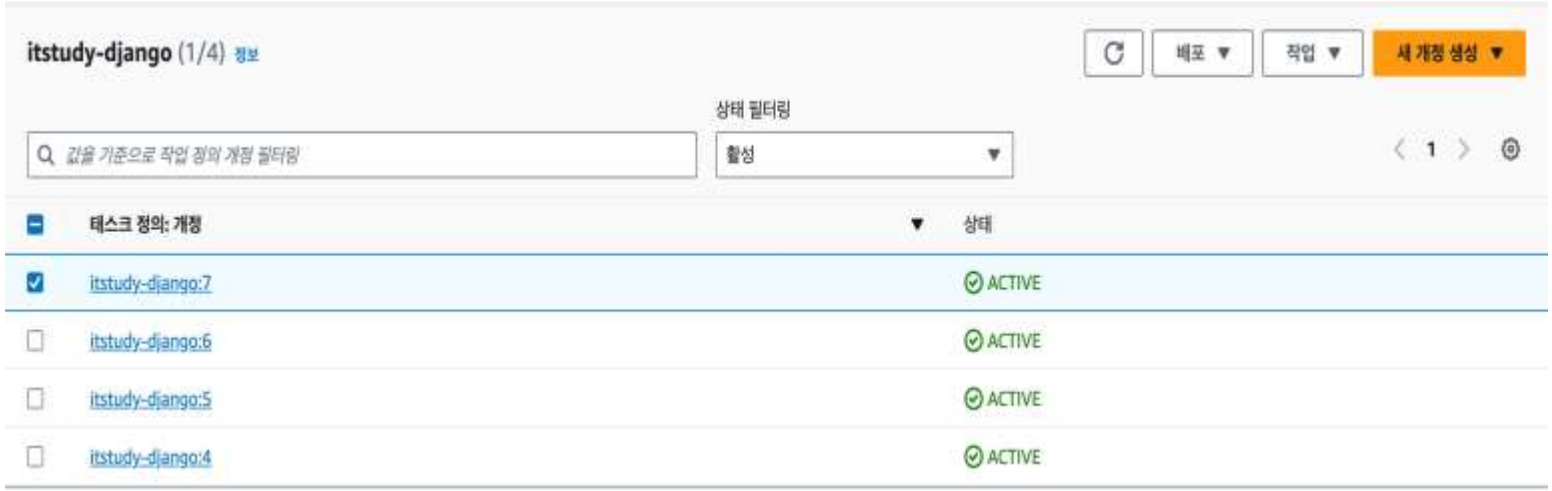
AWS Container Service

❖ ECS 배포 전략 – Green/Blue Strategy

AWS Container Service

❖ ECS 배포 전략 – Green/Blue Strategy

✓ 이전 태스크에서 새 개정 생성



AWS Container Service

- ECS 배포 전략 – Green/Blue Strategy
  - ✓ 태스크에서 새 개정 생성

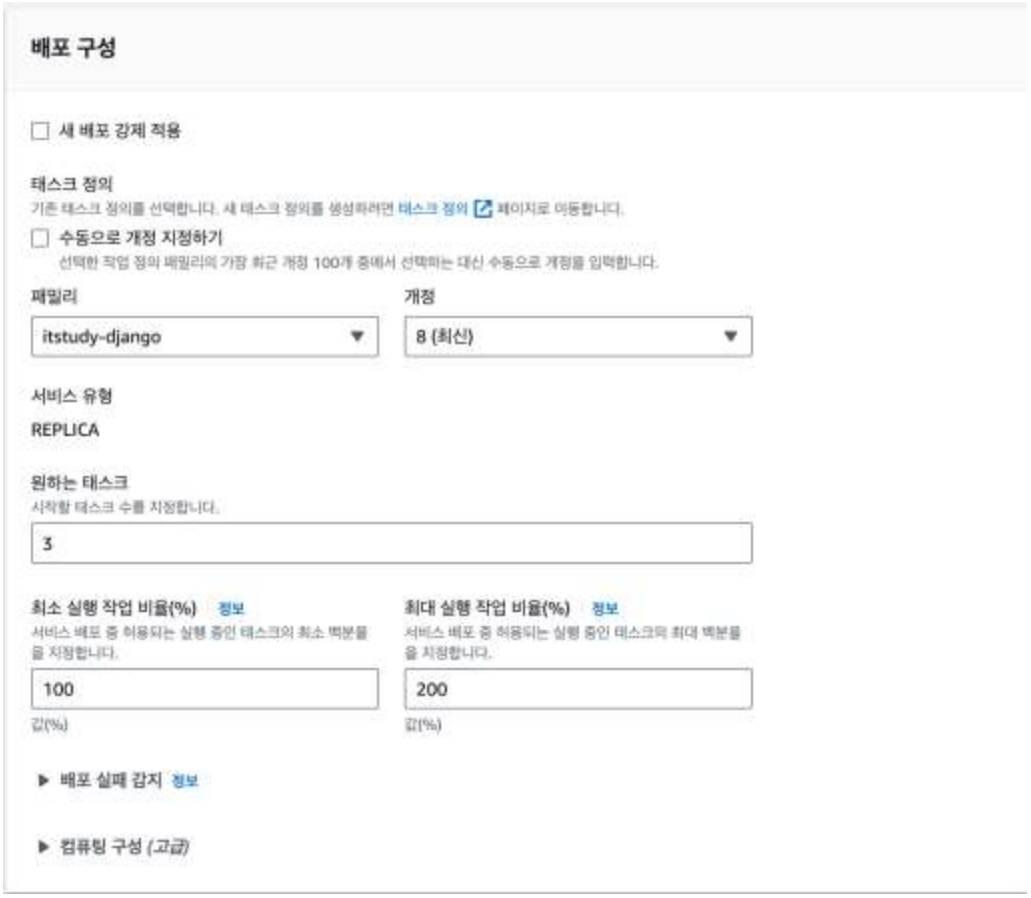
AWS Container Service

❖ ECS 배포 전략 – Green/Blue Strategy

✓ 서비스 업데이트

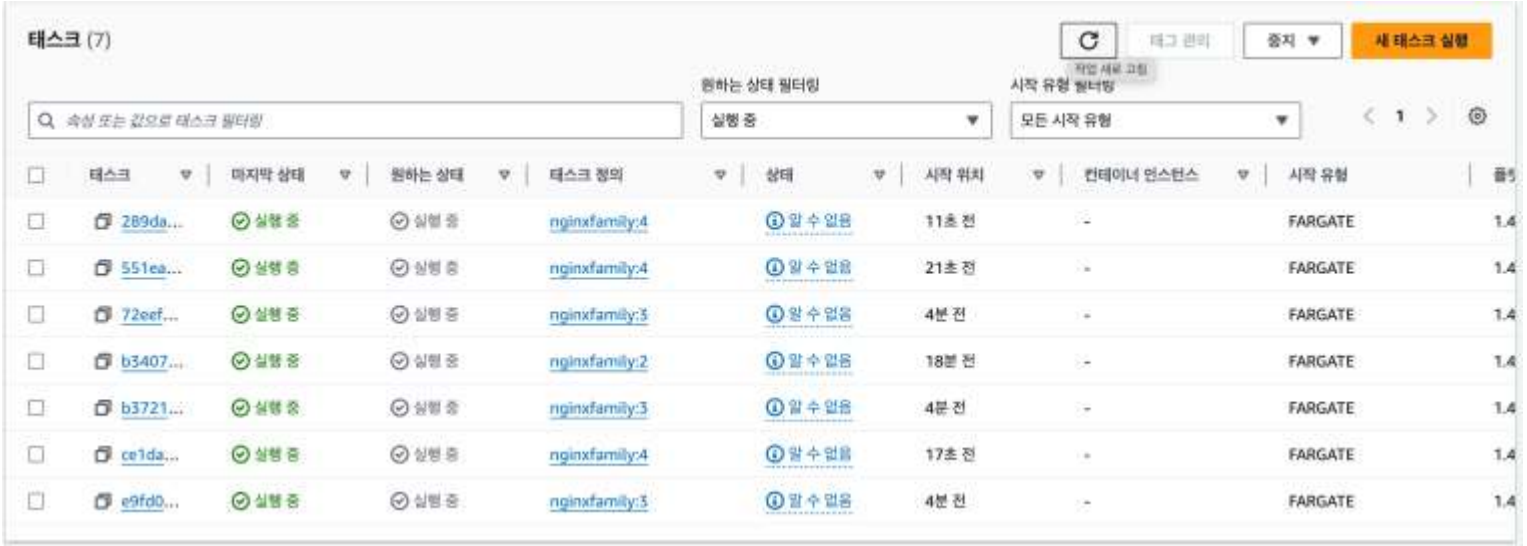
AWS Container Service

- ECS 배포 전략 – Green/Blue Strategy
  - ✓ 서비스 업데이트



AWS Container Service

- ECS 배포 전략 – Green/Blue Strategy
  - ✓ 서비스 업데이트

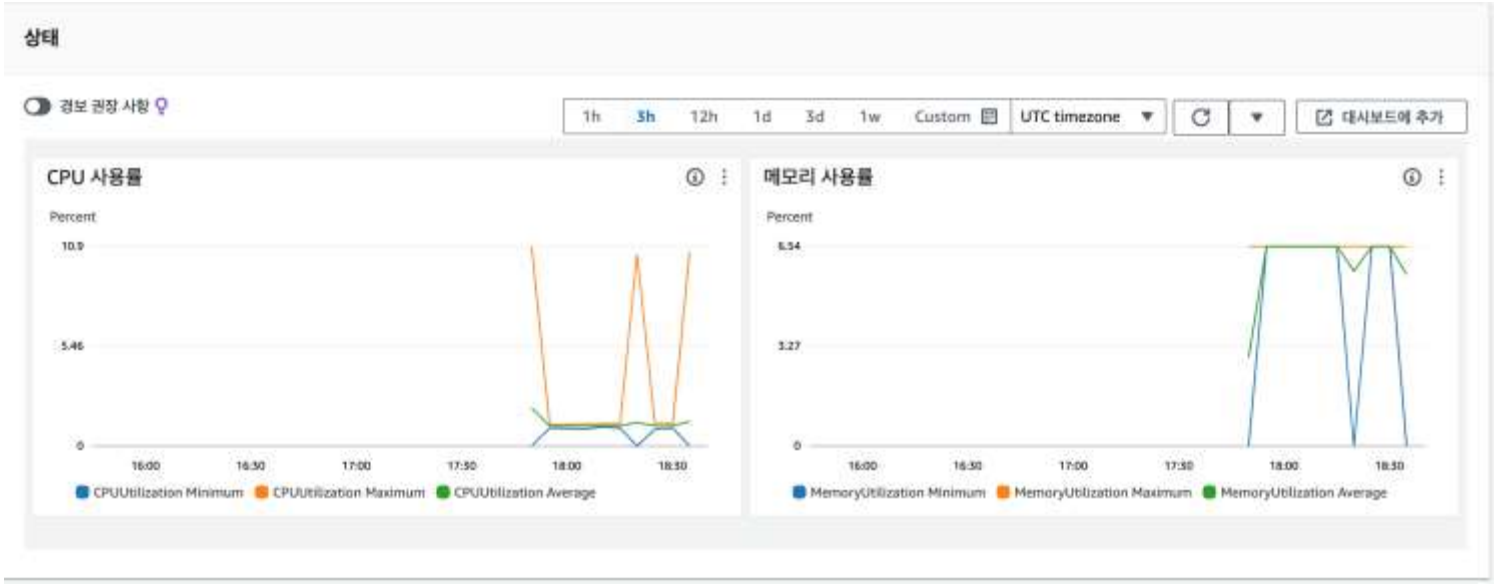


AWS Container Service

- ECS 배포 전략 – Green/Blue Strategy
  - ✓ 서비스 업데이트

AWS Container Service

- ❖ Auto Scaling
  - ✓ 트리거 조건
    - CPU 사용량
    - 메모리 사용량
    - 트래픽



AWS Container Service

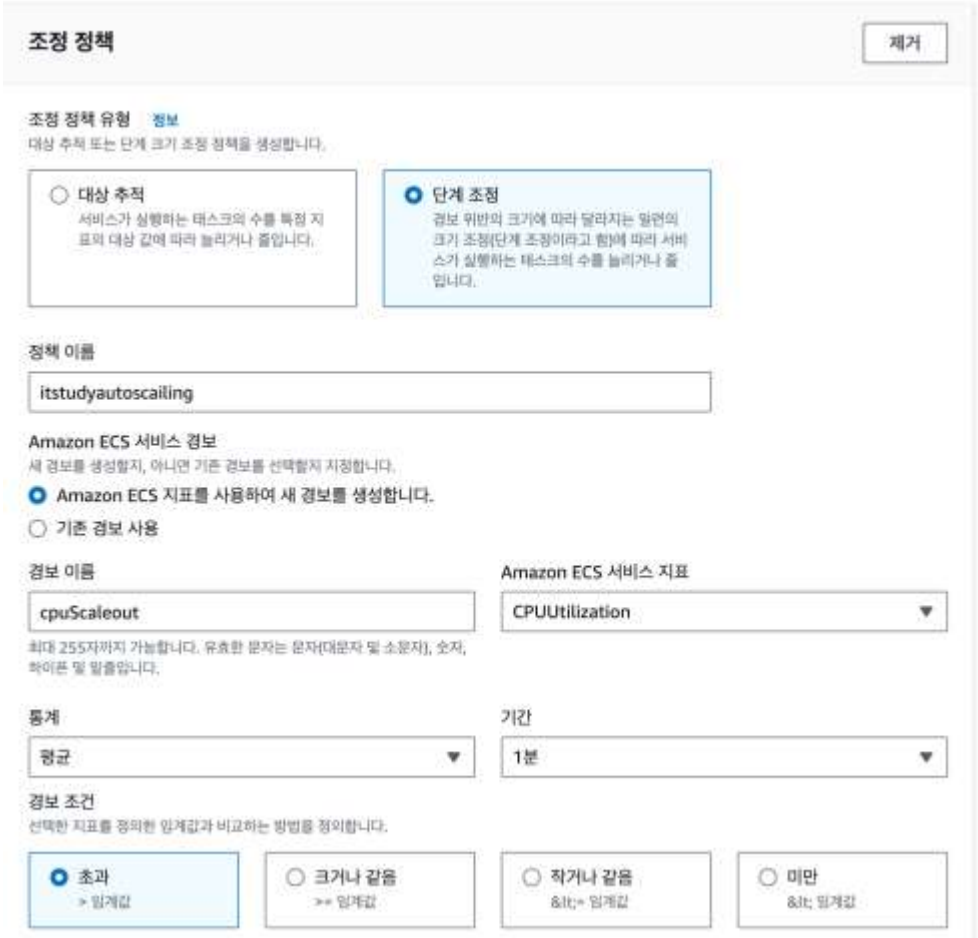
- ❖ Auto Scaling
  - ✓ 설정 내용
    - 트리거 조건
    - 확장 단위
    - 쿨다운: scale out(수평 확장), scale in(수평 축소) 하는데 모니터링 하는 시간

AWS Container Service

❖ Auto Scaling

AWS Container Service

❖ Auto Scaling

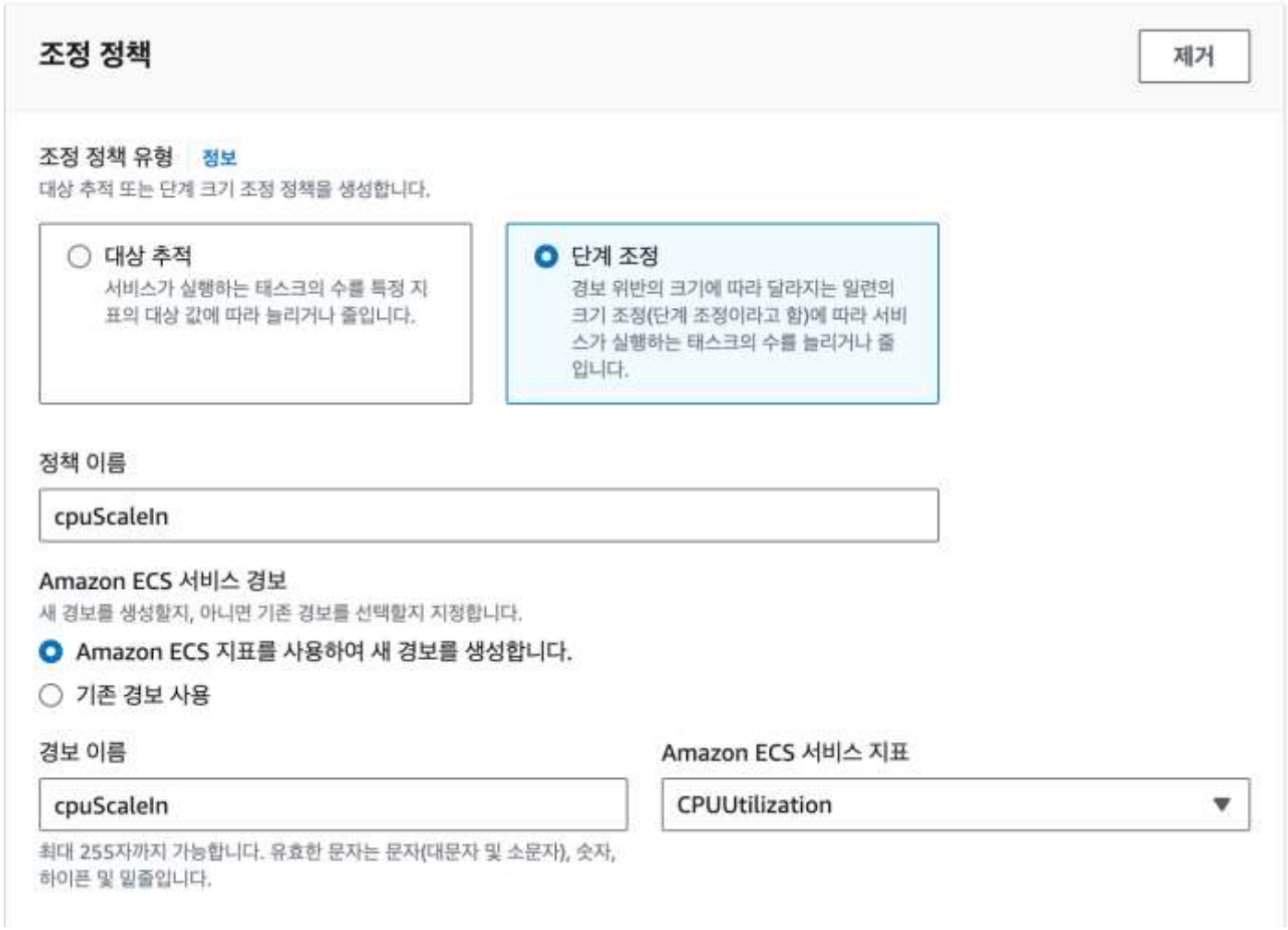


AWS Container Service

❖ Auto Scaling

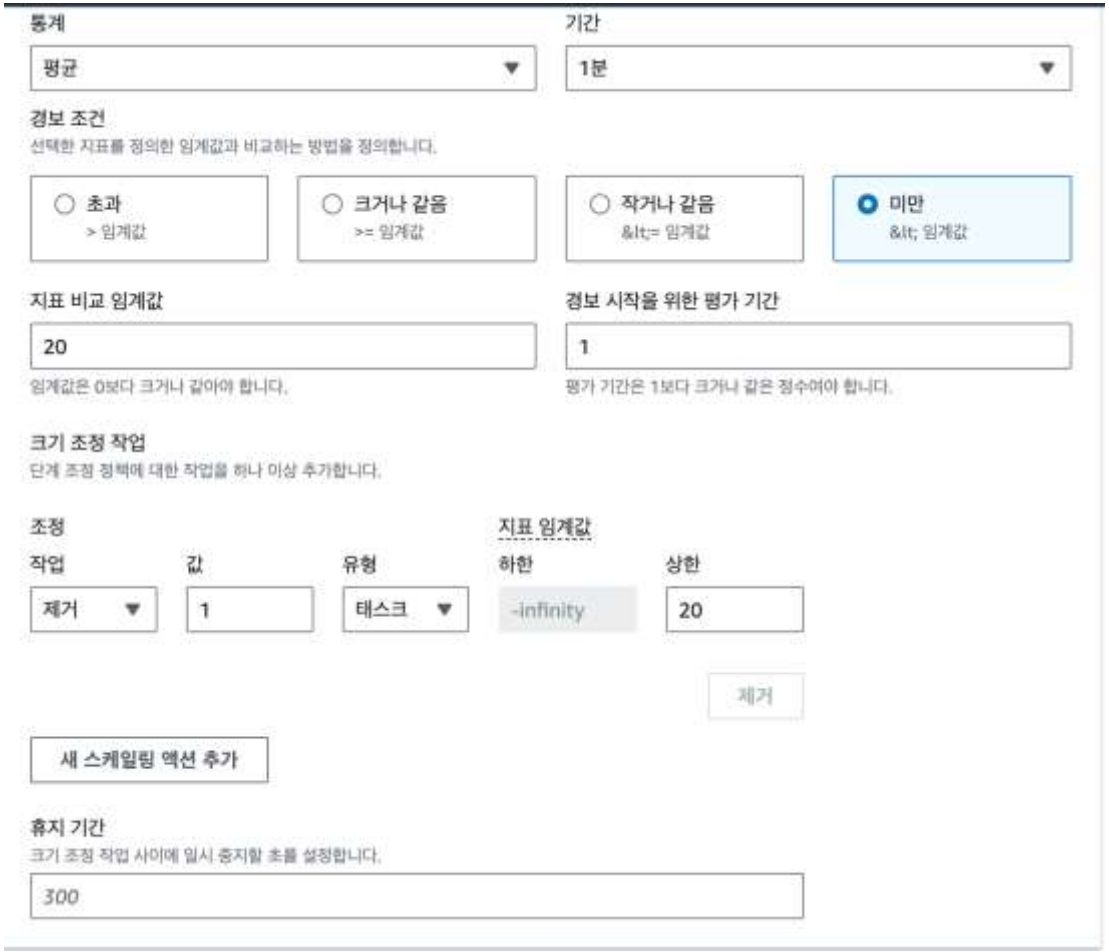
AWS Container Service

❖ Auto Scaling



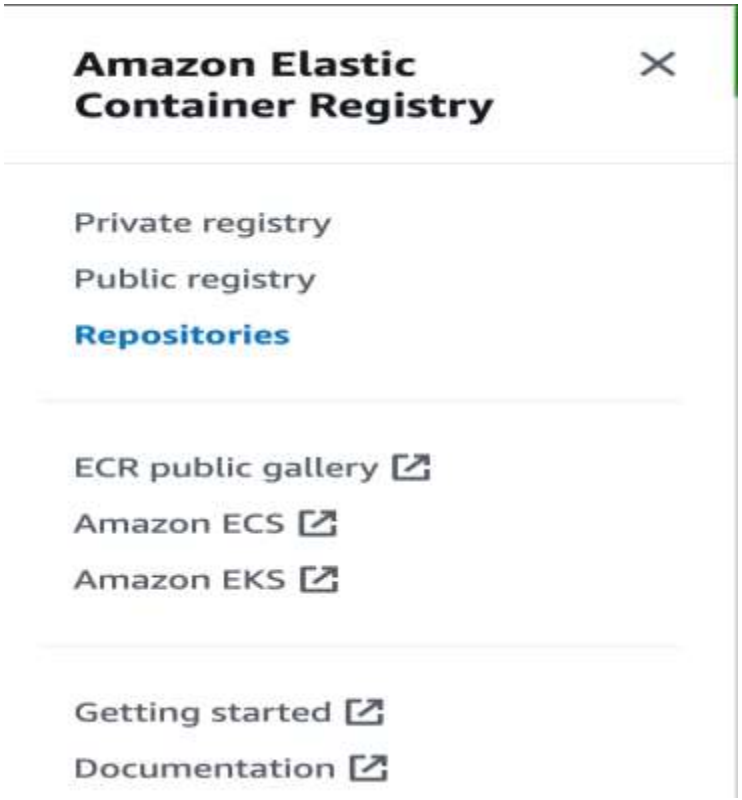
AWS Container Service

❖ Auto Scaling



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성



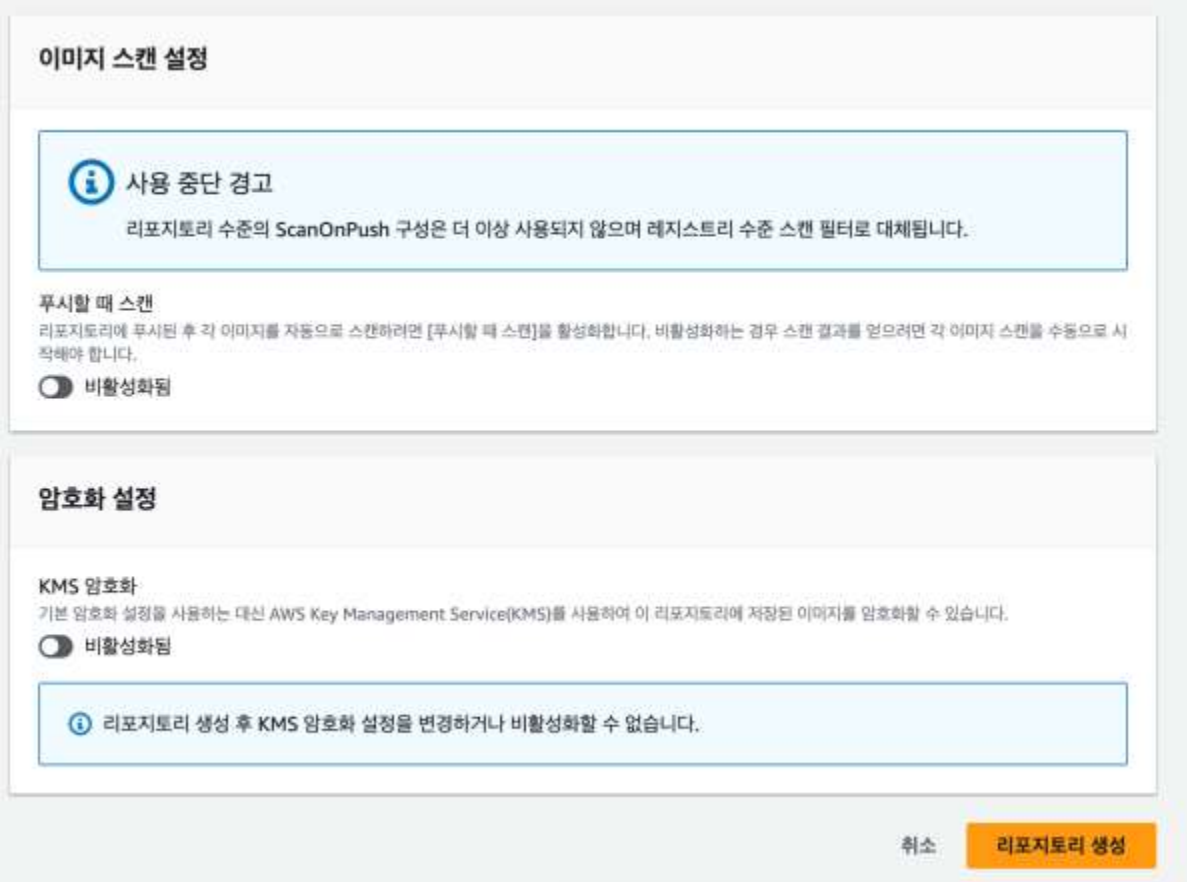
AWS Container Service

- Django Application CI/CD Pipeline 설정
  - ✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성 - apiserver

AWS Container Service

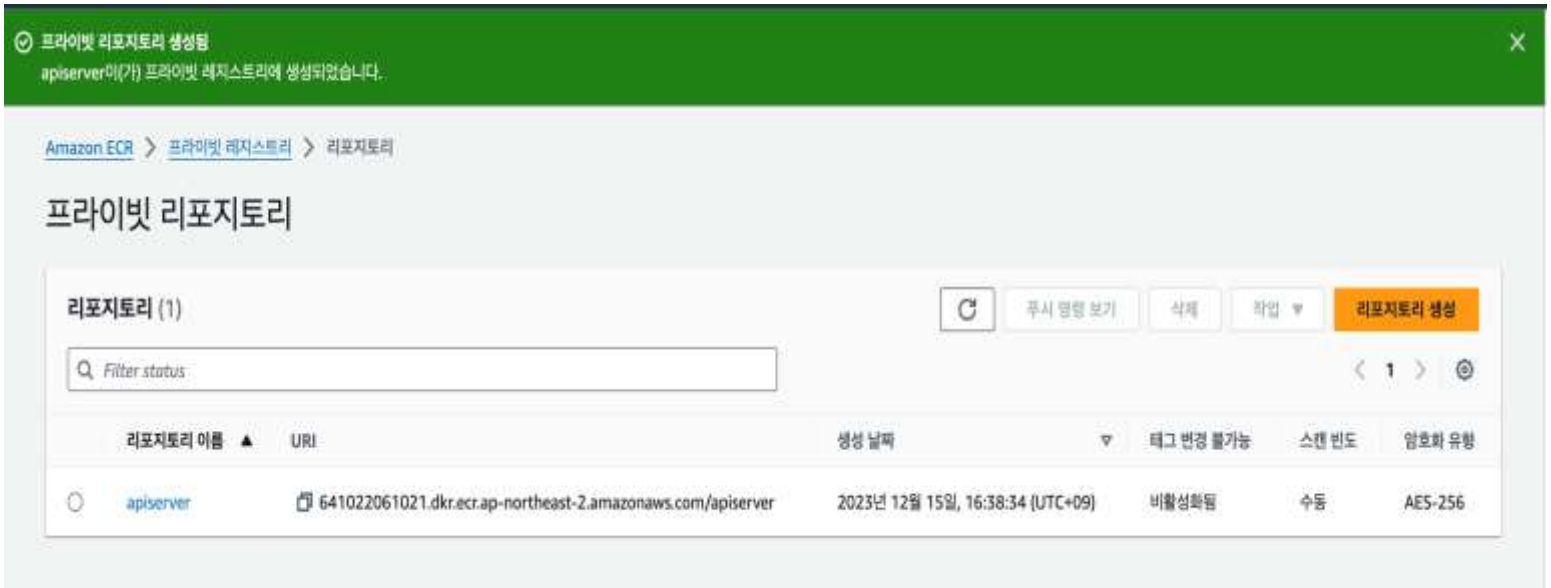
❖ Django Application CI/CD Pipeline 설정

✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성 - apiserver



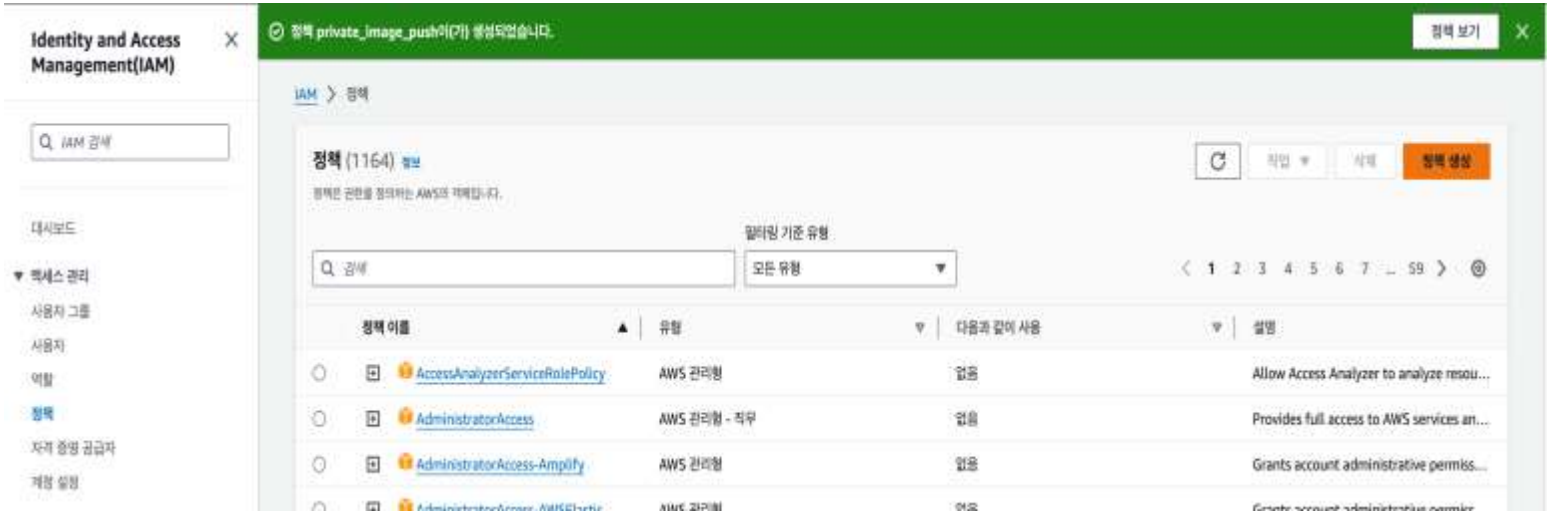
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성 - apiserver



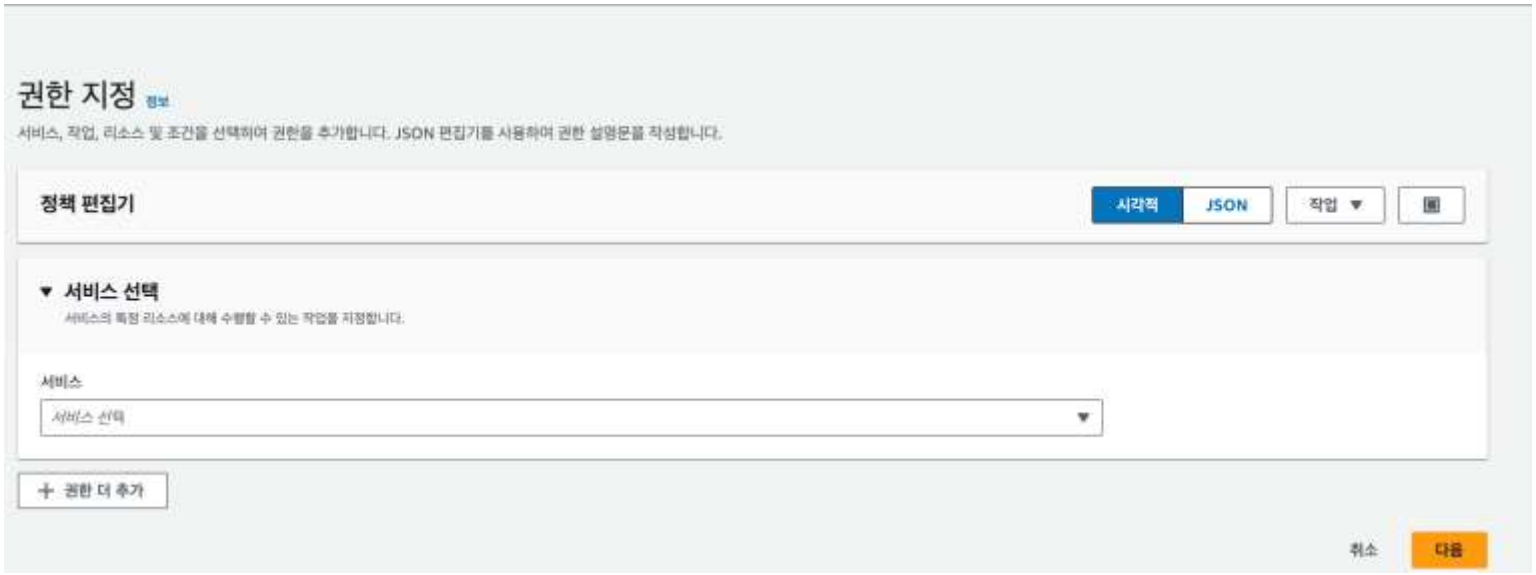
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ IAM 서비스에서 ECR에 이미지를 push할 수 있는 정책 생성
    - ● 정책 생성(private_image_push) – 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성
      - ◆ 정책 생성 클릭



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ IAM 서비스에서 ECR에 이미지를 push할 수 있는 정책 생성
    - ● 정책 생성(private_image_push) – 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성
      - ◆ JSON 클릭



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR에 이미지를 push할 수 있는 정책 생성
    - 정책 생성(private_image_push) – 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:CompleteLayerUpload",
                "ecr:UploadLayerPart",
                "ecr:InitiateLayerUpload",
                "ecr:BatchCheckLayerAvailability",
                "ecr:PutImage"
            ],
            "Resource": "arn:aws:ecr:ap-northeast-2:753510247212:repository/apiserver"
        },
```

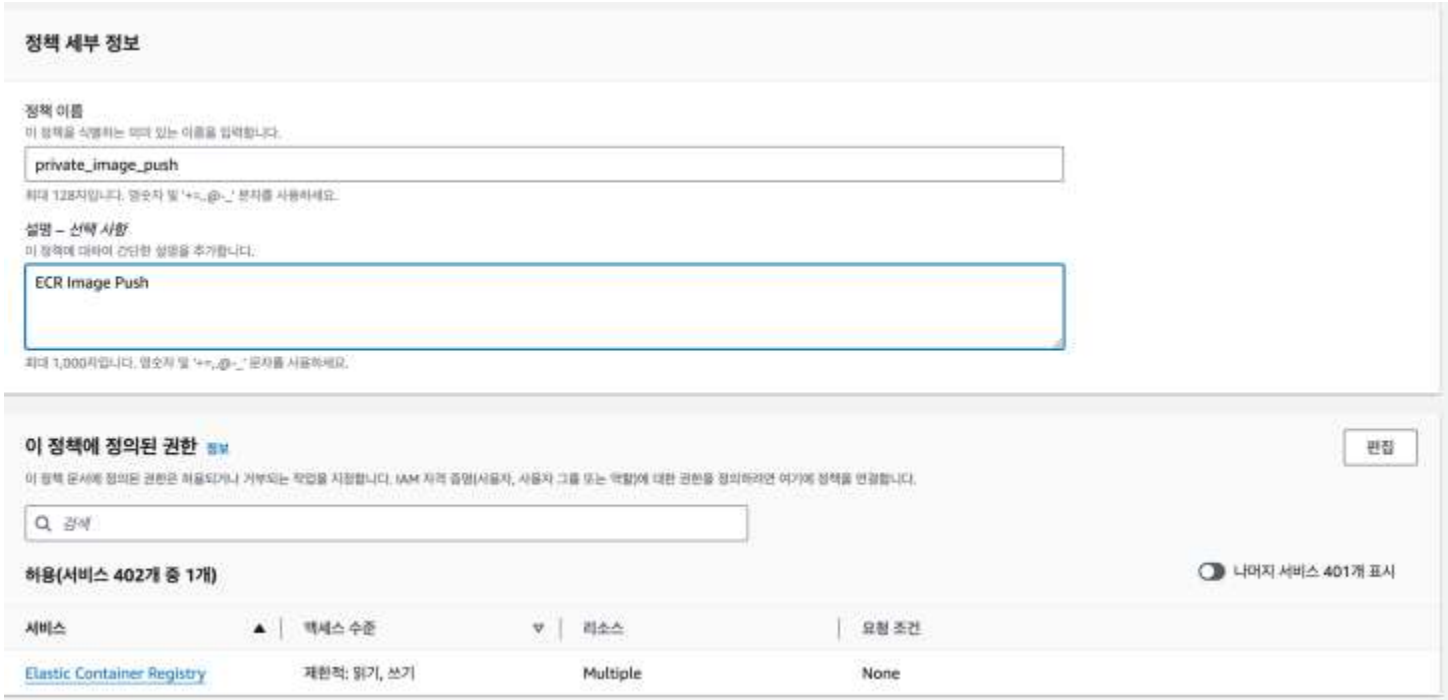
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR에 이미지를 push할 수 있는 정책 생성
    - ● 정책 생성(private_image_push) – 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성

```
            {
                "Effect": "Allow",
                "Action": "ecr:GetAuthorizationToken",
                "Resource": "*"
            }
        ]
    }
```

AWS Container Service

- Django Application CI/CD Pipeline 설정
  - ✓ IAM에서 ECR에 이미지를 push할 수 있는 정책 생성
    - 정책 생성(private_image_push) – 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - IAM 에서 ID 제공업체를 클릭해서 추가

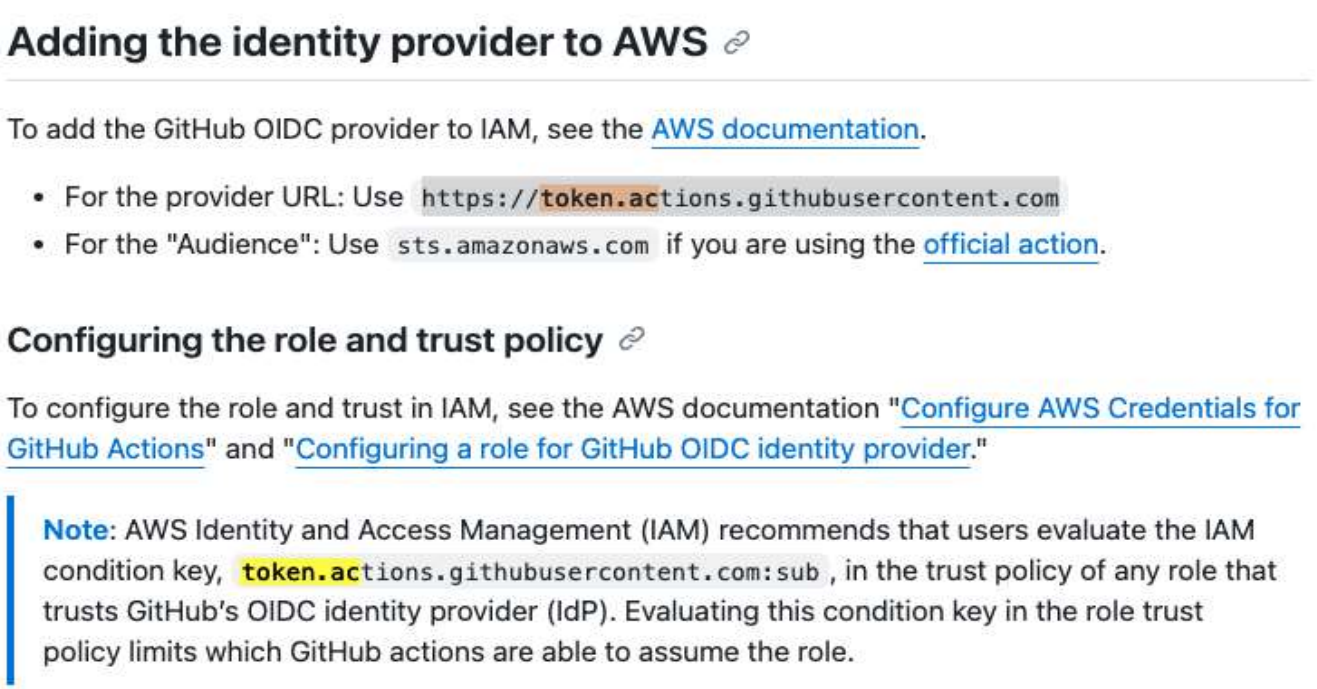
Adding the identity provider to AWS

To add the GitHub OIDC provider to IAM, see the AWS documentation.

- For the provider URL: Use `https://token.actions.githubusercontent.com`
- For the "Audience": Use `sts.amazonaws.com` if you are using the official action.

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● IAM 에서 자격 증명 공급자를 클릭해서 추가
      - ◆ URL 확인: https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services



Adding the identity provider to AWS

To add the GitHub OIDC provider to IAM, see the AWS documentation.

- For the provider URL: Use `https://token.actions.githubusercontent.com`
- For the "Audience": Use `sts.amazonaws.com` if you are using the official action.

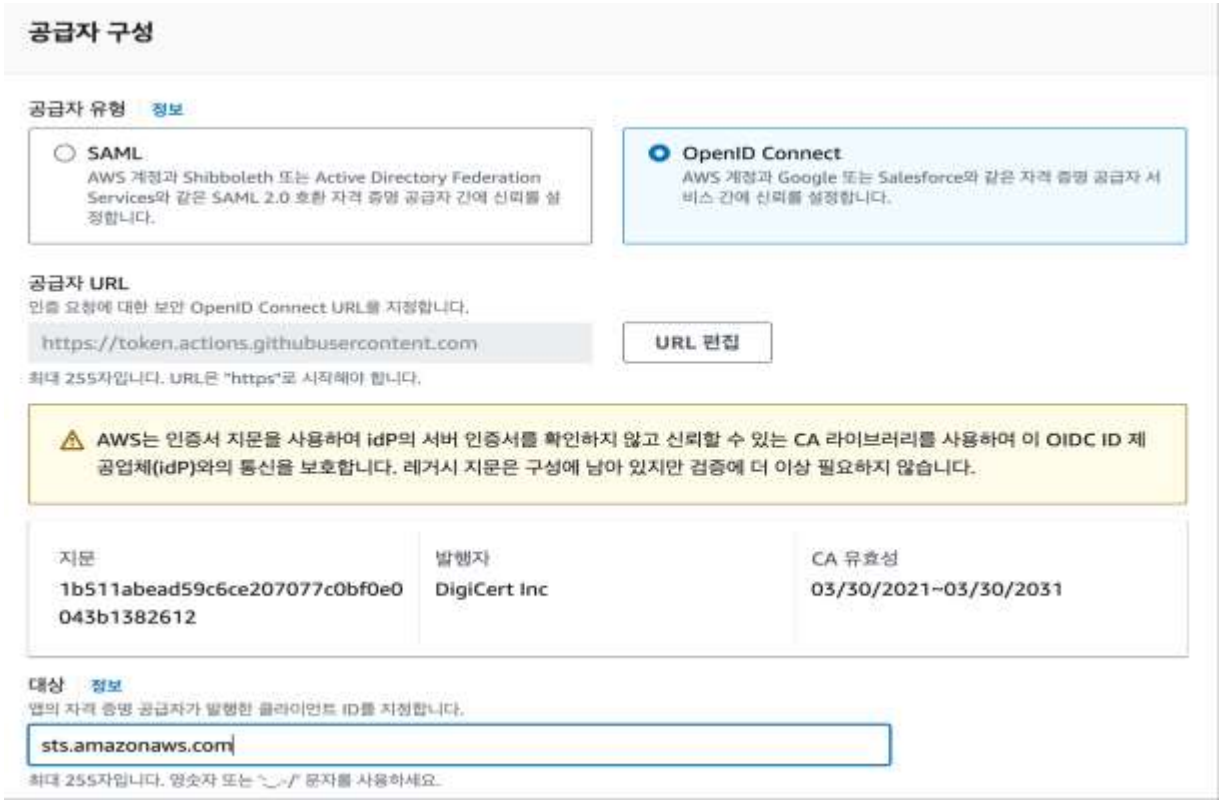
Configuring the role and trust policy

To configure the role and trust in IAM, see the AWS documentation "Configure AWS Credentials for GitHub Actions" and "Configuring a role for GitHub OIDC identity provider."

> **Note**: AWS Identity and Access Management (IAM) recommends that users evaluate the IAM condition key, `token.actions.githubusercontent.com:sub`, in the trust policy of any role that trusts GitHub's OIDC identity provider (IdP). Evaluating this condition key in the role trust policy limits which GitHub actions are able to assume the role.

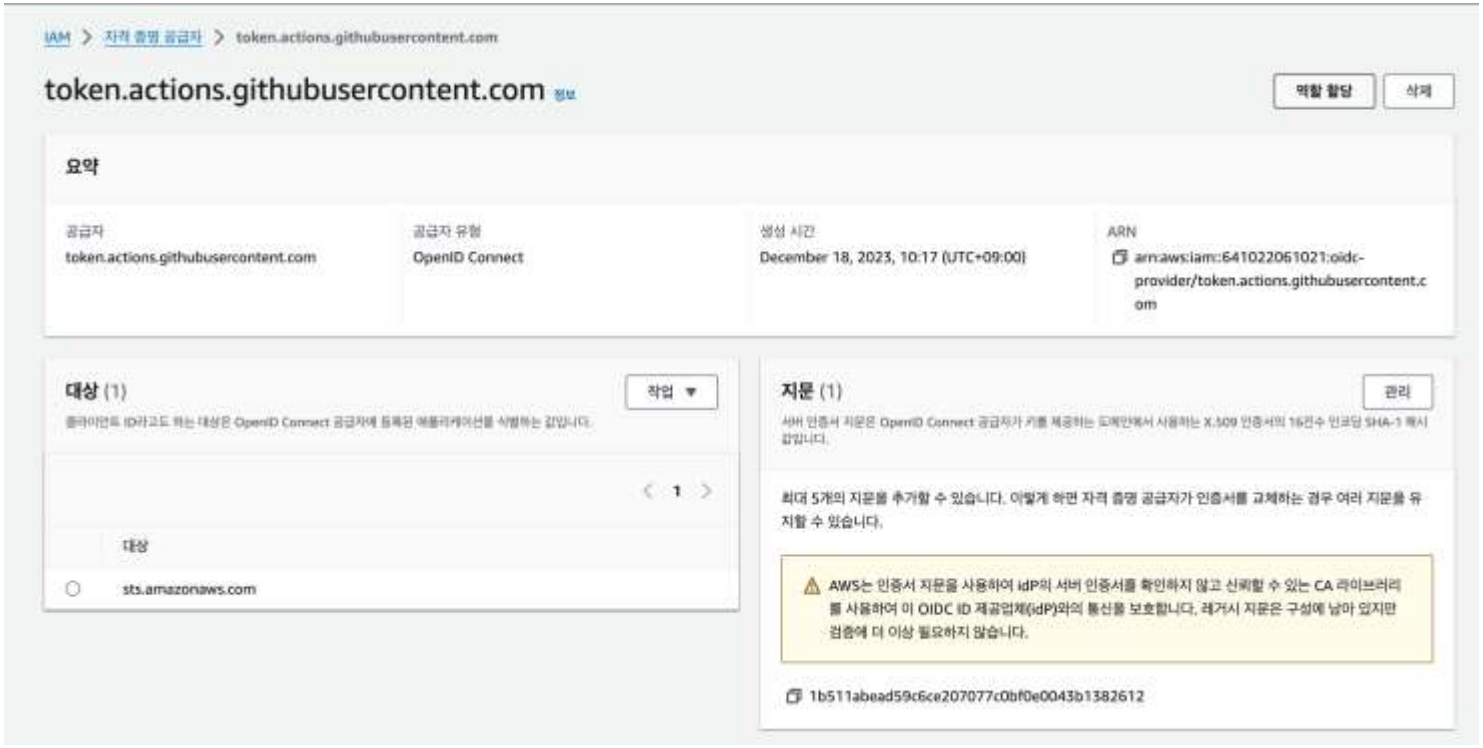
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - IAM 에서 자격 증명 공급자를 클릭해서 추가



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● 자격 증명 공급자에 새로운 역할 할당



AWS Container Service

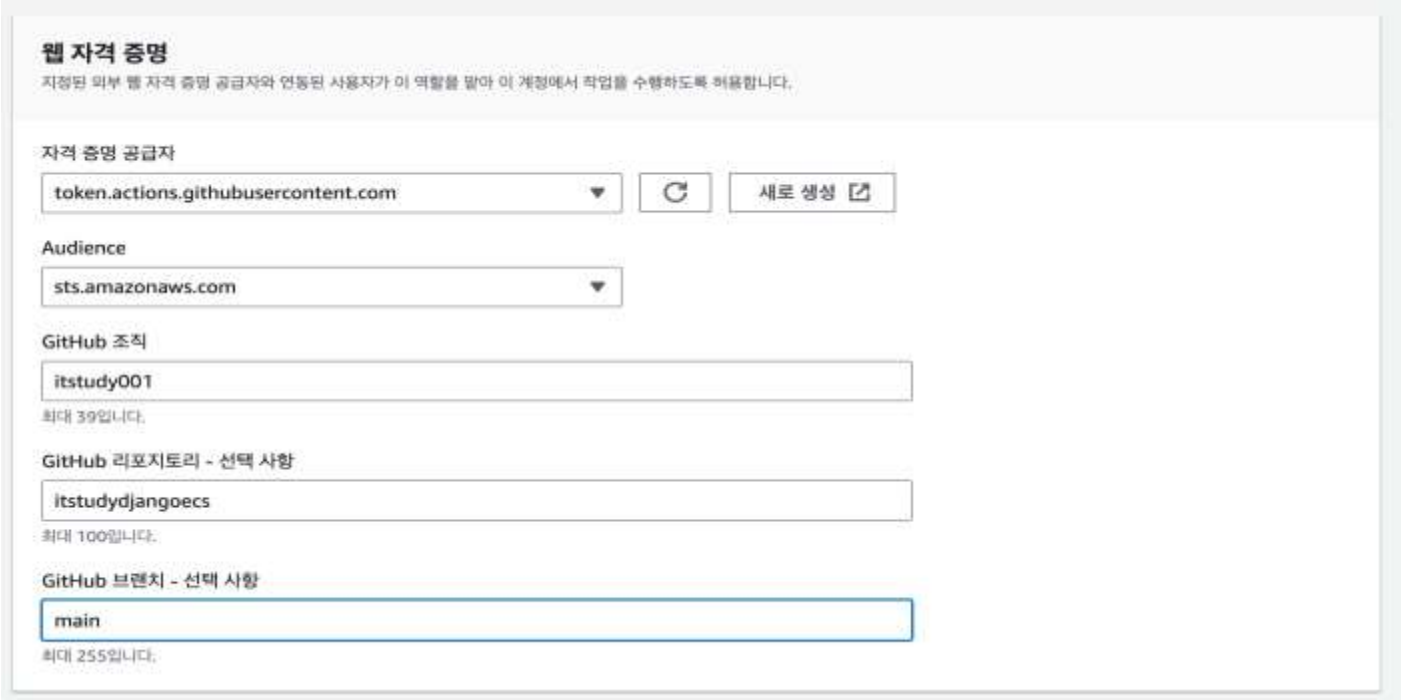
❖ Django Application CI/CD Pipeline 설정

✓ OpenID Connect를 이용한 Git Hub AWS 로그인

● 자격 증명 공급자에 새로운 역할 할당

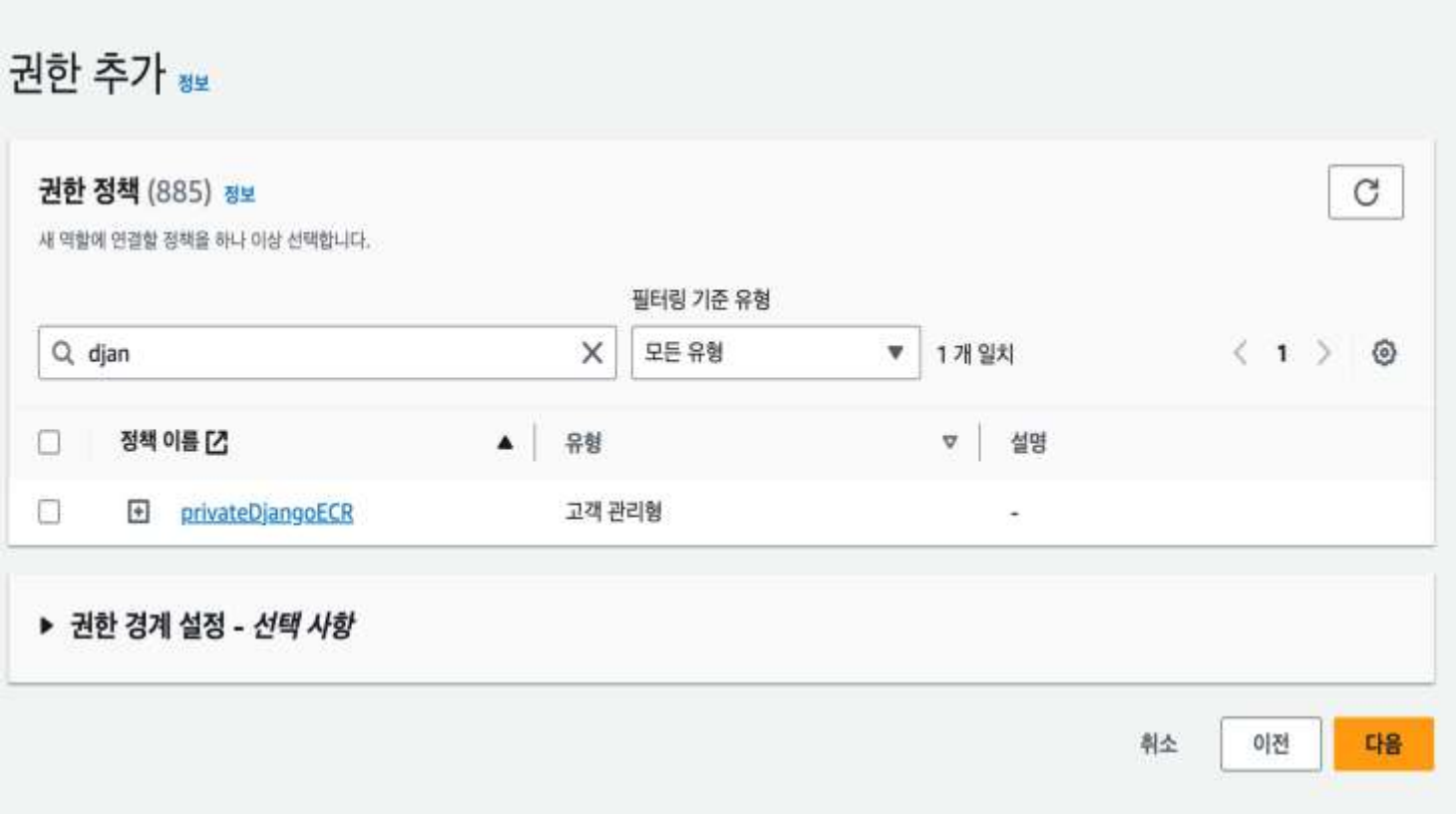
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● 자격 증명 공급자에 새로운 역할 할당



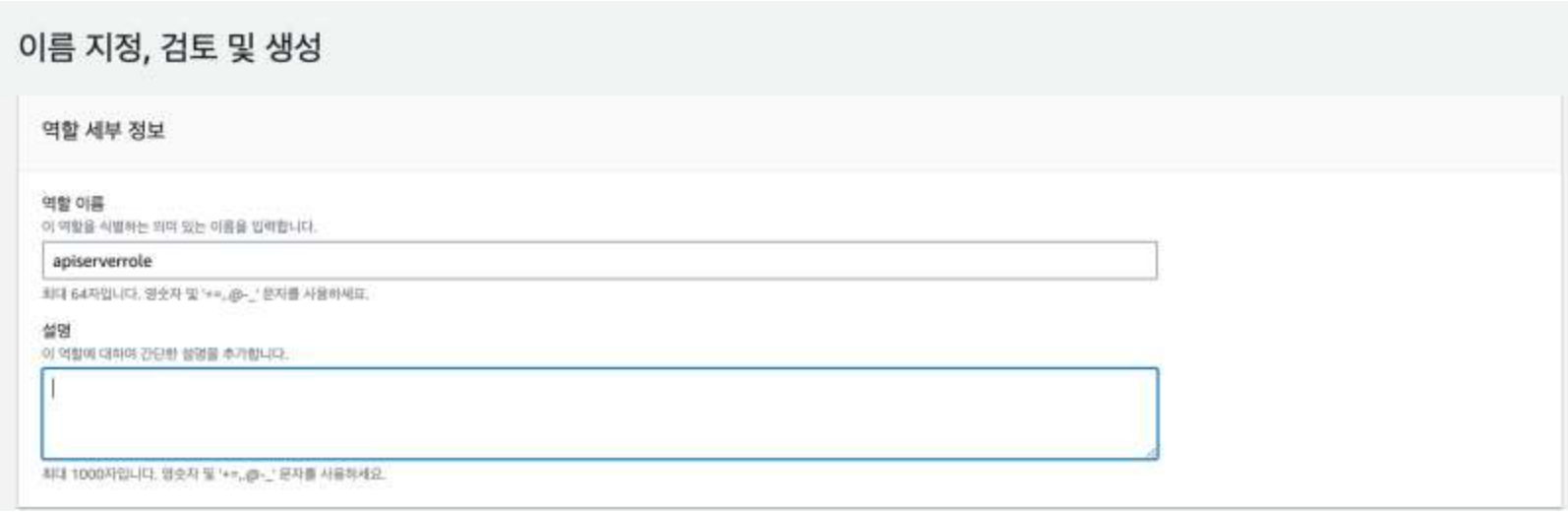
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - Role에 이전에 만든 Policy를 할당



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - Role에 이전에 만든 Policy를 할당



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● GitHub Action.yml 파일을 수정해서 역할을 가져올 수 있는지 확인

```
name: django

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  id-token: write # This is required for requesting the JWT
  contents: read  # This is required for actions/checkout
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● GitHub Action.yml 파일을 수정해서 역할을 가져올 수 있는지 확인

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up Python 3.10
        uses: actions/setup-python@v4
        with:
            python-version: '3.10'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Build Docker Compose
        run: docker-compose
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● GitHub Action.yml 파일을 수정해서 역할을 가져올 수 있는지 확인

```
- name: Configure AWS Credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::641022061021:role/itstudyoicd
    role-session-name: sampleSessionName
    aws-region: ap-northeast-2
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ Image Push
    - GitHub Action.yml 파일을 수정

```
name: django

on:
  push:
    branches: [ "main" ]

permissions:
  id-token: write
  contents: read

env:
  AWS_REGION: ap-northeast-2
  ECR_REPOSITORY: itstudydjangoecs
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ Image Push
    - ● GitHub Action.yml 파일을 수정

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up Python 3.10
        uses: actions/setup-python@v4
        with:
            python-version: '3.10'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ Image Push
    - GitHub Action.yml 파일을 수정

```
name: django

on:
  push:
    branches: [ "main" ]

permissions:
  id-token: write
  contents: read

env:
  AWS_REGION: ap-northeast-2
  ECR_REPOSITORY: itstudydjangoecs
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ Image Push
    - GitHub Action.yml 파일을 수정

```
- name: Configure AWS Credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::641022061021:role/imagepush
    role-session-name: sampleSessionName
    aws-region: ap-northeast-2

- name: Login to Amazon ECR
  id: login-ecr
  uses: aws-actions/amazon-ecr-login@62f4f872db3836360b72999f4b87f1ff13310f3a
```

AWS Container Service

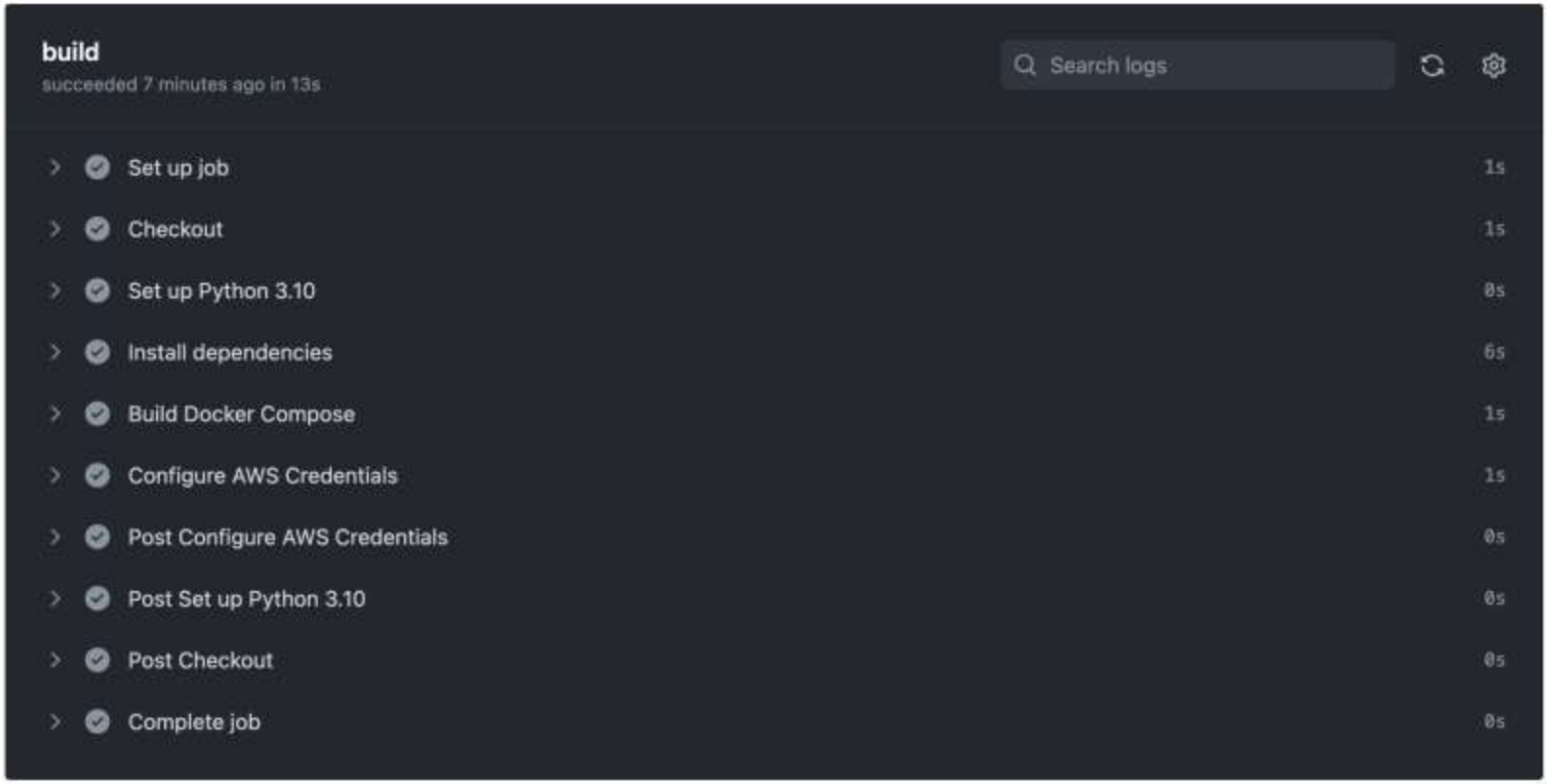
- ❖ Django Application CI/CD Pipeline 설정
  - ✓ Image Push
    - ● GitHub Action.yml 파일을 수정

```
- name: Build, tag, and push image to Amazon ECR
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        IMAGE_TAG: ${{ github.sha }}
      run: |
        # Build a docker container and
        # push it to ECR so that it can
        # be deployed to ECS.
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
        echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >>
$GITHUB_OUTPUT
```

AWS Container Service

❖ Django Application CI/CD Pipeline 설정

✓ OpenID Connect를 이용한 Git Hub AWS 로그인



AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - https://docs.github.com/en/enterprise-cloud@latest/actions/deployment/deploying-to-your-cloud-provider/deploying-to-amazon-elastic-container-service 참고
    - 사전 작업
      - ◆ ECR Repository 생성
      - ◆ ECS Cluster, Task, Service 생성
      - ◆ task-definition.json 파일 생성: ECS에서 생성한 task 의 JSON을 복사해서 프로젝트에 생성 – image 부분은 제거

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
name: django

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  id-token: write
  contents: read

env:
  AWS_REGION: ap-northeast-2
  ECR_REPOSITORY: itstudydjangoecs
  ECS_SERVICE: itstudyservice2
  ECS_CLUSTER: deploytest
  ECS_TASK_DEFINITION: ./task-definition.json
  CONTAINER_NAME: itstudy1
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up Python 3.10
        uses: actions/setup-python@v4
        with:
            python-version: '3.10'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Build Docker Compose
        run: docker-compose
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
- name: Configure AWS Credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::641022061021:role/imagepush
    role-session-name: sampleSessionName
    aws-region: ap-northeast-2

- name: Login to Amazon ECR
  id: login-ecr
  uses: aws-actions/amazon-ecr-
login@62f4f872db3836360b72999f4b87f1ff13310f3a
```

AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
- name: Build, tag, and push image to Amazon ECR
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        IMAGE_TAG: ${{ github.sha }}
      run: |
        # Build a docker container and
        # push it to ECR so that it can
        # be deployed to ECS.
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
        echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >>
$GITHUB_OUTPUT
```

AWS Container Service

❖ Django Application CI/CD Pipeline 설정

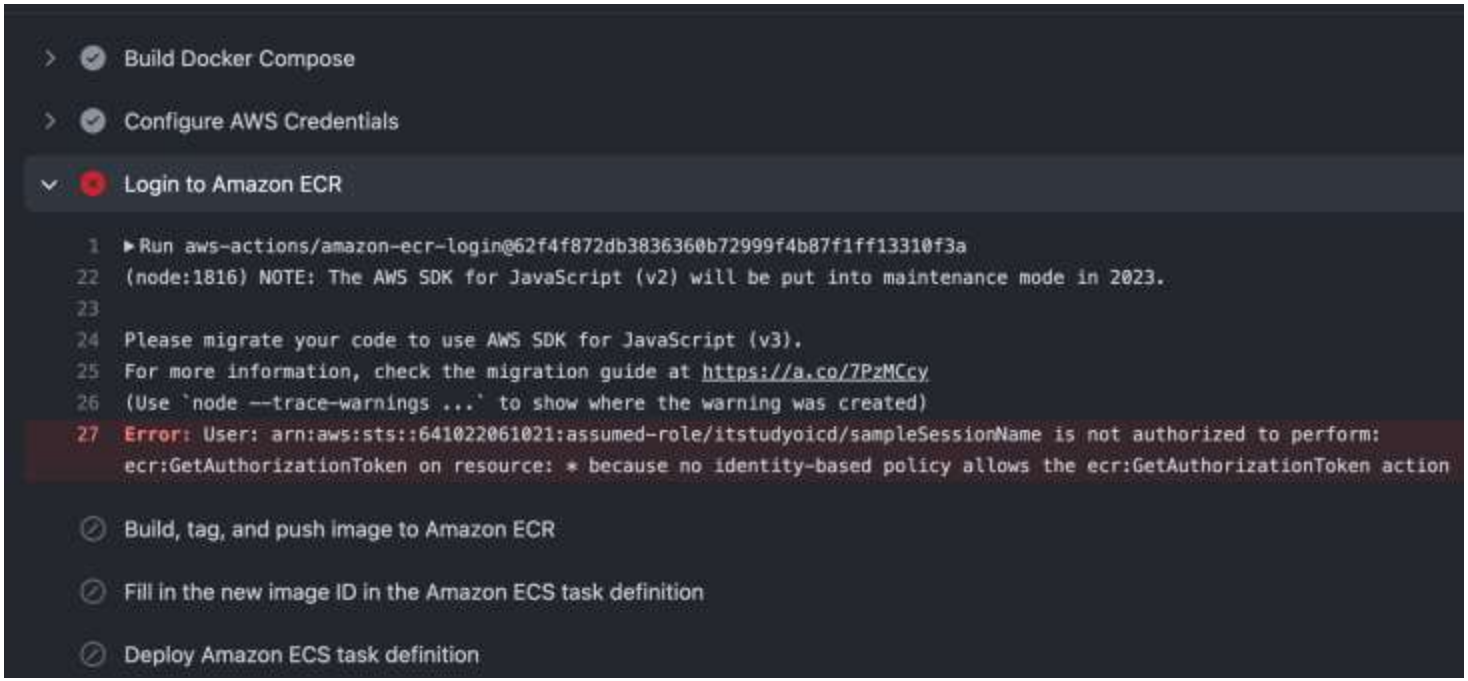
✓ ECR Push And Task Definition 생성 및 Service Update

● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
- name: Fill in the new image ID in the Amazon ECS task definition
        id: task-def
        uses: aws-actions/amazon-ecs-render-task-definition@v2
        with:
          task-definition: ${{ env.ECS_TASK_DEFINITION }}
          container-name: ${{ env.CONTAINER_NAME }}
          image: ${{ steps.build-image.outputs.image }}

      - name: Deploy Amazon ECS task definition
        uses: aws-actions/amazon-ecs-deploy-task-definition@df9643053eda01f169e64a0e60233aacca83799a
        with:
          task-definition: ${{ steps.task-def.outputs.task-definition }}
          service: ${{ env.ECS_SERVICE }}
          cluster: ${{ env.ECS_CLUSTER }}
          wait-for-service-stability: true
```

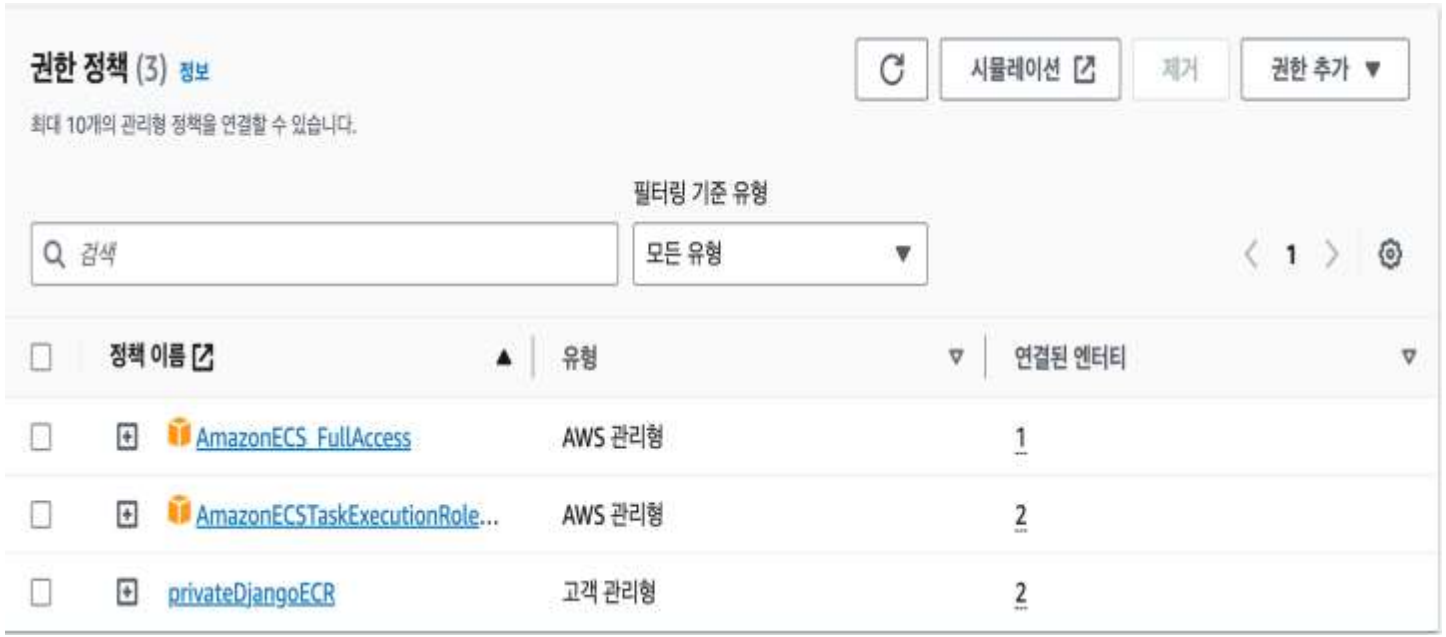
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정



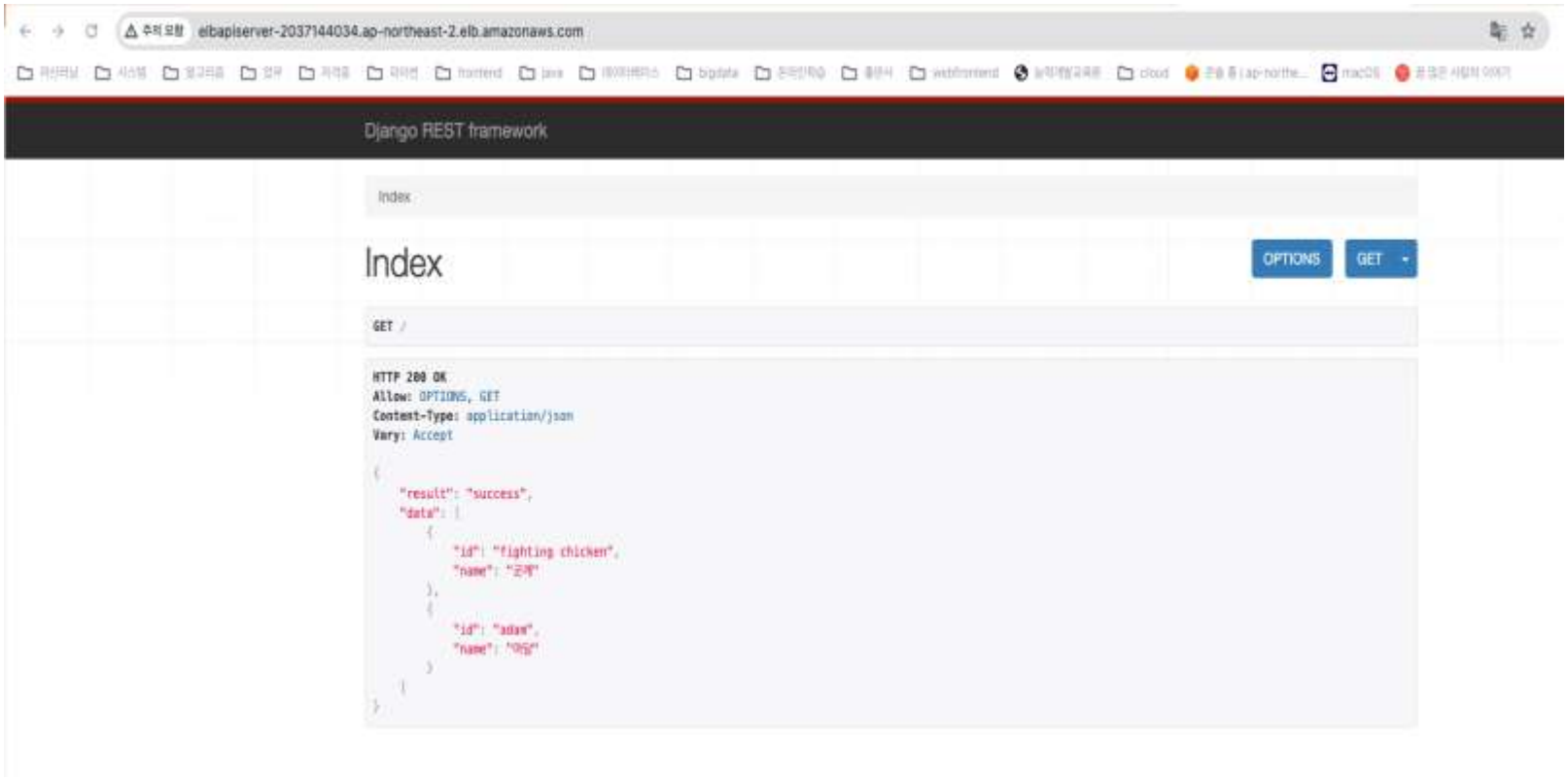
AWS Container Service

- ❖ Django Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● 권한 부여 – IAM에서 이전에 만든 Role에 ECS 관련 권한 추가 후 다시 빌드



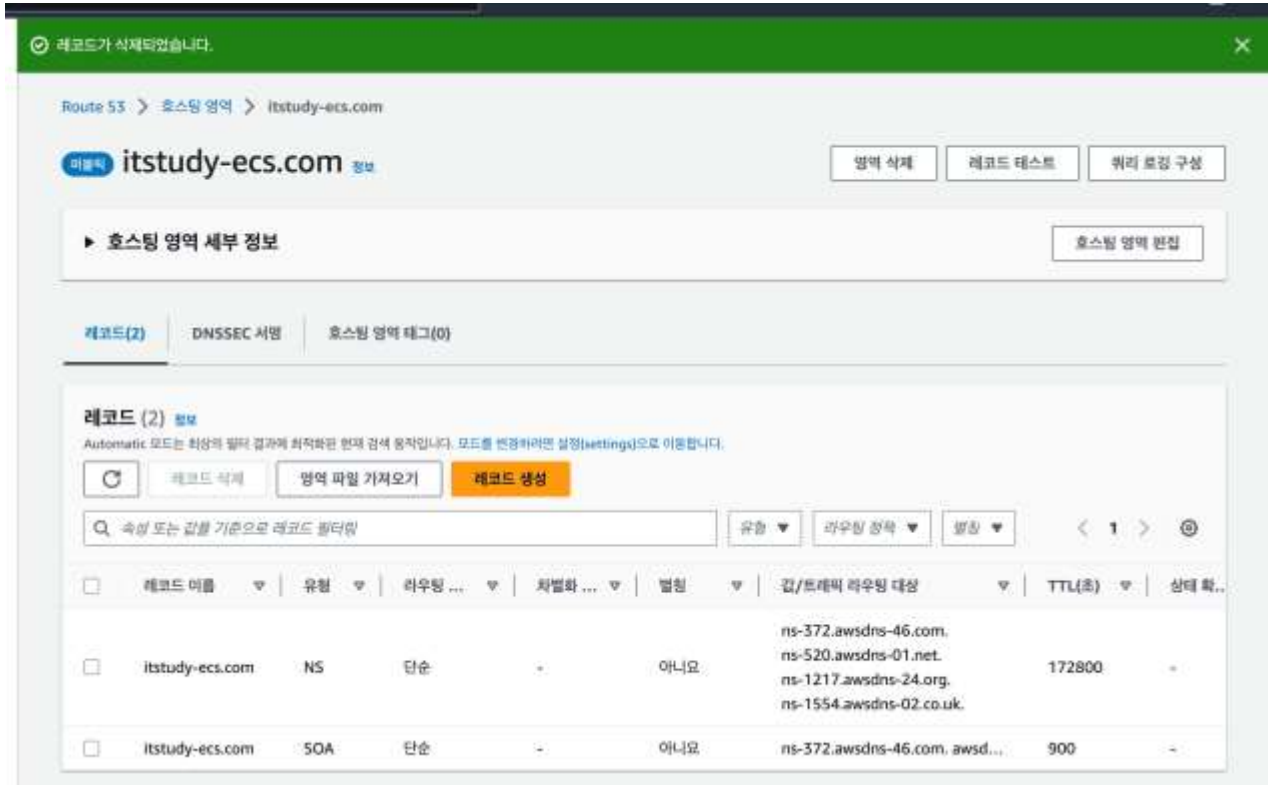
AWS Container Service

- Django Application CI/CD Pipeline 설정
  - ✓ 내용을 수정하고 푸시



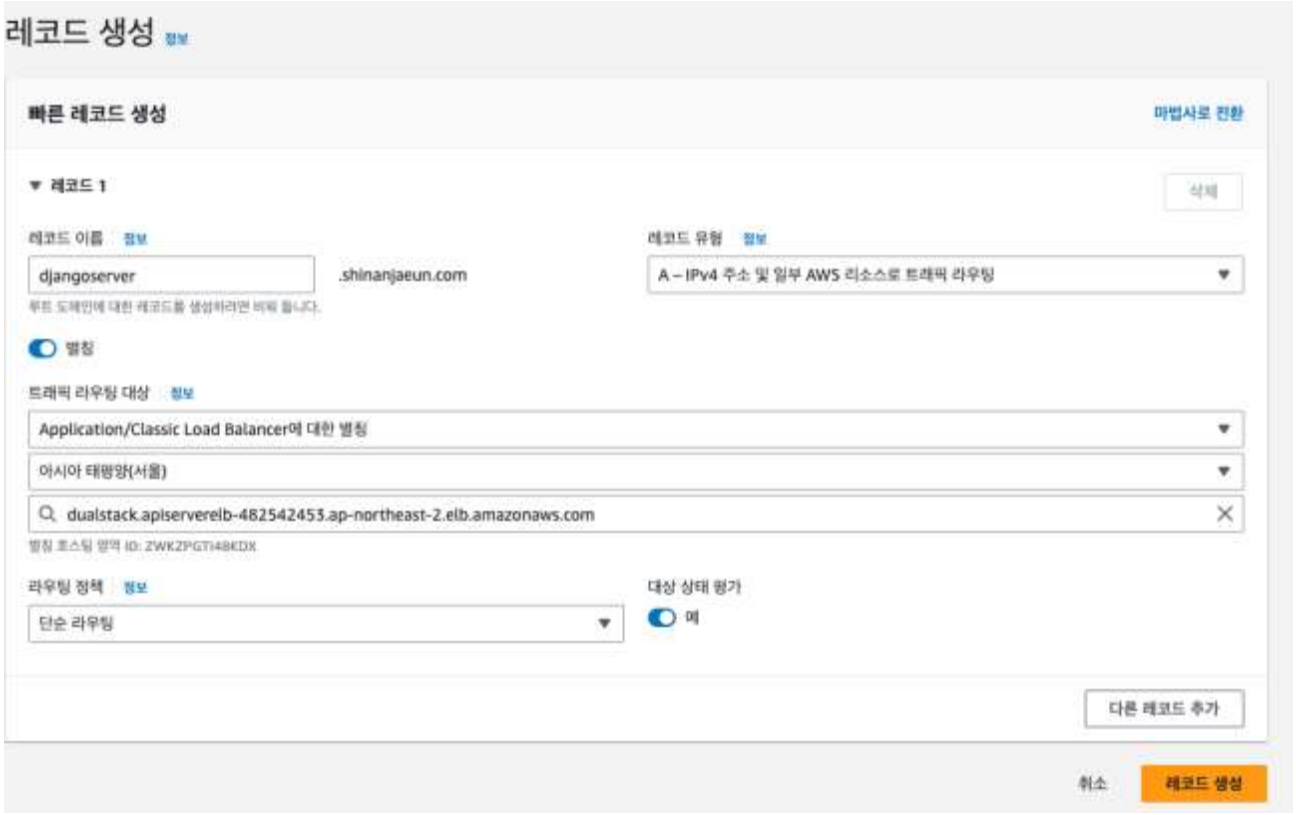
AWS Container Service

- ❖ 도메인 연결
  - ✓ Domain 연결
    - ● 호스팅 영역에서 호스팅 영역 선택



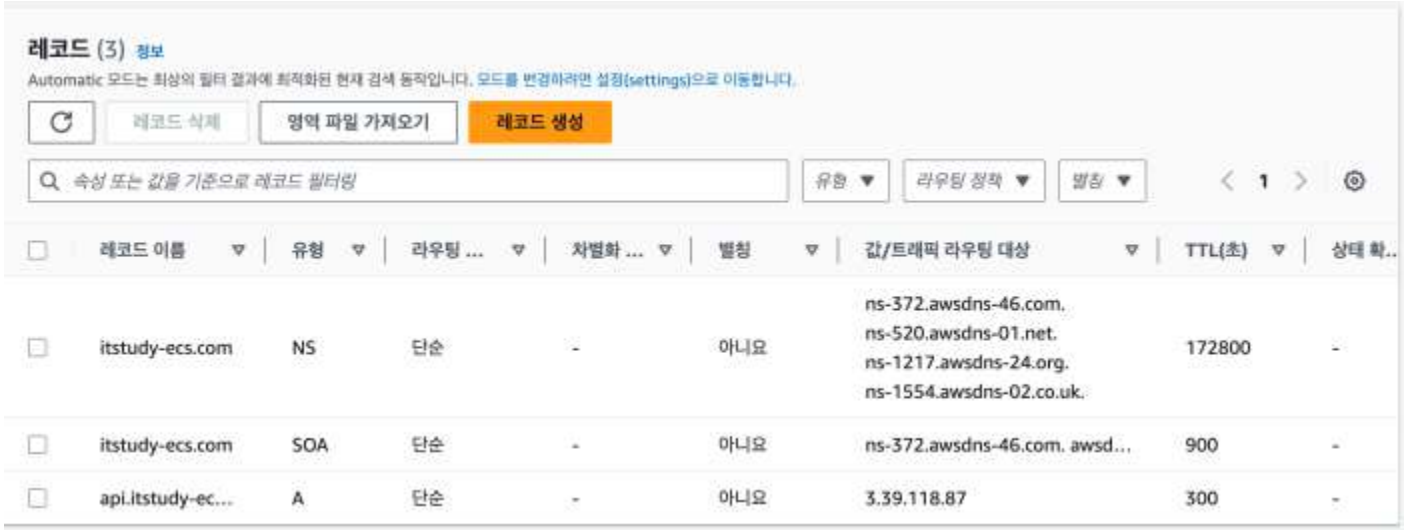
AWS Container Service

- ❖ 도메인 연결
  - ✓ Domain 연결
    - ● 레코드 생성 버튼을 누르고 정보 입력



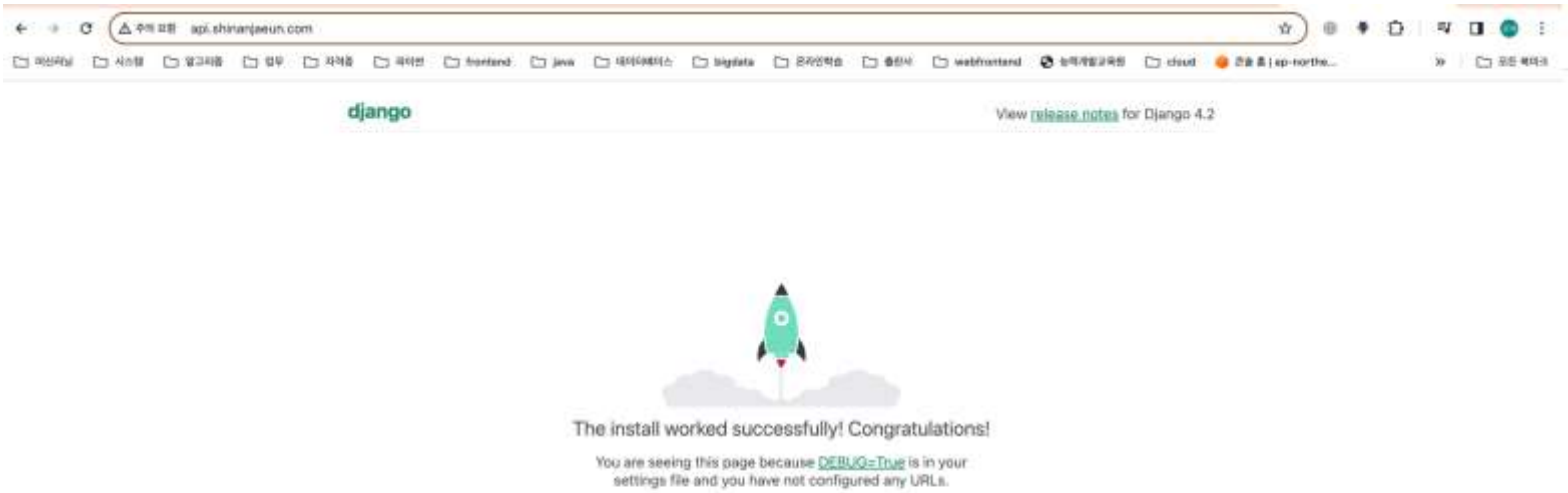
AWS Container Service

- ❖ 도메인 연결
  - ✓ Domain 연결
    - ● 레코드 생성 버튼을 누르고 정보 입력



AWS Container Service

- 도메인 연결
  - ✓ 브라우저에서 확인



AWS Container Service

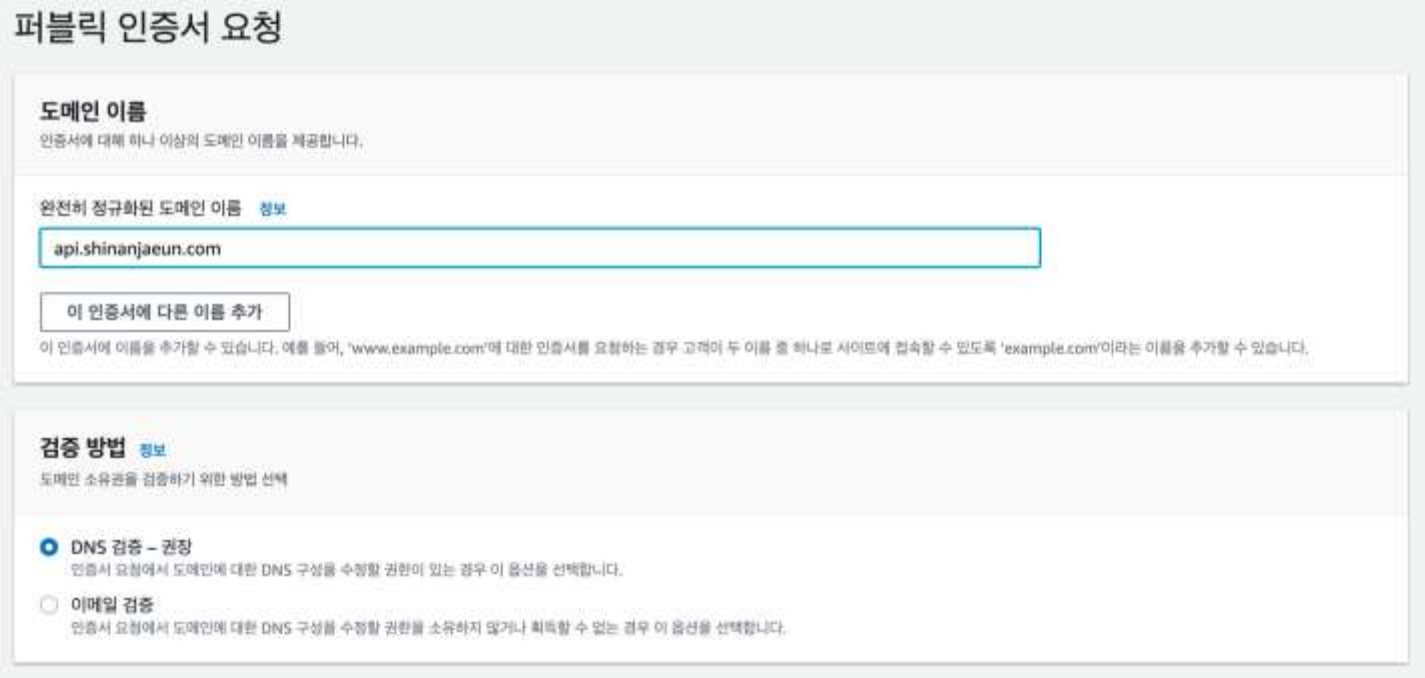
- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - AWS Certificate Manager 서비스에 접속

AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서 요청 클릭

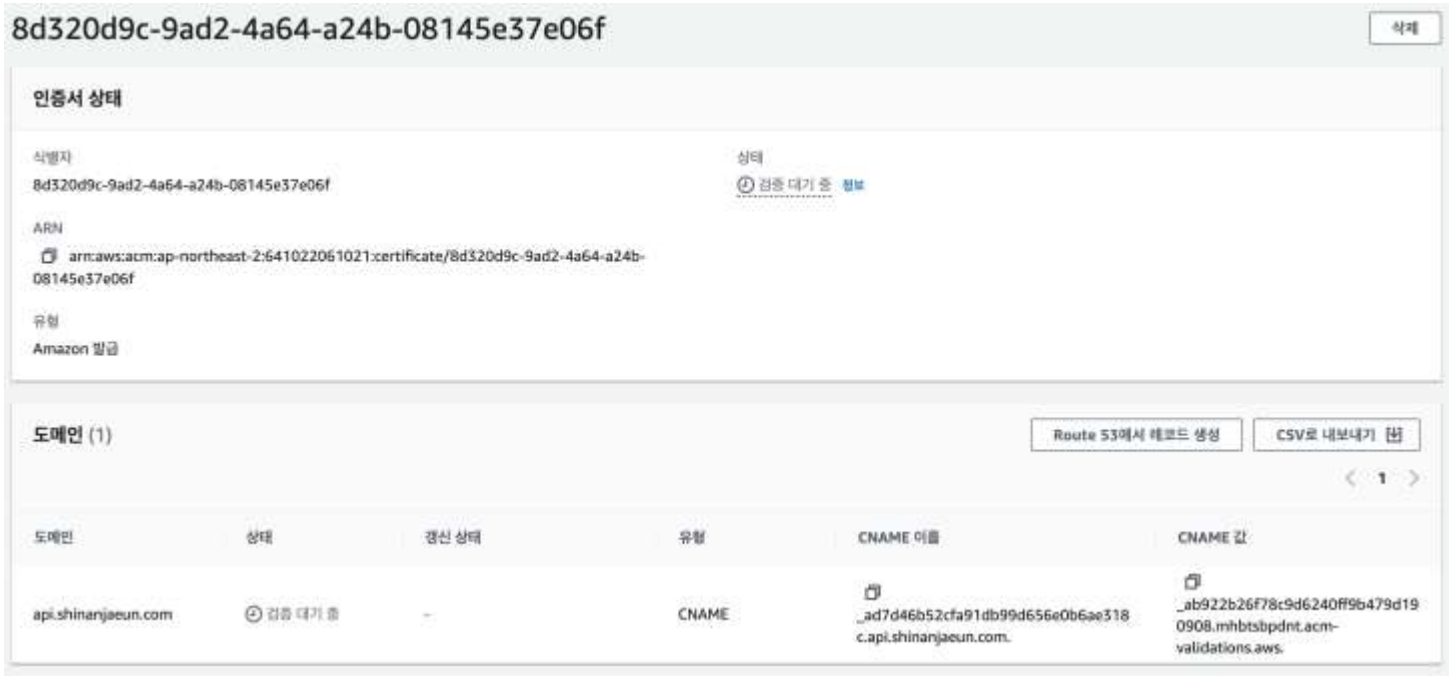
AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서 요청을 위한 도메인 설정



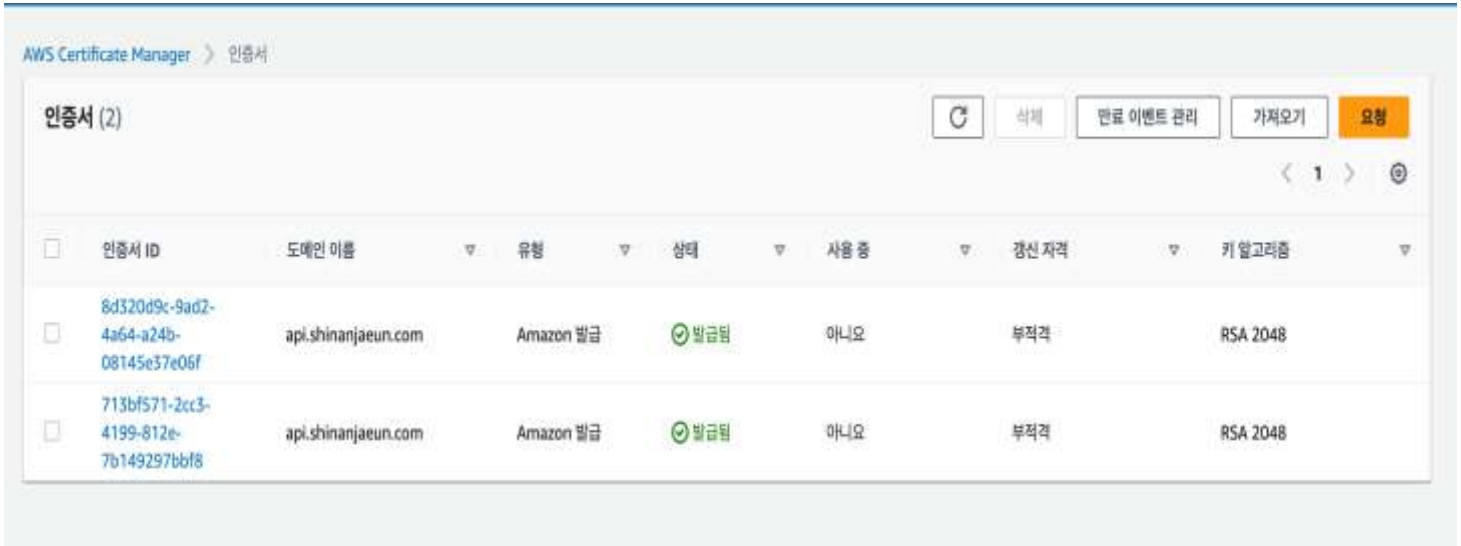
AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서를 클릭하고 Route53에서 레코드 생성 클릭



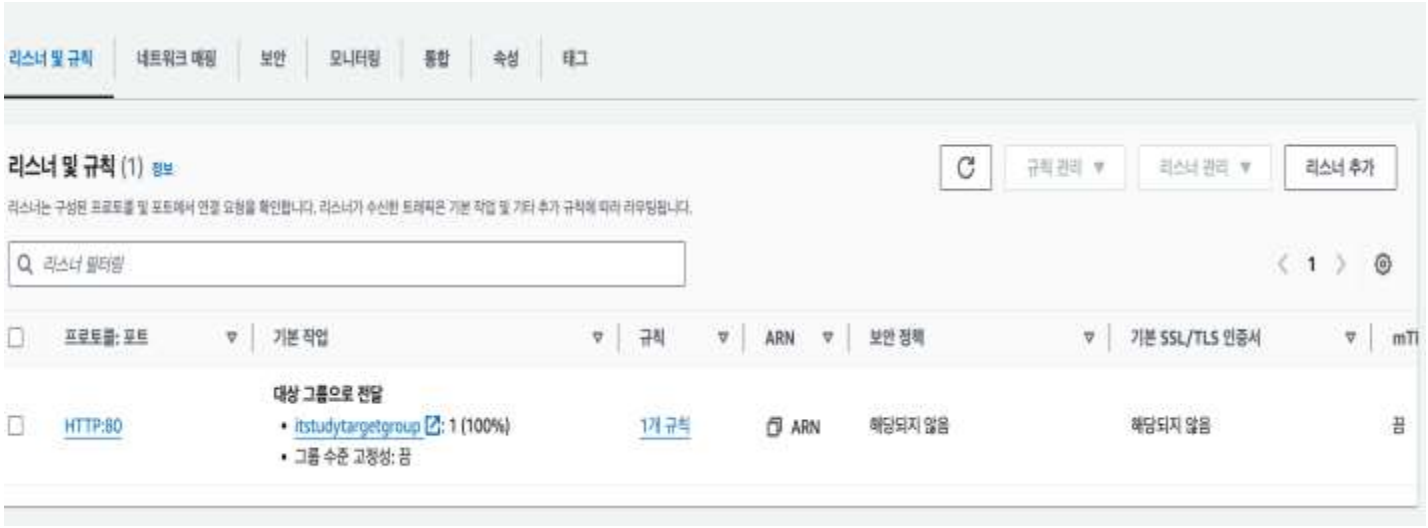
AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서를 클릭하고 Route53에서 레코드 생성 클릭



AWS Container Service

- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 로드 밸런서에 접속해서 [리스너 추가] 버튼 클릭



AWS Container Service

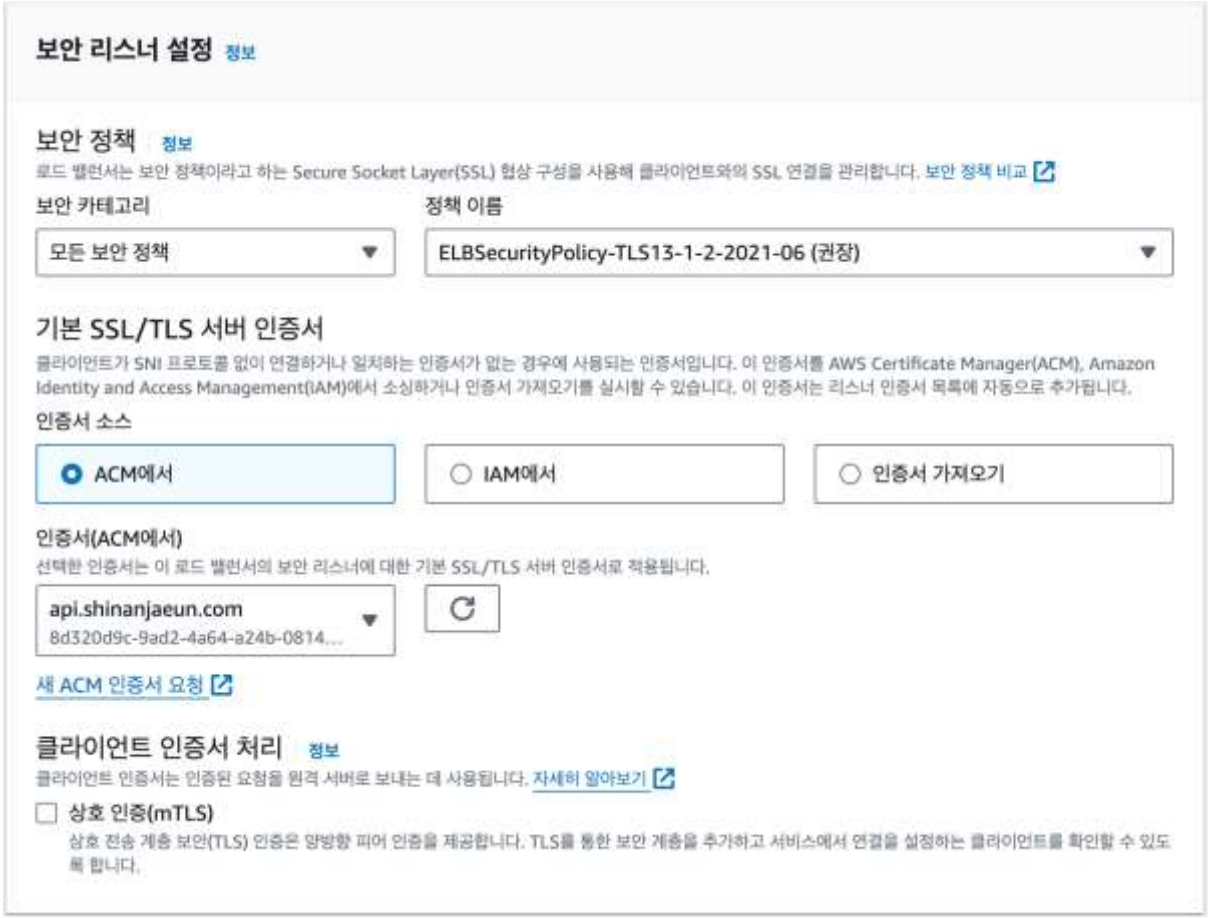
- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 로드 밸런서 설정 – 프로토콜을 HTTPS로 변경

AWS Container Service

- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 로드 밸런서 설정 – 대상 그룹 선택

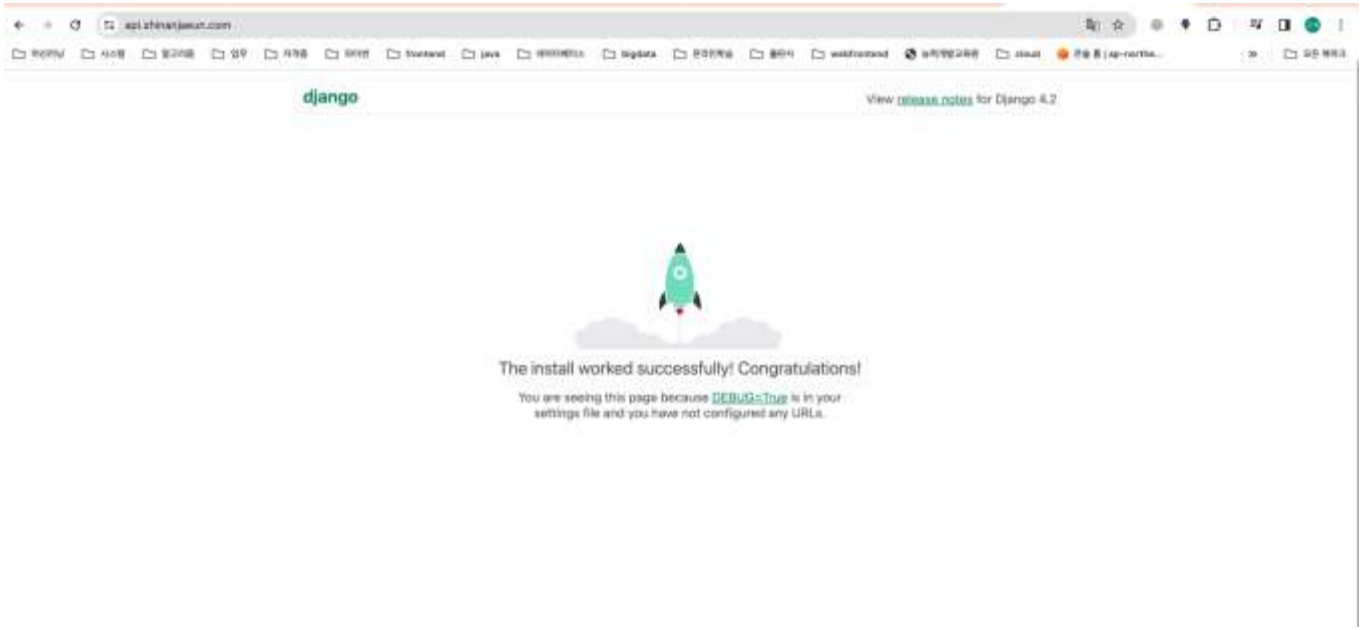
AWS Container Service

- HTTPS 적용
  - ✓ ELB에 인증서 등록
    - 로드 밸런서 설정 – 인증서 선택



AWS Container Service

- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 확인 – 브라우저에서 https 로 접속



AWS Container Service

- Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Spring Boot Application 작성
    - Spring Boot Application 생성
    - 프로젝트에 Controller 추가 - FrontController

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FrontController {
    @GetMapping("/")
    public Map<String, Object> uploadFile() {
        Map<String, Object>data = new HashMap<String, Object>();
        data.put("result", "success");
```

AWS Container Service

- Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Spring Boot Application 작성
    - 프로젝트에 Controller 추가 - FrontController

```
        @SuppressWarnings("rawtypes")
        List <Map> list = new ArrayList<Map>();

        Map<String, String> map1 = new HashMap<>();
        map1.put("id", "itstudy");
        map1.put("name", "군계");

        Map<String, String> map2 = new HashMap<>();
        map2.put("id", "itggangpae");
        map2.put("name", "아담");

        list.add(map1);
        list.add(map2);

        data.put("data", list);

        return data;
    }
}
```

AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Spring Boot Application 작성
    - ● application.yml 파일에 서버 포트 설정

```
server:
  port: 80
```

AWS Container Service

- Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Spring Boot Application 실행 확인

AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ 이미지 생성 및 확인
    - ● Spring Boot Application 의 루트 디렉토리에 Dockerfile을 작성
      - ◆ jar 로 생성

```
FROM amazoncorretto:17
CMD ["./mvnw", "clean", "package"]
ARG JAR_FILE=target/*.jar
COPY ./build/libs/*.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

      - ◆ war 로 생성

```
FROM amazoncorretto:17
CMD ["./mvnw", "clean", "package"]
ARG WAR_FILE=target/*.war
COPY ./build/libs/*.war app.war
ENTRYPOINT ["java", "-jar", "app.war"]
```

AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ 이미지 생성 및 확인
    - 터미널에서 build

      ./gradlew clean build
    - 이미지 빌드 및 생성

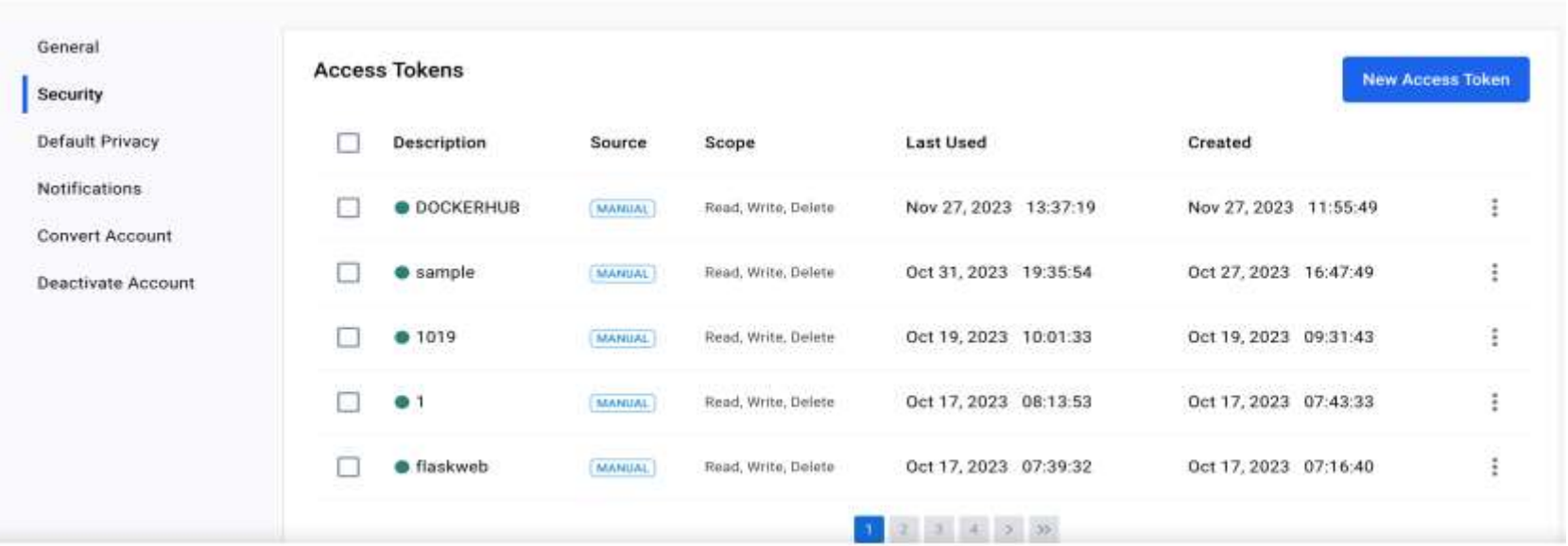
      docker build -f Dockerfile -t springapiserver:0.0.1 .
    - 이미지 확인

      docker images
    - 컨테이너 생성 및 실행

      docker run -d --name springapiserver -p 80:80 springapiserver:0.0.1

AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - Docker Hub에 Repository 생성
    - Docker Hub에서 Access Token 생성 – 오른쪽 상단의 아이디 옆을 클릭한 후 [Account Settings] – [Security] – [Access Token] – [New Access Token]



AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - Git Hub에 자바 코드 업로드
    - GIT HUB의 Repository에 Secret Key 등록: [Settings] – [Secrets and variable] – [Actions] – [New repository secret]
      - ◆ DOCKERHUB_USERNAME
      - ◆ DOCKERHUB_TOKEN

AWS Container Service

- Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - .github/workflows/ 디렉토리에 yaml 파일을 생성하고 push

```
name: Java CI with Gradle

# 동작 조건 설정 : main 브랜치에 push 혹은 pull request가 발생할 경우 동작한다.
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read
```

AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - .github/workflows/ 디렉토리에 yaml 파일을 생성하고 push

```
jobs:
  # Spring Boot 애플리케이션을 빌드하여 도커 허브에 푸시하는 과정
  build-docker-image:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    # 1. Java 17 세팅
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'

    # 2. Spring Boot 애플리케이션 빌드
    - name: Build with Gradle
      uses: gradle/gradle-build-
action@67421db6bd0bf253fb4bd25b31ebb98943c375e1
      with:
        arguments: clean bootJar
```

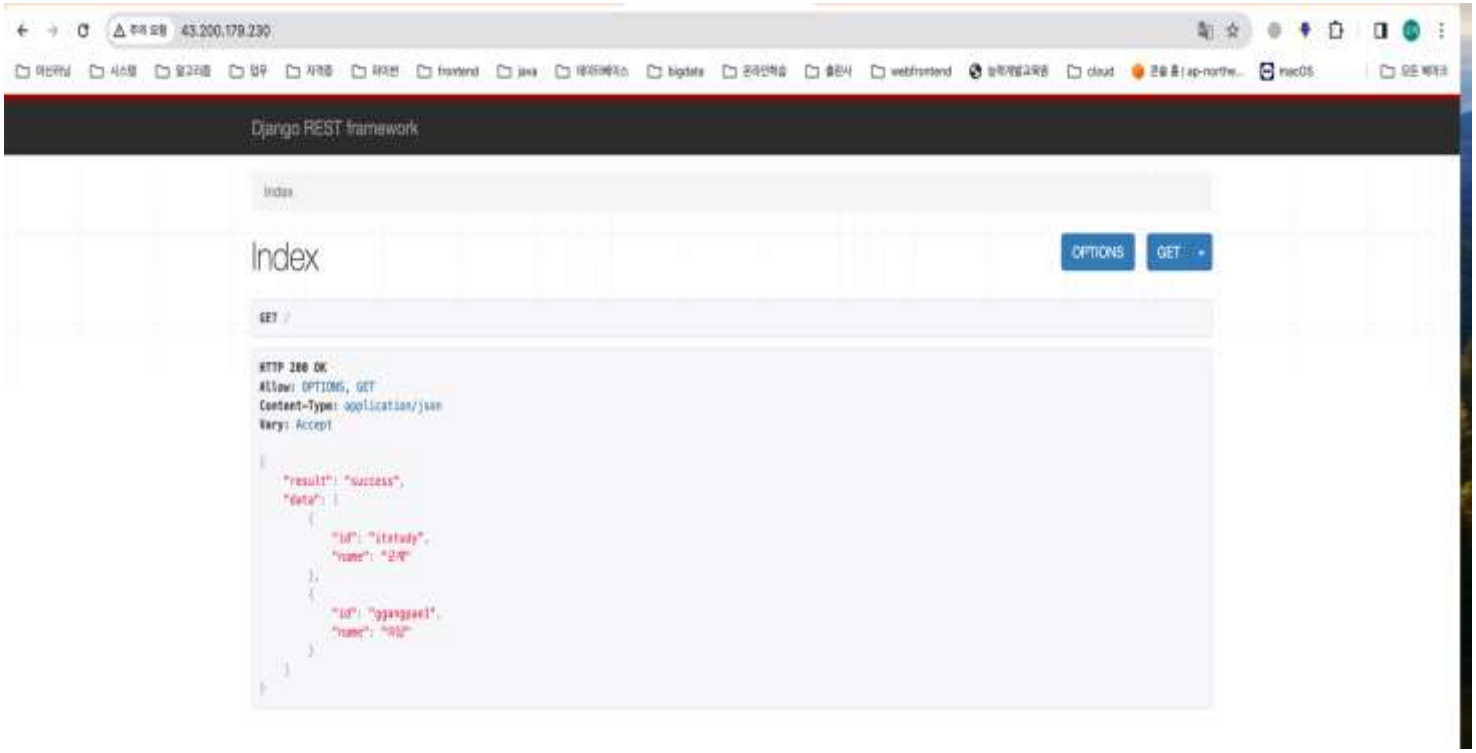
AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ Git Action을 활용한 Docker Hub에 이미지 배포
    - ● .github/workflows/ 디렉토리에 yaml 파일을 생성하고 push

```
# 3. DockerHub 로그인
- name: Login to DockerHub
  uses: docker/login-action@v1
  with:
    username: ${{secrets.DOCKERHUB_USERNAME}}
    password: ${{secrets.DOCKERHUB_TOKEN}}
# 4. 이미지 업로드
- name: build and release to DockerHub
  env:
    NAME: ${{secrets.DOCKERHUB_USERNAME}}
    REPO: springapiserver
  run: |
    docker build -t $REPO .
    docker tag $REPO:latest $NAME/$REPO:latest
    docker push $NAME/$REPO:latest
```

AWS Container Service

- ❖ Spring 애플리케이션을 Docker Image을 이용한 EC2 배포
  - ✓ 이미지 다운로드 및 실행
    - sudo docker run -p 80:80 -d --rm ggangpae1/springapiserver
  - ✓ 브라우저에서 퍼블릭 IP로 확인



AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ Docker-Compose 사용
    - ● 프로젝트에 docker-compose.yml 파일을 생성하고 작성

```
version: "3"

services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    command: "java -jar app.jar"
    ports:
      - "80:80"
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ Docker-Compose 사용
    - ● 콘솔 명령으로 확인

      docker-compose -f docker-compose.yml up --build

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ Docker-Compose 사용
    - ● Git Hub Action을 위한 yml 파일을 수정해서 테스트

```
name: Java CI with Gradle

# 동작 조건 설정 : main 브랜치에 push 혹은 pull request가 발생할 경우 동작한다.
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ Docker-Compose 사용
    - ● Git Hub Action을 위한 yml 파일을 수정해서 테스트

```
jobs:
  # Spring Boot 애플리케이션을 빌드하여 도커 허브에 푸시하는 과정
  build-docker-image:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    # 1. Java 17 세팅
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'

    # 2. Spring Boot 애플리케이션 빌드
    - name: Build with Gradle
      uses: gradle/gradle-build-
action@67421db6bd0bf253fb4bd25b31ebb98943c375e1
      with:
        arguments: clean bootJar
```

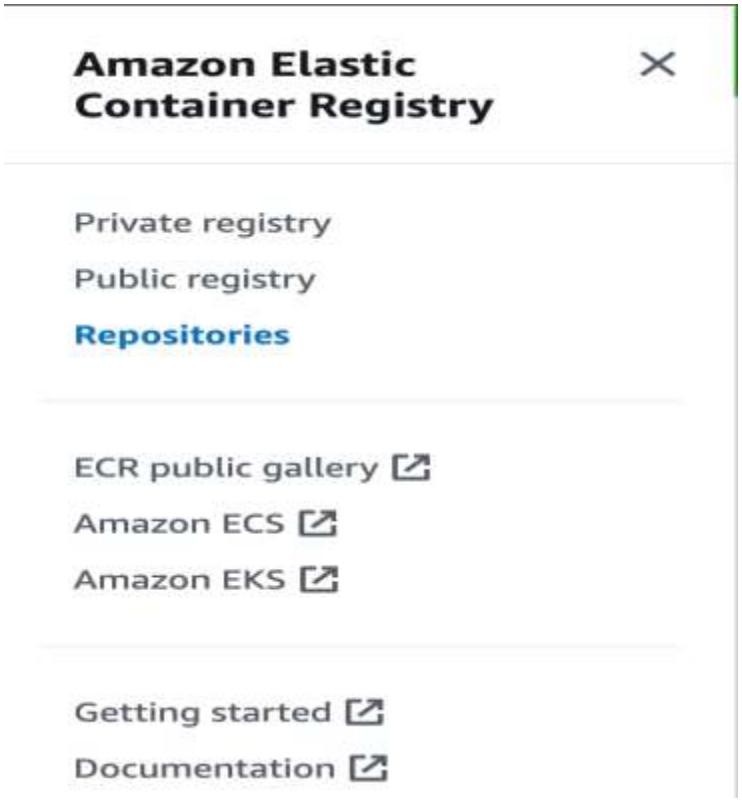
AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ Docker-Compose 사용
    - ● Git Hub Action을 위한 yml 파일을 수정해서 테스트

```
- name: Build Docker Compose
  run: docker-compose
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성 - springapiserver



AWS Container Service

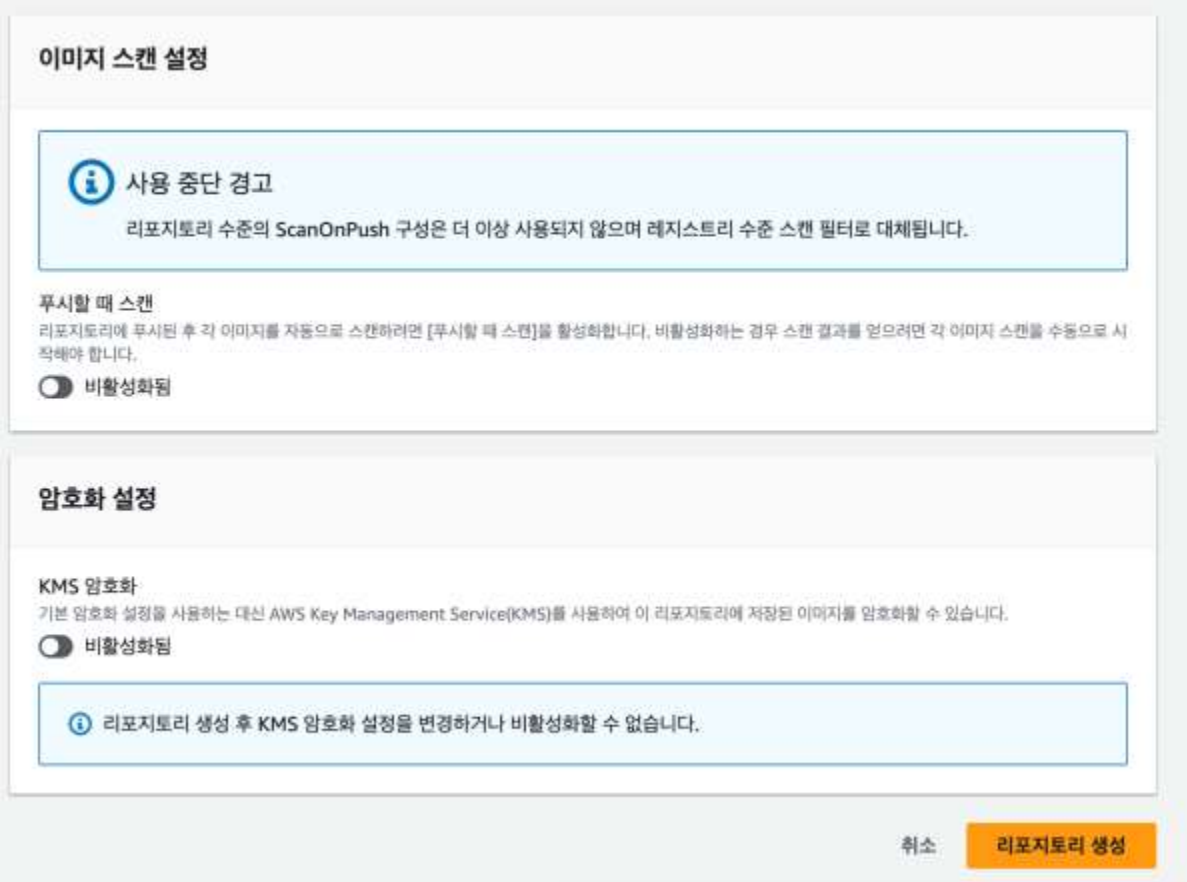
❖ Spring Boot Application CI/CD Pipeline 설정

✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성 - springapiserver

AWS Container Service

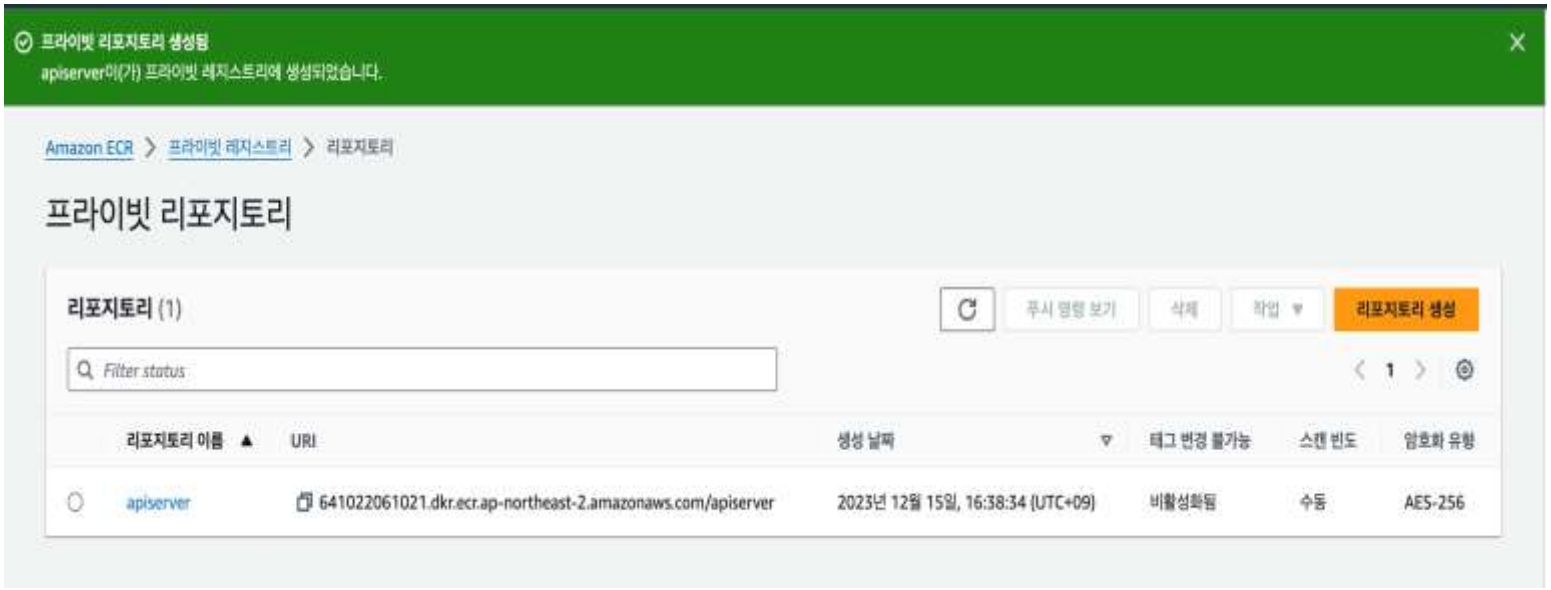
❖ Spring Boot Application CI/CD Pipeline 설정

✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성 - springapiserver



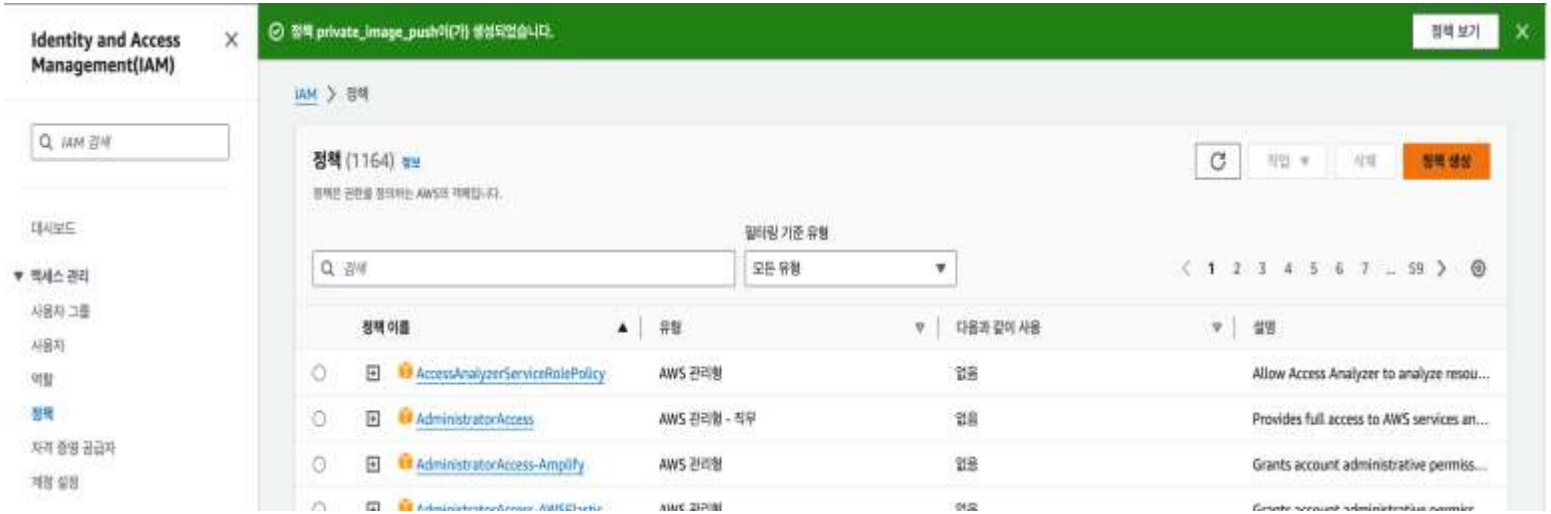
AWS Container Service

- Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR(Elastic Container Registry) 서비스에 접속해서 Repository 생성 - springapiserver



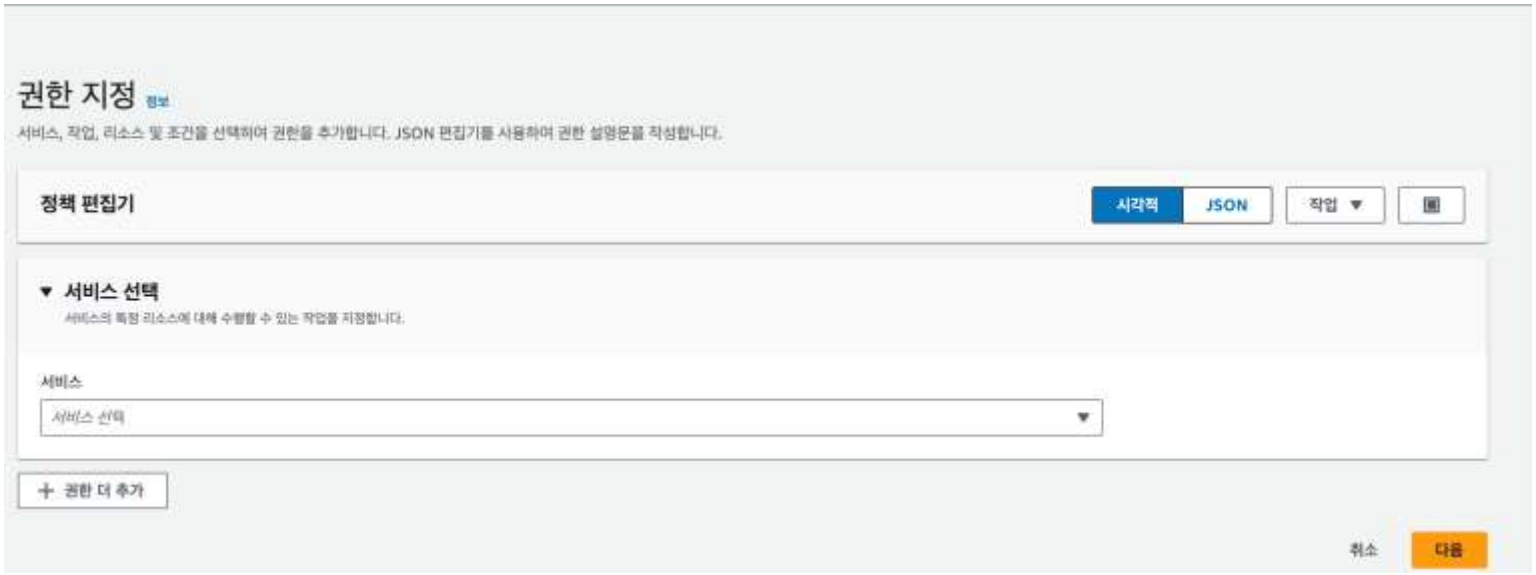
AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ IAM에서 ECR에 이미지를 push할 수 있는 정책 생성(private_image_push): 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성
    - 정책 생성 클릭



AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ IAM에서 ECR에 이미지를 push할 수 있는 정책 생성(private_image_push): 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성
    - JSON 클릭



AWS Container Service

- Spring Boot Application CI/CD Pipeline 설정
  - ✓ IAM에서 ECR에 이미지를 push할 수 있는 정책 생성(private_image_push): 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성
    - JSON 수정

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:CompleteLayerUpload",
                "ecr:UploadLayerPart",
                "ecr:InitiateLayerUpload",
                "ecr:BatchCheckLayerAvailability",
                "ecr:PutImage"
            ],
            "Resource": "arn:aws:ecr:ap-northeast-2:753510247212:repository/apiserver"
        },
```

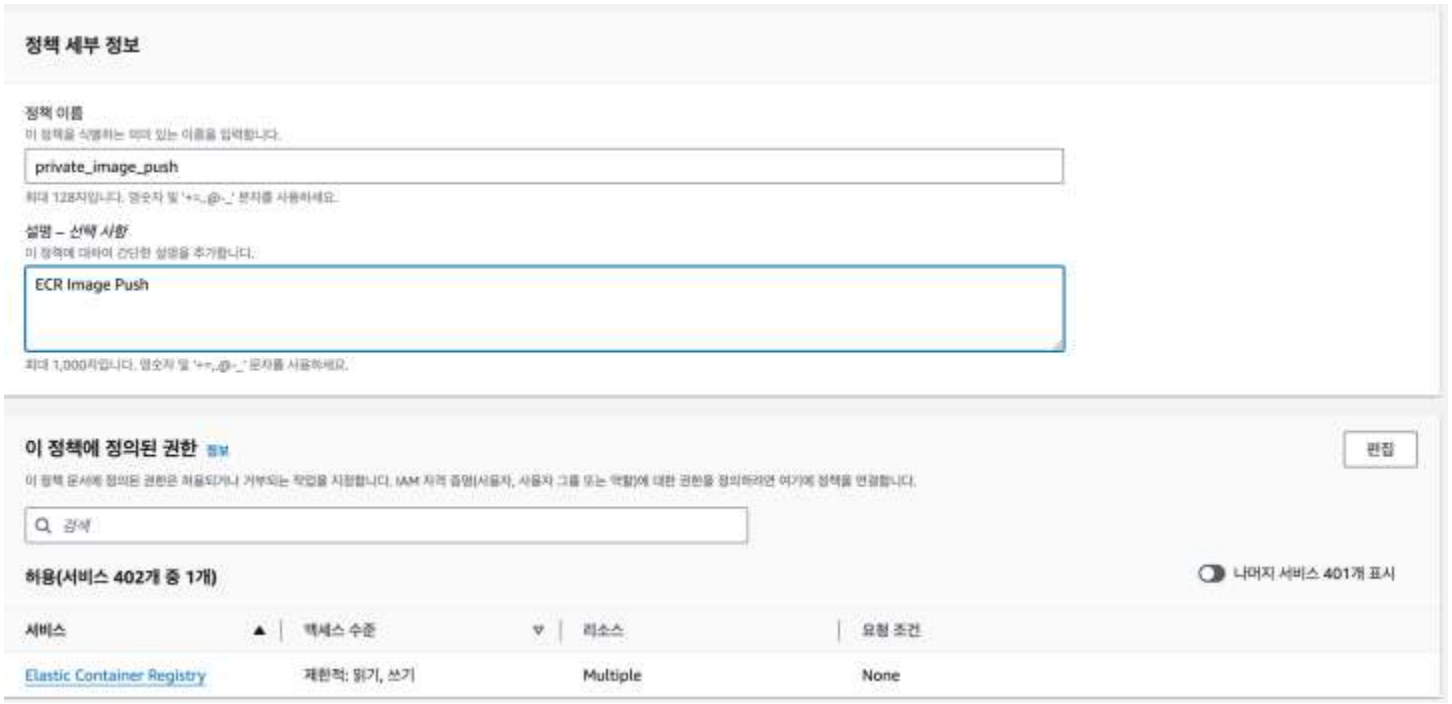
AWS Container Service

- Spring Boot Application CI/CD Pipeline 설정
  - ✓ IAM에서 ECR에 이미지를 push할 수 있는 정책 생성(private_image_push): 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성
    - JSON 수정

```
            {
                "Effect": "Allow",
                "Action": "ecr:GetAuthorizationToken",
                "Resource": "*"
            }
        ]
    }
```

AWS Container Service

- Spring Boot Application CI/CD Pipeline 설정
  - ✓ IAM에서 ECR에 이미지를 push할 수 있는 정책 생성(private_image_push): 리전을 수정하고 숫자 와 이름은 ECR에서 가져와서 작성



AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - IAM 에서 자격 증명 공급자를 클릭해서 추가

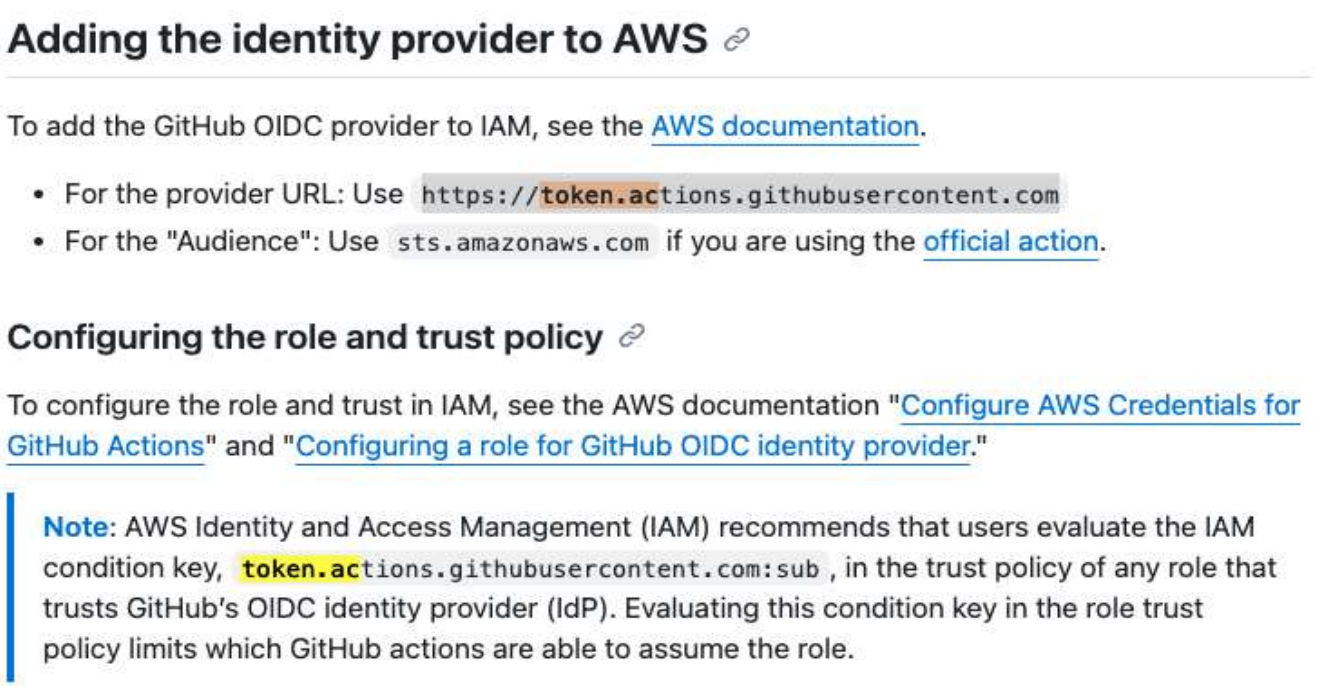
Adding the identity provider to AWS

To add the GitHub OIDC provider to IAM, see the AWS documentation.

- For the provider URL: Use `https://token.actions.githubusercontent.com`
- For the "Audience": Use `sts.amazonaws.com` if you are using the official action.

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● IAM 에서 자격 증명 공급자를 클릭해서 추가
      - ◆ URL 확인: https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services



Adding the identity provider to AWS

To add the GitHub OIDC provider to IAM, see the AWS documentation.

- For the provider URL: Use `https://token.actions.githubusercontent.com`
- For the "Audience": Use `sts.amazonaws.com` if you are using the official action.

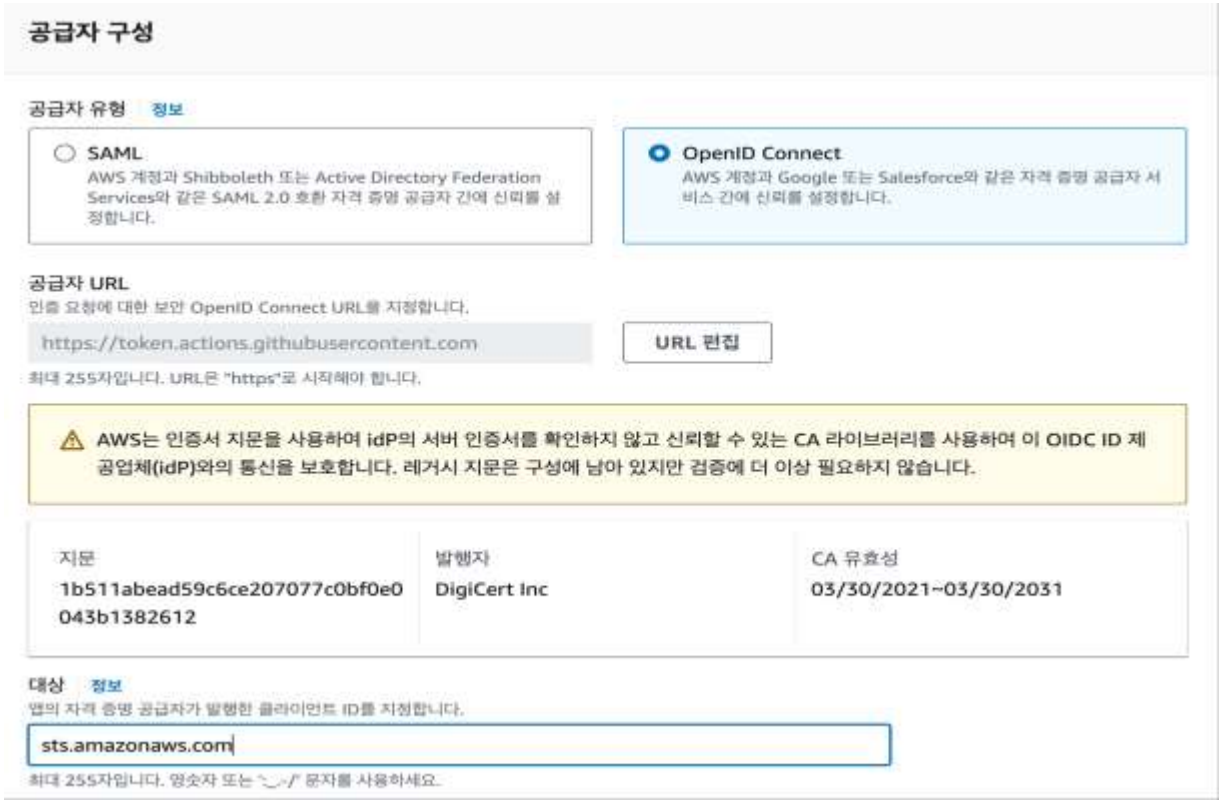
Configuring the role and trust policy

To configure the role and trust in IAM, see the AWS documentation "Configure AWS Credentials for GitHub Actions" and "Configuring a role for GitHub OIDC identity provider."

> **Note**: AWS Identity and Access Management (IAM) recommends that users evaluate the IAM condition key, `token.actions.githubusercontent.com:sub`, in the trust policy of any role that trusts GitHub's OIDC identity provider (IdP). Evaluating this condition key in the role trust policy limits which GitHub actions are able to assume the role.

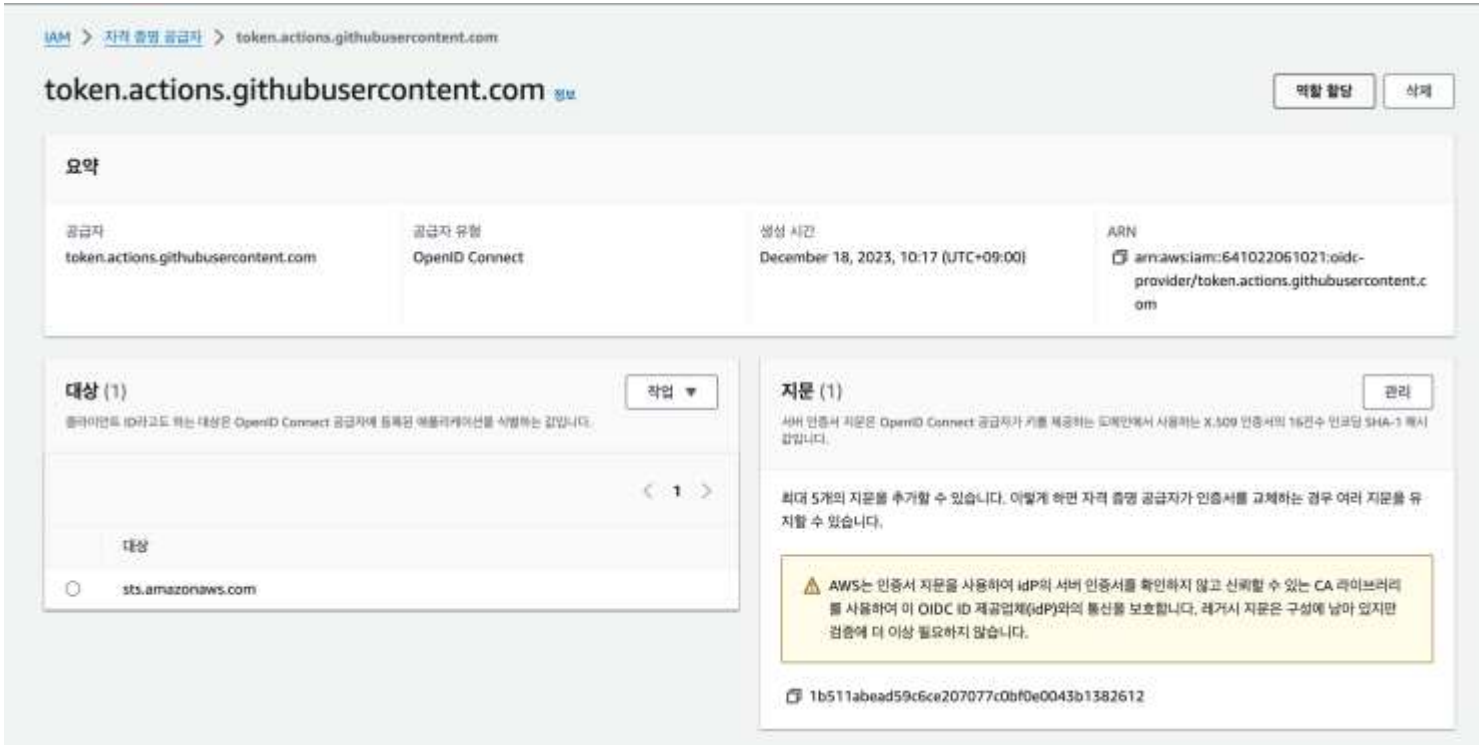
AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - IAM 에서 자격 증명 공급자를 클릭해서 추가



AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● 자격 증명 공급자에 새로운 역할 할당

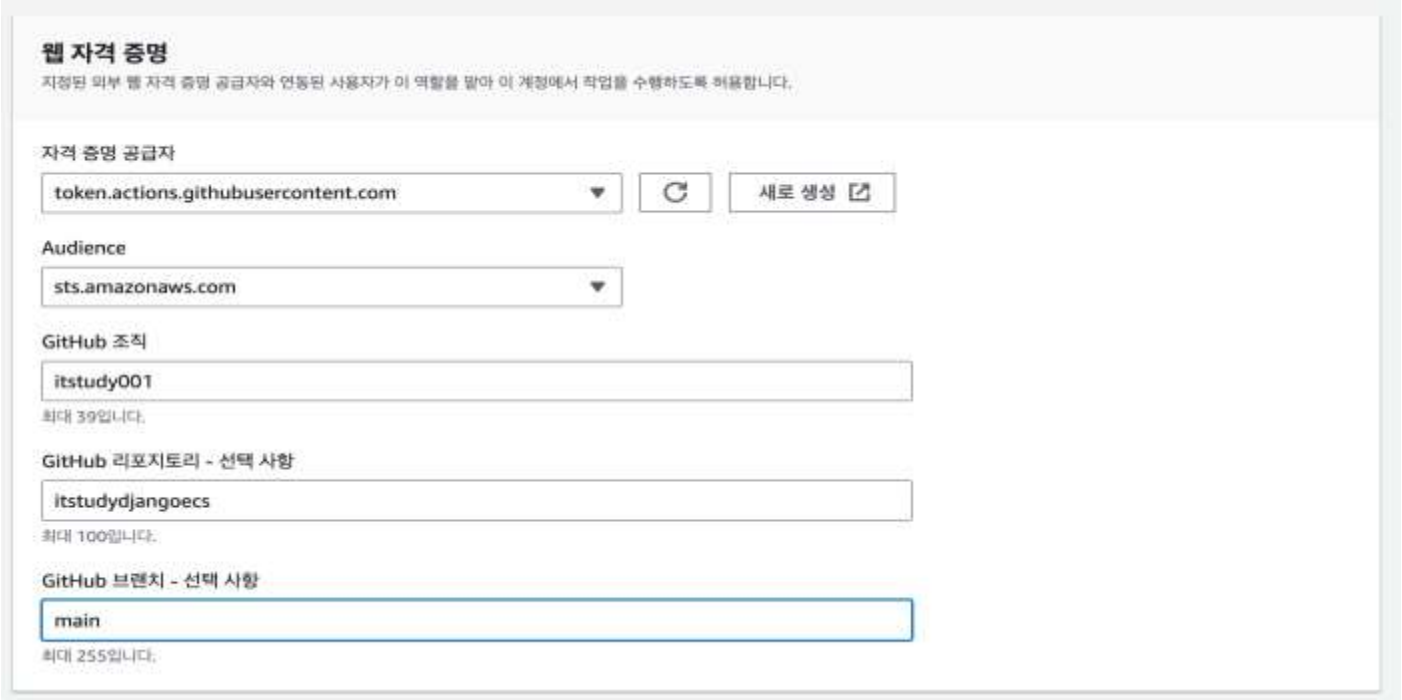


AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● 자격 증명 공급자에 새로운 역할 할당

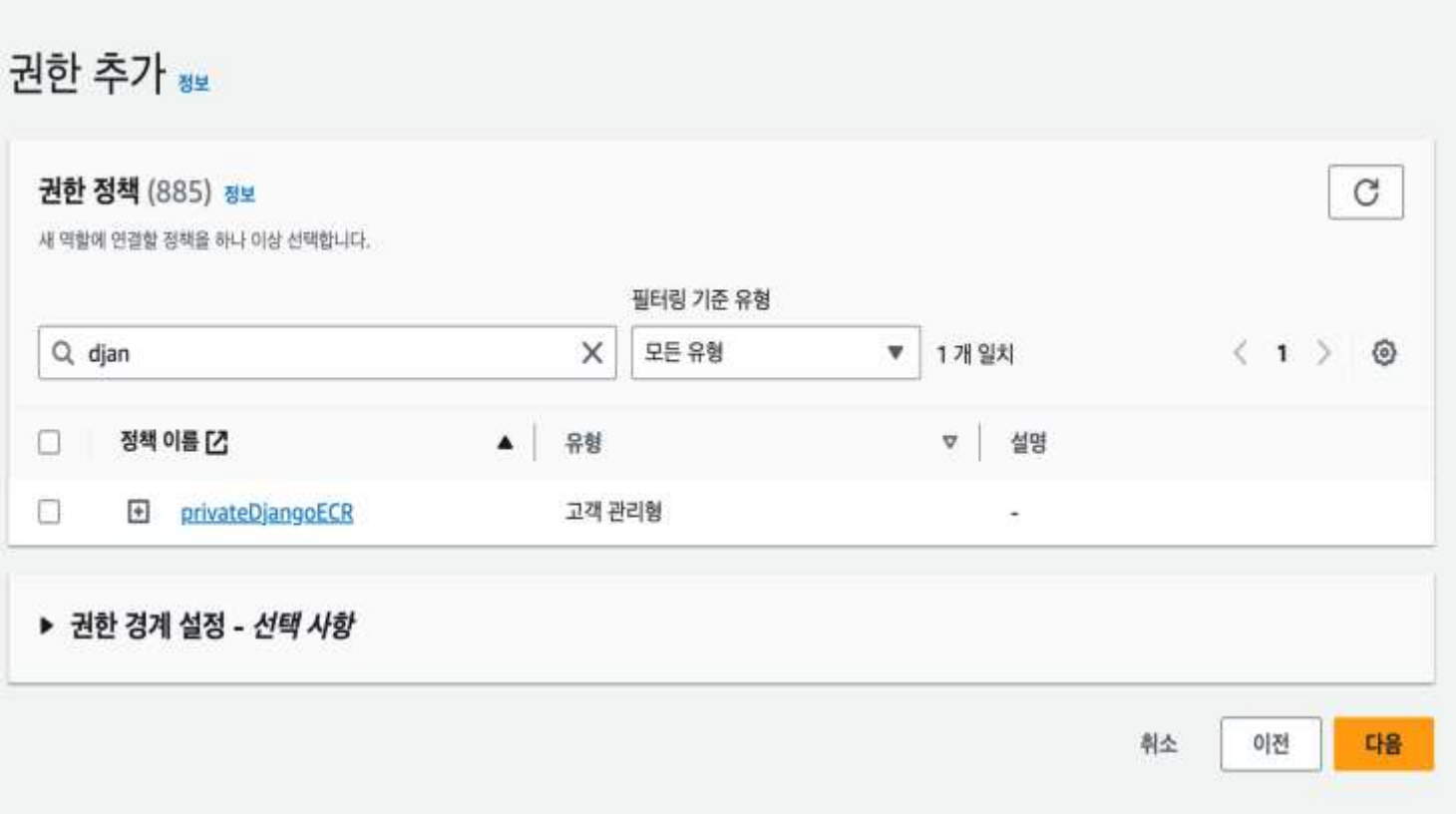
AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● 자격 증명 공급자에 새로운 역할 할당



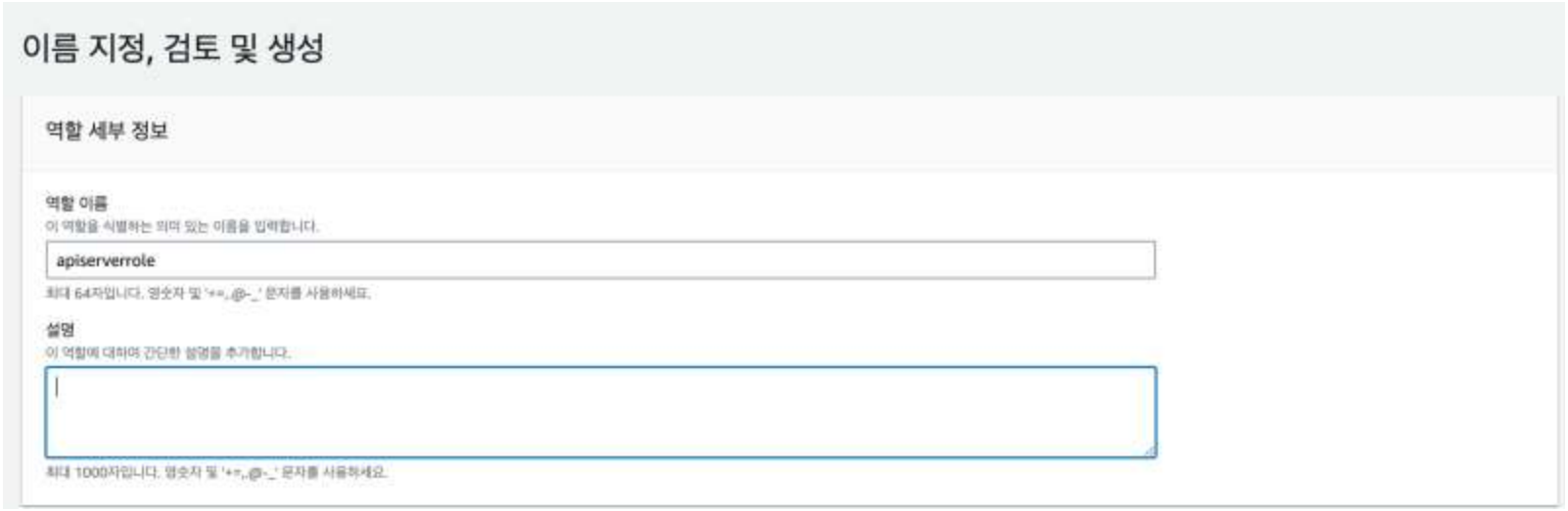
AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - Role에 이전에 만든 Policy를 할당



AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - Role에 이전에 만든 Policy를 할당



AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● Git Action.yml 파일을 수정해서 로그인이 되는지 확인: Role의 ARN 값으로 수정

```
name: Java CI with Gradle

# 동작 조건 설정 : main 브랜치에 push 혹은 pull request가 발생할 경우 동작한다.
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  id-token: write # This is required for requesting the JWT
  contents: read  # This is required for actions/checkout
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ OpenID Connect를 이용한 Git Hub AWS 로그인
    - ● Git Action.yml 파일을 수정해서 로그인이 되는지 확인

```
jobs:
  # Spring Boot 애플리케이션을 빌드하여 도커허 브에 푸시하는 과정
  build-docker-image:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    # 1. Java 17 세팅
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'

    # 2. Spring Boot 애플리케이션 빌드
    - name: Build with Gradle
      uses: gradle/gradle-build-
action@67421db6bd0bf253fb4bd25b31ebb98943c375e1
      with:
        arguments: clean bootJar
```

AWS Container Service

❖ Spring Boot Application CI/CD Pipeline 설정

✓ OpenID Connect를 이용한 Git Hub AWS 로그인

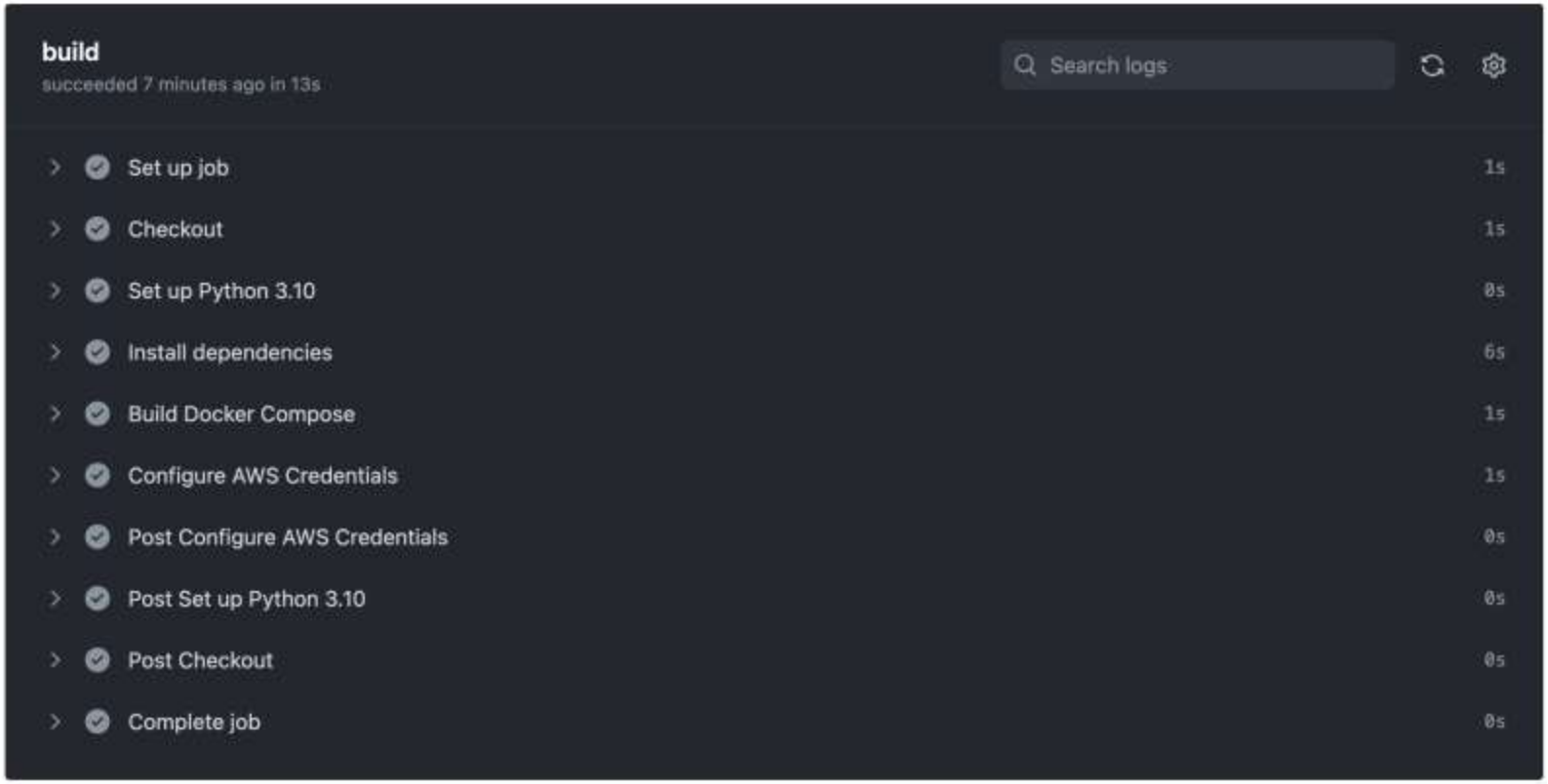
● Git Action.yml 파일을 수정해서 로그인이 되는지 확인

```
- name: Build Docker Compose
  run: docker-compose

- name: Configure AWS Credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::641022061021:role/itstudyoicd
    role-session-name: sampleSessionName
    aws-region: ap-northeast-2
```

AWS Container Service

❖ Spring Boot Application CI/CD Pipeline 설정

✓ OpenID Connect를 이용한 Git Hub AWS 로그인



AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - https://docs.github.com/en/enterprise-cloud@latest/actions/deployment/deploying-to-your-cloud-provider/deploying-to-amazon-elastic-container-service 참고
    - 사전 작업
      - ◆ ECR Repository 생성
      - ◆ ECS Cluster, Task, Service 생성
      - ◆ task-definition.json 파일 생성: ECS에서 생성한 task 의 JSON을 복사해서 프로젝트에 생성 – image 부분은 제거

AWS Container Service

❖ Spring Boot Application CI/CD Pipeline 설정

✓ ECR Push And Task Definition 생성 및 Service Update

● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
name: django

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  id-token: write
  contents: read

env:
  AWS_REGION: ap-northeast-2
  ECR_REPOSITORY: itstudydjangoecs
  ECS_SERVICE: itstudyservice2
  ECS_CLUSTER: deploytest
  ECS_TASK_DEFINITION: ./task-definition.json
  CONTAINER_NAME: itstudy1
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
jobs:
  build-docker-image:
    #ubuntu 환경에서 수행
    runs-on: ubuntu-latest
    steps:
    #소스 코드 가져오기
    - uses: actions/checkout@v3
    #JDK 설치
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
jobs:
  # Spring Boot 애플리케이션을 빌드하여 도커허 브에 푸시하는 과정
  build-docker-image:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    # 1. Java 17 세팅
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'

    # 2. Spring Boot 애플리케이션 빌드
    - name: Build with Gradle
      uses: gradle/gradle-build-
action@67421db6bd0bf253fb4bd25b31ebb98943c375e1
      with:
        arguments: clean bootJar
```

AWS Container Service

❖ Spring Boot Application CI/CD Pipeline 설정

✓ ECR Push And Task Definition 생성 및 Service Update

● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
- name: Configure AWS Credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::641022061021:role/imagepush
    role-session-name: sampleSessionName
    aws-region: ap-northeast-2

- name: Login to Amazon ECR
  id: login-ecr
  uses: aws-actions/amazon-ecr-
login@62f4f872db3836360b72999f4b87f1ff13310f3a
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

```
- name: Build, tag, and push image to Amazon ECR
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        IMAGE_TAG: ${{ github.sha }}
      run: |
        # Build a docker container and
        # push it to ECR so that it can
        # be deployed to ECS.
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
        echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >>
$GITHUB_OUTPUT
```

AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - ● Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정

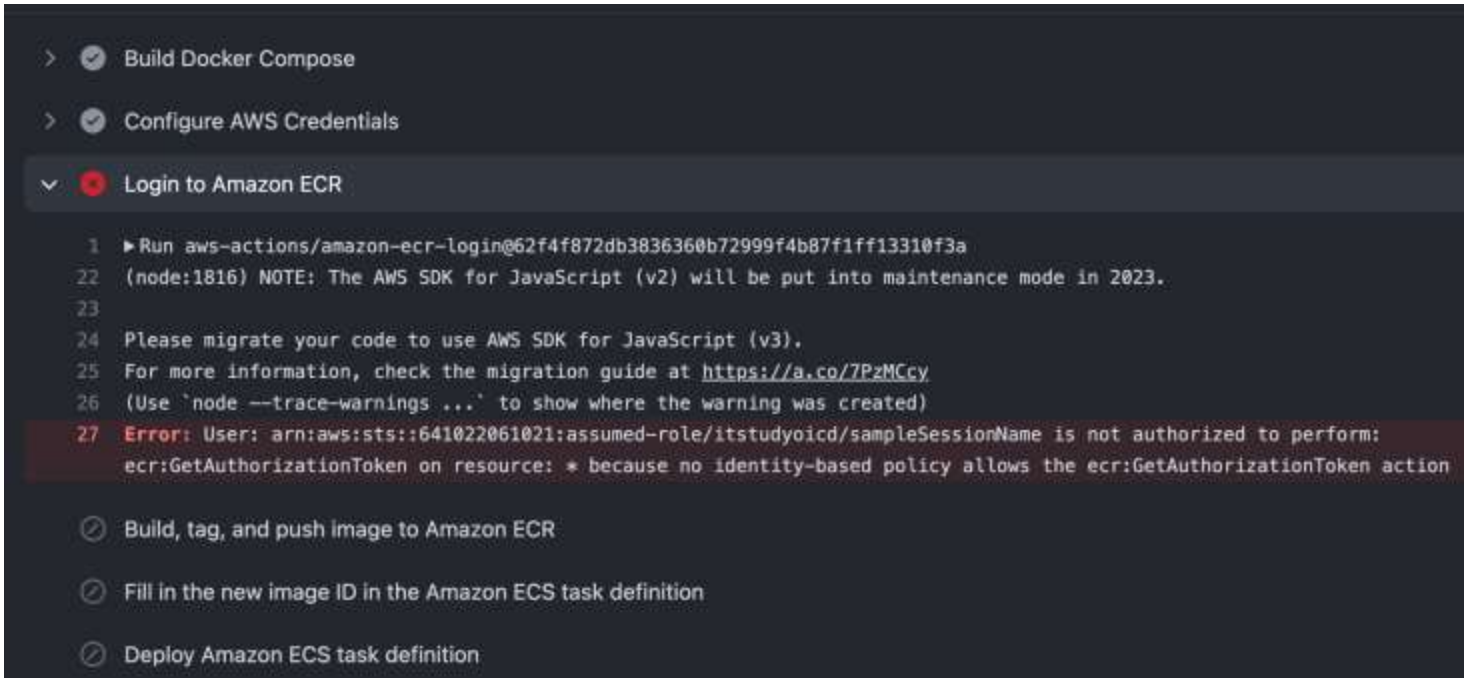
```
- name: Fill in the new image ID in the Amazon ECS task definition
      id: task-def
      uses: aws-actions/amazon-ecs-render-task-
definition@c804dfbdd57f713b6c079302a4c01db7017a36fc
      with:
        task-definition: ${{ env.ECS_TASK_DEFINITION }}
        container-name: ${{ env.CONTAINER_NAME }}
        image: ${{ steps.build-image.outputs.image }}

    - name: Deploy Amazon ECS task definition
      uses: aws-actions/amazon-ecs-deploy-task-definition@v2
      with:
        task-definition: ${{ steps.task-def.outputs.task-definition }}
        service: ${{ env.ECS_SERVICE }}
        cluster: ${{ env.ECS_CLUSTER }}
        wait-for-service-stability: true
```

AWS Container Service

❖ Spring Boot Application CI/CD Pipeline 설정

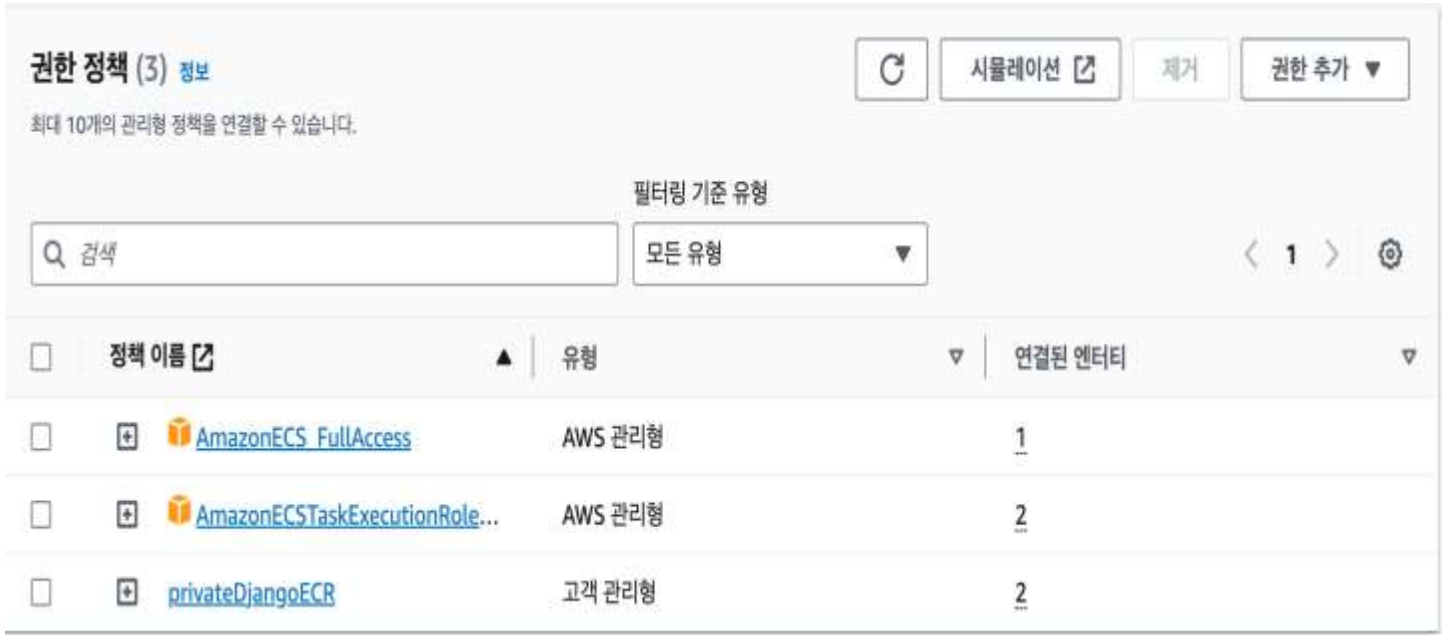
✓ ECR Push And Task Definition 생성 및 Service Update

- Git Action.yml 파일을 수정 – 환경 변수는 적절하게 수정



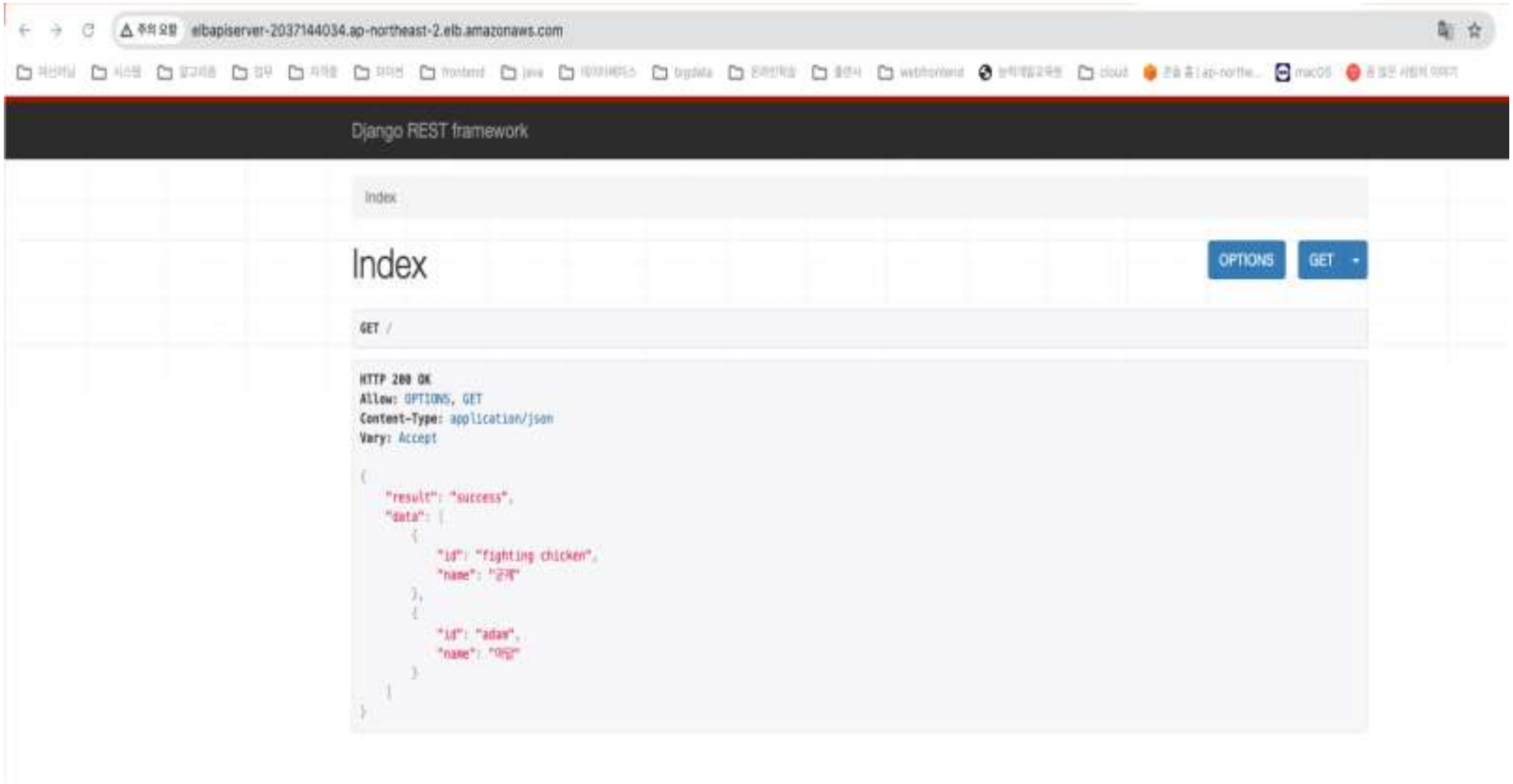
AWS Container Service

- ❖ Spring Boot Application CI/CD Pipeline 설정
  - ✓ ECR Push And Task Definition 생성 및 Service Update
    - 권한 부여 – IAM에서 이전에 만든 Role에 ECS 관련 권한 추가 후 다시 빌드



AWS Container Service

- Spring Boot Application CI/CD Pipeline 설정
  - ✓ 내용을 수정하고 푸시

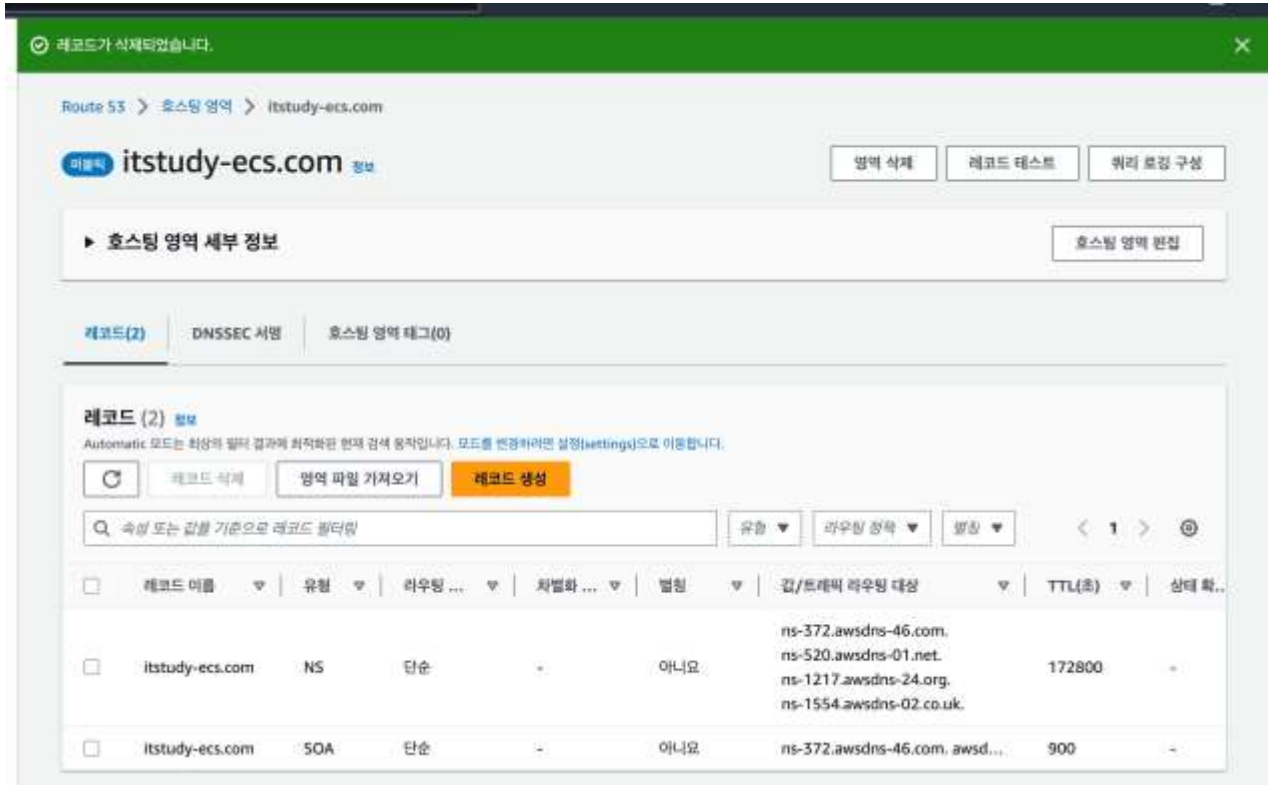


AWS Container Service

❖ 도메인 연결

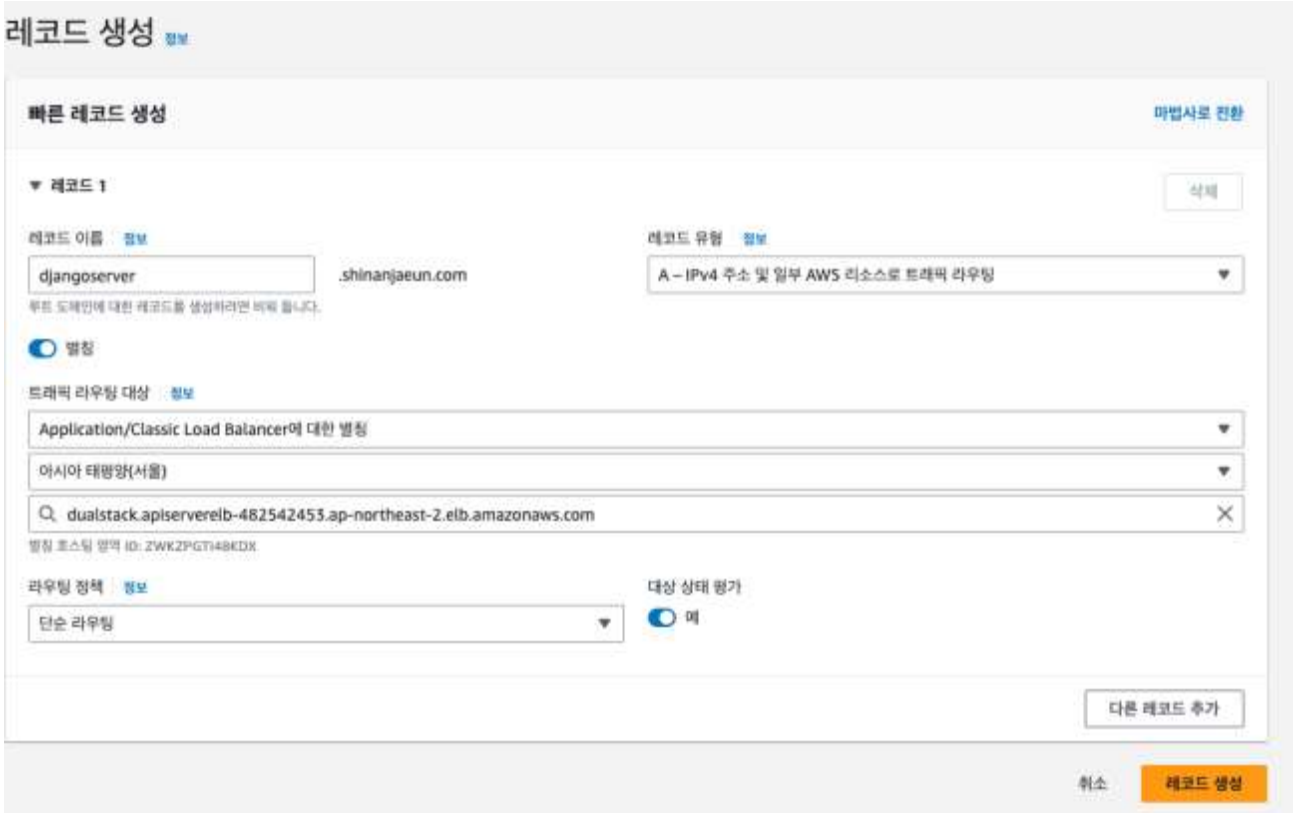
✓ Domain 연결

● 호스팅 영역에서 호스팅 영역 선택



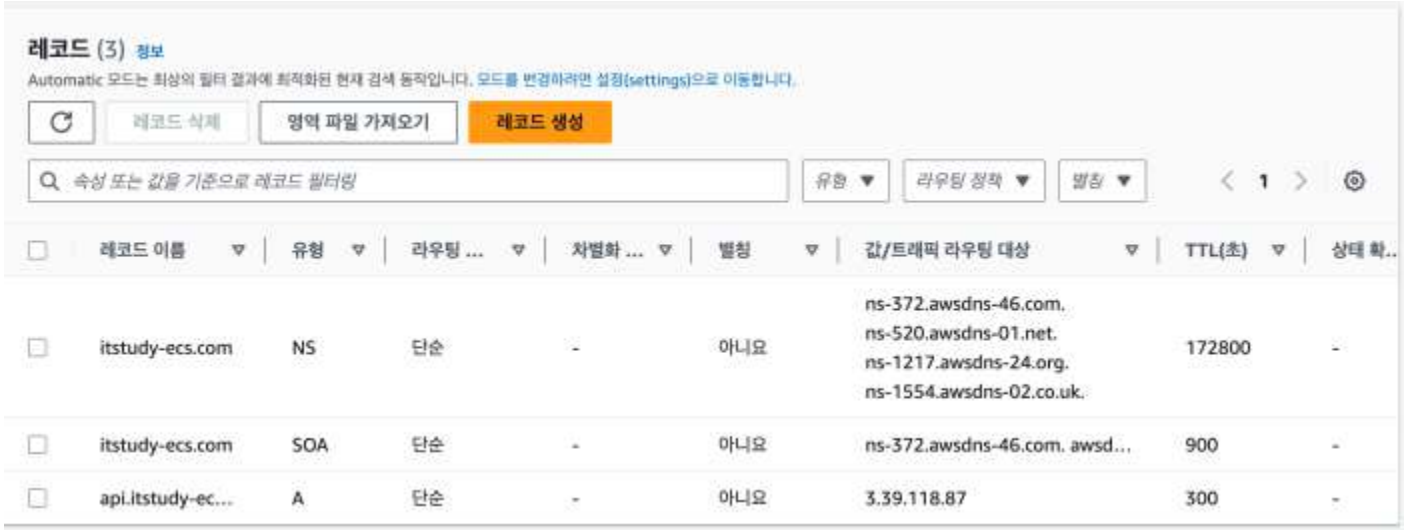
AWS Container Service

- ❖ 도메인 연결
  - ✓ Domain 연결
    - ● 레코드 생성 버튼을 누르고 정보 입력



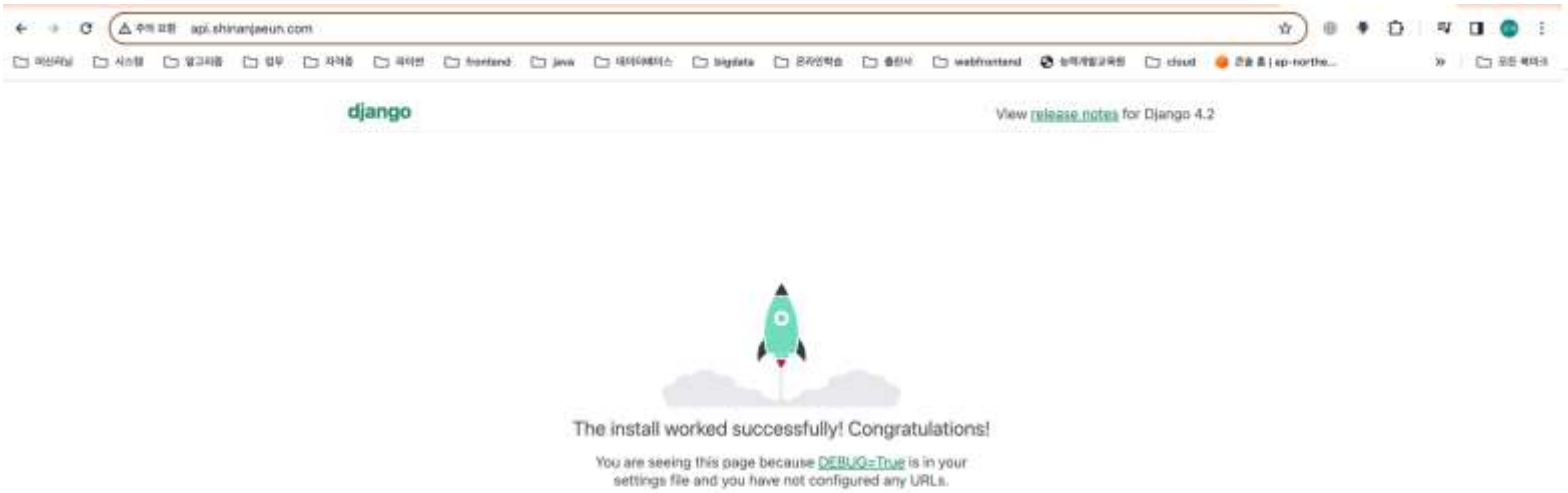
AWS Container Service

- ❖ 도메인 연결
  - ✓ Domain 연결
    - ● 레코드 생성 버튼을 누르고 정보 입력



AWS Container Service

- 도메인 연결
  - ✓ 브라우저에서 확인



AWS Container Service

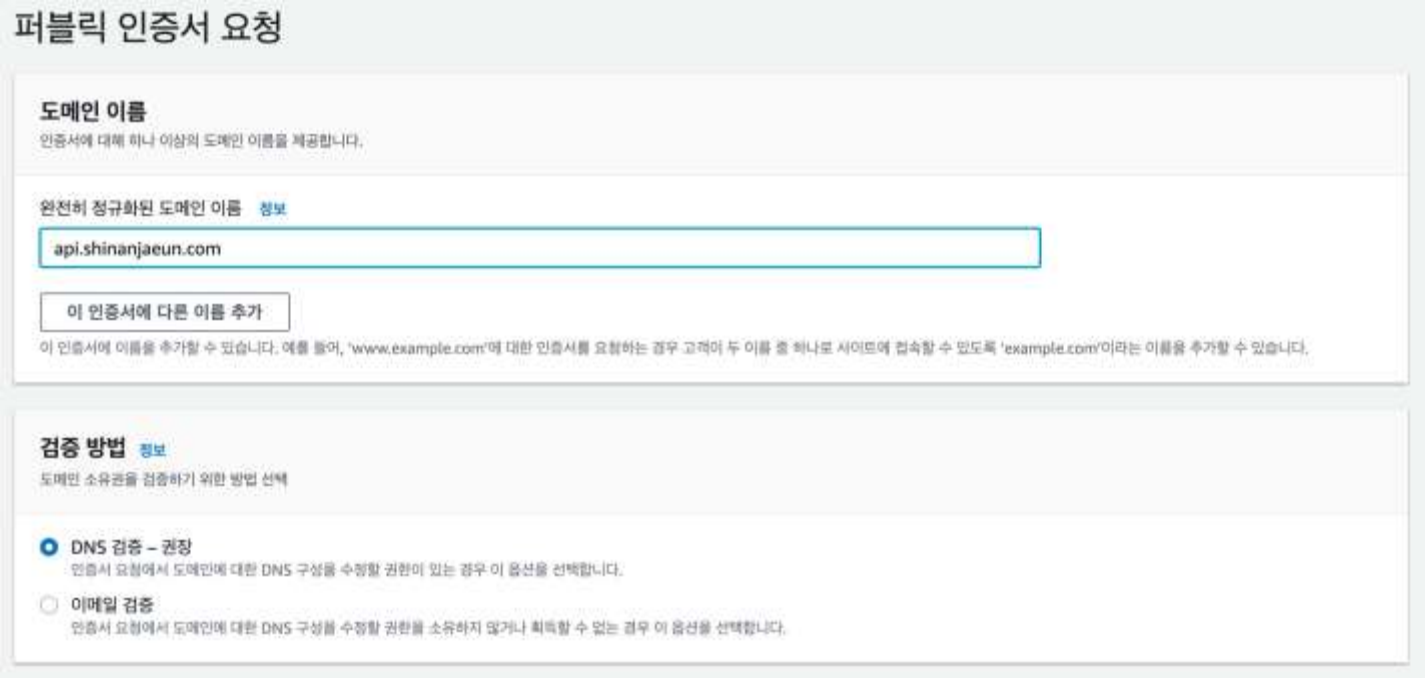
- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - AWS Certificate Manager 서비스에 접속

AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서 요청 클릭

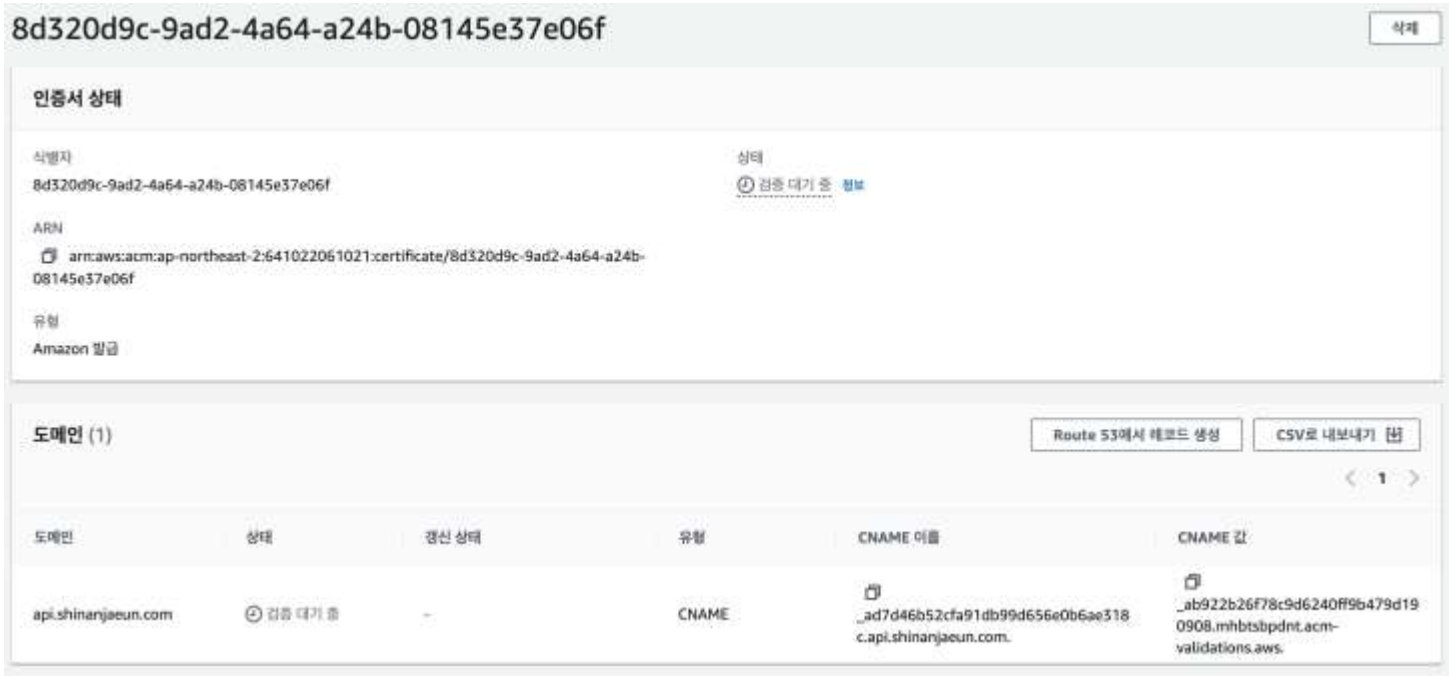
AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서 요청을 위한 도메인 설정



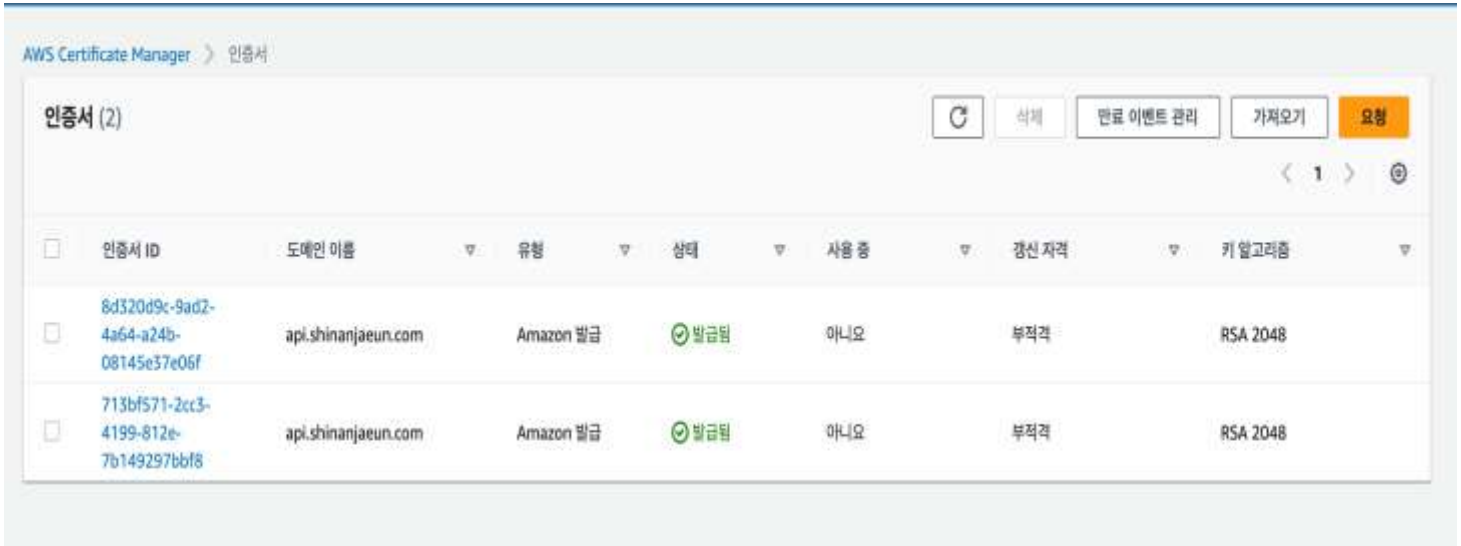
AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서를 클릭하고 Route53에서 레코드 생성 클릭



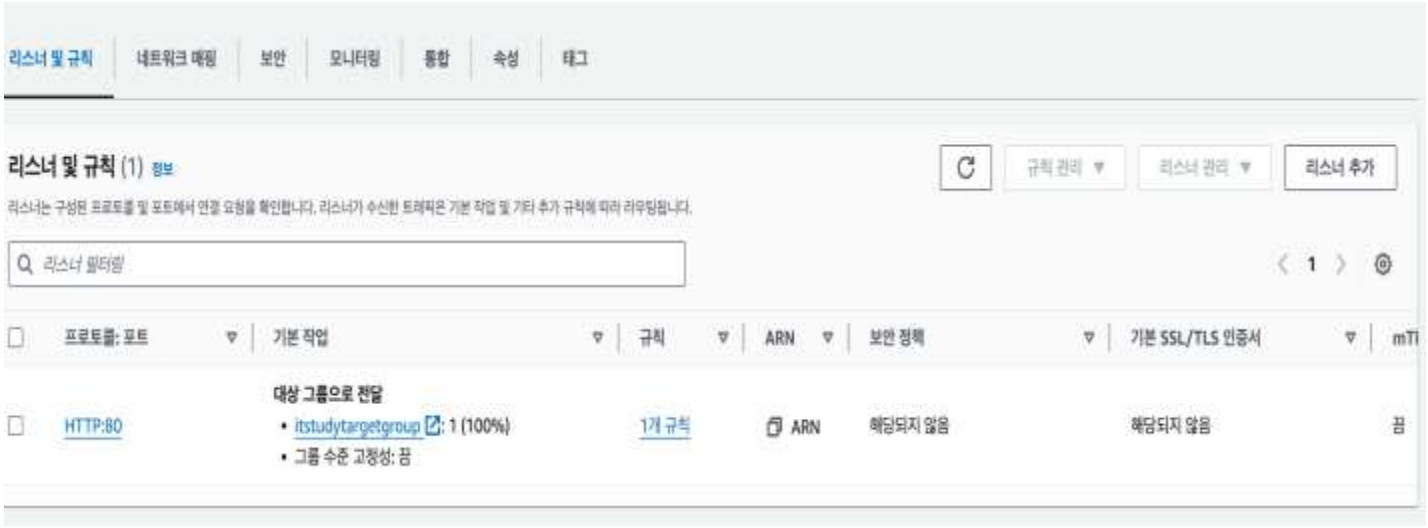
AWS Container Service

- ❖ HTTPS 적용
  - ✓ HTTPS 인증서 발급
    - ● 인증서를 클릭하고 Route53에서 레코드 생성 클릭



AWS Container Service

- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 로드 밸런서에 접속해서 [리스너 추가] 버튼 클릭

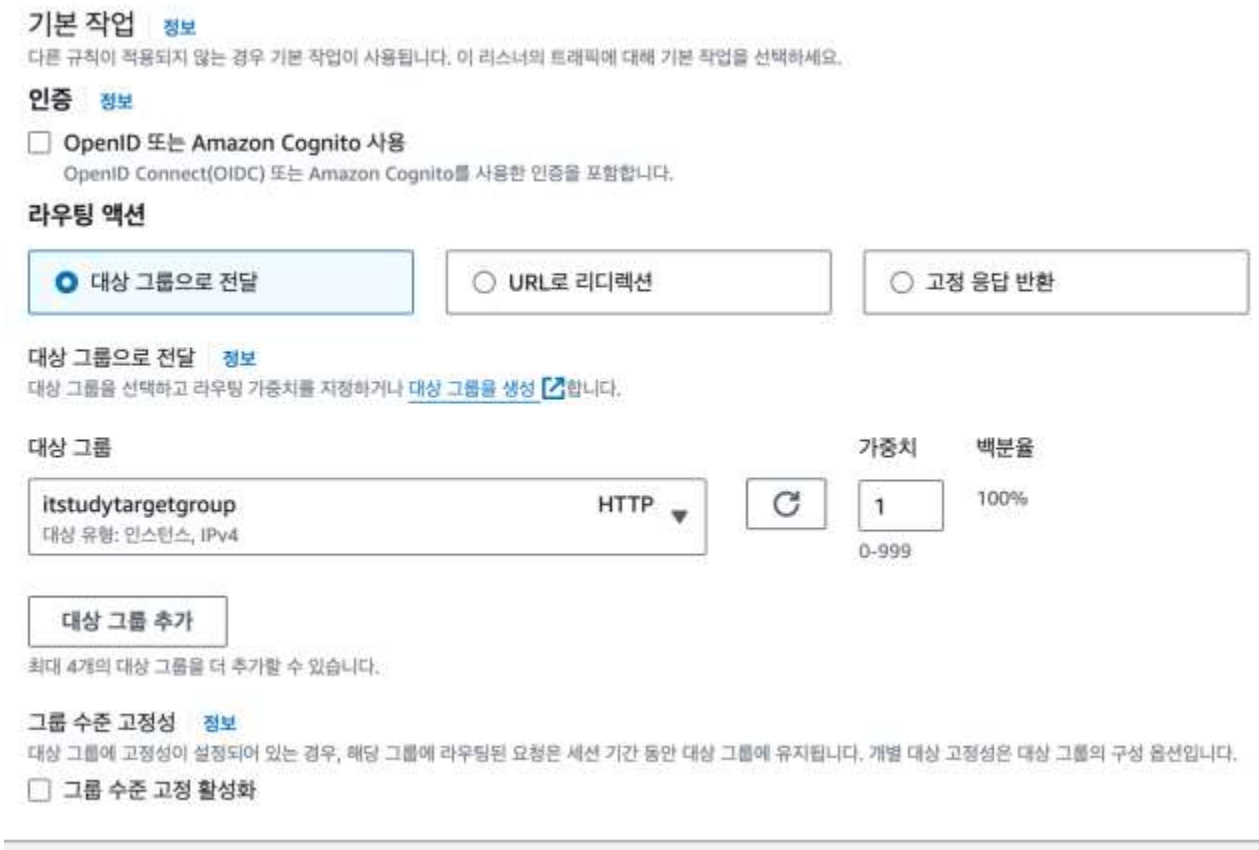


AWS Container Service

- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 로드 밸런서 설정 – 프로토콜을 HTTPS로 변경

AWS Container Service

- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 로드 밸런서 설정 – 대상 그룹 선택



AWS Container Service

- ❖ HTTPS 적용
  - ✓ ELB에 인증서 등록
    - ● 로드 밸런서 설정 – 인증서 선택