

---

# 프로젝트 보고서

고가용성 웹 어플리케이션 구축  
EKS 환경에 블루/그린 파이프라인 구축  
EKS 환경에 Multi Region 배포

---

최종 작성일: 2020-11-18

## &lt;목차&gt;

1. 프로젝트 개요	
1.1 프로젝트 명 및 기간.....	5 쪽
1.2 프로젝트 배경 및 목표.....	5 쪽
1.3 프로젝트 설명.....	5 쪽
1.4 산출물.....	6 쪽
2. 수행 일정	
2.1 추진 일정.....	6 쪽
3. 조직	
3.1 조직도.....	7 쪽
3.2 역할.....	7 쪽
4. 프로젝트 설계	
4.1 프로젝트 #1.....	8 쪽
4.1.1 아키텍처.....	8 쪽
4.1.2 AWS 리소스 생성.....	8 쪽
4.1.3 퍼블릭 서브넷 단계.....	10 쪽
4.1.4 프라이빗 서브넷 단계.....	10 쪽
4.1.5 웹 서버 개발.....	10 쪽
4.1.6 연동.....	11 쪽

4.1.6.1 웹 페이지와 RDS 연동.....	11 쪽
4.1.6.2 Github 공공데이터 Import 및 업데이트 자동화 설계.....	12 쪽
4.1.7 결과.....	13 쪽
4.2 프로젝트 #2.....	14 쪽
4.2.1 아키텍처.....	14 쪽
4.2.2 AWS 리소스 생성.....	14 쪽
4.2.2.1 CDK(Cloud Deployment Kit)를 활용한 리소스 및 서비스 생성.....	15 쪽
4.2.2.2 EKS (Elastic Kubernetes Service) 개념.....	15 쪽
4.2.2.3 AWS CodePipeline 개념.....	16 쪽
4.2.3 결과.....	17 쪽
4.3 프로젝트 #3.....	
19 쪽	
4.3.1 아키텍처.....	19 쪽
4.3.2 AWS 리소스 생성.....	20 쪽
4.3.2.1 CDK 와 EKS 를 사용하기 위한 환경 구성 .....	20 쪽
4.3.2.2 EKS 를 사용한 다중 배포 환경 구성 .....	21 쪽
4.3.3 CI/CD .....	24 쪽

4.3.4 결과 .....	30 쪽
Reference .....	30 쪽

## 1. 프로젝트 개요

### 1.1 프로젝트 명 및 기간

1.1.1 프로젝트 명 : 고가용성 웹 서버 구축

기간 : 2020-09-21 ~ 2020-10-20

1.1.2 프로젝트 명 : EKS 환경에 블루/그린 파이프라인 구축

기간 : 2020-10-15 ~ 2020-11-17

### 1.2 프로젝트 배경 및 목표

저희 프로젝트 목표는 현업에서 일하는 사람들이 자주 사용하는 AWS 클라우드 서비스를 이용하여 구현하는 것에 초점을 맞추었습니다. 그래서 첫 번째 프로젝트는 클라우드 환경에 가장 기본적인 3 티어 계층 구성(어플리케이션-웹 서버-데이터베이스)을 함으로써 클라우드 환경에 웹 서비스를 올리는 것을 하였습니다.

두 번째 프로젝트는 EKS 를 활용한 CDK 로 CI/CD 파이프라인 구축 및 블루/그린 방식의 배포를 하였습니다. CDK 를 통해 EKS 클러스터에 워커노드에 컨테이너화된 Flask 어플리케이션을 Pods 형태로 무중단 블루그린 배포가 잘 진행되는 것을 확인할수 있었습니다. CDK 로 CI/CD 파이프라인을 구축해보았고 소스를 변경시 CodeCommit 으로 버전관리를 하고, 블루 그린 배포 방식을 선택하여 테스트를 진행하였습니다. 파이프라인을 통해 DevOps 조직의 프로세스에 대한 전반적인 이해를 할 수 있었습니다.

### 1.3 프로젝트 설명

구현하고자 하는 웹 서비스는 코로나 현황을 볼 수 있는 웹 사이트입니다. 사용자는 웹 사이트에 접속하여 코로나 일일 확진자, 지역별 확진자 등의 내용을 확인할 수 있습니다.

AWS 쿠버네티스를 생성하여 워커노드와 통신을 확인합니다. 생성된 워커노드에 서비스를 배포함으로써 마스터와 워커노드 간의 프로세스를 확인합니다. 그리고 파이프라인을 구축하여 AWSCodeCommit, CodeBuild, CodeDeploy 를 생성 및 선택합니다. 소스에 변경 시 관리자가 업그레이드된 버전을 운영서버에 Deploy 할지 승인 여부를 확인합니다. 승인이 되면 변경된 소스로 업데이트되고 운영 시 문제가 발생되면 이전 버전으로 쉽게 롤백이 가능합니다. ALB 를 통한 배포 방식을 사용함으로써 사용자 접속 트래픽을 조절할 수 있습니다. 또한 접속 트래픽을 맞게

스케일 아웃되어 가용성을 높이게 설계되었습니다.

## 1.4 산출물

➤ 프로젝트 계획서

## 2. 수행 일정

### 2.1 추진 일정

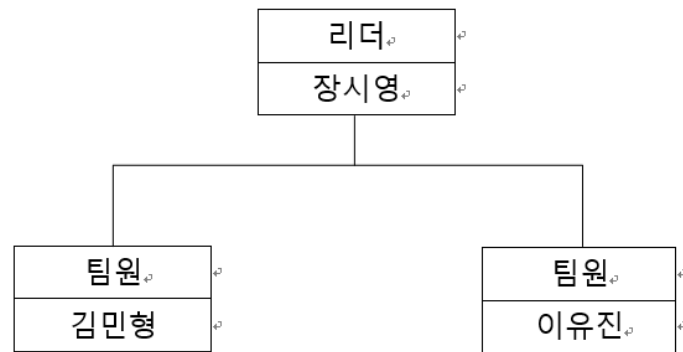
고가용성 웹 어플리케이션 구축, EKS 환경에 블루/그린 파이프라인 구축을 각각 한 달 정도의 시간이 소요되었습니다. 모두 처음 AWS Cloud 교육을 들으면서 진행한 프로젝트로 주제 선정에 어려움을 겪었습니다. 그리고 교육 외의 서비스를 구현하기 위해 많은 레퍼런스를 참조하였습니다.

내용 / 기간	소내용	9월		10월					11월		역할
		4W	5W	1W	2W	3W	4W	5W	1W	2W	
주제 선정 #1	코로나 웹 페이지 구현										팀원 전부
자료 조사	코로나 공공데이터 활용										팀원 전부
개발	로컬에서 테스트 환경 구축										이유진
	웹 페이지 시뮬레이션										이유진
AWS 환경 구성	Cloud Formation으로 리소스 생성										김민형
	EC2, RDS 생성 및 연동										김민형, 장시영
	ALB 및 Auto Scaling 적용										이유진
운영서버 적용	로컬 테스트 소스 EC2에 업로드										이유진
	RDS 연동 테스트 및 데이터 I/O확인										이유진
	웹 페이지 확인										이유진
주제 선정 #2	CDK를 통해 EKS 구축을 통한 어플리케이션 배포와 Pipeline 설계										팀원 전부
자료 조사	EKS 주제 선정 후 구체적인 구축 가이드 설계										팀원 전부
AWS 환경 구성	CDK를 이용한 리소스 생성										김민형,이유진
	리소스 : EC2(Master), EKS(WorkerNode), CodePipeline										팀원 전부
	블루/그린 배포 테스트										팀원 전부
문서 작성	발표 PPT 및 문서 작업										팀원 전부

[프로젝트 추진 일정]

### 3. 조직

#### 3.1 조직도



[팀 조직도]

#### 3.2 역할

담당자명	역할
장시영	프로젝트 총괄 책임자 프로젝트 일정 관리 및 마무리 프로젝트 기획
김민형	프로젝트 설계 프로젝트 시스템 비용 예측
이유진	프로젝트 설계 Logic 개발

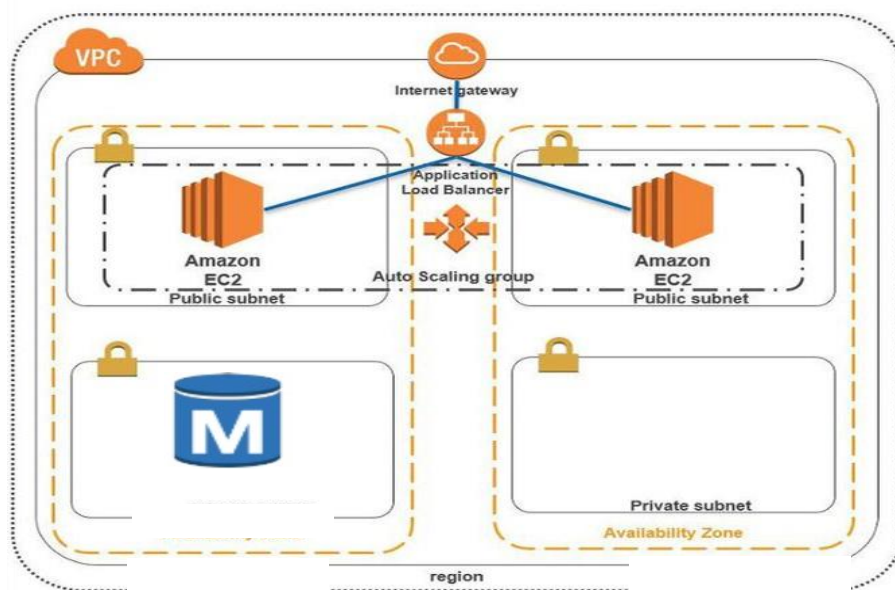
[팀 역할 내용]

## 4. 프로젝트 설계

### 4.1 프로젝트 #1 : 고가용성 웹 어플리케이션 구축

#### 4.1.1 아키텍처

도쿄 리전에 가용영역 2 개를 설계하고 각 가용영역마다 외부망과 연결되는 퍼블릭 서브넷 1 개와 프라이빗 서브넷 1 개를 생성하였습니다. 외부에서 웹 서비스로 접속할 때는 ALB 를 통해 트래픽을 분산하도록 유도하였고, 유동적인 트래픽 양에 따라 EC2 서버 수가 자동적으로 늘어나거나 줄어 들 수 있도록 하였습니다. 그리고 프라이빗 서브넷에는 RDS 를 구성하여 외부에서 접속하지 못하도록 보안 설정하였습니다.



[아키텍처 설계]

#### 4.1.2 AWS 리소스 생성

리소스를 생성하기 위해 CloudFormation 와 웹 콘솔을 활용하였습니다. 위의 구성도와 같이 필요한 리소스는 VPC, Public Subnet, Private Subnet, EC2, RDS, ALB, Auto Scaling 입니다. VPC, 서브넷과 EC2 는 CloudFormation 으로 생성을



하고, RDS, ALB, Auto Scaling 은 웹 콘솔로 생성하였습니다.

Key	Value	Description	Export name
InstanceId	i-0c1fcd6fd48b10aff	The EC2 instance id.	al2-mutable-public-InstanceId
PrivateIPAddress	10.0.1.93	The private IP address of the EC2 instance.	al2-mutable-public-PrivateIPAddress
PublicIPAddress	54.238.78.242	The public IP address of the EC2 instance.	al2-mutable-public-IPAddress
StackName	al2-mutable-public	Stack name.	-
TemplateID	ec2/al2-mutable-public	cloudonaut.io template id.	-
TemplateVersion	12.12.0	cloudonaut.io template version.	-

[CloudFormation 스택으로 생성된 리소스 결과]

Name	Subnet ID	State	VPC
<input checked="" type="checkbox"/> A private	subnet-0904b4f09e333096a	Available	vpc-09e909ed923ca4db2   10.0.0.0/16
<input type="checkbox"/> A public	subnet-0f5765924adc28552	Available	vpc-09e909ed923ca4db2   10.0.0.0/16
<input type="checkbox"/> B private	subnet-0c7bbe2581cb91908	Available	vpc-09e909ed923ca4db2   10.0.0.0/16
<input type="checkbox"/> B public	subnet-053577cb15c2cc5c7	Available	vpc-09e909ed923ca4db2   10.0.0.0/16

[CloudFormation 으로 실제 생성된 화면]

#### 4.1.3 퍼블릭 서브넷 단계

퍼블릭 서브넷은 인터넷 게이트웨이를 통하여 퍼블릭 서브넷과 외부와의 통신을 가능하게 합니다. EC2 웹 서버를 퍼블릭 서브넷에 위치하여 사용자의 웹 사이트 접속을 가능하게 하였습니다. 사용자들에게 코로나 최신 정보를 제공하는 웹 사이트로서 별도 회원 등록 및 관리를 필요하지 않는 서비스로 보안 설정은 별도

없습니다.

그리고 ALB 을 추가하여 사용자 접속 트래픽을 분산하여 웹 사이트로 이동하게 합니다. 또한 Auto Scaling 을 통해 사용자 접속이 많을 경우 웹 서버를 스케일 아웃하고 다시 사용자 접속이 줄어들면 Scale In 합니다. 사용자 접속 트래픽에 따른 자동화 구성으로 사용자들에게 고 가용성 서비스를 제공합니다.



[Auto Scaling 생성하는 화면]

#### 4.1.4 프라이빗 서브넷 단계

프라이빗 서브넷은 외부와의 접속은 차단하고 EC2 서버를 통해서만 접속할 수 있도록 설정하였습니다. RDS DB 서버를 프라이빗 서브넷에 구축하여 DB 서버의 데이터의 안정성을 고려하였습니다. RDS 는 MySQL 을 설치하고 웹 페이지에 데이터 정보를 보여줍니다.

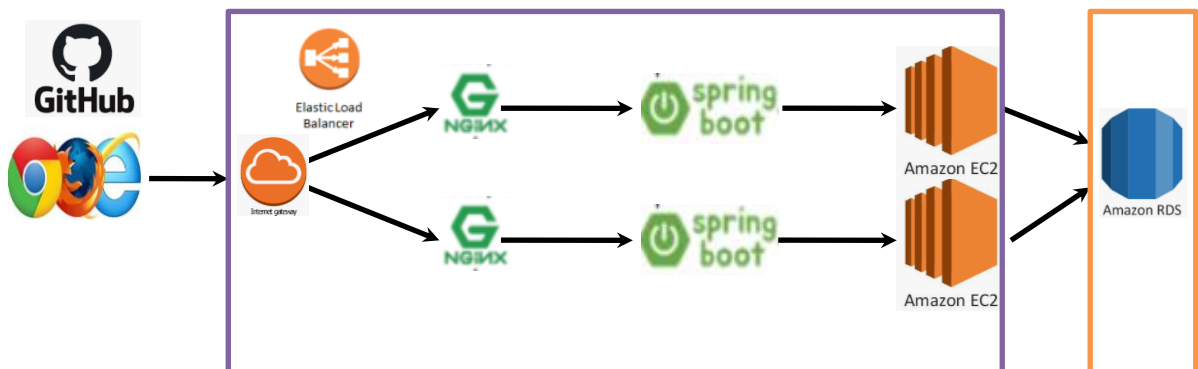
#### 4.1.5 웹 서버 개발

코로나 최신 정보 웹 사이트를 Java 언어를 사용하여 개발하였습니다.

클라이언트는 서버에 웹 페이지 호출을 요청할 때 Nginx 는 응용프로그램 EC2 서버에 요청을 보내는 전달자 역할 또는 프록시 서버 역할을 합니다. 응용 프로그램 서버는 반대로 리버스 서버이고 데이터를 보내는 역할을 합니다.

EC2 에 Nginx 패키지를 설치하고 Nginx 설정 파일을 아래와 같이 수정합니다.

ELB DNS 주소:80 으로 들어온 요청은 Nginx 를 통해 스프링부트:8080 으로 리다이렉션되어 호출됩니다.



[웹 서버 배포 구성]

```

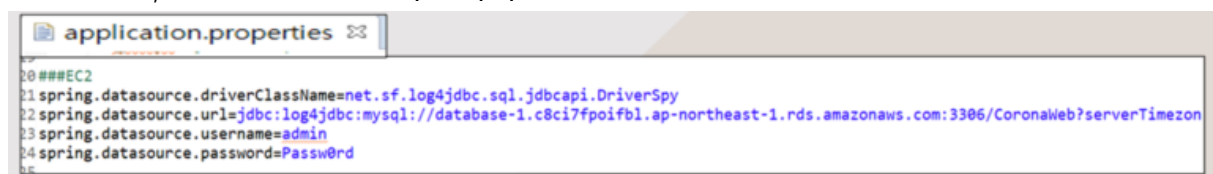
proxy_pass http://localhost:8080;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header Host $http_host;
  
```

[Nginx 설정 파일 (nginx.conf)]

## 4.1.6 연동

### 4.1.6.1 웹 페이지와 RDS 연동

스프링부트 애플리케이션에서 Java Config 를 통해 DataSource 설정을 합니다. DataSource 설정은 application.properties 파일에서 spring.datasource.\* 문법으로 RDS 의 연동 설정이 가능합니다. 그리고 RDS 연결을 위해 RDS 엔드포인트, RDS Username, Password 를 입력합니다.



[RDS 연결을 위한 설정 파일 (application.properties)]

### 4.1.6.2 Github 공공데이터 Import 및 업데이트 자동화 설계

코로나 확진자 자료는 현재 예민한 이슈로 사용자들에게 오픈 데이터로 Github 에서 공유되고 있습니다. 해당 데이터는 오전 11 시에 매일 업데이트되기 때문에 Github 과 동일하게 저희 RDS 에도 데이터를 매일 업데이트하려고 합니다. 사람이 매일 같은 작업을 하는 것은 번거롭기 때문에 서버가 Github 에 접속하여 EC2 로 데이터를 .csv 파일로 가져오게끔 Python 스크립트를 작성하였습니다. 그리고 가져온 .csv 파일을 RDS 데이터베이스에 자동 Import 할 수 있도록 하였습니다.

```

WebServer#1
# !bin/bash
sudo su <<EOF
cd /home/ec2-user/pythoncode/githubdata && echo "success" || echo "failed"
rm -rf coronaboard_kr && echo "success" || echo "failed"
git clone https://github.com/jooeungen/coronaboard_kr.git && echo "success" || echo "failed"
cd /home/ec2-user/pythoncode/ && echo "success" || echo "failed"
python3 test.py && echo "success" || echo "failed"
EOF
  
```

[공공데이터 및 데이터 Import 파이썬 스크립트]

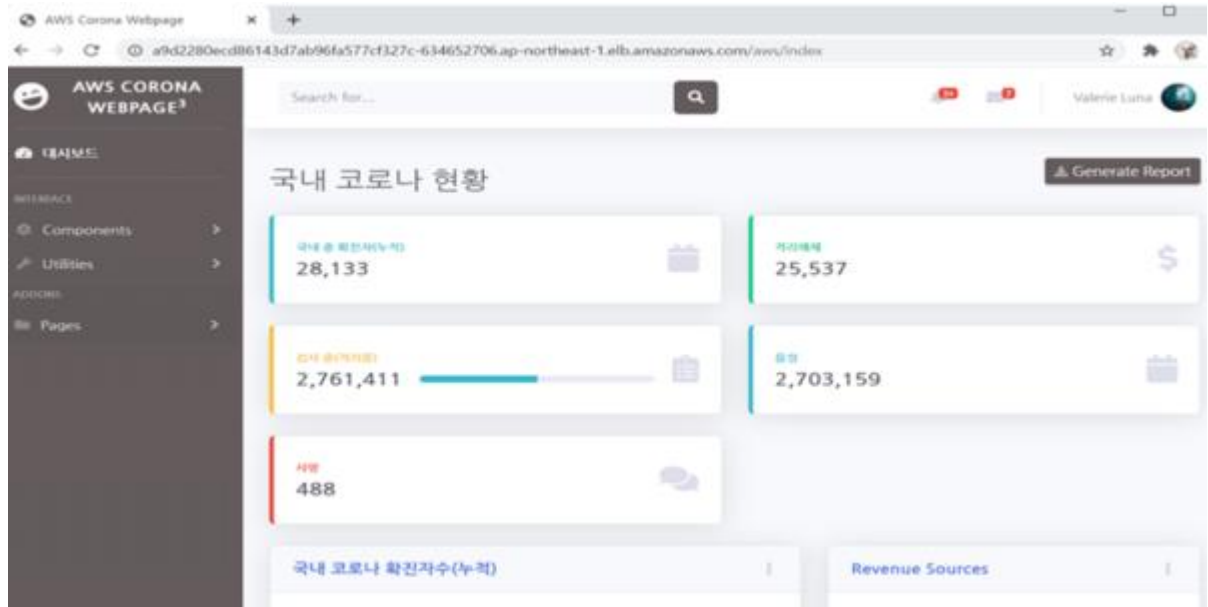
```

[ec2-user@ip-10-0-1-93 coronaboard_kr]$ crontab -l
03 11 * * * sh /home/ec2-user/pythoncode/autodata.sh
  
```

[공공데이터 및 데이터 Import 크론템 등록]

#### 4.1.7 결과

사용자들이 코로나 웹 사이트 접속 시 코로나에 대한 최신 정보를 확인할 수 있습니다. 결과적으로 사용자 측면에서는 해당 코로나 웹 페이지에 대한 신뢰성을 얻을 수 있습니다. 왜냐하면 요청하는 사용자 접속 트래픽에 맞게 자동적으로 서버가 확장되므로 지연 또는 에러 페이지를 호출하지 않습니다. 그리고 고객 측면에서는 실제 온프레미스 환경에 설계하였을 때보다 비용면에서 효과적입니다. 또한 AWS 서비스는 자동 관리되기 때문에 장애에 대한 염려를 하지 않아도 된다는 장점이 있습니다. 다음 프로젝트에서는 웹 어플리케이션을 DevOps 환경에서 쿠버네티스 클러스터에 구축하고 무중단 배포하는 내용에 다뤄 해당 프로젝트와 비교해보겠습니다.

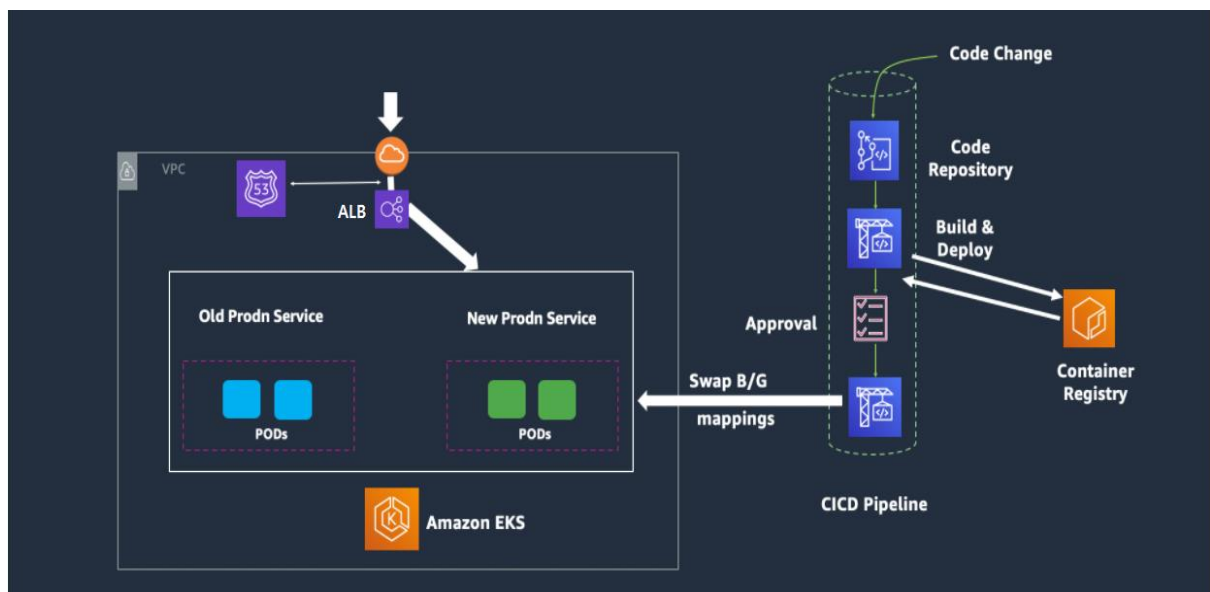


[코로나 웹 페이지 호출 화면]

## 4.2 프로젝트 #2 : EKS 환경에 블루/그린 파이프라인 구축

### 4.2.1 아키텍처

EKS(Elastic Kubernetes Service) 환경에서 CI/CD 파이프라인을 구축하였습니다. 쿠버네티스는 컨테이너 오케스트레이션 툴로 컨테이너화된 애플리케이션을 자동화해 배포하며 스케일링을 통하여 효과적으로 관리해주는 방식의 오픈소스 플랫폼입니다. 쿠버네티스 마스터와 노드 간의 파이프라인 구축을 통해 노드를 자동으로 감지하고 교체하며, 자동화된 버전 업그레이드 및 패치 기능들을 제공하고, 무중단 서비스를 통한 배포가 가능합니다. 클러스터와 워커노드의 통신을 위해 네트워크 로드밸런서를 사용하게 설계되었습니다. 그래서 사용자의 트래픽을 분산 또는 한 쪽으로만 접속하도록 유도할 수 있습니다.



[아키텍처 설계]

### 4.2.2 AWS 리소스 생성

#### 4.2.2.1 CDK(Cloud Deployment Kit)를 활용한 리소스 및 서비스 생성

웹 콘솔 또는 Cli 를 통해 EKS 를 구축할 수 있지만 저희는 CDK 를 사용하여 모든 리소스를 구성하였습니다. CDK 애플리케이션이 실행되면 CloudFormation YAML 템플릿으로 컴파일된 다음 프로비저닝을 위해 CloudFormation 서비스에 제출합니다. 삭제 시에도 CloudFormation 에서 생성된 스택을 삭제하면 되기 때문에 간편합니다.



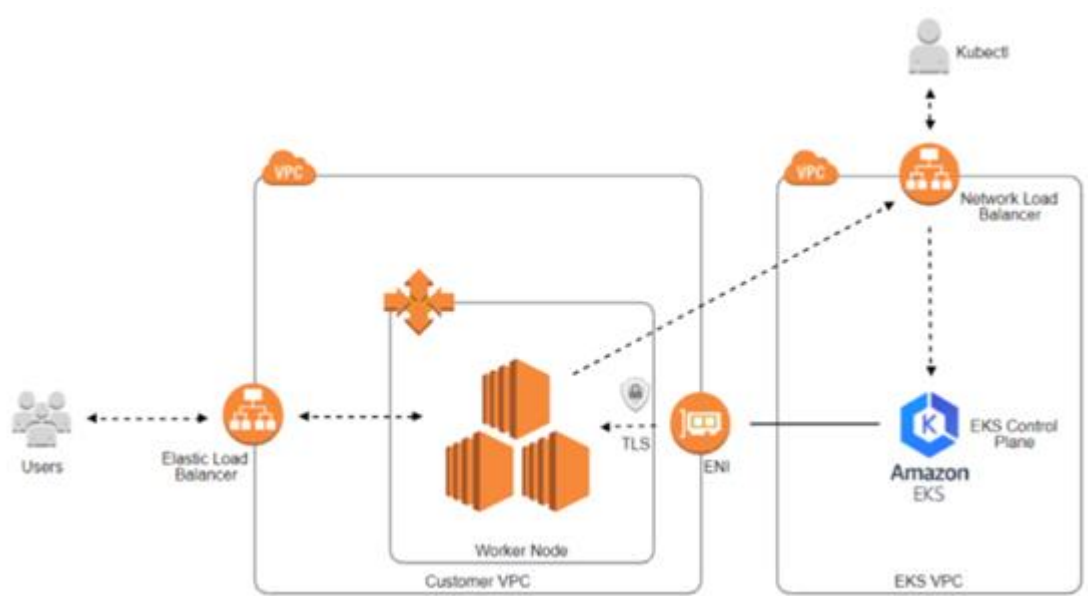
[CDK 과정]

```
cd cdk
cdk init
npm install
npm run build
cdk ls
cdk synth
cdk bootstrap aws://$ACCOUNT_ID/$AWS_REGION
cdk deploy
```

[cdk 명령어 예시]

#### 4.2.2.2 EKS (Elastic Kubernetes Service) 개념

아래 그림처럼 쿠버네티스에 워커노드와 ControlPlane 노드 간의 소통하는 아키텍처입니다. 좀 더 설명을 하자면, EKS의 ControlPlane은 ENI를 관리합니다. 클러스터가 워커노드 또는 워커노드 위에서 실행되어지는 앱과 통신을 하기 위해서는 ENI가 필요합니다.

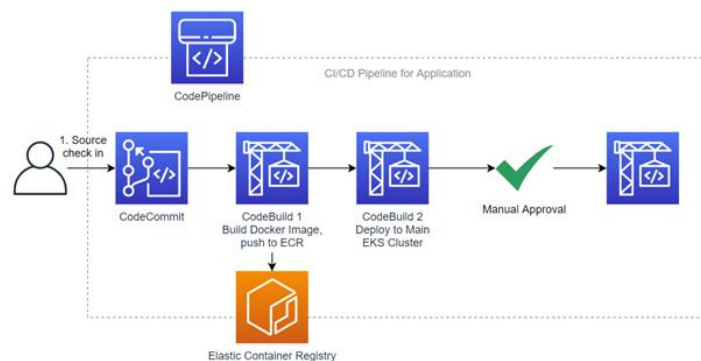


[EKS 구성요소]

#### 4.2.2.3 AWS CodePipeline 개념

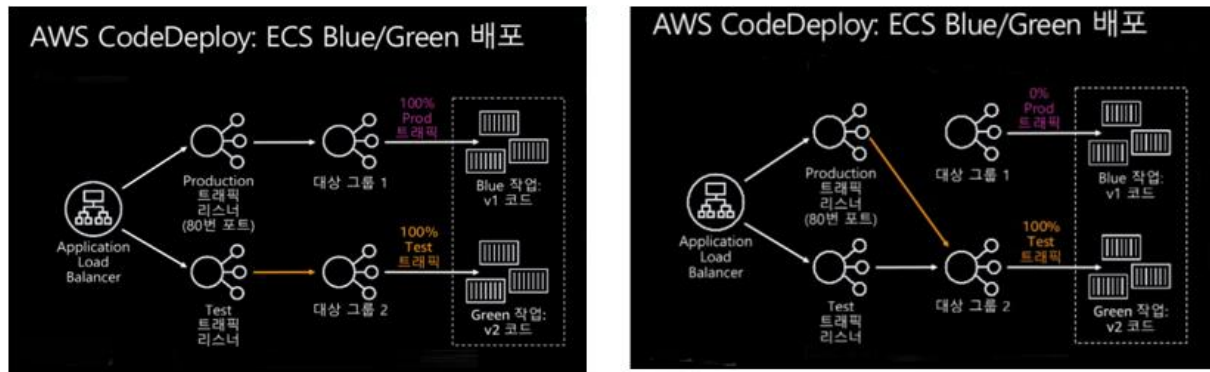
지속적인 통합, 지속적인 서비스 제공, 지속적인 배포를 해주는 CI/CD CodePipeline 의 구조는 아래 그림과 같고, 동작하는 순서 및 기능에 대해서 나열하였습니다.

- 1) AWS CodeCommit 에서 커밋된 Flask 앱 코드를 체크한 후에 빌드를 실행합니다.
- 2) AWS CodeBuild 는 새로운 빌드를 실행 및 앱을 도커화하기 위해 빌드 후 AWS ECR 에 푸시합니다
- 3) AWS CodeBuild 에서 도커이미지가 ECR 로 푸시되어지면, CodeBuild 는 EKS 클러스터에 블루/그린 서비스를 생성해주게됩니다
- 4) 그린 서비스에서 배포될 앱이 테스트 절차를 밟는 동안 CodePipeline 에 Swap B/G 승인이 멈추게되고, 앱이 잘 작동이되는게 확인되면 승인을 해줍니다.
- 5) 승인으로 인해 CodePipeline 이 트리거가되면서 CodeBuild 에서 "Swap and Deploy" 단계에서 그린 환경쪽에 Pod 형태로 CodeDeploy 를 통한 배포가 이루어지게 됩니다.
- 6) 마지막으로 인그레스 컨트롤러 형태의 어플리케이션 로드 밸런서의 타겟 그룹의 트래픽을 업데이트된 버전인 블루 환경의 서비스로 전환이 되어지는걸 확인할 수 있습니다.



[파이프라인 구성요소]

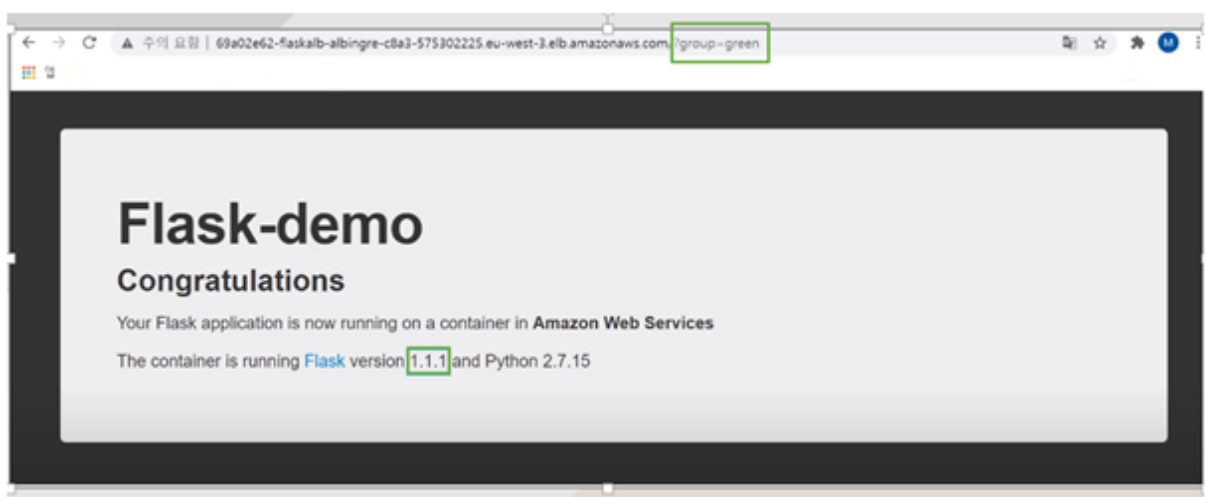




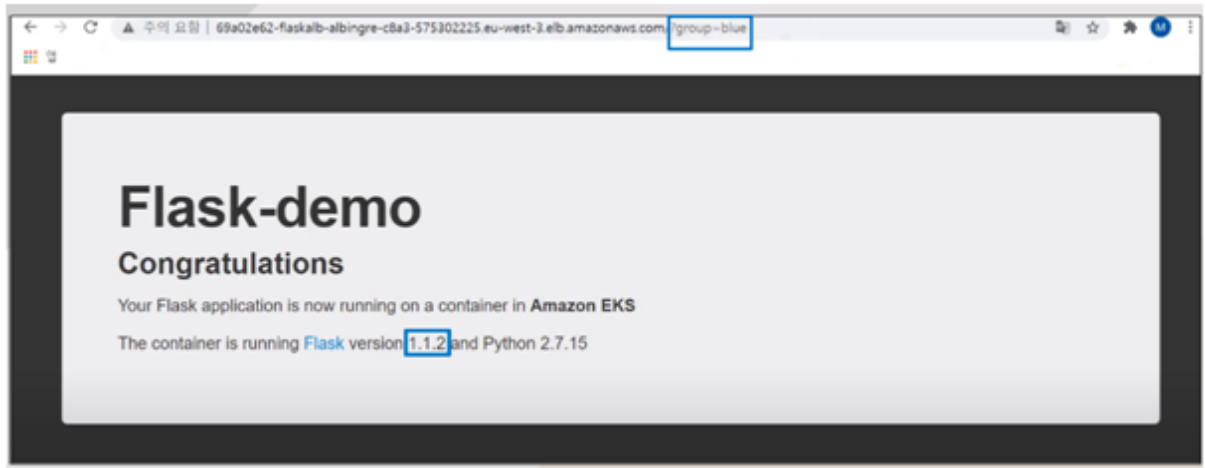
[CodeDeploy Blue/Green 배포방식]

#### 4.2.3 결과 화면

파리 리전에 Cloud9 을 사용하여 EKS 클러스터에 CI/CD 파이프라인을 만들어 웹 기반의 앱을 컨테이너화시켜 블루그린 무중단 방식으로 배포를 해보았습니다. 무중단 방식으로는 크게 Rolling, Blue/Green, Canary 가 있습니다. 여기서의 무중단 배포란 서비스의 중단 없이 구버전 환경인 그린 서비스에서 신규 버전을 동시에 서비스하는 것을 의미합니다. 문제 발생 시 기존 버전으로 로드밸런서 타겟 그룹의 트래픽을 그린 테스트 환경 쪽으로 전환이 되어지면서 롤백이 쉽게 되도록 해줍니다. 하지만 단점으로는 두 배의 Pod 를 실행해줘야 하므로 많은 리소스가 필요하여 비용이 많이 듭니다. 도커화된 Flask 앱을 배포하기위한 사전 작업으로 ControlPlane 보안그룹에 인바운드 포트를 열어주었고 결과물은 잘 나왔습니다.



[그린 서비스에서 배포된 앱 화면]

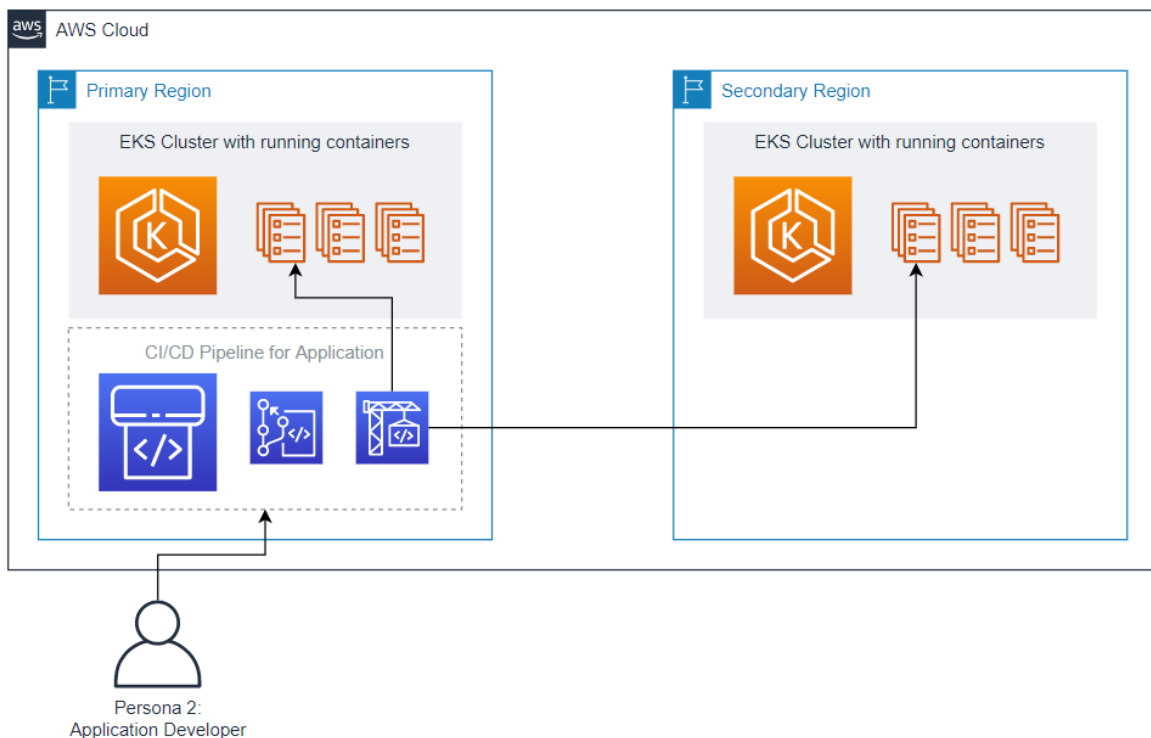


[블루 서비스에서 배포된 앱 화면]

### 4.3 프로젝트 #3 : EKS 를 활용한 멀티 리전 구성

EKS 를 활용한 멀티 리전 배포 방식을 구현해보았습니다. 이 방법은 여러 리전에 kubernetes cluster 을 구성하여 다양한 리전에서 컨테이너화 된 애플리케이션 서비스를 사용할 수 있도록 하는 구성 방법입니다. 그리고 CICD 를 사용해 첫번째 primary 리전과 secondary 리전에 배포를 자동으로 할 수 있게 합니다. 이런 멀티리전 방식을 사용하는 이유는 하나의 리전에 장애가 있더라도 다른 리전에서 서비스할 수 있도록 하기 위함입니다. 최근에 서울 리전 C 에서 장애가 발생해 그 리전에만 배포한 서비스는 이용이 불가했던 적이 있는데 이런 문제점을 해결하기 위해서는 다른 리전에도 동시에 배포를 하여 리다이렉션을 시키는 방식으로 해결한다고 합니다. 이런 재해에 대한 문제를 융통성 있게 해결하는 방법을 모색하고자 다음 구성도를 바탕으로 코로나 웹페이지를 배포해보았습니다. 하지만 이 방식의 주의점은 cluster 을 두 리전에 생성하여 관리하는 방식은 금전적으로 많이 들기 때문에 요금을 고려해서 아키텍처를 설계해야 한다는 것입니다.

### 4.3 아키텍처



구성도는 이렇게 이루어져 있고 Primary Region 은 도쿄 ap-northeast-1 리전 그리고 오레곤 us-west-2 리전을 사용했습니다.

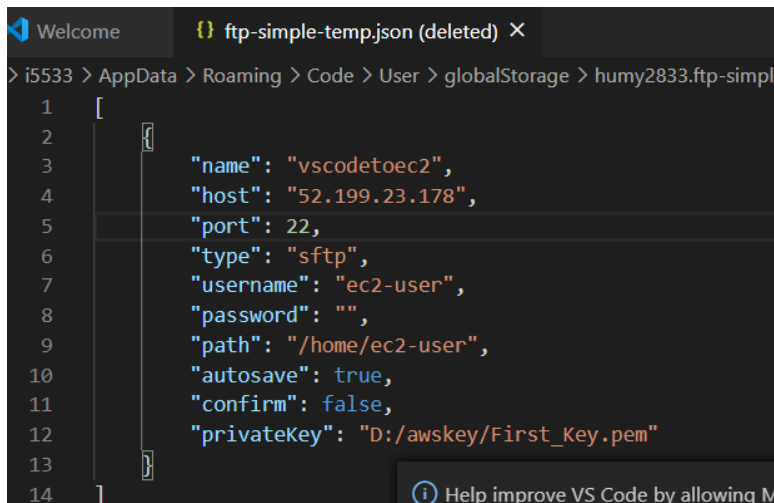
## 4.3.2 AWS 리소스 생성

### 4.3.2.1 CDK 와 EKS 를 사용하기 위한 환경 구성

#### CDK 를 설치하기 전 환경구성

1. AWS CLI 설치
2. AWS configuration 설정
3. Node.js 설치
4. CDK Toolkit 설치
5. Docker-cli 설치
6. Kubectl 설치
7. git, jq 설치
8. cdk 프로젝트의 코드 수정할 IDE 준비

(이 프로젝트에서는 VS code 를 sftp 연결했습니다.)



```
1  [
2    {
3      "name": "vscode-to-ec2",
4      "host": "52.199.23.178",
5      "port": 22,
6      "type": "sftp",
7      "username": "ec2-user",
8      "password": "",
9      "path": "/home/ec2-user",
10     "autosave": true,
11     "confirm": false,
12     "privateKey": "D:/awskey/First_Key.pem"
13   }
14 ]
```

### 4.3.2.2 EKS 를 사용한 다중 배포 환경 구성

<https://github.com/yjw113080/aws-cdk-eks-multi-region-skeleton>

이 github 레포지토리에 존재하는 스켈레톤 코드를 바탕으로 구현해보았습니다.

#### EKS Cluster 생성

EKS cluster 은 Node 들의 집합이라고 할 수 있는데 Master Node 와 Worker Node 들로 이루어져 있습니다. Master Node 는 Worker Node 를 관리하기 위한 Node 로 Kubernetes API 를 Worker Node 에게 전달해주는 역할을 합니다. 이 worker Node 들은 Docker 기반으로 파일시스템을 생성해주어 Docker 기반의 Node 에서 pod 들을 생성해 pod 위에 있는 앱 container 들을 관리한다고 보면 됩니다.

그래서 결국 EKS Cluster 을 생성한다는 것은 Master Node 와 Worker Node 들의 집합을 생성하기 위한 인스턴스를 생성한다고 보면 됩니다.

lib/cluster-stack.ts 파일의 구성은 다음과 같이 진행했습니다.

```

1  import * as cdk from '@aws-cdk/core';
2  import * as iam from '@aws-cdk/aws-iam';
3  import * as eks from '@aws-cdk/aws-eks';
4  import * as ec2 from '@aws-cdk/aws-ec2';
5  import { PhysicalName } from '@aws-cdk/core';
6
7  export class ClusterStack extends cdk.Stack {
8      public readonly cluster: eks.Cluster; //자신의 클러스터를 가져올 수 있게 한다.
9      public readonly firstRegionRole: iam.Role;
10     public readonly secondRegionRole: iam.Role;
11     constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
12         super(scope, id, props);
13
14         const primaryRegion = 'ap-northeast-1';
15         const clusterAdmin = new iam.Role(this, 'cluster-admin', { //iam 설정
16             assumedBy: new iam.AccountRootPrincipal
17         });
18         const cluster = new eks.Cluster(this, 'yj-cluster', { //논리적인 cluster
19             clusterName: 'yj-cluster',
20             version: eks.KubernetesVersion.V1_16, //자기꺼에 설치된 kuber버전(?)
21             defaultCapacity: 2, //cluster에 생성한 인스턴스 개수
22             mastersRole: clusterAdmin, //cluster-admin이라는 애가 이 kubernetes도 제어할 수 있는 롤을 준 것
23             defaultCapacityInstance: new ec2.InstanceType('t3.small') //인스턴스 type은 뭘로 둘건지
24         });
25         this.cluster = cluster;
26
27         if (cdk.Stack.of(this).region == primaryRegion) {
28             this.firstRegionRole = createDeployRole(this, 'for-1st-region', cluster);
29         }
30         else {
31             // 스택이 두 번째 리전값을 가지고 있으면, 새로운 롤을 생성하고 그 리전 클러스터에만 접근할 수 있는 권한을 부여합니다.
32             this.secondRegionRole = createDeployRole(this, 'for-2nd-region', cluster);
33         }
34     }
35
36 }
37 }
```

eks cluster 을 생성하는 stack 코드입니다. 여기서 eks cluster 의 인스턴스 크기에 따라

Pods를 생성할 수 있는 개수가 달라지므로 생성하려는 pod 개수에 맞게 인스턴스 타입을 정해주어야 합니다. 여기서 인스턴스 마다 생성 가능한 pod 개수는 수식을 사용해 구할 수 있습니다.

Elastic Network Interface를  $N$ ,

ENI 마다 할당할 수 있는 IP address 개수를  $M$

이라고 했을 때

$N \times (M-1) + 2$ 가 가용 pod 개수라고 할 수 있습니다.

t3.small은 인터페이스 수 3( $N$ ) 인터페이스당 프라이빗 IPv4 주소수가 4( $M$ )이므로 수식으로 pod 가용영역을 구한다면 11이 됩니다.

인스턴스당 ENI 관련 정보는

[https://docs.aws.amazon.com/ko\\_kr/AWSEC2/latest/UserGuide/using-eni.html#AvailableIpPerENI](https://docs.aws.amazon.com/ko_kr/AWSEC2/latest/UserGuide/using-eni.html#AvailableIpPerENI)

에서 확인할 수 있습니다.

이렇게 cluster를 정의한 후에는 bin/ multi-cluster-ts.ts에 lib/cluster-stack.ts에 작성한 리소스들을 생성하기 위해 생성자를 만들어줍니다.

```
//여기서 생성해줘야지만 생성된다.
const primaryCluster = new ClusterStack(app, `ClusterStackYJv2-${primaryRegion.region}`, {env: primaryRegion});
new ContainerStack(app, `ContainerStackYJv2-${primaryRegion.region}`, {env: primaryRegion, cluster: primaryCluster.cluster});

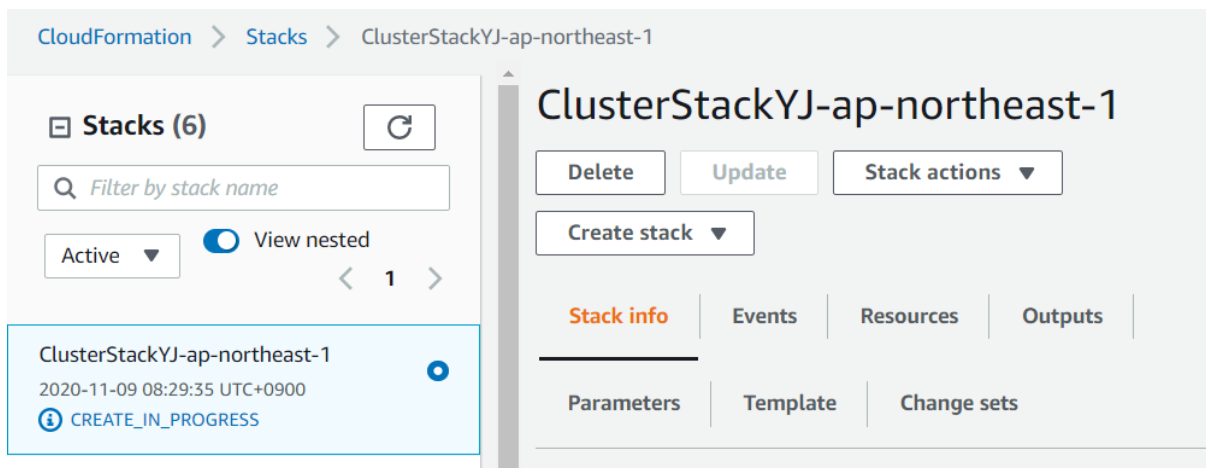
const secondaryCluster = new ClusterStack(app, `ClusterStackYJv2-${secondaryRegion.region}`, {env: secondaryRegion});
new ContainerStack(app, `ContainerStackYJv2-${secondaryRegion.region}`, {env: secondaryRegion, cluster: secondaryCluster.cluster});
```

첫번째 primaryCluster와 SecondaryCluster를 각각 생성해줍니다.

그 후에는 cdk diff를 사용해서 생성 되려는 리소스들을 확인해줍니다. 이때 VPC나 Subnet, Route table, EIP 등 네트워크 관련 설정들은 굳이 사용자가 설정해주지 않는다면 default로 생성해주는 경우가 많기 때문에 이 부분은 cdk의 장점이라고 할 수 있습니다.

```
Resources
+ AWS::IAM::Role cluster-admin clusteradmin648C9E8C
+ AWS::EC2::VPC yj-cluster/DefaultVpc yjclusterdefaultvpc5522F95
+ AWS::EC2::Subnet yj-cluster/defaultVpc/PublicSubnet1/Subnet yjclusterdefaultvpcPublicSubnet1Subnet8C342E5A
+ AWS::EC2::RouteTable yj-cluster/DefaultVpc/PublicSubnet1/RouteTable yjclusterdefaultvpcPublicSubnet1RouteTable19A81CFD
+ AWS::EC2::SubnetRouteTableAssociation yj-cluster/DefaultVpc/PublicSubnet1/RouteTableAssociation yjclusterdefaultvpcPublicSubnet1RouteTableAssociation7F94CA76
+ AWS::EC2::Route yj-cluster/DefaultVpc/PublicSubnet1/DefaultRoute yjclusterdefaultvpcPublicSubnet1DefaultRouteCF3EE8DD
+ AWS::EC2::EIP yj-cluster/DefaultVpc/PublicSubnet1/EIP yjclusterdefaultvpcPublicSubnet1EIP83AD64D2
+ AWS::EC2::NatGateway yj-cluster/DefaultVpc/PublicSubnet1/NATGateway yjclusterdefaultvpcPublicSubnet1NATGateway7CCEA10A
+ AWS::EC2::Subnet yj-cluster/DefaultVpc/PublicSubnet2/Subnet yjclusterdefaultvpcPublicSubnet2Subnet5696F7F1
+ AWS::EC2::RouteTable yj-cluster/DefaultVpc/PublicSubnet2/RouteTable yjclusterdefaultvpcPublicSubnet2RouteTableCC959A0E
+ AWS::EC2::SubnetRouteTableAssociation yj-cluster/DefaultVpc/PublicSubnet2/RouteTableAssociation yjclusterdefaultvpcPublicSubnet2RouteTableAssociationA68749C5
+ AWS::EC2::Route yj-cluster/DefaultVpc/PublicSubnet2/DefaultRoute yjclusterdefaultvpcPublicSubnet2DefaultRoute2C112FAD
+ AWS::EC2::EIP yj-cluster/DefaultVpc/PublicSubnet2/EIP yjclusterdefaultvpcPublicSubnet2EIP9687C223
+ AWS::EC2::NatGateway yj-cluster/DefaultVpc/PublicSubnet2/NATGateway yjclusterdefaultvpcPublicSubnet2NATGateway3C4BBEFA
+ AWS::EC2::Subnet yj-cluster/DefaultVpc/PublicSubnet3/Subnet yjclusterdefaultvpcPublicSubnet3Subnet63B5AD45
+ AWS::EC2::RouteTable yj-cluster/DefaultVpc/PublicSubnet3/RouteTable yjclusterdefaultvpcPublicSubnet3RouteTableA9043027
+ AWS::EC2::SubnetRouteTableAssociation yj-cluster/DefaultVpc/PublicSubnet3/RouteTableAssociation yjclusterdefaultvpcPublicSubnet3RouteTableAssociationA20C2077
+ AWS::EC2::Route yj-cluster/DefaultVpc/PublicSubnet3/DefaultRoute yjclusterdefaultvpcPublicSubnet3DefaultRoute35E50F21
+ AWS::EC2::EIP yj-cluster/DefaultVpc/PublicSubnet3/EIP yjclusterdefaultvpcPublicSubnet3EIP80C73B3
+ AWS::EC2::NatGateway yj-cluster/DefaultVpc/PublicSubnet3/NATGateway yjclusterdefaultvpcPublicSubnet3NATGatewayE1D0C0C
```

cdk deploy 를 통해 cdk diff 로 생성되려는 리소스들을 cloudformation 의 stack 으로 생성되도록 실행시킵니다.



그러면 이렇게 cluster stack 이 cloudformation 에 생성되는 것을 확인 할 수 있습니다.

## Container 생성

```

1 import * as fs from 'fs';
2 import * as yaml from 'js-yaml';
3 import * as eks from '@aws-cdk/aws-eks';
4
5
6 export function readYamlFromDir(dir: string, cluster: eks.Cluster) {
7   fs.readdir(dir, 'utf8', (err, files) => {
8     if (files!=undefined) {
9       files.forEach((file) => {
10        if (file.split('.').pop()=='yaml') {
11          fs.readFile(dir+file, 'utf8', (err, data) => {
12            if (data!=undefined) {
13              let i = 0;
14              yaml.loadAll(data).forEach((item) => {
15                cluster.addManifest(file.substr(0,file.length-5)+i, item);
16                i++;
17              })
18            }
19          })
20        }
21      }) else {
22        console.log(`${dir} is empty`);
23      }
24    }
25  })
26 }

```

```

1 import * as cdk from '@aws-cdk/core';
2 import { readYamlFromDir } from '../utils/read-file';
3 import { EksProps } from './cluster-stack';
4
5
6 export class ContainerStack extends cdk.Stack {
7   constructor(scope: cdk.Construct, id: string, props: EksProps) {
8     super(scope, id, props);
9     const cluster = props.cluster; //cluster 변수 선언해서 넣어주기
10    const commonFolder = './yaml-common'; //어떤 환경이든 동일하게 적용될 yaml
11    const regionFolder = `./yaml-${cdk.Stack.of(this).region}`; //region에 따라 다르게 적용될 yaml
12
13    readYamlFromDir(commonFolder, cluster);
14    readYamlFromDir(regionFolder, cluster);
15
16
17  }
18 }

```

[사진] Container Stack 파일에서 nginx yaml 파일과 namespace yaml 파일을 읽어오는 함수

생성된 Cluster 에 nginx 웹서버에 대한 pod 를 생성해보았습니다. 매커니즘은 Container 이라는 stack 을 정의하고 여기서 yaml 파일을 읽는 코드를 호출하여 nginx 웹서버에 대한 deployment(pod 와 replicaset 에 대한 정보)를 cluster 에 추가시켜줍니다.

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5   labels:
6     app: nginx
7 spec:
8   replicas: 2
9   selector:
10    matchLabels:
11      app: nginx
12   template:
13     metadata:
14       labels:
15         app: nginx
16     spec:
17       containers:
18       - name: nginx
19         image: nginx:1.7.9
20         ports:
21         - containerPort: 80

```

nginx 웹서버 yaml 파일



```
[root@ip-172-31-38-228 aws-cdk-eks-multi-region-skeleton]# kubectl get ns
NAME                STATUS   AGE
default              Active   44m
kube-node-lease      Active   44m
kube-public          Active   44m
kube-system          Active   44m
management           Active   24m
[root@ip-172-31-38-228 aws-cdk-eks-multi-region-skeleton]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-54f57cf6bf-kqfnr   1/1     Running   0           24m
nginx-deployment-54f57cf6bf-v8w6c   1/1     Running   0           24m
```

결과적으로 namespace 와 nginx deployment 가 pod 로 선언된 것을 확인 할 수 있습니다.

### 4.3.3 CI/CD

#### codecommit Repository 설정하기

CI/CD stack 을 로컬 git 레포지토리와 aws repository 인 codecommit 과 연동하고 repository 에 존재하는 코드를 docker 로 이미지화 하여 aws 의 ecr 이미지 repository 에 올린후 pod 에 배포하는 작업을 진행합니다.

```
cluster-stacks.ts  TS ccd-stacks.ts  TS container-stacks.ts  TS multi-cluster-ts.ts
aws-cdk-eks-multi-region-skeleton > lib > TS ccd-stacks.ts > CcdStack > constructor
1  import * as cdk from '@aws-cdk/core';
2  import codecommit = require('@aws-cdk/aws-codecommit');
3  import ecr = require('@aws-cdk/aws-ecr');
4  import codepipeline = require('@aws-cdk/aws-codepipeline');
5  import pipelineAction = require('@aws-cdk/aws-codepipeline-actions');
6  import { codeToECRSpec, deployToEKSSpec } from '../utils/buildspecs';
7  import { CcdProps } from './cluster-stack';
8
9
10
11 export class CcdStack extends cdk.Stack {
12
13     constructor(scope: cdk.Construct, id: string, props: CcdProps) {
14         super(scope, id, props);
15
16         const primaryRegion = 'ap-northeast-1';
17         const secondaryRegion = 'us-west-2';
18         const helloPyRepo = new codecommit.Repository(this, 'hello-py-for-demo', {
19             repositoryName: 'hello-py-${cdk.Stack.of(this).region}'
20         });
21
22         new cdk.CfnOutput(this, 'codecommit-uri', {
23             exportName: 'CodeCommitURL',
24             value: helloPyRepo.repositoryCloneUrlHttp
25         });
26
27     }
28 }
29
30
31
```

codecommit 에 대한 repository 를 생성하고 cdk.cfnoutput 으로 code commit repository 에 대한 url 을 출력하도록 하면 로컬 git repository 와 연동할 수 있는 url 을 출력하게 되는데 이것은 내가 만든 코드를 clone 한 후 git remote 으로 연동해주면 됩니다.

```

1 FROM openjdk:8-jdk-alpine as build
2
3 ENV MAVEN_VERSION 3.5.4
4 ENV MAVEN_HOME /usr/lib/mvn
5 ENV PATH $MAVEN_HOME/bin:$PATH
6
7 RUN wget http://archive.apache.org/dist/maven/maven-3/$MAVEN_VERSION/binaries/apache-maven-$MAVEN_VERSION-bin.tar.gz && \
8     tar -zxvf apache-maven-$MAVEN_VERSION-bin.tar.gz && \
9     rm apache-maven-$MAVEN_VERSION-bin.tar.gz && \
10     mv apache-maven-$MAVEN_VERSION /usr/lib/mvn
11
12 RUN ["mkdir","-p","/app"]
13 WORKDIR /app
14 COPY mvnw ./mvnw
15 COPY .mvn ./mvn
16 COPY pom.xml ./pom.xml
17 COPY src ./src
18 RUN mvn clean package
19
20 FROM openjdk:8-jdk-alpine
21 ARG DEP=/app/target
22 COPY --from=build ${DEP}/AWSCoronaWebpage-0.0.1-SNAPSHOT.war /app/app.war
23 EXPOSE 8080
24 ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app/app.war"]

```

dockerfile

내 코드에는 dockerfile 이 존재해야하며 이것이 있어야 내 코드를 앱 컨테이너로 변환할 수 있습니다.

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: coronaweb-spring
5 spec:
6   type: LoadBalancer
7   ports:
8     - port: 80
9     targetPort: 8080
10  selector:
11    app: coronaweb-spring
12  ---
13 apiVersion: apps/v1
14 kind: Deployment
15 metadata:
16   name: coronaweb-spring
17 spec:
18   replicas: 1
19   strategy:
20     type: RollingUpdate
21   rollingUpdate:
22     maxUnavailable: 2
23     maxSurge: 3
24   selector:
25     matchLabels:
26       app: coronaweb-spring
27   template:
28     metadata:
29       labels:
30         app: coronaweb-spring
31     spec:
32       containers:
33         - name: coronaweb-spring
34           image: CONTAINER_IMAGE
35           imagePullPolicy: Always
36           securityContext:
37             privileged: false
38             readOnlyRootFilesystem: false
39             allowPrivilegeEscalation: false
40           ports:
41             - containerPort: 8080

```

app-deployment.yaml 파일

Service(load balancer)와 deployment(corona webpage)에 대한 yaml 파일도 생성해야 deploy 하는 과정에서 이 yaml 파일을 바탕으로 application 을 생성하게 됩니다. Service 는 들어오는 포트 80 에서 8080 의 targetport 로 redirection 을 시켜주는 작업을 해주고 deployment 에서 replica 는 1 로 설정 해 놓았는데 가용성 서버를 위해 2 개 이상으로 설정해 놓아도 되지만 클러스터의 인스턴스에 따른 pod 개수를 최대한 최소로 맞추기 위해 1 로 설정했습니다. 앱에서 로그를 docker 로 생성되는 파일시스템에 써야 하므로 readOnlyRootFilesystem 이 false 로 설정 되어있어야 합니다.

이런 과정들을 마치게 되면 codecommit repository 에 해당 코드를 올려야 합니다.

git add, git commit -m "message", git push codecommit master

이런 git 과정으로 repository 에 올릴 수 있습니다.

## ECR 이미지 생성과 build

```
//ECR 레포지토리 생성
const ecrForMainRegion = new ecr.Repository(this, `ecr-for-coronaweb`);

//docker 이미지를 빌드하는 codbuild 프로젝트 생성
const buildForECR = codeToECRSpec(this, ecrForMainRegion.repositoryUri);
///utils 폴더에 이 워크샵에서 사용할 빌드 스펙을 미리 정의해두었습니다. 자세한 빌드 스펙이 궁금하

ecrForMainRegion.grantPullPush(buildForECR.role!); //ECR에 이미지를 push할 수 있는 권한
```

codecommit 의 코드를 이미지로 만들어 이미지 repository 에 올리기 위해 ecr. repository 를 생성합니다. ecr 에 push 할 수 있는 권한 설정을 합니다.

codeToECRSpec 함수 (사용자 정의 함수)를 사용해 ecr repository 의 url 로 docker 이미지로 build 합니다.

```
export function codeToECRSpec (scope: cdk.Construct, apprepo: string) : PipelineProject {
  const buildForECR = new codebuild.PipelineProject(scope, 'build-to-ecr', {
    projectName: 'build-to-ecr',
    environment: {
      buildImage: codebuild.LinuxBuildImage.UBUNTU_14_04_DOCKER_18_09_0,
      privileged: true
    },
    environmentVariables: { 'ECR_REPO_URI': {
      value: apprepo
    } },
    buildSpec: codebuild.BuildSpec.fromObject({
      version: "0.2",
      phases: {
        pre_build: {
          commands: [
            'env', '$(aws ecr get-login --region $AWS_DEFAULT_REGION --no-include-email)',
            'IMAGE_TAG=$CODEBUILD_RESOLVED_SOURCE_VERSION'
          ]
        },
        build: {
          commands: [
            'docker build -t $ECR_REPO_URI:latest .',
            'docker tag $ECR_REPO_URI:latest $ECR_REPO_URI:$IMAGE_TAG'
          ]
        },
        post_build: {
          commands: [
            'docker push $ECR_REPO_URI:latest',
            'docker push $ECR_REPO_URI:$IMAGE_TAG'
          ]
        }
      }
    })
  });
  return buildForECR;
}
```

위의 코드는 codeToECRSpec 함수입니다. docker build 를 해서 docker 이미지로 만들고 docker push 로 ecr repository 의 url 에 올리는 작업을 하면 build 한 후에 이미지 올리는 과정이 완료됩니다.

## ECR 이미지를 배포하는 코드 구성

```
//deploy하기 위한 코드
const deployToMainCluster = deployToEKSpec(this, primaryRegion, props.firstRegionCluster, e
///utils 폴더에 이 워크샵에서 사용할 빌드 스펙을 미리 정의해두었습니다. 자세한 빌드 스펙이 궁금하

const deployTo2ndCluster = deployToEKSpec(this, secondaryRegion, props.secondRegionCluster,
//코드 pipeline 생성하는 코드
const sourceOutput = new codepipeline.Artifact();
```

```
44 new codepipeline.Pipeline(this, 'multi-region-eks-dep-coronawebjd', {
45   stages: [ {
46     stageName: 'Source',
47     actions: [ new pipelineAction.CodeCommitSourceAction({
48       actionName: 'CatchSourceFromCode',
49       repository: 'helloPyRepo',
50       output: sourceOutput,
51     })]
52   }, {
53     stageName: 'Build',
54     actions: [ new pipelineAction.CodeBuildAction({
55       actionName: 'BuildAndPushToECR',
56       input: sourceOutput,
57       project: buildForECR
58     })]
59   },
60   {
61     stageName: 'DeployToMainEKSCluster',
62     actions: [ new pipelineAction.CodeBuildAction({
63       actionName: 'DeployToMainEKSCluster',
64       input: sourceOutput,
65       project: deployToMainCluster
66     })]
67   }, {
68     stageName: 'ApproveToDeployTo2ndRegion',
69     actions: [ new pipelineAction.ManualApprovalAction({
70       actionName: 'ApproveToDeployTo2ndRegion'
71     })]
72   },
73   {
74     stageName: 'DeployTo2ndRegionCluster',
75     actions: [ new pipelineAction.CodeBuildAction({
76       actionName: 'DeployTo2ndRegionCluster',
77       input: sourceOutput,
78       project: deployTo2ndCluster
79     })]
80   }
81 ]
82 });
83
84
```

cluster 에 deploy 하는 함수 deployToEKSpec(사용자 정의 함수)를 사용해 eks cluster 에 pod 로 배포합니다. 이 과정에서는 command 에서의 AWS configuration 에 대한 설정을 먼저 하고 이전에 코드에 만들었던 app-deployment.yaml 파일에 CONTAINER\_IMAGE 부분을 ecr 에 있는 이미지 파일에 대한 uri 를 넣고 kubectl apply 를 통해 변경된 yaml 파일을 deploy 하도록 하면 ecr 이미지에 대한 앱컨테이너가 배포됩니다.

## CodePipeline 을 생성하고 WAS 서버 배포

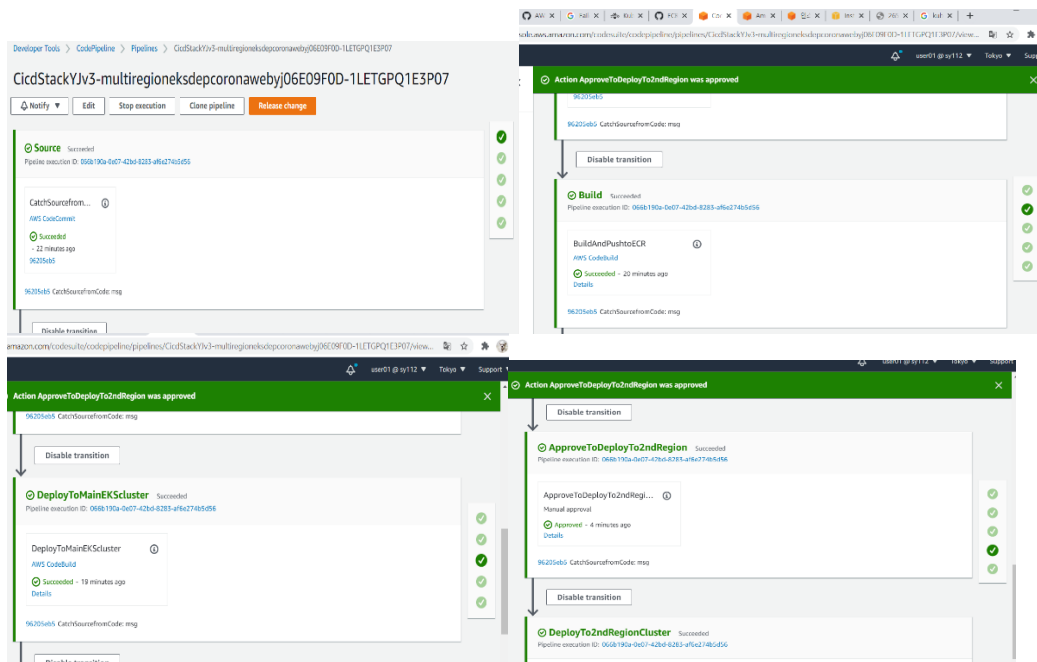
이런 build, deploy 과정을 담은 함수의 결과들을 변수에 넣어 놓고 code pipeline 에 각 과정들의 이름을 넣고 변수들을 넣으면 과정이 어떻게 진행이 되고 이에 대한 로그, 결과를 확인 할 수 있게 됩니다.

```

43
44     new codepipeline.Pipeline(this, 'multi-region-eks-dep-coronawebj', {
45         stages: [ {
46             stageName: 'Source',
47             actions: [ new pipelineAction.CodeCommitSourceAction({
48                 actionName: 'CatchSourceFromCode',
49                 repository: helloPyRepo,
50                 output: sourceOutput,
51             })]
52         }, {
53             stageName: 'Build',
54             actions: [ new pipelineAction.CodeBuildAction({
55                 actionName: 'BuildAndPushToECR',
56                 input: sourceOutput,
57                 project: buildForECR
58             })]
59         },
60         {
61             stageName: 'DeployToMainEKSCluster',
62             actions: [ new pipelineAction.CodeBuildAction({
63                 actionName: 'DeployToMainEKSCluster',
64                 input: sourceOutput,
65                 project: deployToMainCluster
66             })]
67         }, {
68             stageName: 'ApproveToDeployTo2ndRegion',
69             actions: [ new pipelineAction.ManualApprovalAction({
70                 actionName: 'ApproveToDeployTo2ndRegion'
71             })]
72         },
73         {
74             stageName: 'DeployTo2ndRegionCluster',
75             actions: [ new pipelineAction.CodeBuildAction({
76                 actionName: 'DeployTo2ndRegionCluster',
77                 input: sourceOutput,
78                 project: deployTo2ndCluster
79             })]
80         }
81     ]
82 }
83 });

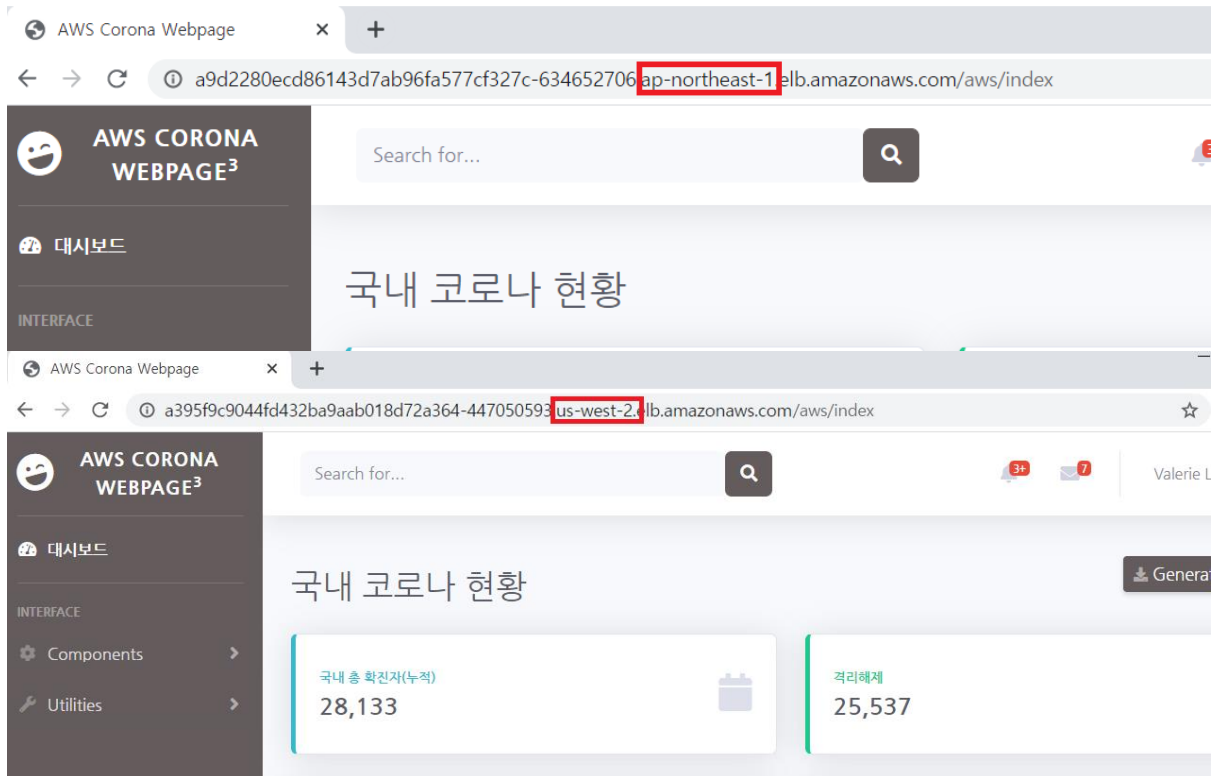
```

배포는 첫번째 PrimaryRegion 에서 진행하고 두번째 Region 에서는 배포자의 허가를 받아야만 배포가 가능할 수 있도록 설정 해놓으면 됩니다.



build 의 상태와 deploy 의 상태를 위와 같이 웹콘솔에서 확인 할 수 있습니다.

#### 4.3.4 결과



결과적으로 이렇게 각 region 에 정상적인 배포를 하게 된 것을 확인 할 수 있습니다.

#### Reference

- <https://nirsa.tistory.com/129>
- <https://blog.nuricloud.com/aws-amazon-eks-intro-usages/>
- <https://x-team.com/blog/introduction-kubernetes-architecture/>
- <https://dev.classmethod.jp/articles/summit-korea-report-eks/>
- <https://cdk-eks-devops.workshop.aws/ko/10-intro.html>