

Serverless Architecture on

with



본 프레젠테이션에선 자세히 다루지 못한 내용은
velog 블로그에서 따로 확인하실 수 있습니다. (QR코드)

본 프레젠테이션과 함께 읽어보시면 더욱 더 쉽게 이해하실 수 있습니다.



목차

00

Overview

- 1.** Presenter Introduction
- 2.** What is the Topic?

01

What is AWS?

- 1.** Cloud
- 2.** AWS(Amazon Web Service)

02

What is Serverless?

- 1.** AWS Lambda
- 2.** Why use Serverless?
- 3.** MSA(MicroService Architecture)
- 4.** Stateless
- 5.** When use Serverless?

목차

03

Project Concept

1. API Endpoint

04

AWS Architecture

1. Lambda

2. API Gateway

3. DynamoDB

4. Cognito

5. S3 + CloudFront (Frontend)

6. IaC + Serverless Framework

7. Github Actions CI/CD

05

Let's write the Code

1. Tech Stacks

2. AWS SDK

3. Explain Lambda Function Code

목차

06

Let's build the Infra

1. Deploy Lambda Function
2. With API Gateway
3. Cognito Authorization
4. CRUD with DynamoDB
5. Serverless Framework
6. CI/CD With Github Actions
7. Frontend with S3 + CloudFront

07

API Testing with Postman

1. Authorizations API
2. Posts CRUD API
3. Users API

08

Calculate Price

1. Lambda
2. API Gateway
3. Cognito
4. DynamoDB
5. S3
6. CloudFront
7. TCO(Total Cost of Ownership)

목차

09

Cleaning AWS Services

10

Conclusion & Insights

Presenter Introduction

20206 김준영

- 세명컴고 보안과 & [넬포유](#)
- 웹([FE](#) · [BE](#) · [DevOps](#)) 공부 중 (특히 DevOps, AWS 등)
 - [타입스크립트](#) 사랑꾼(FE · BE 전부 타입스크립트만 씀: React, NestJS 등)



@yulmwu



이펙티브 타입스크립트 << 매우 강추 도서 (한국어 번역본 있음)

What is the Topic?



퀴즈 게임을 리액트로 제작해봤는데, 유저끼리 경쟁할 수 있게 **랭킹 기능 구현**은 못할까?

⇒ 쓰는 사람도 거의 없는데 고작 그거 하나가지고 EC2 인스턴스 키고 백엔드 배포하고..
돈아깝다 + 서버관리 귀찮다 (계획 철회)

What is the Topic?



인터넷에서 말로만 듣던 **서버리스(Serverless)**를 활용해볼까?

" Serverless "

Cloud

전통적으로 서버를 구축하려면 서버(컴퓨터)를 구매하고 펌웨어, OS 등을 **하나하나 다루고 유지보수**해야 했음

⇒ 초기 인프라 구축 비용 큼

⇒ 유지보수도 어렵고 보안 솔루션을 직접 만들어야함

⇒ 규모 확장도 어려움 (새로 서버를 사야함)

⇒ 인터넷 데이터센터(IDC) 비용도 만만치 않음(전기세, 인터넷 사용료, 냉각 시스템 등등)

⇒ **온 프레미스(On Premies)**

Cloud

그럼 업체 등을 이용해 인터넷을 통해
가상 컴퓨터(서버)로 빌려쓸 순 없을까?

⇒ 클라우드(Cloud)



==테스토스테론이 많은 남자

- 관심사: 리눅스 커널 분석,
(직접)온프레미스/토폴로지 구축
- 서버실가서 서버 랙마운트로 쇠질함
- 단순하고 남자답다
- 이상형이 어셈블리어 처럼 Low Level



==에스트로겐론이 많은 남자

- 관심사: 트렌디한 기술, 클라우드
- 스타벅스에서 자주 출몰, 맥북 씬
- 깊게 공부해본 기술 없음
- 섬세하다
- 타입스크립트 좋아함

AWS(Amazon Web Service)



클라우드 업체 중 1위

사실상 백엔드 구축하는데 필요한 모든게 다 있음

단점: 러닝커브 뽕뽕, 비용도 썸

AWS(Amazon Web Service)



AWS에서 가상 컴퓨터를 사용할때 주로 **EC2(Elastic Cloud Computer)**를 사용함

On-site	IaaS	PaaS	SaaS
Applications	Applications	Applications	Applications
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
O/S	O/S	O/S	O/S
Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking
 You manage			
 Service provider manages			

EC2 = IaaS

가상 컴퓨터까진 제공해주나 OS부터 알잘딱하게 관리 해야함

실행 시간 만큼 요금이 부과됨

What is Serverless?

" 나는 서버 관리도 못하고 365일 24시간 서버 켜두는것도 돈 때문에 싫어요 "

⇒ 그럴 때 **FaaS(Function as a Service)** (ECS 또는 EKS를 통해 컨테이너로 작동하는 Fargate도 서버리스)

⇒ 이게 곧 일반적인 **서버리스(Serverless)**

(서버가 아예 없다는게 아님, 관리가 서버 불필요한 구조라는거)

What is Serverless?

FaaS =

함수 단위로 코드가 구성되며, 지속적으로 실행되는게 아닌 **특정한 이벤트에 트리거되면 실행**되고 종료됨
(직접 서버를 프로비저닝 할 필요 없이 코드만 작성하면 됨)

⇒ AWS 람다(Lambda)



⇒ 서버리스의 핵심

이벤트 트리거 예시)

S3 파일 업로드 → 람다 실행

API Gateway의 HTTP 요청 → 람다 실행 (서버리스 API)

AWS Lambda

AWS에서 제공하는 대표적인 서버리스 서비스이자 FaaS 서비스

⇒ 앞서 말했 듯 특정한 이벤트에 트리거되면 작동함



AWS Lambda

람다 요금의 경우 **요청 수 + 함수 실행 시간**으로 계산함

⇒ 찐 온디맨드(EC2는 켜두는 시간만큼 요금이 나감, 사실상 24시간 요금 부과)

AWS Lambda 요금

리전:

아시아 태평양(서울)

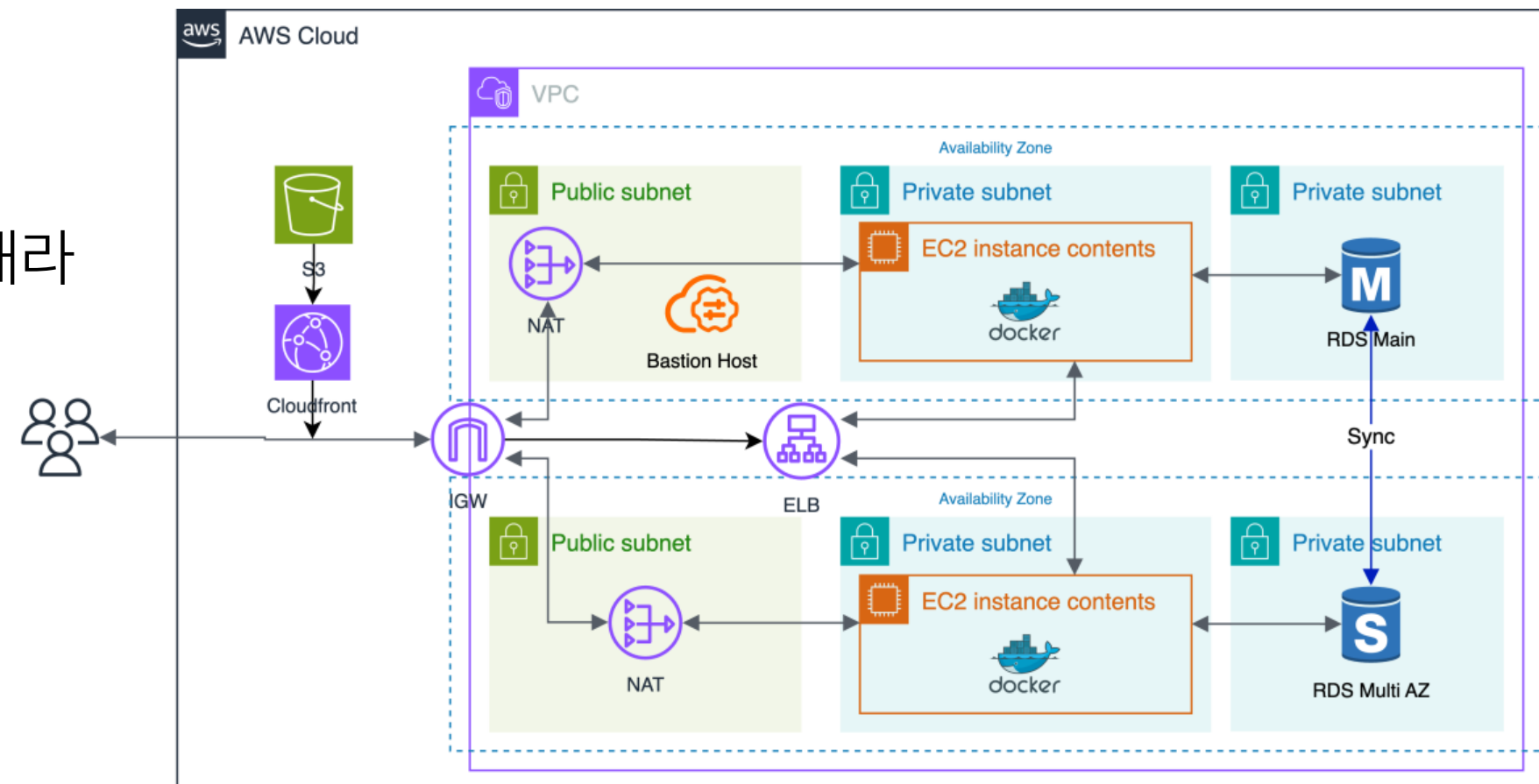
아키텍처	시간	요청
x86 요금		
처음 6십억GB-초/월	GB-초당 USD 0.0000166667	요청 1백만 건당 USD 0.20
다음 9십억GB-초/월	GB-초당 USD 0.000015	요청 1백만 건당 USD 0.20
다음 150억GB-초/월	GB-초당 USD 0.0000133334	요청 1백만 건당 USD 0.20

Why use Serverless?

1. 확장성

기존의 Serverful한 구조(EC2 등)에선 오토스케일링 + 로드 밸런싱으로 확장 가능함

아이고 복잡해라



Why use Serverless?

1. 확장성

기존의 Serverful한 구조(EC2 등)에선 오토스케일링 + 로드 밸런싱으로 확장 가능함

반면 람다에선 서버를 항상 켜두는 구조가 아니기 때문에 알아서 확장됨

+ 동시성 설정 ⇒ 병렬 처리도 간단, CloudFront 연동으로 Lambda@Edge 사용 가능(CDN)

Why use Serverless?

2. 비용 효율적

람다는 요청 수 + 함수 실행 시간으로 계산됨

⇒ 딱 실행한 만큼만 부과되는 구조, 평상시엔 아무 비용도 안나감
(프로비저닝된 동시성 제외)

Why use Serverless?

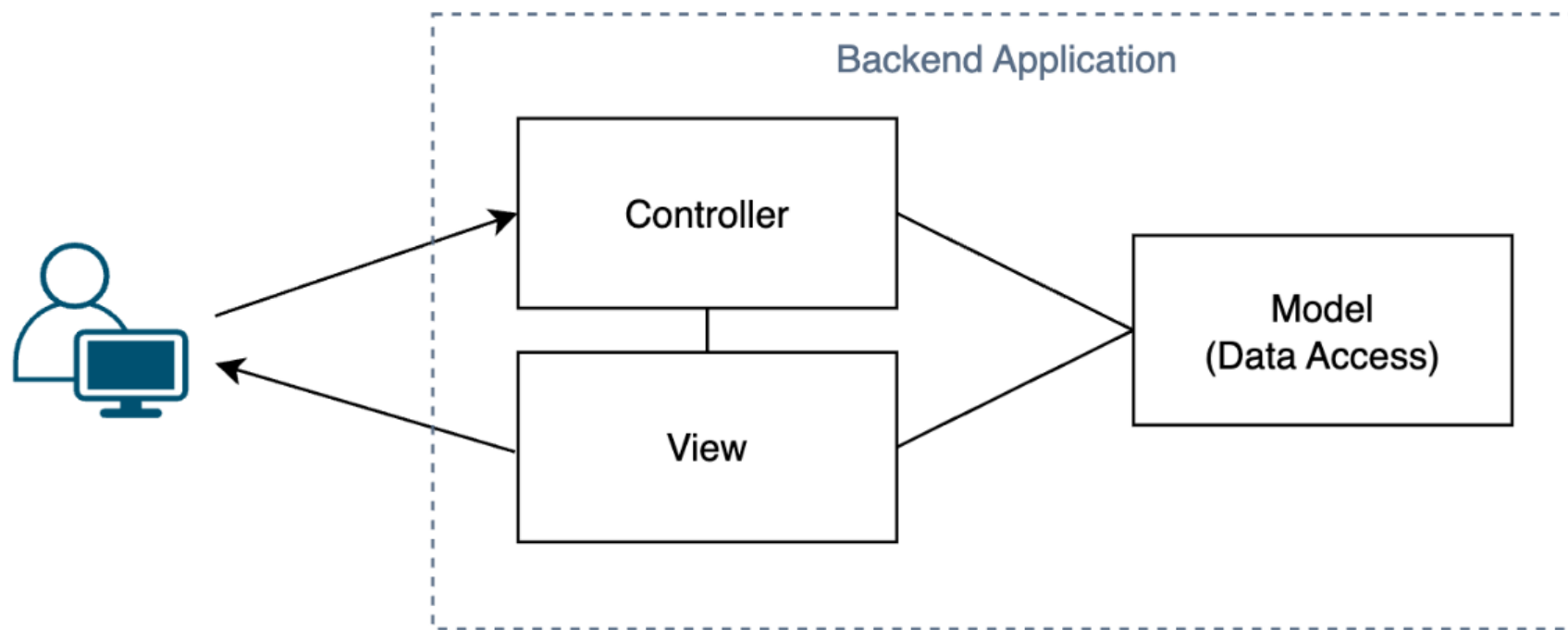
3. 서버 관리 불필요

한번 인프라를 구축해두면 추후 서버를 관리할 필요가 거의 없어짐

다만 함수를 개별적으로 실행하기 때문에 전체적인 디버깅의 어려움, 의존성 관리 어려움 등의 단점

MSA(MicroService Architecture)

전통적으로 백엔드 서버를 구축하고 하나의 백엔드 애플리케이션을 서버에 배포함.



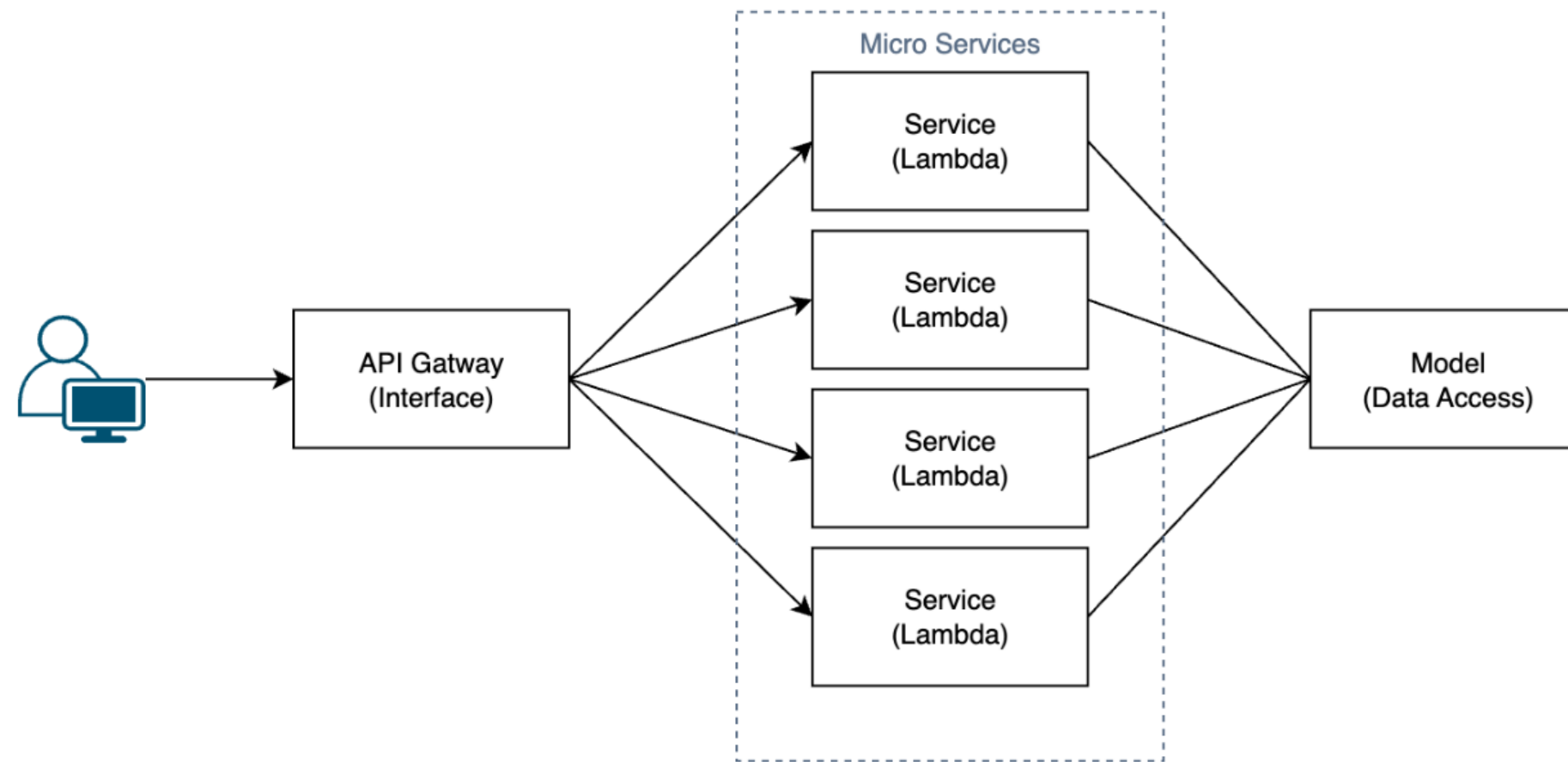
코드 수정하면 다시 배포해야함,
어디 하나 뺏가면 애플리케이션 전체가 다운됨

예: MVC(Model, View, Controller) 아키텍처를 사용한 하나의 큰 백엔드 애플리케이션

⇒ 이런걸 **모놀리식(Monolithic)** 아키텍처라 함

MSA(MicroService Architecture)

MSA = 기능 하나하나를 **작은 서비스 단위(마이크로서비스)**로 개발하고 따로 관리함



서비스 수정 시 변경된 마이크로 서비스만
변경하고 다시 배포하면 됨

⇒ 이러한 구조는 FaaS, 서버리스와 궁합이 좋음

MSA (MicroService Architecture)

MSA = 기능 하나하나를 **작은 서비스 단위(마이크로서비스)**로 개발하고 따로 관리함

다만 서버리스라고 무조건 MSA는 아님(모놀로식으로 코드 작성 가능)

Stateless

FaaS의 특징, 상태를 저장하지 않는 **Stateless**임

⇒ 즉 실행 환경에 데이터를 저장해도 함수 실행 시 환경이 다시 만들어지니 의미가 없음

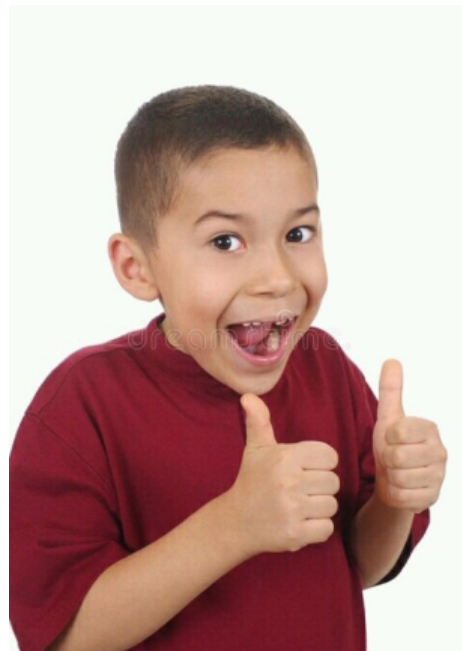
반대로 애플리케이션 내부에 상태를 저장한다면 **Stateful**하다고 함

(다만 추후 설명할 람다의 특징으로 항상 Stateless한건 아님)

다만 DynamoDB와 같이 외부의 리소스에 데이터를 저장하는건 따지지 않음.

When use Serverless?

- 인프라 구축과 관리(DevOps)의 부담을 줄이거나 그러한 관리가 어려울 경우
- 스타트업 등 초기 인프라 구축 비용 부담을 줄이고 확장성 및 유연성을 챙기고 싶은 경우
- 비교적 자주 사용되지 않는 서비스를 구축할 경우
- 특정한 이벤트에 트리거될 때 실행되는 이벤트 기반 애플리케이션을 제작하고 싶은 경우



Project Concept

조건

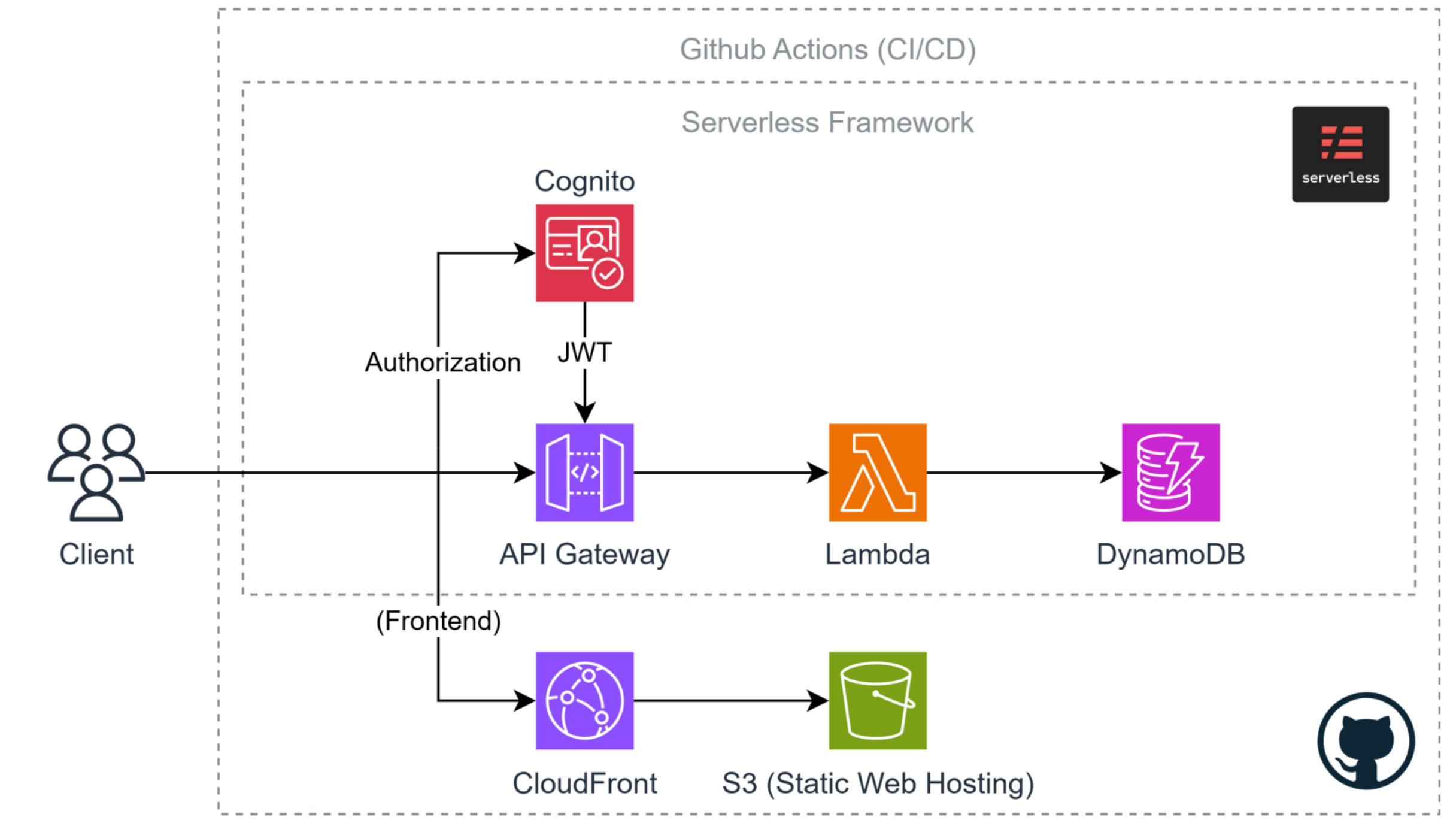
- 전체적으로 **서버리스 환경**에 적용시키기 쉬운 것
- REST API와 CRUD 적용이 될 것
- 인증 시스템을 사용하는 서비스 일 것 (JWT)
- 너무 복잡하진 않을 것

⇒ **게시판 구현 프로젝트**

API Endpoint

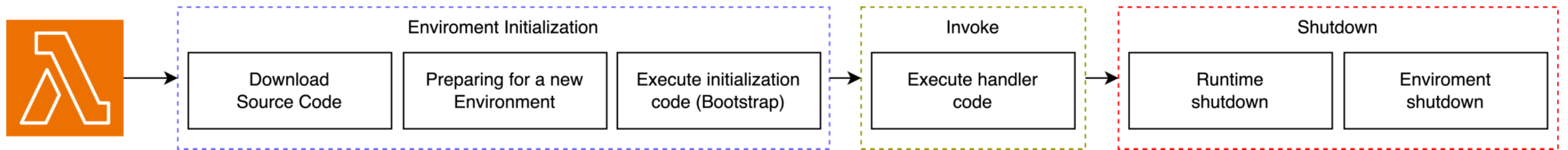
엔드포인트	설명	요청 데이터	응답 데이터	JWT인증 여부
POST /auth/login	로그인	username, password	accessToken,idToken, (Set-Cookie)refreshToken	
POST /auth/confirmEmail	이메일 인증	username, code		
POST /auth/resendEmail	이메일 인증 재전송	username		
POST /auth/logout	로그아웃 (리프레시 토큰 무효화)			0
POST /auth/refresh	엑세스 토큰 재발급		accessToken,idToken	
GET /posts	모든 글 조회		(생략)	
GET /posts/{postId}	특정 글 조회		(생략)	
POST /posts	글 작성	title, content	(생략)	0
PUT /posts/{postId}	글 수정	title, content	(생략)	0
DELETE /posts/{postId}	글 삭제			0
GET /myinfo	로그인된 현재 유저 정보 확인		(생략)	0

AWS Architecture



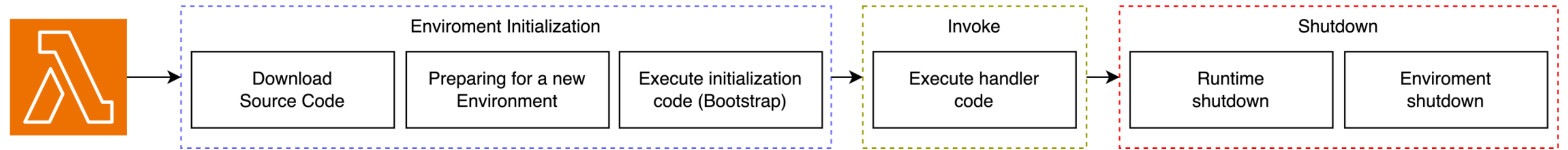
Lambda

앞서 말했 듯 AWS의 대표적인 FaaS 서비스, 동작 과정은 아래와 같음.



1. Environment Initialization
2. Invoke
3. Shutdown

Lambda: Environment Initialization



람다 함수를 실행하기 위한 런타임 플랫폼 등의 **실행 환경(컨테이너)**을 **초기화**하는 단계

소스코드(람다 함수)를 다운받고 초기화 코드를 실행하여 `handler()`가 실행(Invoke) 될 준비를 함
초기화 코드 = 부트스트랩

런타임 플랫폼 구축하는 시간은 비용이 포함되지 않지만 초기화 코드 실행부터 비용에 포함됨

```
// --- 초기화 코드 ---

import { DynamoDBClient } from '@aws-sdk/client-dynamodb'
import { DynamoDBDocumentClient, ScanCommand } from '@aws-sdk/lib-dynamodb'
import { APIGatewayProxyResultV2 } from 'aws-lambda'
import { error, internalServerError } from '../utils/httpError'

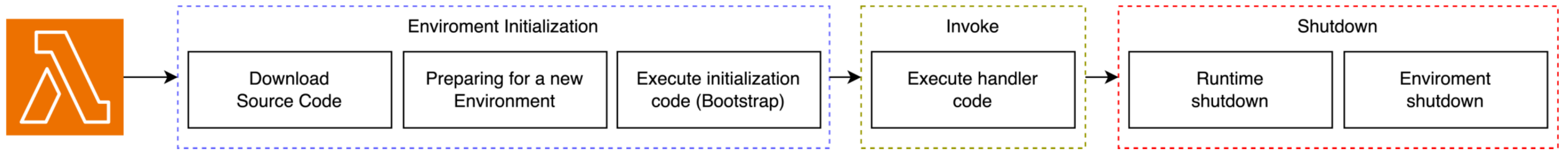
const dynamoDB = DynamoDBDocumentClient.from(new DynamoDBClient({}))

// --- 초기화 코드 ---

// Invoke(실행) 시 실행될 handler() 함수
export const handler = async (): Promise<APIGatewayProxyResultV2> => {
  try {
    const command = new ScanCommand({ TableName: 'Posts' })
    const result = await dynamoDB.send(command)

    return {
      statusCode: 200,
      body: JSON.stringify(result.Items),
    }
  } catch (err) {
    return error(internalServerError((err as Error).message), 'ERR_GET_POSTS_INTERNAL_SERVER_ERROR')
  }
}
```

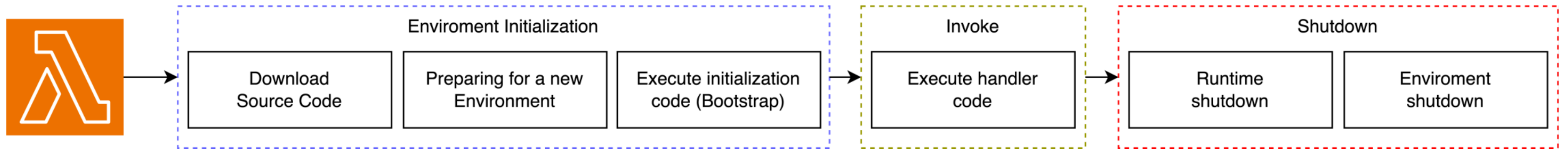
Lambda: Invoke



람다 함수의 핸들러, 즉 (NodeJS 기준) 코드의 `handler()` 함수가 **실행(Invoke)**되는 단계

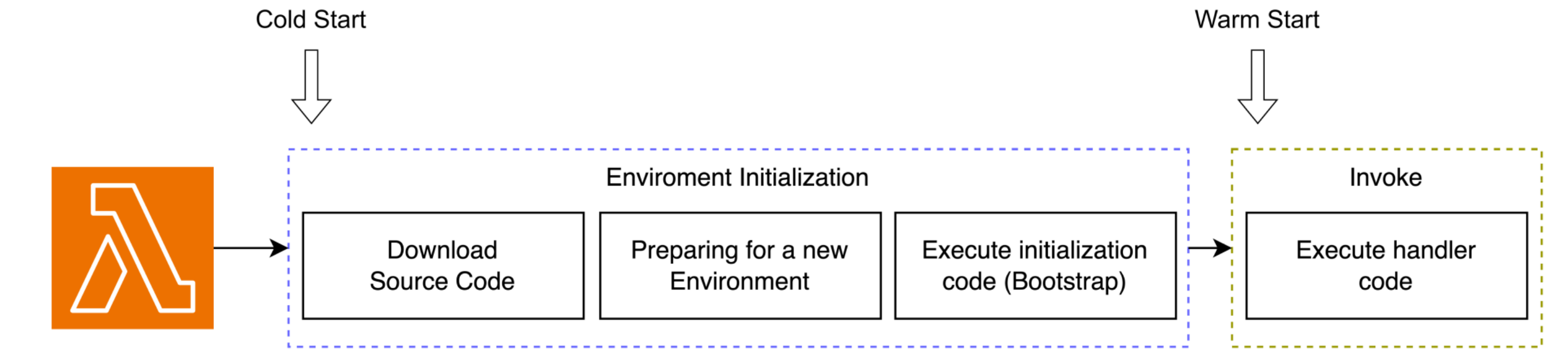
람다 함수의 엔트리 포인트라 생각하면 됨

Lambda: Shutdown



람다 함수가 종료되고 런타임과 실행 환경(컨테이너)이 **종료**되는 단계

Lambda: Cold Start



일반적인 FaaS 구조 자체의 단점인데, 함수를 실행하기 위해선 환경을 세팅하는 것 부터 시작해야함.

문제는 이 과정이 꽤나 오래걸림

⇒ Cold Start

Lambda: Cold Start

로그 이벤트

작업 ▼

테일링 시작

지표 필터 생성

아래의 필터 막대를 사용하여 로그 이벤트의 용어, 구문 또는 값을 검색하고 매칭할 수 있습니다. [필터 패턴에 대해 자세히 알아보기](#)

이벤트 필터링 - Enter 키를 눌러 검색

1분1시간현재 시간대 ▼디스플레이 ▼

▶ | 타임스탬프 | 메시지

현재 이전 이벤트가 없습니다. [재시도](#)

▶

2025-06-24T09:26:24.573+09:00

INIT_START Runtime Version: nodejs:18.v71 Runtime Version ARN: arn:aws:lambda:ap-northeast-2::runtime:bcd7c0a...

▶

2025-06-24T09:26:24.878+09:00

START RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1 Version: \$LATEST

▶

2025-06-24T09:26:25.584+09:00

END RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1

▼

2025-06-24T09:26:25.584+09:00

REPORT RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1 Duration: 704.66 ms Billed Duration: 705 ms Memory Siz...

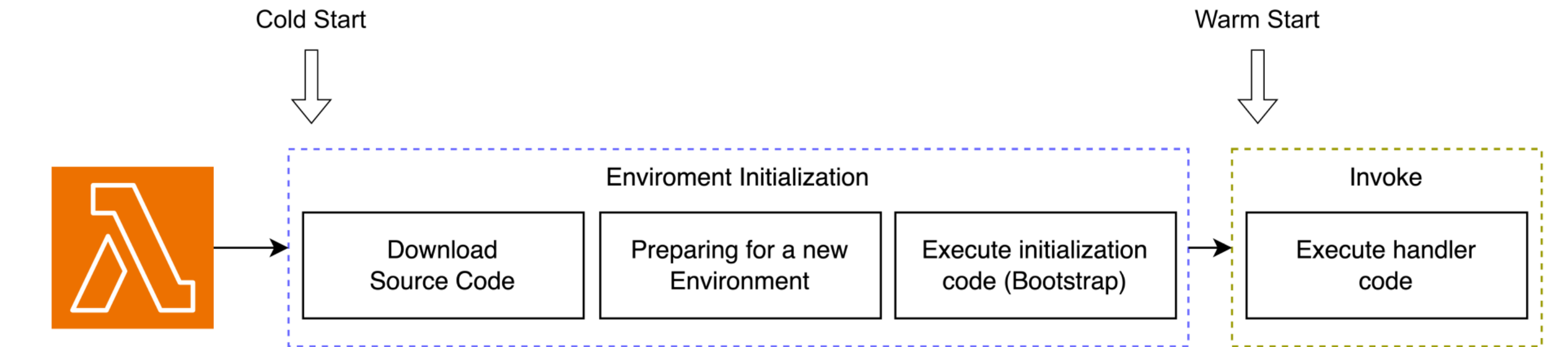
REPORT RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1 Duration: 704.66 ms Billed Duration: 705 ms Memory Size: 128 MB

Max Memory Used: 79 MB Init Duration: 302.54 ms

총 실행 시간(Duration) 700ms - 300ms = 400ms = handler() 실행 시간

Null4U 2학년 김준영

Lambda: Warm Start



그런데 항상 Cold Start되면 초기화 되는 시간이 너무 길고 한번 쓰고 버리기엔 아까움

그래서 람다는 같은 요청에 대해 일정 시간 동안 초기화 단계를 거친 환경을 유지해둬 (약 5분)

⇒ Warm Start

Lambda: Warm Start

로그 이벤트

아래의 필터 막대를 사용하여 로그 이벤트의 용어, 구문 또는 값을 검색하고 매칭할 수 있습니다. [필터 패턴에 대해 자세히 알아보기](#)

이벤트 필터링 - Enter 키를 눌러 검색

1분 1시간

현재 시간대

디스플레이

지표 필터 생성

▶	타임스탬프	메시지
현재 이전 이벤트가 없습니다. 재시도		
▶	2025-06-24T09:26:24.573+09:00	INIT_START Runtime Version: nodejs:18.v71 Runtime Version ARN: arn:aws:lambda:ap-northeast-2::runtime:bcd7c0a...
▶	2025-06-24T09:26:24.878+09:00	START RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1 Version: \$LATEST
▶	2025-06-24T09:26:25.584+09:00	END RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1
▼	2025-06-24T09:26:25.584+09:00	REPORT RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1 Duration: 704.66 ms Billed Duration: 705 ms Memory Siz...
		REPORT RequestId: 371cc355-8cd6-4e33-8e70-afd2554ceec1 Duration: 704.66 ms Billed Duration: 705 ms Memory Size: 128 MB Max Memory Used: 79 MB Init Duration: 302.54 ms
▶	2025-06-24T09:26:51.414+09:00	START RequestId: df954bba-bc6c-46f1-8d05-be583bcad2fc Version: \$LATEST
▶	2025-06-24T09:26:51.824+09:00	END RequestId: df954bba-bc6c-46f1-8d05-be583bcad2fc
▼	2025-06-24T09:26:51.824+09:00	REPORT RequestId: df954bba-bc6c-46f1-8d05-be583bcad2fc Duration: 408.63 ms Billed Duration: 409 ms Memory Siz...
		REPORT RequestId: df954bba-bc6c-46f1-8d05-be583bcad2fc Duration: 408.63 ms Billed Duration: 409 ms Memory Size: 128 MB Max Memory Used: 80 MB
현재 최신 이벤트가 없습니다. 자동 재시도를 일시 중지했습니다. 재개		

같은 요청에 대해 일정 시간 동안은 환경을 유지하는 모습 (두번째 요청엔 초기화 생략)

Lambda: Warm Start

이러한 Warm Start 현상으로 인해 Stateful하게 작동할 수 도 있음

```
let a = 0

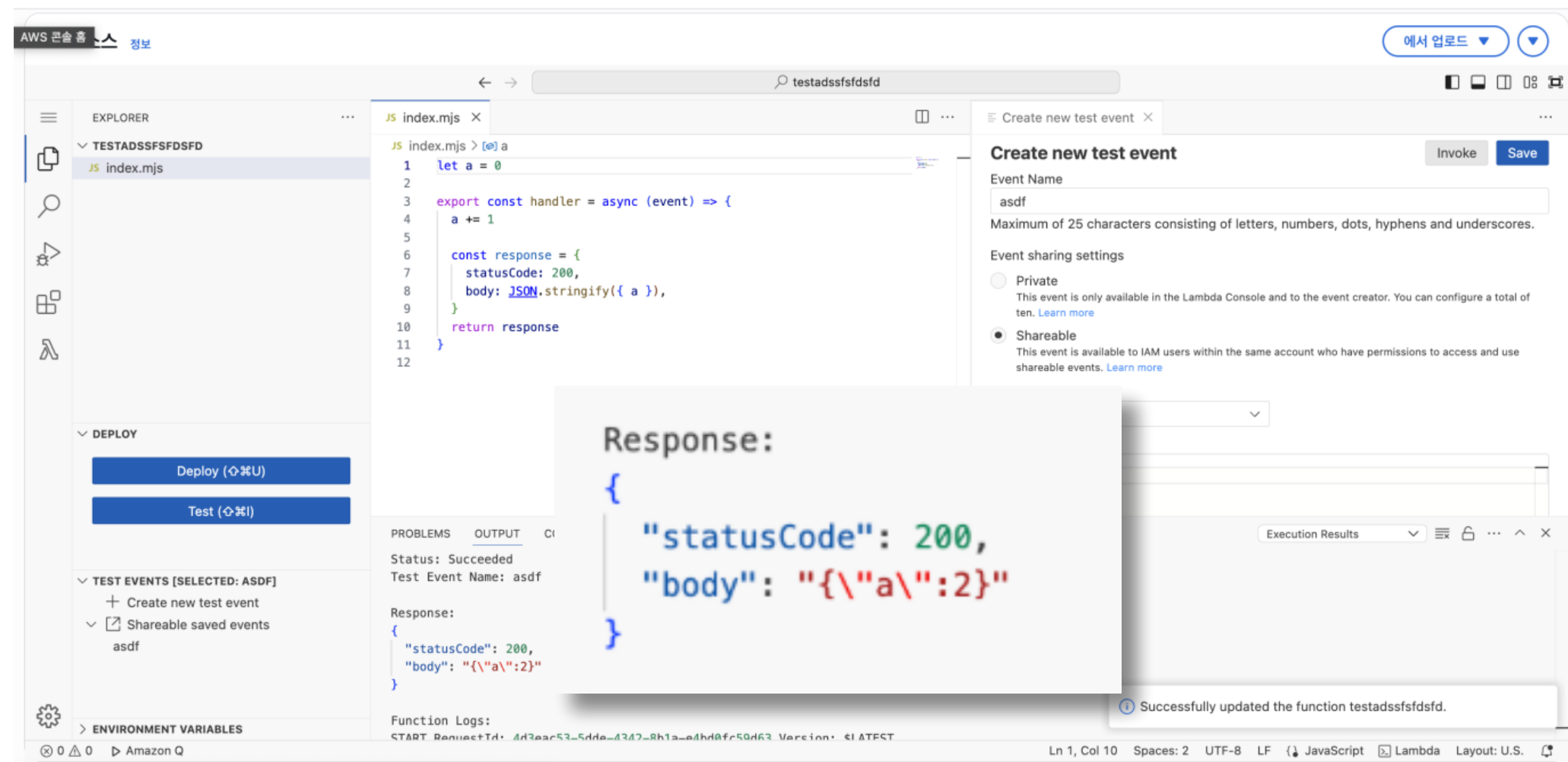
export const handler = async (event) => {
  a += 1

  const response = {
    statusCode: 200,
    body: JSON.stringify({ a }),
  }
  return response
}
```

함수를 빠르게 2번 실행해보면

Lambda: Warm Start

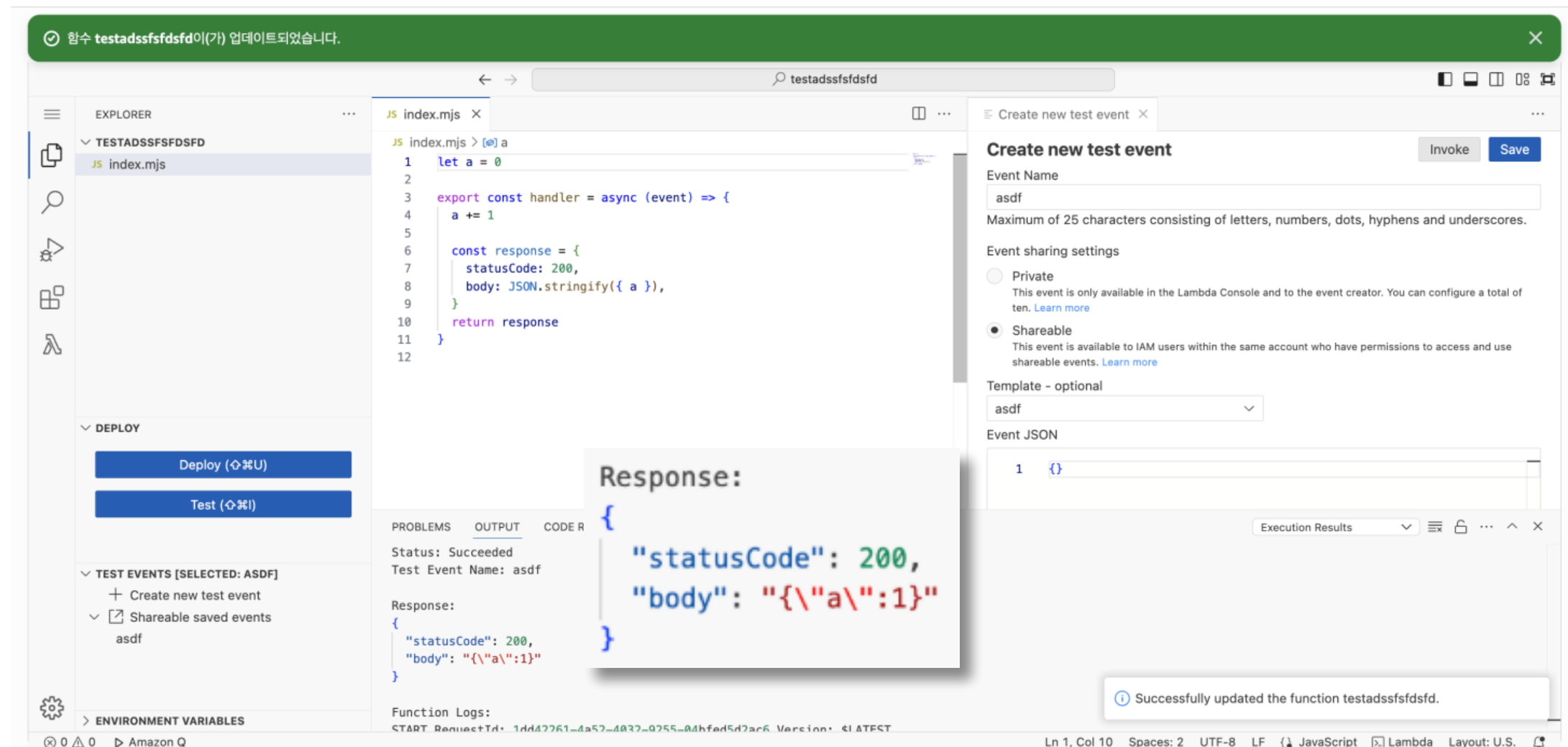
이러한 Warm Start 현상으로 인해 Stateful하게 작동할 수 도 있음



Stateful하게 작동하는 모습, 그래서 Stateless한 코드를 작성해야함

Lambda: Warm Start

이러한 Warm Start 현상으로 인해 Stateful하게 작동할 수 도 있음



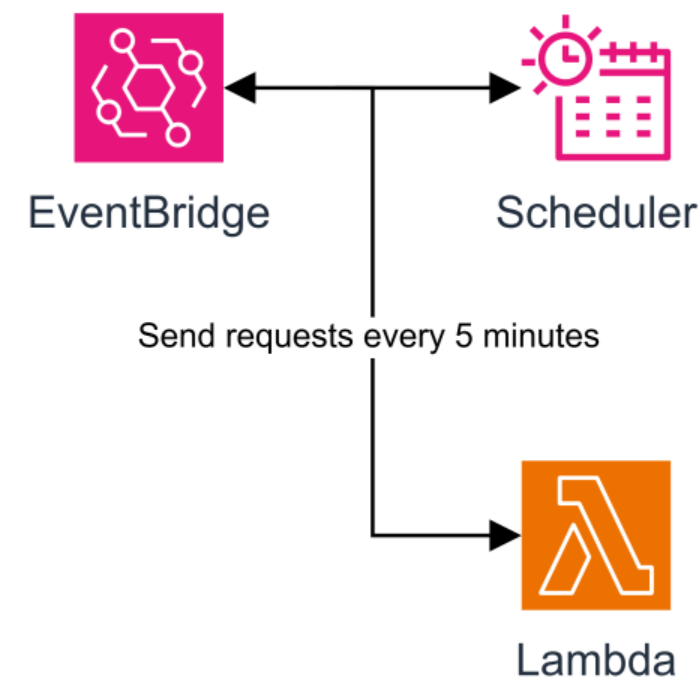
참고로 좀 기다리거나 다시 배포해보면 초기화되는 모습

Lambda: Cold Start Prevention

Cold Start 현상 방지 방법

람다 함수는 일정 기간 동안 환경을 유지하고, 새로운 요청이 들어오면 시간이 초기화됨

1. EventBridge Scheduler

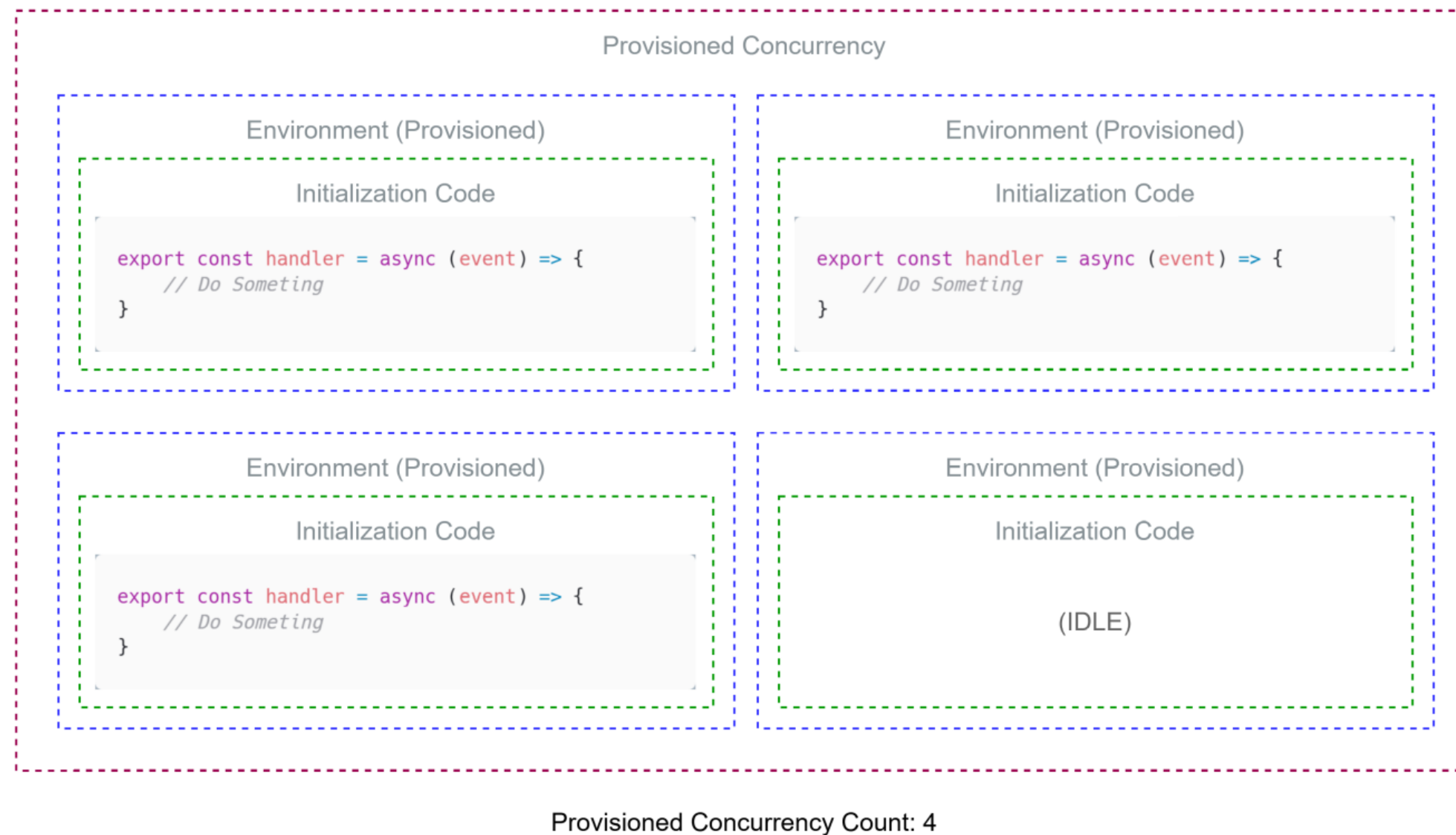


이벤트를 감지해서 다른 서비스로 넘겨주는 AWS EventBridge
EventBridge의 스케줄러를 사용하여 **일정한 간격으로 람다를 실행함**

⇒ 100% 정확한건 아니지만 어느정도 Warm Start 상태가 유지 됨

Lambda: Cold Start Prevention

2. Provisioned Concurrency (프로비저닝된 동시성)



초기화 상태를 실행해둔 환경을
미리 프로비저닝해둠 (세팅해둠)

비싸지만 돈낸 만큼 확실하게 보장함
(SLA 정책 = 돈낸만큼 보장)

(여기서도 오토스케일링 사용 가능)

Lambda: Cold Start Prevention

3. Increase Memory

람다에선 메모리를 늘리면 CPU 성능도 같이 늘어남

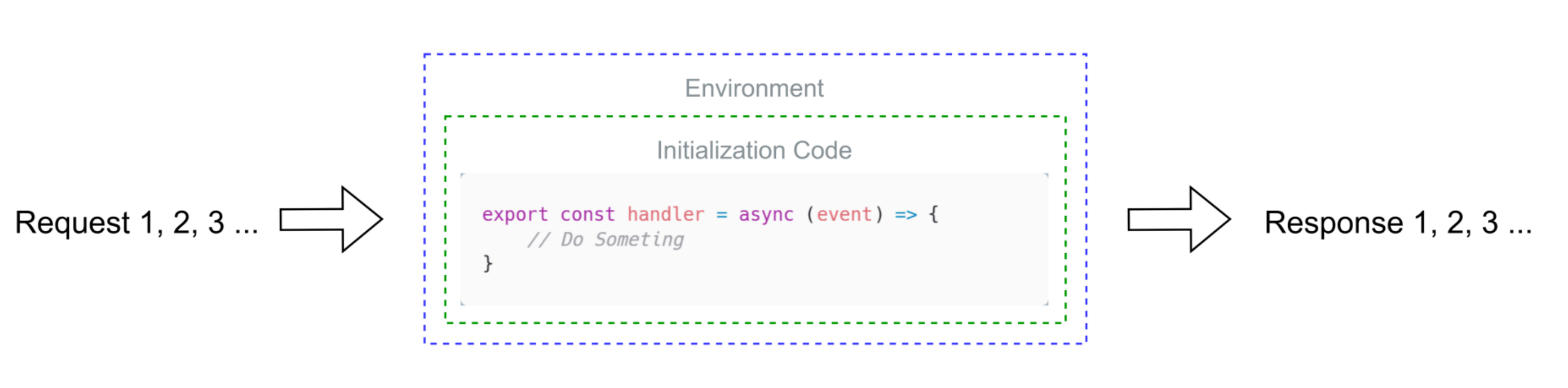
⇒ 초기화 속도 빨라짐

Configured Memory (MB)	Allocated cores
128-1769	1
1770-3538	2
3539-5307	3
5308-7076	4
7077-8845	5
8846-10240	6

Lambda: Concurrently

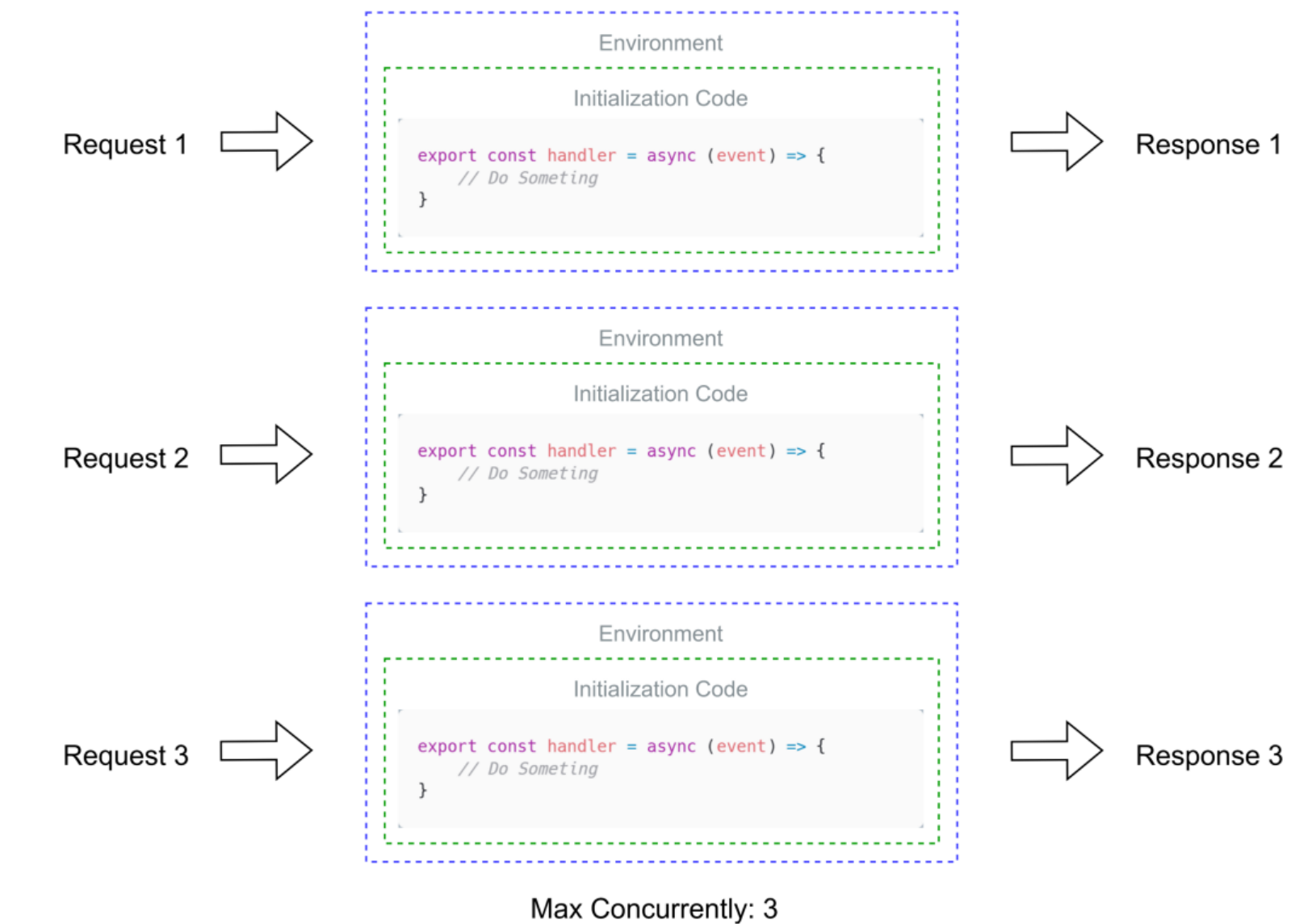
람다에서 동시성은 한번에 몇개의 람다 함수를 처리할 수 있는지를 의미함

람다가 `handler()`를 처리하고 있으면 추가적인 이벤트를 받을 수 없음.



Lambda: Concurrently

람다가 처리 되기를 기다리는건 비효율적이니 여러 람다를 구성하여 병렬로 실행함



Lambda: Limits

람다엔 크게 배포 시 제한과 실행 시 제한이 있음 (실행 시 제한은 따로 설정 가능)

항목	제한	기타
메모리	128MB ~ 10GB	메모리에 따라 CPU 성능이 같이 증가함, 1MB 단위로 조절 가능
실행 시간	최대 15분(900초)	기본 값은 3초
/tmp 용량	512MB	임시 파일을 저장하는 용도로 람다에서 /tmp 디렉토리를 사용할 수 있음.
동시성	최대 1000개	리전 단위 당 최대 1000개 제한

실행 시 제한

Lambda: Limits

람다엔 크게 배포 시 제한과 실행 시 제한이 있음 (실행 시 제한은 따로 설정 가능)

방식	제한	기타
ZIP 파일 직접 업로드	최대 50MB	AWS 내부에서 언팩된 코드의 사이즈 ECR 업로드 후 람다에서 컨테이너 이미지 사용 가능
ZIP 파일 S3 업로드 및 연동	최대 250MB	
ZIP 압축 해제 이후 코드	최대 250MB	
도커 컨테이너 이미지	최대 10GB	
레이어 결합	최대 5개	
개별 레이어	압축 후 최대 50MB	

배포 시 제한

Lambda: Limits

람다엔 크게 배포 시 제한과 실행 시 제한이 있음 (실행 시 제한은 따로 설정 가능)

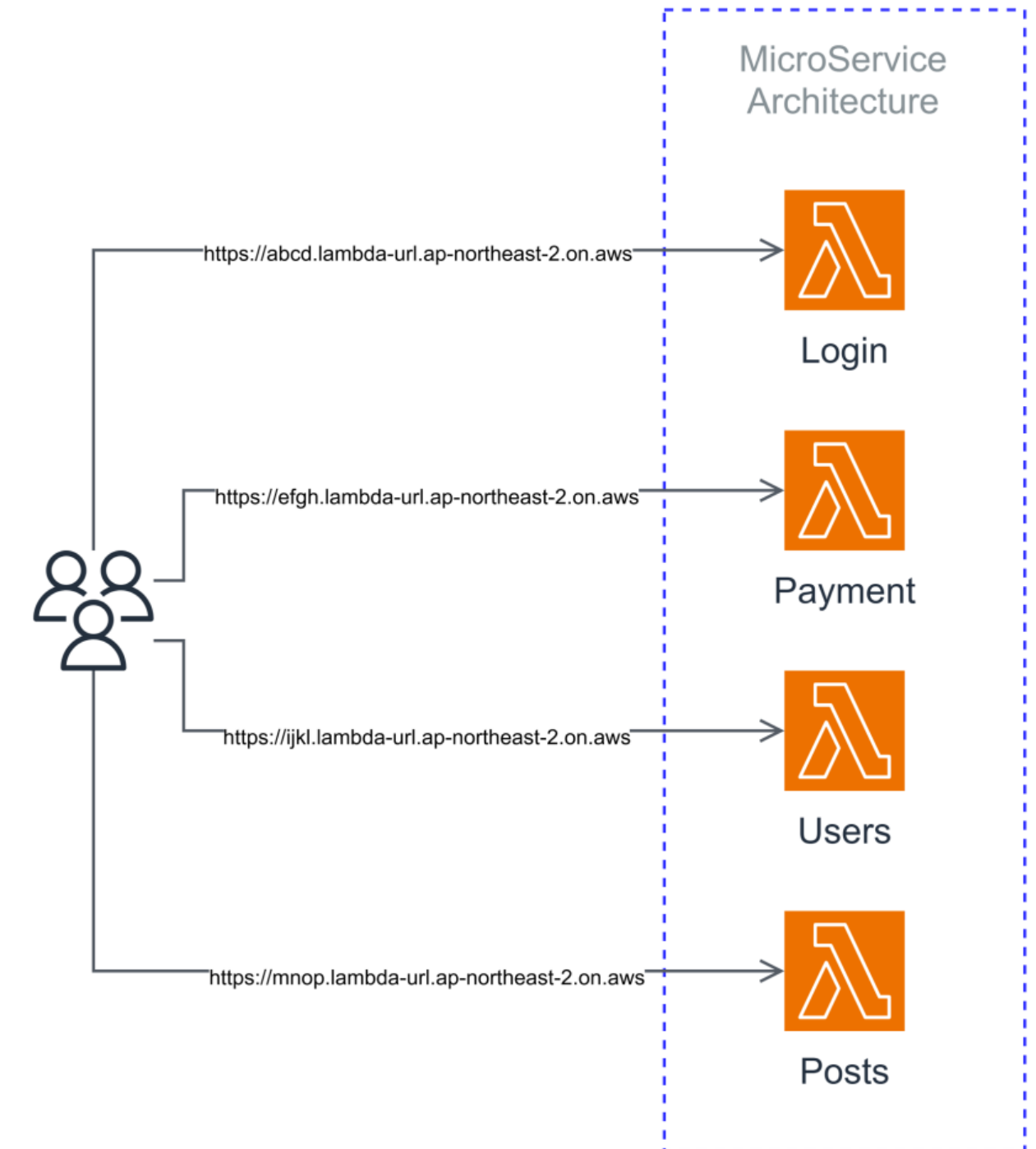
방식	제한	기타
ZIP 파일 직접 업로드	최대 50MB	
ZIP 파일 S3 업로드 및 연동	최대 250MB	
ZIP 압축 해제 이후 코드	최대 250MB	AWS 내부에서 언팩된 코드의 사이즈
도커 컨테이너 이미지	최대 10GB	ECR 업로드 후 람다에서 컨테이너 이미지 사용 가능
레이어 결합	최대 5개	
개별 레이어	압축 후 최대 50MB	레이어 총합 압축 해제 후 최대 250MB 이내

그래서 S3로 업로드하거나 그보다 크면 ECR(Elastic Container Registry)로 도커 컨테이너 사용해야됨

API Gateway

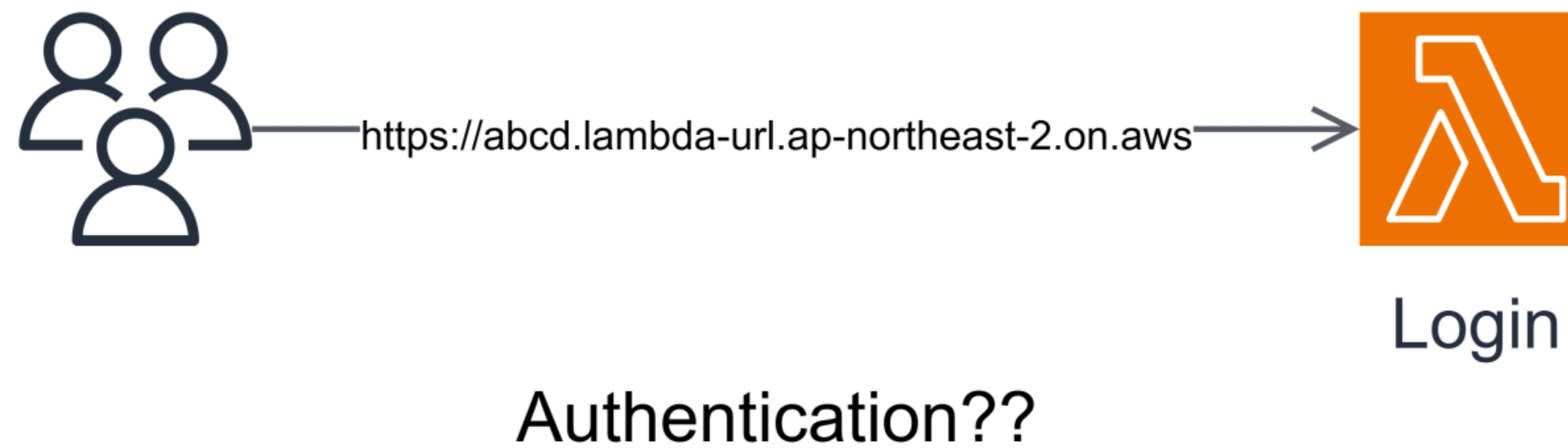
MSA 구조에서 각 마이크로서비스들이 따로 따로 존재함

그래서 이 상태로 함수를 실행하려면 각 함수를 직접 실행해야됨



API Gateway

MSA 구조에서 각 마이크로서비스들이 따로 따로 존재함



그리고 그냥 호출하면 각 함수마다 인증을 처리하는 로직이 필요함
(= 복잡도 증가, 유지보수 어려움)

API Gateway

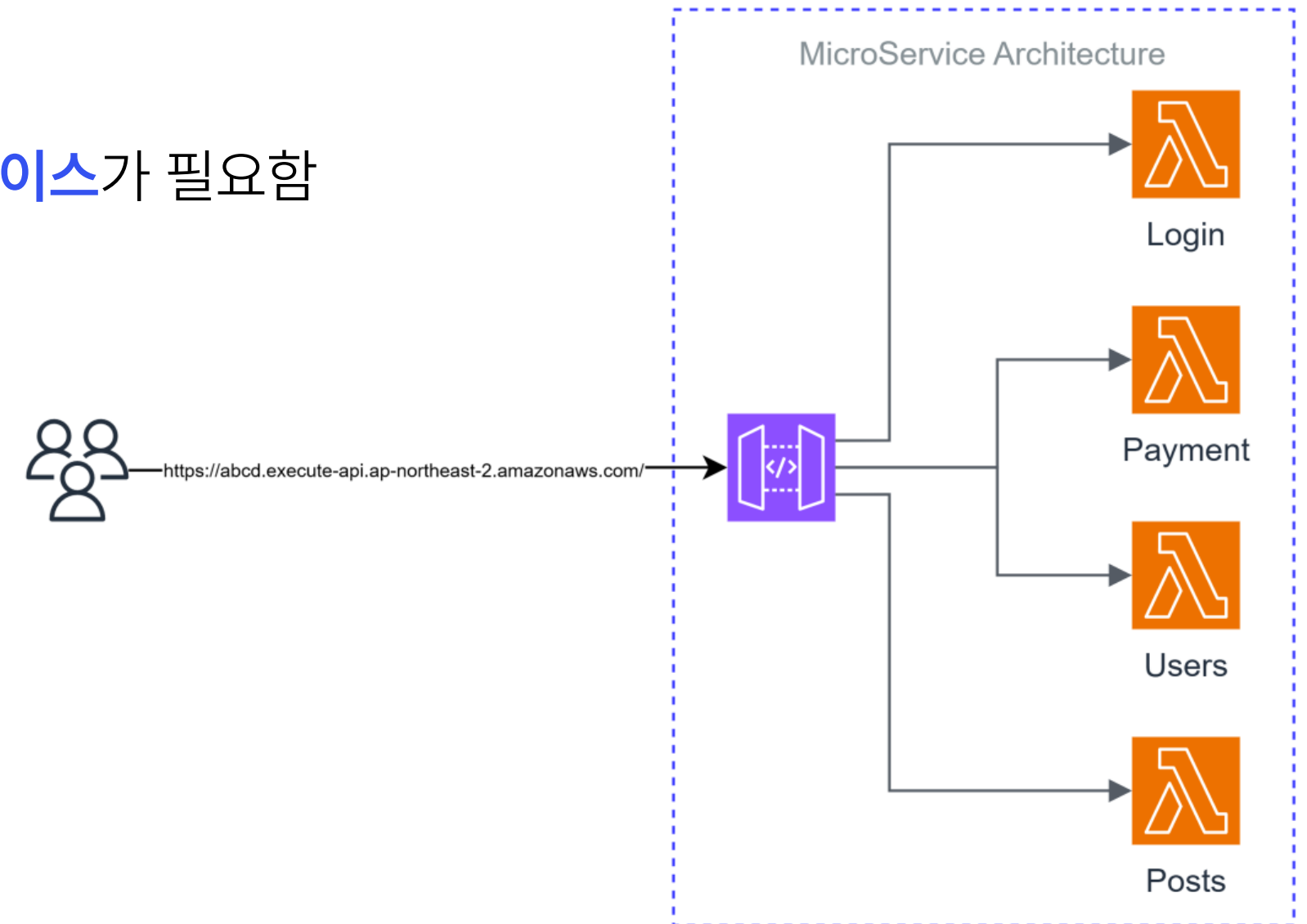
MSA 구조에서 각 마이크로서비스들이 따로 따로 존재함

그래서 각 마이크로서비스(=람다)를 **모아주는(통합해주는) 인터페이스**가 필요함

AWS에선 **API Gateway**로 제공



그리고 **Cognito**라는 인증 서비스(IDaaS)를 사용하여
쉽게 JWT 인증을 연동하고 사용할 수 있음



아쉽게도 CORS에 대한 내용은 분량 상 생략하나, 자세한 내용은 블로그에서 확인 가능함

DynamoDB

AWS에서 제공하는 서버리스 기반의 NoSQL 및 Key-Value 구조의 데이터베이스



1. 서버리스 기반

데이터베이스도 마찬가지로 서버 관리가 필요하지 않은 [서버리스 데이터베이스](#)가 있음

AWS에서 제공하는 서버리스 데이터베이스엔 NoSQL인 DynamoDB와 RDS인 Aurora Serverless가 있음

DynamoDB

AWS에서 제공하는 서버리스 기반의 NoSQL 및 Key-Value 구조의 데이터베이스



2. NoSQL

전통적인 RDBMS는 테이블을 구조화하고 테이블들끼리 상호작용하며 관계를 유지함

NoSQL ⇒ 비관계형 데이터베이스

장점: 스키마(구조) 설정을 안하기 때문에 유연함

단점: 반대로 일관성이 부족함

DynamoDB

AWS에서 제공하는 서버리스 기반의 NoSQL 및 Key-Value 구조의 데이터베이스



3. Key-Value

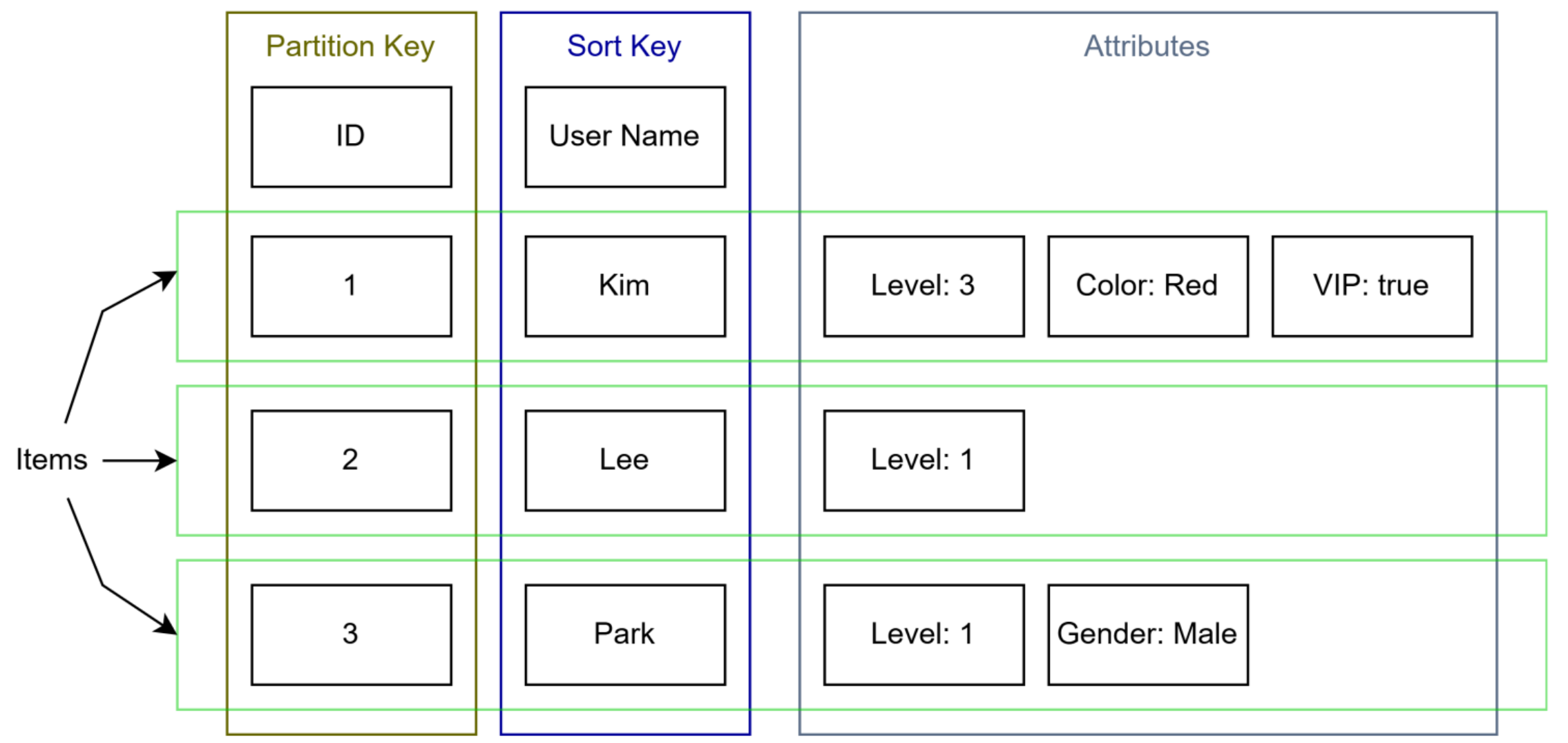
DynamoDB에도 **기본 키**(**파티션 키**)는 존재함, 복합키인 **정렬 키**도 존재함

그래서 기본 키(Key)와 값(Value)으로 존재함

DynamoDB에선 값은 속성(Attributes)이라 함

DynamoDB

그리고 DynamoDB는 스키마(데이터가 구성되는 방식/구조) 정의가 없고 **유연하게** 처리 할 수 있음



DynamoDB

마지막으로 DynamoDB는 **HTTP API**를 이용하여 통신함

전통적인 데이터베이스는 TCP Connection을 마친 후 통신하는데
DynamoDB는 HTTP API로 통신하기 때문에 Connectionless 함

장점: 유연함 (TCP Connection 없이 HTTP API 요청만으로도 가능 ⇒ AWS SDK 사용 시 다 해줌)

단점: HTTP API 호출로 인한 네트워크적 오버헤드나 레이턴시(지연) 발생 가능

Cognito

전통적인 인증 시스템의 경우 직접 구현하는 경우가 많음

모놀리식 아키텍처에선 가능한 상황이었으나, **FaaS 서버리스의 경우 함수마다** 구현을 해줘야함

⇒ 복잡도 증가, 유지 보수의 어려움

또 직접 구현한다면 보안 솔루션에 대해 직접 마련해야 했었는데, 쉽지 않음

Cognito: IDaaS

그래서 이런 경우 대부분 직접 구현하지 않고, **인증과 인가 기능을 다른 클라우드에 분리시킴**

⇒ **IDaaS(Identity as a Service)**, 클라우드 형태로 인증과 인가 기능을 제공하는 것

AWS에선 IDaaS 서비스로 대표적인 **Cognito**가 있음



서드파티 인증을 가능하게 하는 OAuth, OIDC 등의 기술도 지원하나 이 발표에선 인증 기능만 사용함

Cognito: User Pool

Cognito에서 사용자를 관리하면서 사용하는 공간(일종의 데이터베이스)

기본적으로 JWT 인증 방식을 사용함. 그 외에 MFA, 이메일 검증 등의 여러 기능을 제공함

⇒ AWS에서 제공하는 Cognito SDK를 사용하면 코드 몇줄로 인증 기능 구현 가능

인가에서 사용하는 자격 증명 풀에 대한 내용은 생략

⇒ 블로그 참조

Cognito: User Pool

유저 데이터베이스 생성, 관리, JWT 검증,
시크릿 키 관리, 이메일 인증 전부 직접
해야된다::::

종료(과로사)



ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ

ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ

ㅋㅋ

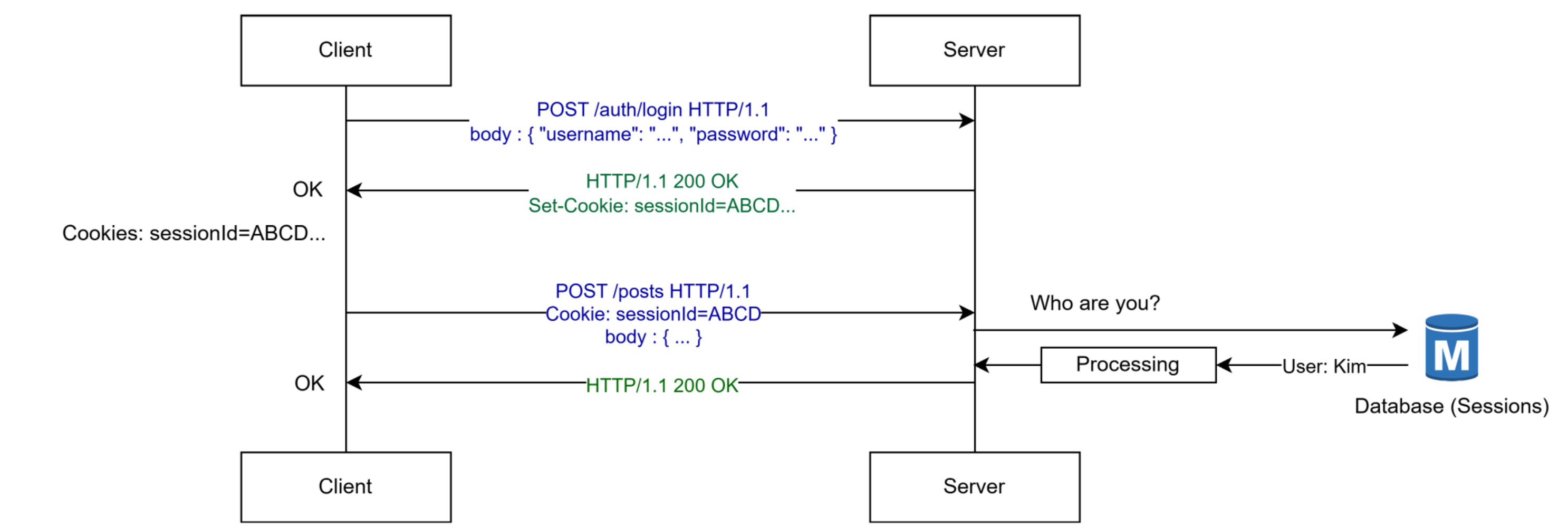
AWS Cognito



달걀

+ JWT(JSON Web Token)

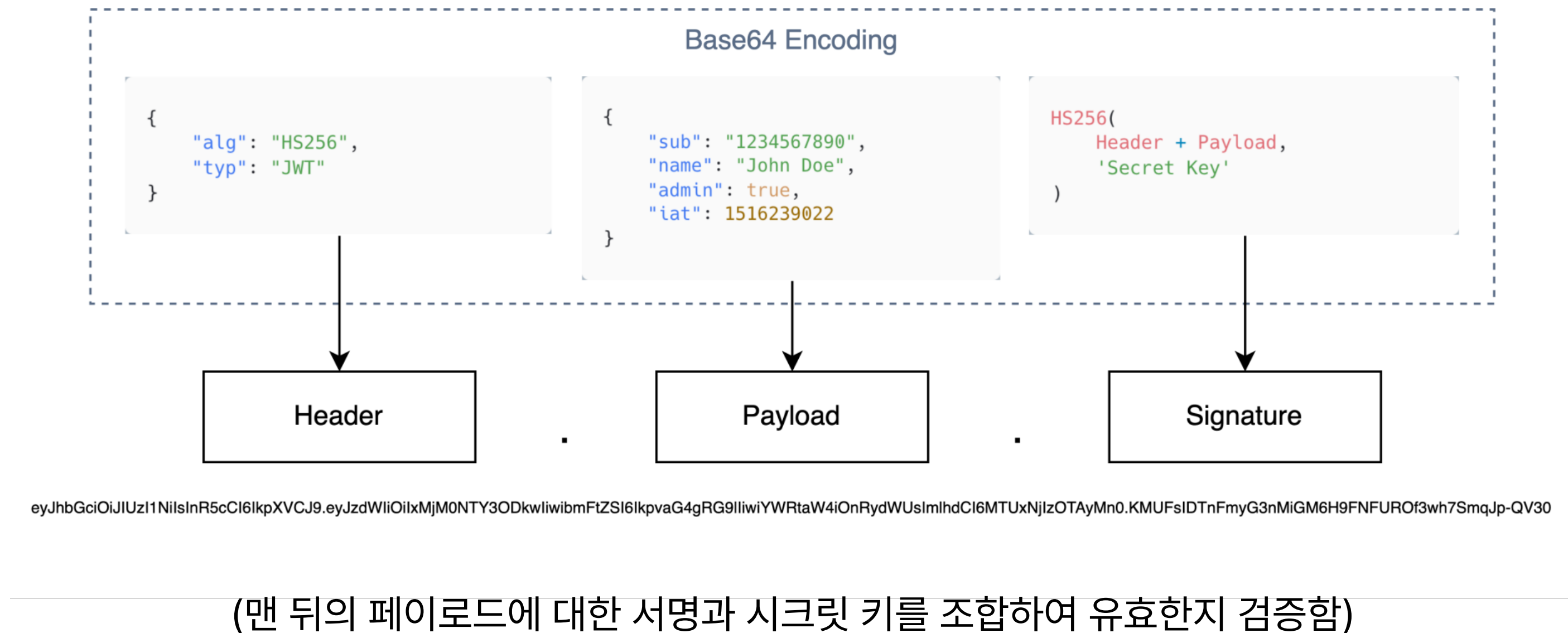
기존엔 **세션 방식**으로 로그인을 유지했었음



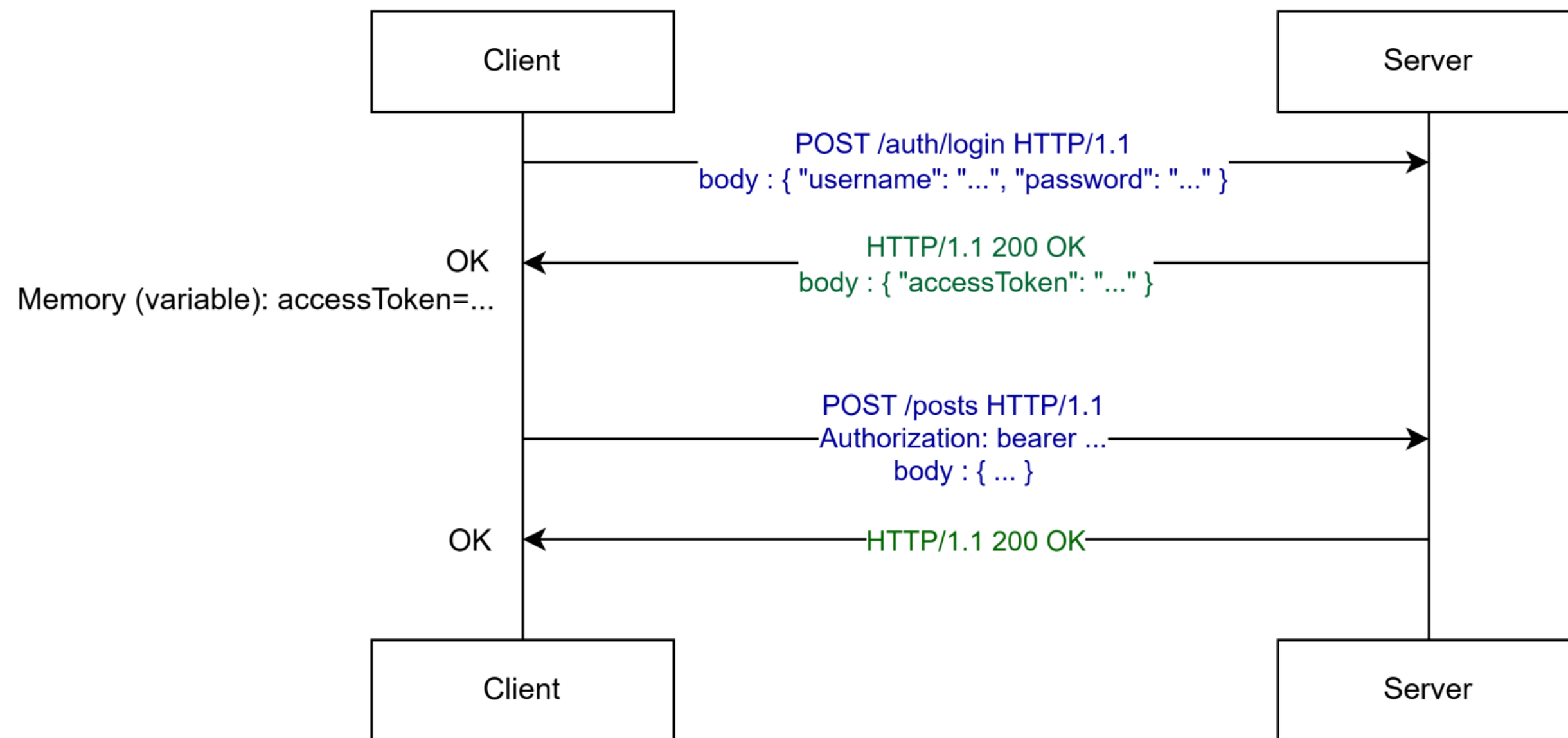
로그인 시 세션 키를 받아오고, 그 세션 키를 서버에 보내 서버에서 확인하는 방식 ⇒ 세션 키 저장을 위한 DB 필요 (오버헤드)

+ JWT(JSON Web Token)

반면 JWT(JSON Web Token)은 그 인증 키(JWT 액세스 토큰) 안에 정보가 어느정도 저장됨 (Claims)



+ JWT(JSON Web Token)



JWT 토큰 안에 기본적인 유저 정보가 있기 때문에 서버에서 세션 키 저장을 위한 데이터베이스 같은건 없음

+ JWT(JSON Web Token)

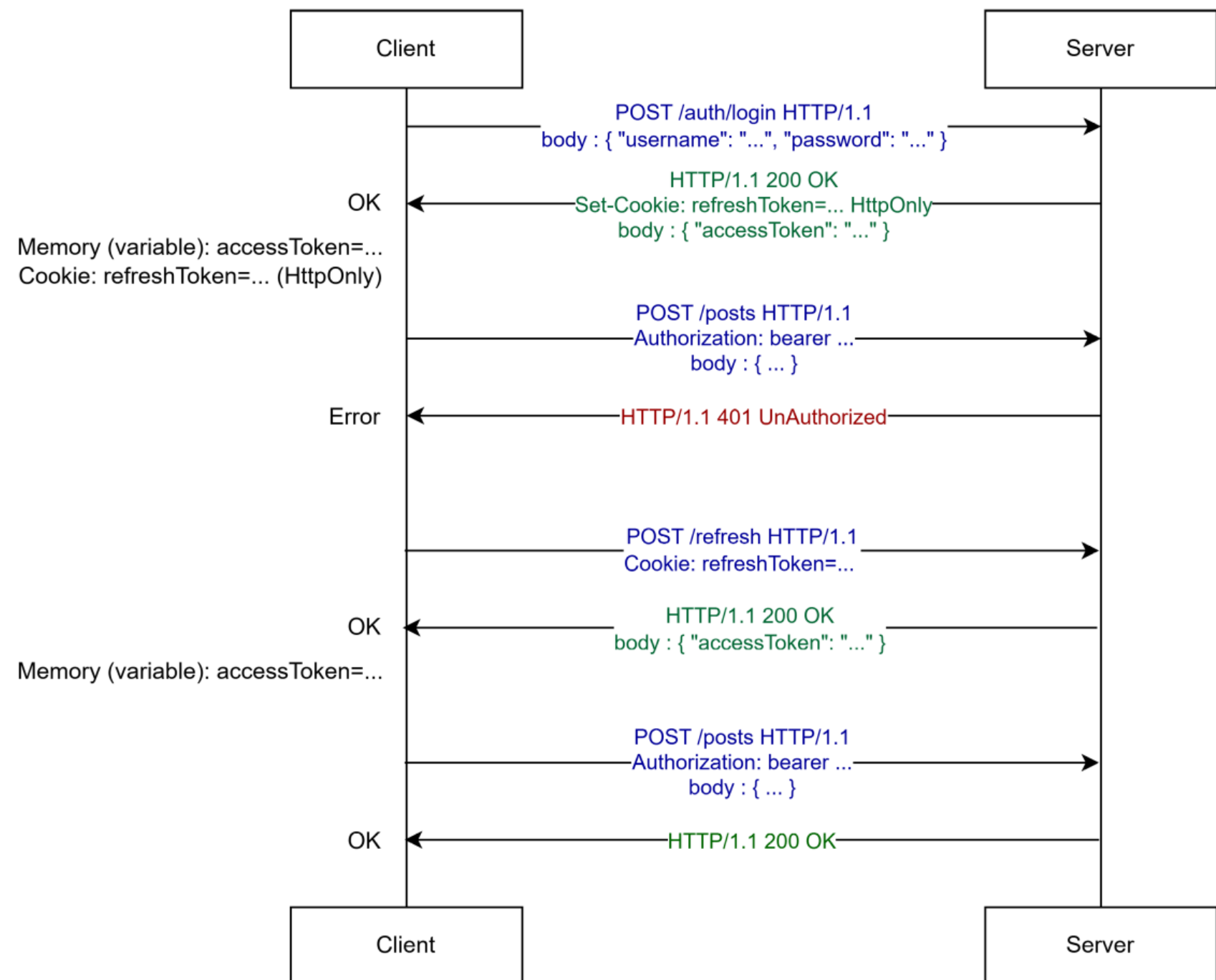
엑세스 토큰은 보안 상 만료 기한을 짧게 설정함

⇒ " 그럼 만료되면 로그아웃되고 다시 로그인을 해야 하나? "

⇒ 그럴 때 **리프레시(Refresh)** 토큰을 사용함, 이걸로 엑세스 토큰 재생성 가능

+ JWT(JSON Web Token)

리프레시 토큰은 보통 HTTPOnly 쿠키에 저장되고
/refresh 등의 엔드포인트 호출 시 자동으로 넘어감



자세한 내용은 블로그 참조 바람

S3 (Frontend)

AWS에서 제공하는 스토리지 서비스



파일 등의 데이터를 저장할 수 있는데 그 공간을 **버킷(Bucket)**이라 함

파일은 디렉토리 구조 없이 **객체(Object)**로 존재

중요한건 S3는 **정적 웹 호스팅**을 지원함

S3 (Frontend)

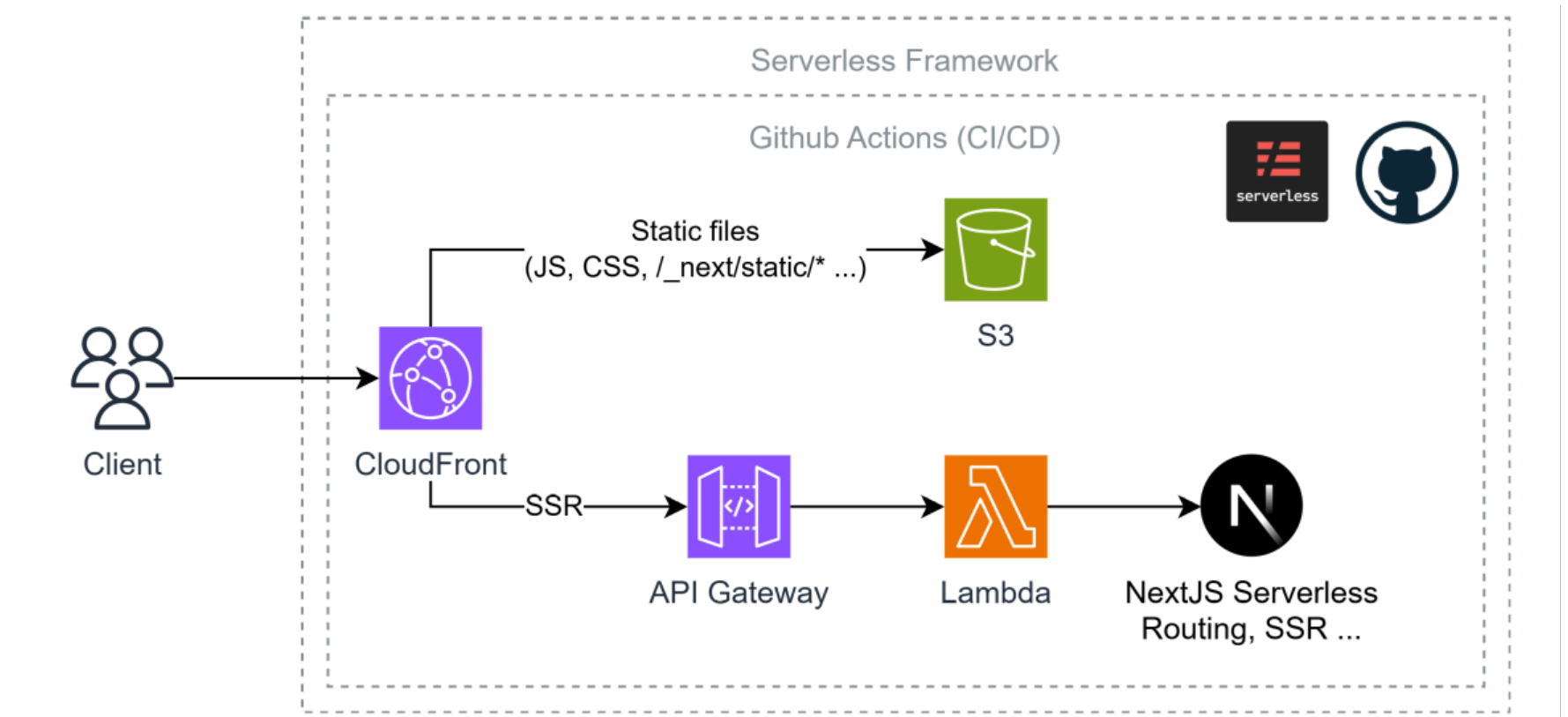
정적 웹 호스팅 = 정적(Static)인 리소스(HTML, CSS, JS ...)를 띄워주는 것

동적 웹 = 사용자의 요청에 따라 보내져오는 내용이 다른 것 (Dynamic)

요즘엔 대부분 동적 웹 + SSR(서버에서 페이지 렌더링) ↔ CSR(클라이언트에서 렌더링) (자료는 해당 아키텍처 예시)

⇒ SEO(검색 엔진 최적화) 등의 장점이 있음

⇒ 복잡해지니 발표에선 정적 웹 + CSR로



S3 (Frontend)

물론 작은 규모에 정적 웹 호스팅"만" 할거면 Github Pages 같은 무료 서비스도 있음



⇒ 발표에선 AWS S3 소개 및 CloudFront 소개를 중심으로 하여 AWS를 선택함

그리고 Github Pages 등의 서비스는 세밀한 조정이 어려움 (단순히 배포 → 호스팅만 있음, 공개 레포라면 유료)

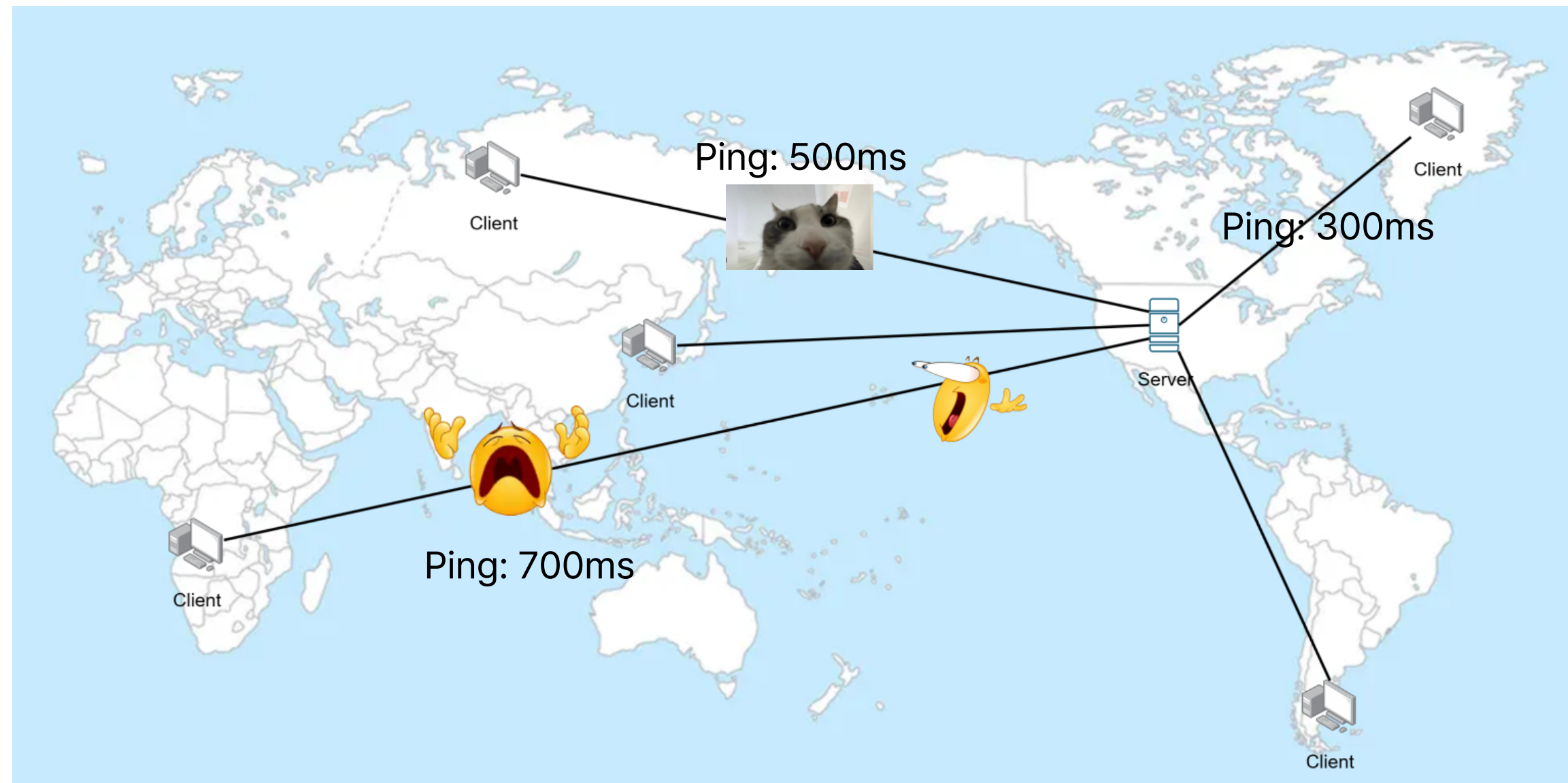
S3 + CloudFront = 나중에 CDN 저장 기능 구현 시 똑같이 쓰이는 서비스임

CloudFront (Frontend)

AWS에서 제공하는 **CDN(Content Delivery Network)** 서비스

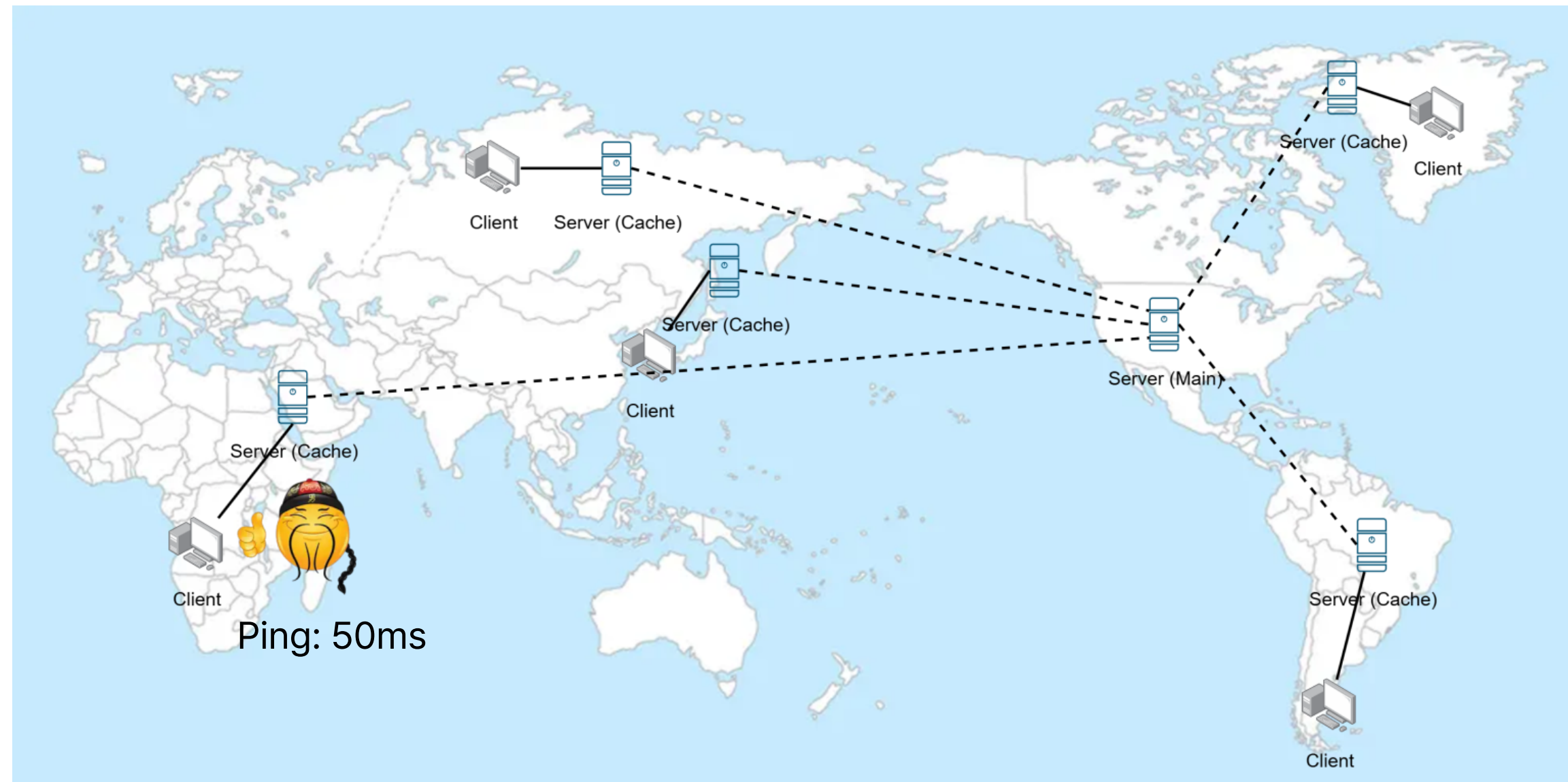


CloudFront (Frontend)



서버는 미국 어딘가 한곳에만 있어서 먼 거리의 클라이언트는 **레이턴시(지연)**가 생김 + 서버의 부하 심해짐

CloudFront (Frontend)



CDN을 사용하면 여러 곳에 캐시 서버를 복제해둌م ⇒ 클라이언트는 가까운 캐시 서버에서 받아옴

CloudFront (Frontend)

AWS에서 제공하는 **CDN(Content Delivery Network)** 서비스

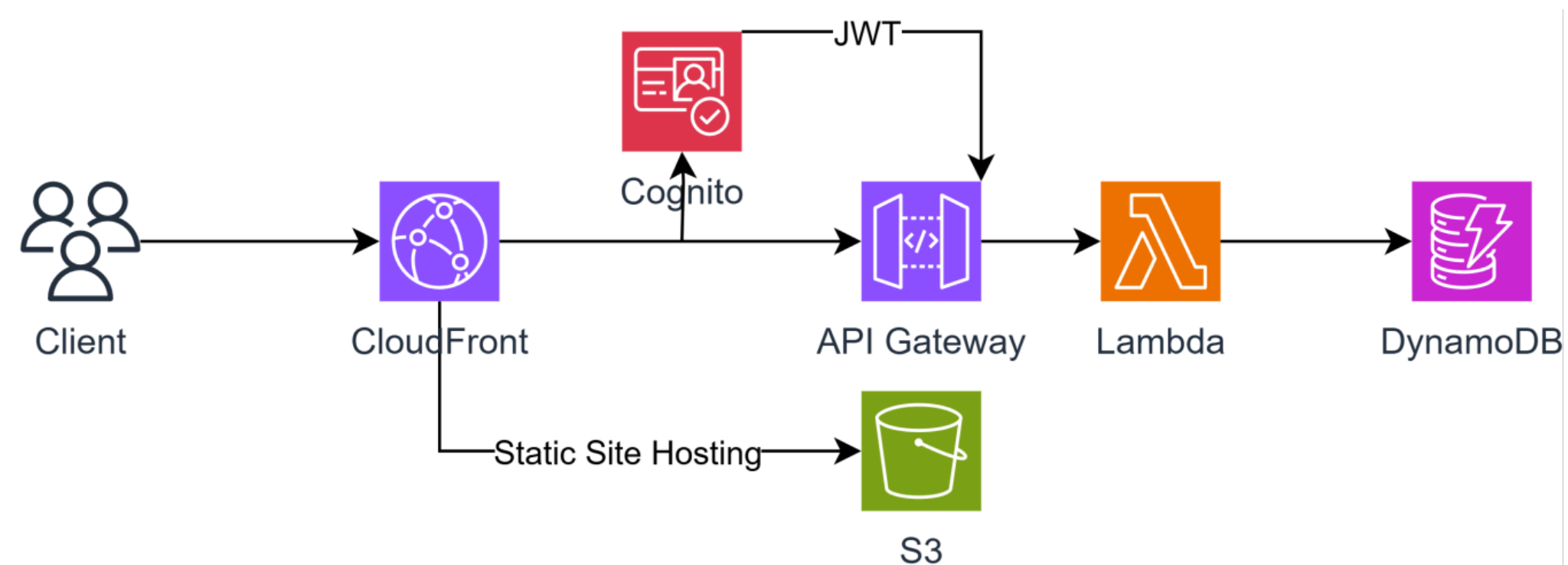


각 캐시 서버를 엣지 로케이션(Edge Location), 배포 시엔 캐싱된걸 무효화할 수 있음 (캐시 무효화)

CloudFront에서 캐싱 할 대상 = 오리진(Origin)

CloudFront (Frontend)

그리고 CDN 서비스로 쓰는 것 말고 여러 서비스(예: API Gateway)를 하나의 CloudFront 오리진으로 통합 가능



⇒ AWS WAF(방화벽) 적용으로 DDoS 등의 공격 → 모니터링, Rate Limit 등의 보안 솔루션 적용 가능

+ CloudFront(CDN)에서 돌아가는 람다 함수인 Lambda@Edge 사용 가능

laC(Infrastructure as Code)

" 이 복잡한 인프라를 직접 하나하나 만들라고? "

" 수정된 부분만 찾아서 직접 그 인프라만 수정하라고? "

" IAM 권한 정책을 직접 만들고 하나하나 적용시키라고? 사용하는 리소스만 100개가 넘는데?? "



laC(Infrastructure as Code)

" 리소스(인프라)를 코드로 작성해서 관리하면 너무 편하겠다.. "

⇒ laC(Infrastructure as Code)

코드를 통해 서비스들을 정의하고 프로비저닝 하는 것

배포도 쉬워지고 인프라 수정 시 수정된 부분만 변경되는 등 매우 편리함

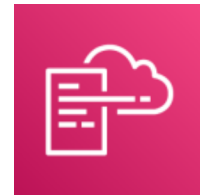


AWS를 사용하지 않는다면 대표적으로 테라폼(Terraform)이 있음



CloudFormation

AWS에서 제공하는 IaC 서비스



테라폼은 자체적인 언어(DSL)인 HCL 언어를 쓰지만 여기선 JSON이나 YAML을 씀 (대부분 후자)

CloudFormation에서 IaC 코드를 **템플릿(Template)**이라 부름

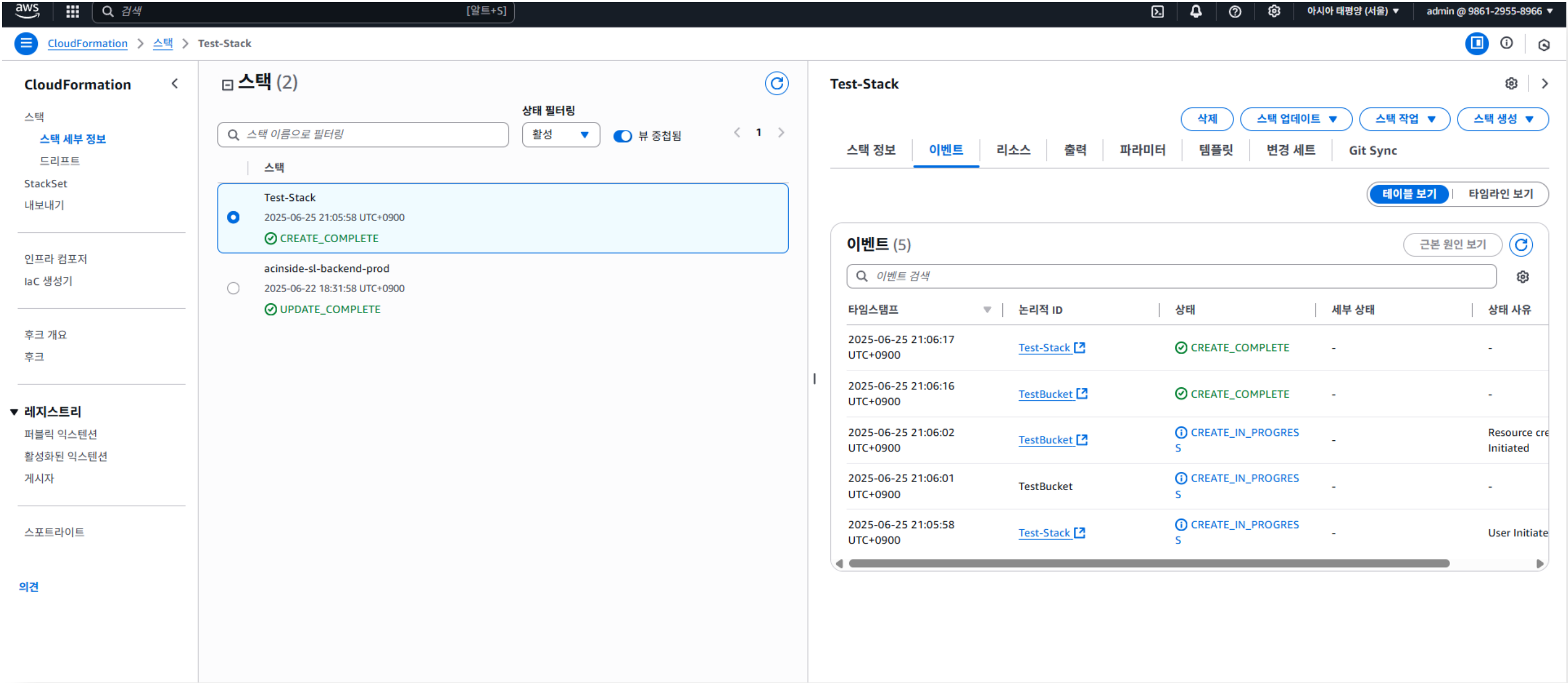
이걸 CloudFormation에서 실행시키면 **스택(Stack)**이 생기고 그 안에서 실제 리소스들이 생성됨

CloudFormation

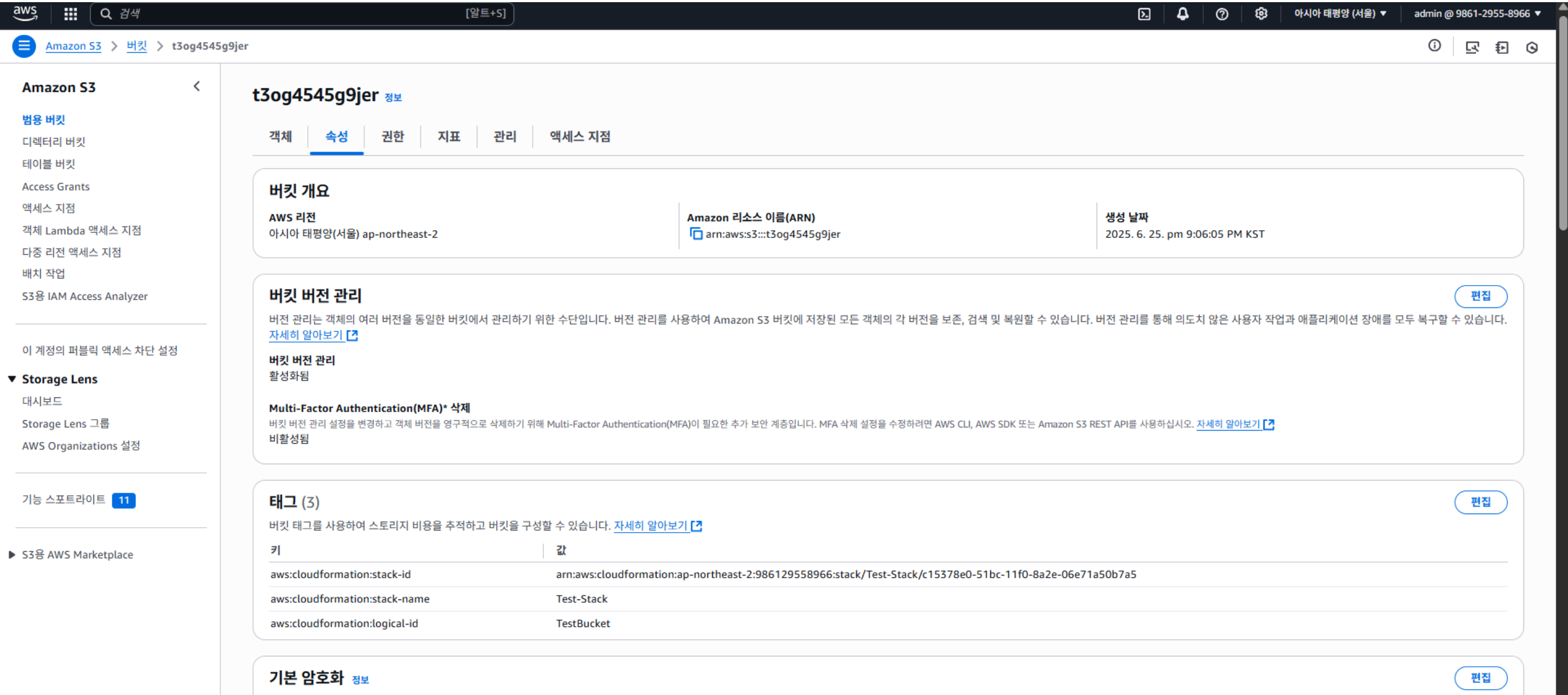
```
AWSTemplateFormatVersion: '2010-09-09'           # 템플릿 버전 (현재까진 2010-09-09가 유일함)
Description: 'Example Cloudformation Template'      # 설명
Resources:                                          # 리소스 목록
  TestBucket:                                       # 논리적인 리소스 이름: TestBucket
    Type: AWS::S3::Bucket                          # 서비스 이름: S3
    Properties:                                     # 속성
      BucketName: test1234                         # 버킷 이름 지정: test1234
      AccessControl: Private                       # 프라이빗으로
      VersioningConfiguration:                    # 버전 관리 사용
        Status: Enabled
```

AWS S3 버킷을 생성하는 CloudFormation 템플릿 예시
Resources 속성 안에 서비스를 정의하고 리소스 생성 가능

CloudFormation



CloudFormation



SAM(Serverless Application Model)

CloudFormation 템플릿 작성하고 AWS 콘솔 들어가서 직접 업로드해서 써야하나요? 좀 짜치는데..

⇒ AWS에서 제공하는 SAM(Serverless Application Model) 등의 "프레임워크" 사용 가능

AWS 공식 지원 및 서버리스(람다 함수 중심) 배포에서 주로 사용함

CloudFormation 기반의 IaC 템플릿 작성 가능

SAM(Serverless Application Model)

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs20.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /hello
            Method: get
```

람다 함수 자동으로 배포해줌 + API Gateway까지 자동으로 연동해줌 (로컬 테스트도 가능)

그런데 SAM보단 플러그인도 지원하고 생태계도 더 활성화된 Serverless Framework 사용을 추천

Serverless Framework

AWS SAM과 비슷하게 AWS 서버리스 배포에 특화된 프레임워크

⇒ 애도 결국엔 CloudFormation 기반이라 템플릿 구조 자체는 비슷함



AWS에 최적화(사실상 AWS에서만 쓰임) + 다양한 플러그인 사용 가능 + 생태계 활발

템플릿도 SAM보다 간결하고 가독성 높음

Serverless Framework

```
service: HelloWorld
provider:
  name: aws
  runtime: nodejs20.x
functions:
  hello:
    handler: handler.hello
    events:
      - http:
          path: hello
          method: get
```

provider 속성 = 람다에 기본으로 적용할 IAM이나 서비스(API Gateway, Cognito 등) 정의

functions 속성 = 람다 리소스 정의

Serverless Framework

```
> npm install -g serverless-esbuild
```

```
plugins:
  - serverless-esbuild      # 플러그인 추가

custom:
  esbuild:
    bundle: true            # 번들링
    minify: true            # 코드 간소화
    target: 'node20'        # NodeJS 20버전 사용 (람다에서 지원하는 최신 버전)
    platform: 'node'
    treeShaking: true       # 사용하지 않는 코드 삭제
    packager: npm           # npm 패키지 매니저 사용
    format: 'cjs'           # CommonJS 사용
```

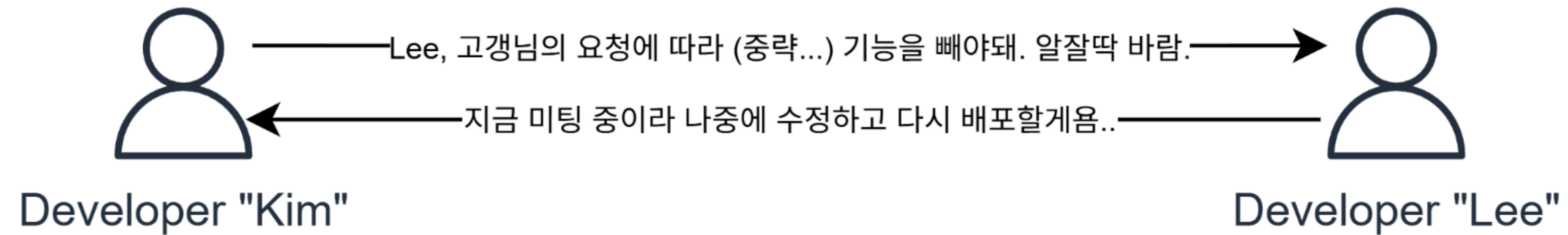
JS 코드 번들링(코드 하나로 묶는 것) 등을 제공하는 플러그인(esbuild)의 예제, 자세한건 추후 설명

Github Actions CI/CD

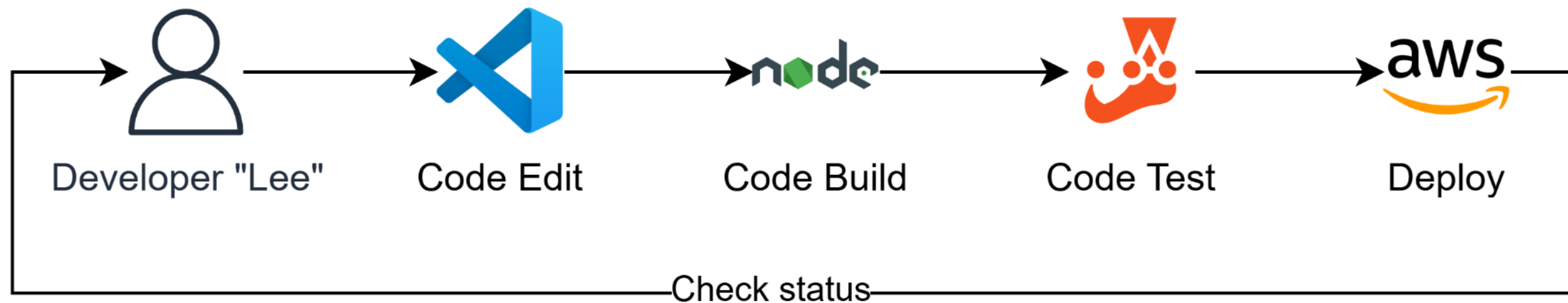
CI/CD = Continuous Integration/Continuous Deployment = 지속적 통합 / 지속적 배포

⇒ 쉽게 말해 개발, 테스트, 배포 등 배포까지의 과정(파이프라인)을 **자동화** 시키는 것

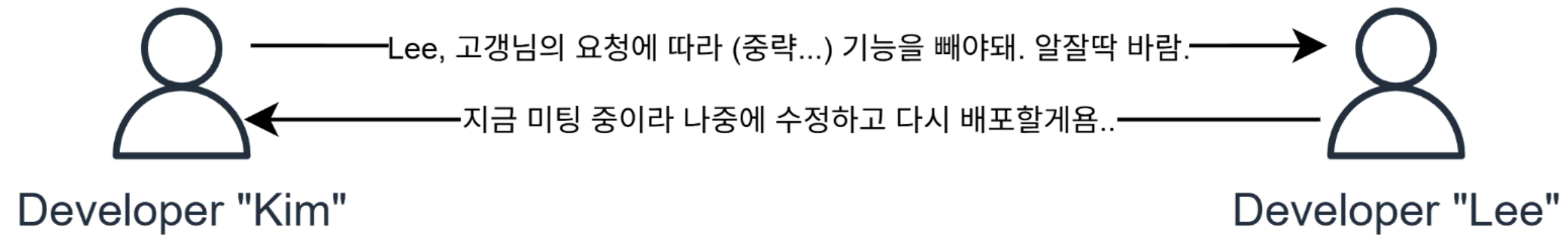
Github Actions CI/CD



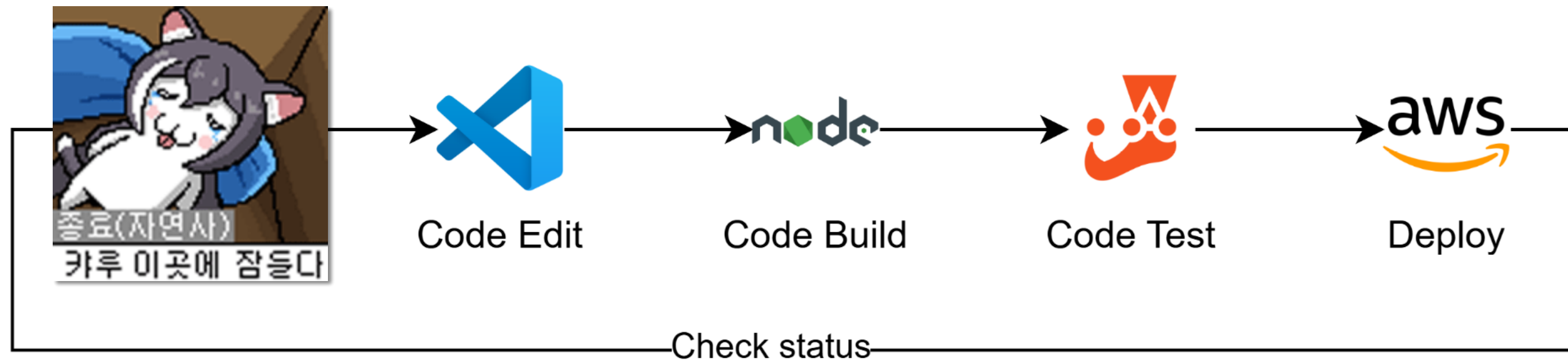
After the meeting ...



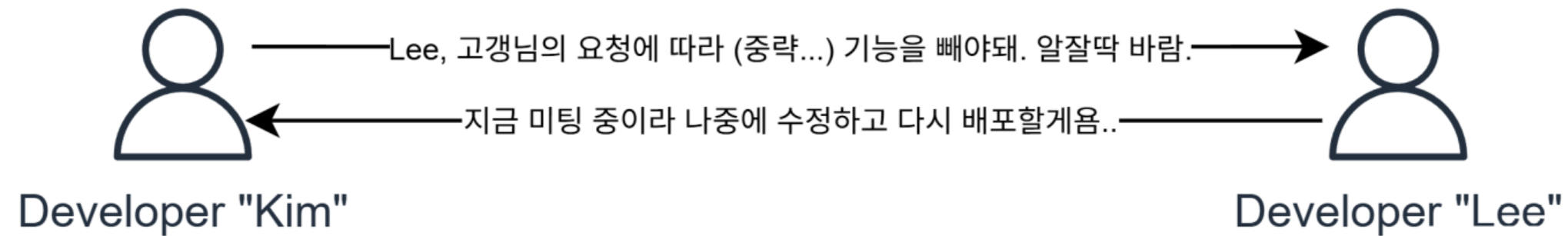
Github Actions CI/CD



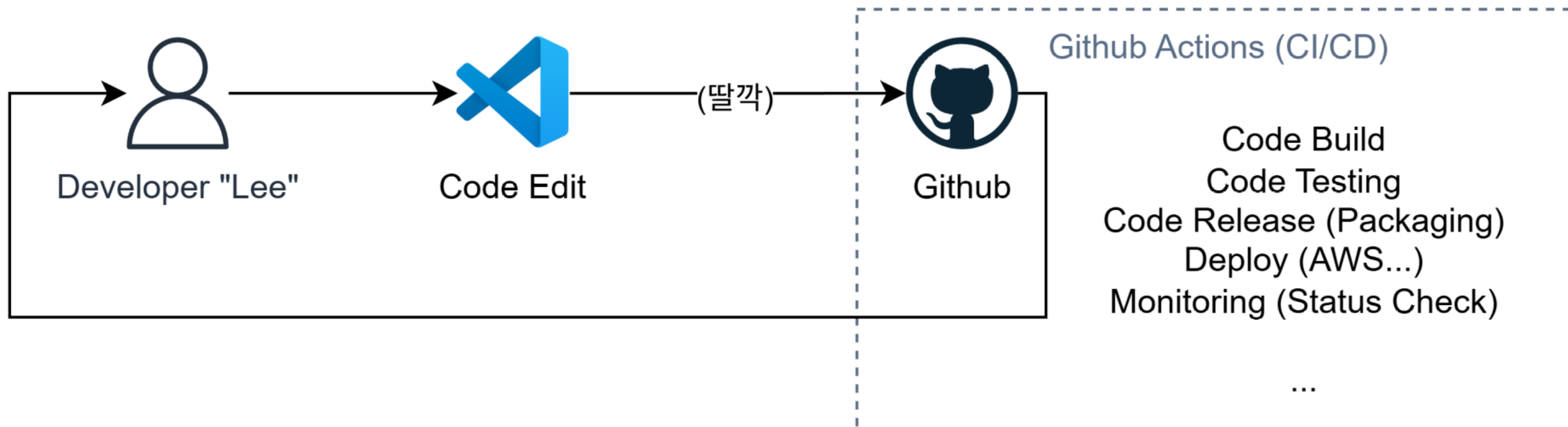
After the meeting ...



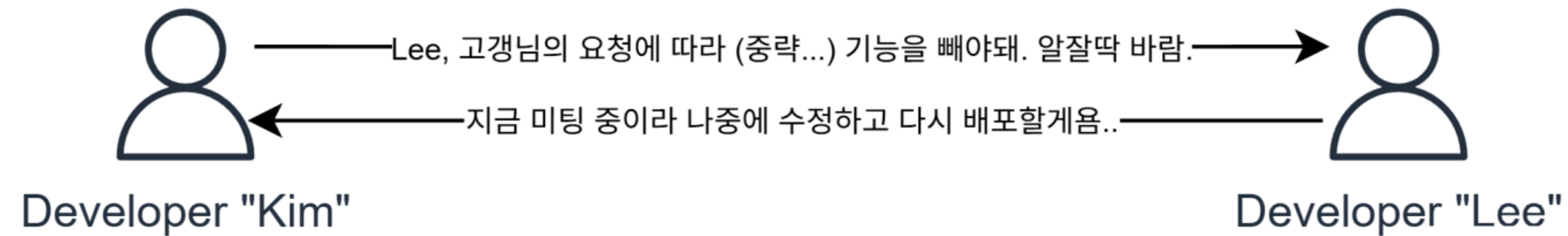
Github Actions CI/CD



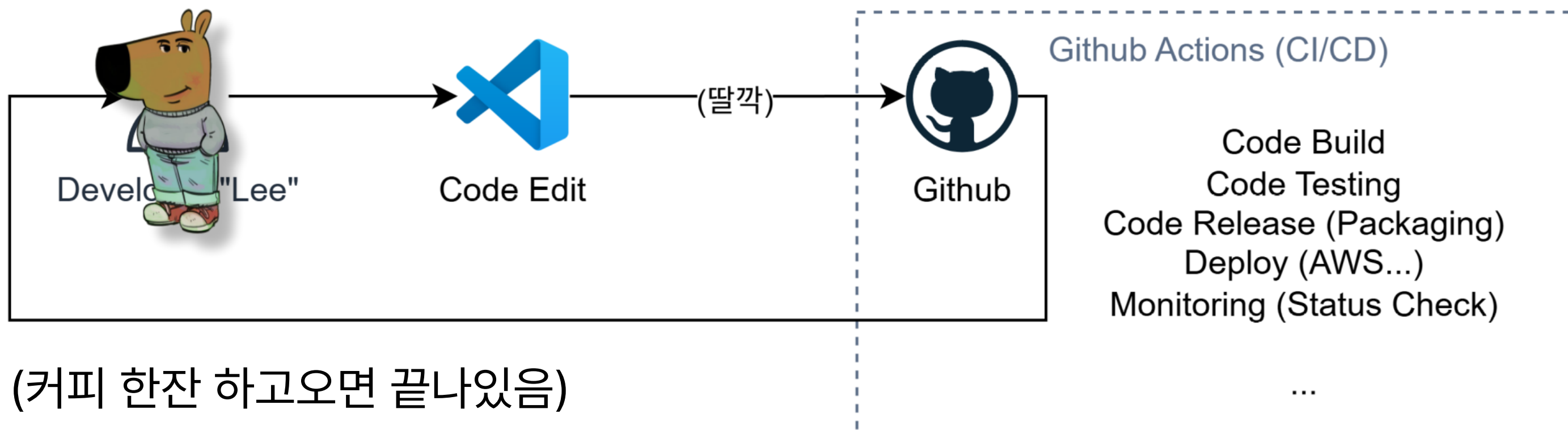
After the meeting ...



Github Actions CI/CD



After the meeting ...



Github Actions CI/CD



CI/CD 플랫폼으로 Jenkins나 Travis CI 등이 있지만 Github에서 제공하는 **Github Actions**도 대표적
(깃허브 = 코드 저장소, 개발자들의 픽시브)

Github 레포지토리에서 특정 작업이 발생 시 (이벤트) 작업들이 실행됨 = **워크플로우(Workflow)**

워크플로우 안에선 **잡(Job)**, 그 안에선 실제 작업인 **스텝(Step)**으로 이루어짐

워크플로우가 실행되는 환경을 Runner, Self Hosting 사용도 가능함 (개발자 컴퓨터에서 돌리는 것)

워크플로우 예) 레포지토리 prod 브랜치에 push되면 테스트 후 AWS에 배포해라

Github Actions CI/CD

```
name: Github Pages Deploy

on:
  push:
    branches:
      - main

permissions:
  contents: write

jobs:
  publish:
    runs-on: ubuntu-latest

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

      - name: Install dependencies
        run: npm install

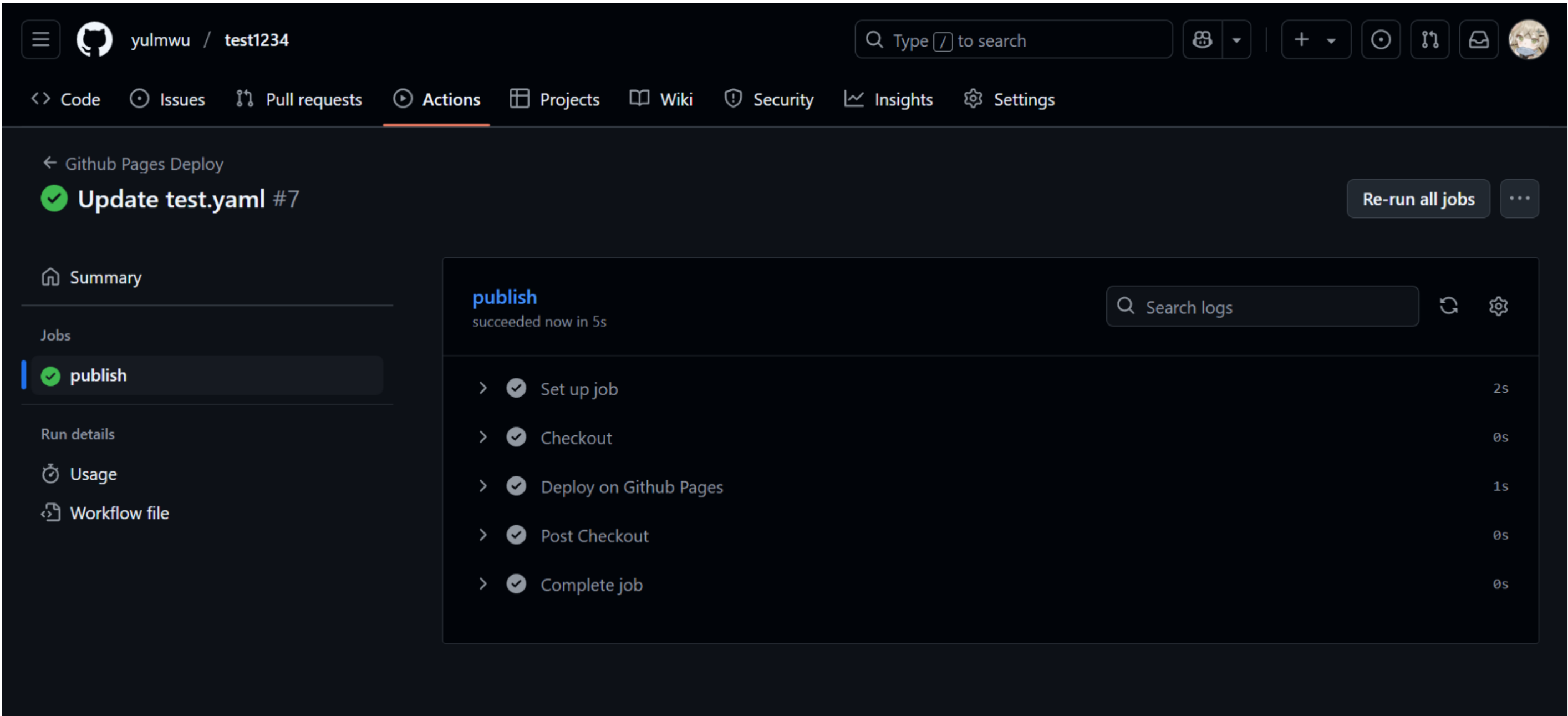
      - name: Build
        run: npm run build

      - name: Deploy on Github Pages
        uses: JamesIves/github-pages-deploy-action@v4
        with:
          folder: build
```

워크플로우도 YAML로 작성됨

왼쪽 워크플로우는 Github Pages 자동 배포 예시

Github Actions CI/CD



Let's write the Code

작성할 API 코드를 카테고리로 나눠보면 인증 기능(/auth), 포스트 CRUD 기능(/posts), 유저 기능(/users)가 있음.

모든걸 설명하기엔 무리가 있으니 중요한 개념만 등장하는 코드 일부만 소개할 예정 (깃허브엔 있지만 분량 상 예외 처리 안함)

- /auth/login.ts
- /post/createPost.ts
- /user/myinfo.ts



모든 코드는 깃허브 레포지토리오에서 확인 가능
<https://github.com/yulmwu/aws-lambda-example>

Tech Stacks



NodeJS



TypeScript



**AWS CDK
V3 (SDK)**

AWS SDK



**AWS CDK
V3 (SDK)**

AWS에선 **SDK(Software Development Kit)**을 대부분의 언어에서 지원함

특히 NodeJS에서 지원하는 AWS SDK엔 레거시 버전 **V2**와 신버전 **V3**가 있음

AWS SDK



AWS CDK
V3 (SDK)

```
import AWS from 'aws-sdk'
const dynamoDB = new AWS.DynamoDB()

dynamoDB.getItem({
  TableName: 'Users',
  Key: {
    userId: { S: '123' },
  },
}, (err, data) => {
  // Do Someting
})
```

V2: aws-sdk 라이브러리 내에 모든 서비스가 들어가있음

AWS SDK



AWS CDK
V3 (SDK)

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb'
import { DynamoDBDocumentClient, GetCommand } from '@aws-sdk/lib-dynamodb'

const dynamoDB = DynamoDBDocumentClient.from(new DynamoDBClient({}))

const command = new GetCommand({
  TableName: 'Users',
  Key: { '123' },
})

const result = await dynamoDB.send(command)

// Do Something
```

V3: 서비스들 단위로 모듈화가 되어있음, 기능을 명령어로 분류되어 있음

⇒ 라이브러리 코드 용량 줄어듦(번들링, Tree Shaking 등) + 코드 가독성 증가

Explain Lambda Function Code

```
"dependencies": {
  "@aws-sdk/client-cognito-identity-provider": "^3.830.0",
  "@aws-sdk/client-dynamodb": "^3.830.0",
  "@aws-sdk/lib-dynamodb": "^3.830.0"
},
"devDependencies": {
  "@types/aws-lambda": "^8.10.150",
  "@types/node": "^24.0.3"
}
```

프로젝트에서 사용한 NodeJS 의존성, TypeScript의 타입만 제공해주는 라이브러리는 dev 디펜더시로

Explain Lambda Function Code

```
export const handler = async (event: APIGatewayProxyEventV2): Promise<APIGatewayProxyResultV2> => {  
    // ...  
}
```

람다 함수의 선언 코드, ESMODULE을 사용하기 때문에 export 키워드로 내보냄

이벤트로는 API Gateway의 프록시 구조로 받음, 비동기 함수이기 때문에 반환 값도 Promise로

Explain Lambda Function Code - Login

로그인을 담당하는 람다 함수, Cognito SDK를 다루는 예제임

```
import { CognitoIdentityProviderClient, InitiateAuthCommand } from '@aws-sdk/client-cognito-identity-provider'
import { APIGatewayProxyEventV2, APIGatewayProxyResultV2 } from 'aws-lambda'

const cognitoClient = new CognitoIdentityProviderClient({})
```

유저 풀을 사용하기 위한 Cognito 클라이언트 선언 (초기화 코드)

Explain Lambda Function Code - Login

```
const { username, password } = JSON.parse(event.body ?? '{}')
if (!username || !password)
  return {
    statusCode: 400,
    body: JSON.stringify({
      error: 'Bad Request',
      message: 'Username and password are required',
    }),
  }
```

요청에 유저 이름과 비밀번호 속성이 있는지 확인하는 기초적인 에러 핸들링

Explain Lambda Function Code - Login

```
const command = new InitiateAuthCommand({
  AuthFlow: 'USER_PASSWORD_AUTH',
  ClientId: process.env.COGNITO_CLIENT_ID!,
  AuthParameters: {
    USERNAME: username,
    PASSWORD: password,
  },
})

const result = await cognitoClient.send(command)

const accessToken = result.AuthenticationResult?.AccessToken
const idToken = result.AuthenticationResult?.IdToken
const refreshToken = result.AuthenticationResult?.RefreshToken
```

Cognito 클라이언트에 보낼 로그인 명령어, 이후 성공 시 액세스 토큰, ID 토큰, 리프레시 토큰을 받아옴

Explain Lambda Function Code - Login

```
const maxAge = 30 * 24 * 60 * 60 // 30 days

return {
  statusCode: 200,
  headers: {
    'Set-Cookie': `refreshToken=${refreshToken}; HttpOnly; Path=/; Max-Age=${maxAge}; SameSite=None; Secure`,
  },
  body: JSON.stringify({
    accessToken,
    idToken,
  }),
}
```

받은 JWT 토큰을 API를 요청한 클라이언트에게 200 OK와 함께 반환함

이때 리프레시 토큰은 HTTPOnly 쿠키에 담아서 보냄

Explain Lambda Function Code - Login

```
const maxAge = 30 * 24 * 60 * 60 // 30 days

return {
  statusCode: 200,
  headers: {
    'Set-Cookie': `refreshToken=${refreshToken}; HttpOnly; Path=/; Max-Age=${maxAge}; SameSite=None; Secure`,
  },
  body: JSON.stringify({
    accessToken,
    idToken,
  }),
}
```

받아온 JWT 토큰을 API를 요청한 클라이언트에게 200 OK와 함께 반환함

이때 리프레시 토큰은 HTTPOnly 쿠키에 담아서 보냄

나머지 인증 관련 기능은 Cognito SDK 명령어만 바뀌어서 구현하기 때문에 설명은 생략

Explain Lambda Function Code - Create Post

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb'
import { DynamoDBDocumentClient, UpdateCommand, PutCommand } from '@aws-sdk/lib-dynamodb'
import { APIGatewayProxyEventV2WithJWTAuthorizer, APIGatewayProxyResultV2 } from 'aws-lambda'

const dynamoDB = DynamoDBDocumentClient.from(new DynamoDBClient({}))
```

DynamoDB 클라이언트를 사용하는데 DynamoDBDocumentClient를 사용함

⇒ 기존 DynamoDBClient보다 사용하기도 쉽고 High Level임 (AWS 문서 왓)

Explain Lambda Function Code - Create Post

```
const getNextId = async (): Promise<number> => {  
  const command = new UpdateCommand({  
    TableName: 'Counter',  
    Key: { name: 'post' },  
    UpdateExpression: 'SET #v = if_not_exists(#v, :init) + :inc',  
    ExpressionAttributeNames: { '#v': 'value' },  
    ExpressionAttributeValues: {  
      ':inc': 1,  
      ':init': 0,  
    },  
    ReturnValues: 'UPDATED_NEW',  
  })  
  
  const result = await dynamoDB.send(command)  
  return result.Attributes?.value  
}
```

게시글 ID에서 게시글을 순차적으로 카운팅하기 위한 로직, 별도의 Counter 테이블을 만듦

MySQL 같은 RDBMS의 Auto Increment 같은거 쓰면 안되냐? ⇒ DynamoDB에선 지원 안함

Explain Lambda Function Code - Create Post

```
const { title, content } = JSON.parse(event.body ?? '{}')
if (!title || !content)
  return {
    statusCode: 400,
    body: JSON.stringify({
      error: 'Bad Request',
      message: 'Title and content are required',
    }),
  }

const user = event.requestContext.authorizer?.jwt?.claims
if (!user || !user.sub || !user.username)
  return {
    statusCode: 401,
    body: JSON.stringify({
      error: 'Unauthorized',
      message: 'User is not authenticated',
    }),
  }
```

기초적인 에러 핸들링: JWT 토큰 확인(검증은 Cognito에서 해줌), 요청 바디 확인

Explain Lambda Function Code - Create Post

```
const item = {
  id: String(await getNextId()),
  title,
  content,
  userId: user.sub,
  userName: user.username,
  createdAt: new Date().toISOString(),
}

const command = new PutCommand({
  TableName: 'Posts',
  Item: item,
})

await dynamoDB.send(command)
```

Posts 테이블에 아이템(게시글)을 작성하는 로직

Explain Lambda Function Code - My Info (me)

헤더의 JWT 토큰(Authorization)을 바탕으로 현재 유저의 정보를 가져오는 함수

```
const authHeader = event.headers?.Authorization ?? event.headers?.authorization

if (!authHeader || !authHeader.startsWith('Bearer '))
  return {
    statusCode: 401,
    body: JSON.stringify({
      error: 'Unauthorized',
      message: 'No valid authorization header provided',
    }),
  }
```

Authorization 헤더가 포함되어 있는지 확인하는 기초적인 에러 핸들링

Explain Lambda Function Code - My Info (me)

헤더의 JWT 토큰(Authorization)을 바탕으로 현재 유저의 정보를 가져오는 함수

```
const authHeader = event.headers?.Authorization ?? event.headers?.authorization

if (!authHeader || !authHeader.startsWith('Bearer '))
  return {
    statusCode: 401,
    body: JSON.stringify({
      error: 'Unauthorized',
      message: 'No valid authorization header provided',
    }),
  }
```

Authorization 헤더가 포함되어 있는지 확인하는 기초적인 에러 핸들링

Explain Lambda Function Code - My Info (/me)

```
const accessToken = authHeader.substring('Bearer '.length)

const getUserCommand = new GetUserCommand({ AccessToken: accessToken })
const result = await cognitoClient.send(getUserCommand)

const userAttributes = Object.fromEntries(
  (result.UserAttributes ?? []).map((attr) => [attr.Name, attr.Value])
)

return {
  statusCode: 200,
  body: JSON.stringify({
    username: result.Username,
    ...userAttributes,
  }),
}
```

엑세스 토큰을 추출해서 그걸 바탕으로 유저의 정보를 가져옴(이메일, 이메일 인증 여부 등)

Let's build the Infra

1. ~~Lambda 코드 배포하기 (SF)~~
2. ~~API Gateway 연동하기 (SF)~~
3. ~~Cognito 설정하기 (SF)~~
4. ~~DynamoDB 생성하기 (SF)~~
5. Serverless Framework로 배포 방식 바꾸기
6. Github Actions를 사용한 자동화 배포 워크플로우 작성
7. CloudFront + S3로 프론트엔드 배포하기

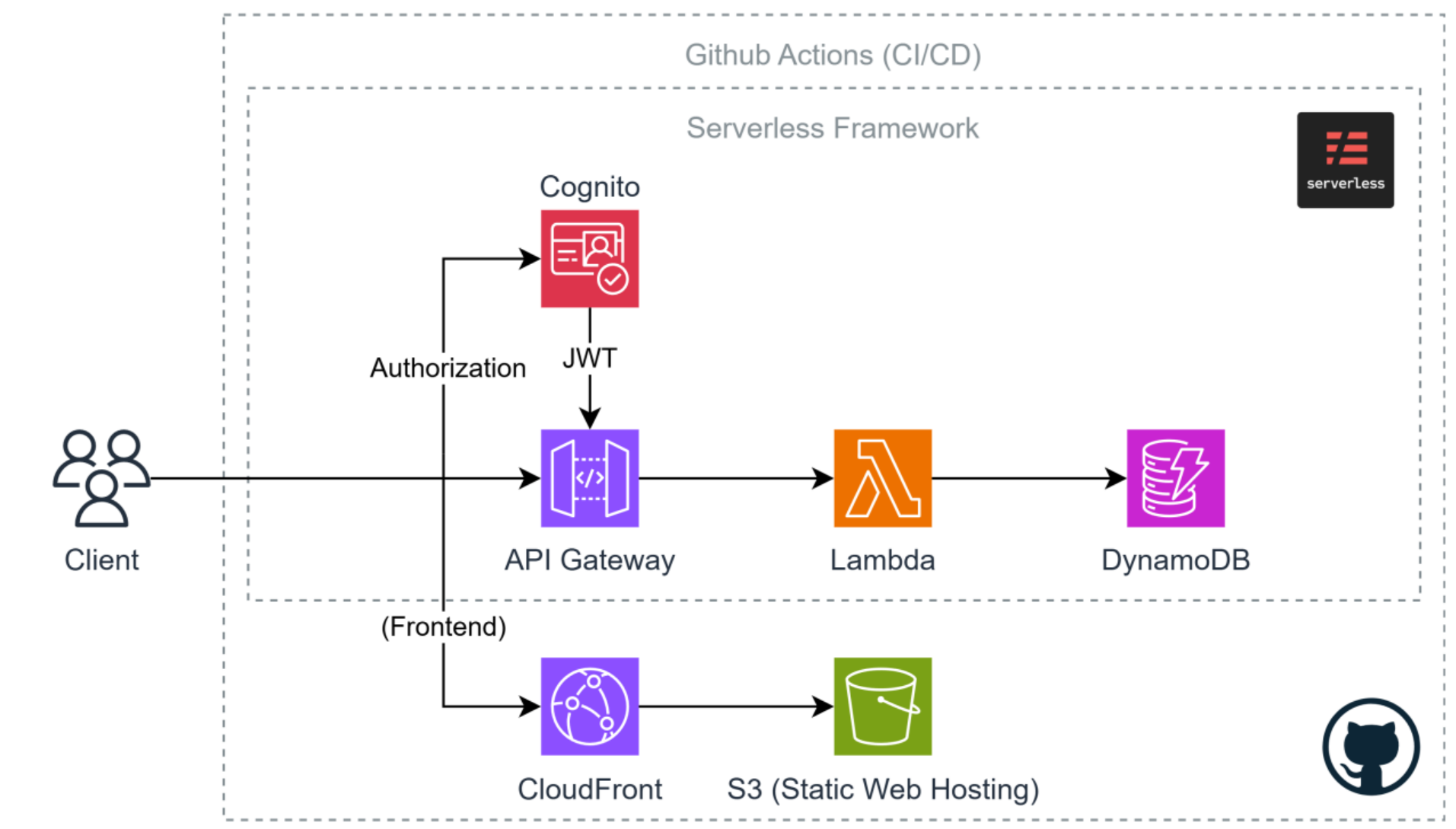
(SF) = Serverless Framework로 구축할 예정

분량 상 생략하지만 세부적인 인프라 구축 과정은 앞선 블로그에서 확인할 수 있음



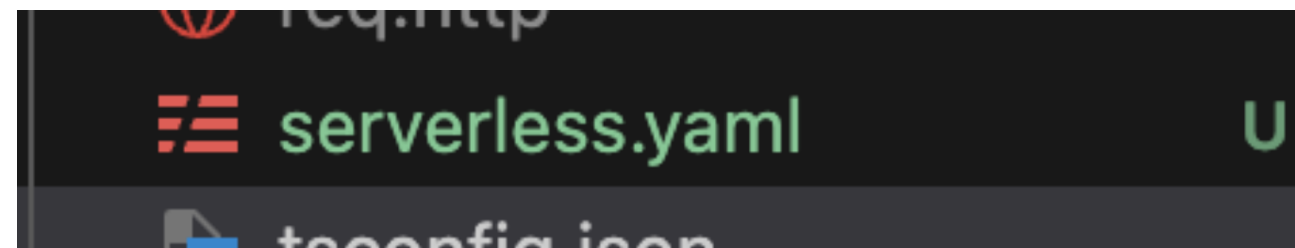
블로그 QR코드

with Serverless Framework



여기서 Serverless Framework 제목의 그룹을 IaC로 직접 작성해볼거임

with Serverless Framework



파일 이름은 일반적으로 `serverless.yml`

with Serverless Framework - Provider Property

```
service: acinside-sl-backend

provider:
  name: aws
  runtime: nodejs20.x
  region: ap-northeast-2
  memorySize: 128
  timeout: 10
  stage: prod
  environment:
    COGNITO_CLIENT_ID: !Ref CognitoClient
    COGNITO_USER_POOL_ID: !Ref CognitoUserPool
    AWS_ACCOUNT_ID: ${env:AWS_ACCOUNT_ID}
```

람다 함수의 **기본 값(provider)**을 정해줌. 람다 함수 내에서 위 코드와 같은 환경 변수를 사용하니 환경 변수도 추가

!Ref(Fn::Ref)는 함수로, 해당 이름(별칭)의 리소스의 기본 값(아이디 등)을 가져옴

with Serverless Framework - Provider Property

```
# provider:
# (생략)
httpApi:
  name: acinside-sl-api
  cors:
    allowedOrigins:
      - https://XXX.cloudfront.net
    allowedHeaders:
      - Content-Type
      - Authorization
      - X-Requested-With
      - X-Csrftoken
      - Set-Cookie
    allowedMethods:
      - GET
      - POST
      - PUT
      - DELETE
      - PATCH
      - HEAD
      - OPTIONS
  maxAge: 86400 # 1 day
  allowCredentials: true
```

람다 함수에 적용할 API Gateway의 설정을 정해줌 (CORS 설정)

with Serverless Framework - Provider Property

```
# provider:
# (생략)
# httpApi:
# (생략)
  authorizers:
    cognitoAuthorizer:
      type: jwt
      identitySource: '$request.header.Authorization'
      issuerUrl: !Sub "https://cognito-idp.${AWS::Region}.amazonaws.com/${CognitoUserPool}"
      audience: !Ref CognitoClient
```

API Gateway에서 사용 할 권한 부여자(Cognito 연동) 설정

대상(Audience)인 Cognito Client는 이후 Resources 속성에서 설정 함

with Serverless Framework - Provider Property

```
# provider:
# (생략)
iam:
  role:
    statements:
      - Effect: Allow
        Action:
          - dynamodb:PutItem # Create
          - dynamodb:GetItem # Read
          - dynamodb:Scan # Read(all)
          - dynamodb:Query # Read
          - dynamodb:UpdateItem # Update
          - dynamodb>DeleteItem # Delete
        Resource:
          - !GetAtt PostsTable.Arn
          - !GetAtt CounterTable.Arn
```

람다 함수에 대한 IAM 권한 정책 설정, 람다는 DynamoDB에 접근해야 되기 때문에 IAM 권한이 필요함

with Serverless Framework - Funcitons Property

```
functions:
  functions:
    functions:
      acinsideSL_CreatePost:
        handler: dist/functions/post/createPost.handler
        events:
          - httpApi:
              path: /posts
              method: post
              authorizer: cognitoAuthorizer
```

Serverless Framework에선 람다 함수 선언을 위한 **functions** 속성이 있음

TypeScript 빌드 후 `dist` 디렉토리에서 자바스크립트를 핸들러로 설정하고 배포함 (`handler` 속성)

Cognito JWT 검증이 필요할 경우 `authorizer` 옵션을 통해 API Gateway에서 선언해뒀던 Cognito 사용 가능

with Serverless Framework - Resources Property

Serverless Framework에선 Provider, Functions 속성에서 제공하는 서비스(전부 람다 관련)만 선언 가능

그래서 DynamoDB, Cognito 등의 서비스는 **resources** 속성에 별도로 선언해야함 (CloudFormation 템플릿 구조)

with Serverless Framework - Resources Property

```
resources:
  Resources:
    CognitoUserPool:
      Type: AWS::Cognito::UserPool
      DeletionPolicy: Retain
      UpdateReplacePolicy: Retain
      Properties:
        UserPoolName: acinside-cognito-user-pool
        AutoVerifiedAttributes:
          - email
        Schema:
          - Name: email
            Required: true
        UserPoolTier: LITE
```

```
# resources:
#   Resources:
#     (생략)
    CognitoClient:
      Type: AWS::Cognito::UserPoolClient
      Properties:
        ClientName: acinside-cognito-client
        UserPoolId: !Ref CognitoUserPool
        ExplicitAuthFlows:
          - USER_PASSWORD_AUTH
```

Cognito 유저 풀 및 해당 유저 풀에 접근할 클라이언트 선언

회원 가입 시 이메일을 제공하도록 하여 이메일 인증 사용 (로그인 시엔 유저 네임/패스워드 사용)

Cognito의 플랜(요금제)는 Lite로 설정, 에센셜/프로 대비 2배 이상 저렴한데 대부분의 기능 사용 가능

with Serverless Framework - Resources Property

```
DeletionPolicy: Retain  
UpdateReplacePolicy: Retain
```

기본적으로 스택 삭제 또는 리소스 변경 시(예: 이름 변경) 기존의 리소스는 삭제됨

⇒ Cognito 유저 풀이나 DynamoDB는 삭제되면 안됨

위와 같은 속성 사용 시 삭제를 방지하고 **유지(Retain)**할 수 있음

with Serverless Framework - Resources Property

```
# resources:
#   Resources:
#     (생략)
    PostsTable:
      Type: AWS::DynamoDB::Table
      DeletionPolicy: Retain
      UpdateReplacePolicy: Retain
      Properties:
        TableName: Posts
        AttributeDefinitions:
          - AttributeName: id
            AttributeType: S
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        BillingMode: PAY_PER_REQUEST
```

DynamoDB 선언, 마찬가지로 스택 삭제/변경 시 리소스 유지 옵션을 활성화하고 파티션 키를 정해주었음

PAY_PER_REQUEST = 온디맨드

with Serverless Framework - Plugins

SAM 대신 Serverless Framework를 이용하는 가장 큰 이유: [플러그인의 다양함](#)

플러그인은 Serverless Framework CLI나 npm으로 설치 가능, 사용 시 아래와 같이 명시함

```
plugins:  
  - serverless-esbuild  
  - serverless-prune-plugin
```

with Serverless Framework - serverless-esbuild

NodeJS로 개발하다 보면 필연적으로 node_modules가 매우 거대해지는걸 볼 수 있음

그래서 분산되어 있는 코드를 하나의 파일로 묶는 **번들링(Bundling)**이나 사용하지 않는 코드를 제거하는 **Tree Shaking**, 코드의 공백이나 긴 변수 명 등을 짧게 하여 용량을 줄여주는 **경량화(Minify)** 작업 등이 필요함

⇒ 웹 개발에선 Vite를 주로 사용하나 **ESBuild**라는 도구를 사용할 수 도 있음.

with Serverless Framework - serverless-esbuild

```
custom:
  esbuild:
    bundle: true
    minify: true
    sourcemap: false
    target: 'node20'
    platform: 'node'
    concurrency: 10
    treeShaking: true
    packager: npm
    format: 'cjs'
```

플러그인은 custom 속성을 통해 사용 가능

번들링, Tree Shaking, 경량화 등의 작업을 플러그인을 통해 자동으로 해줌

CI/CD with Github Actions

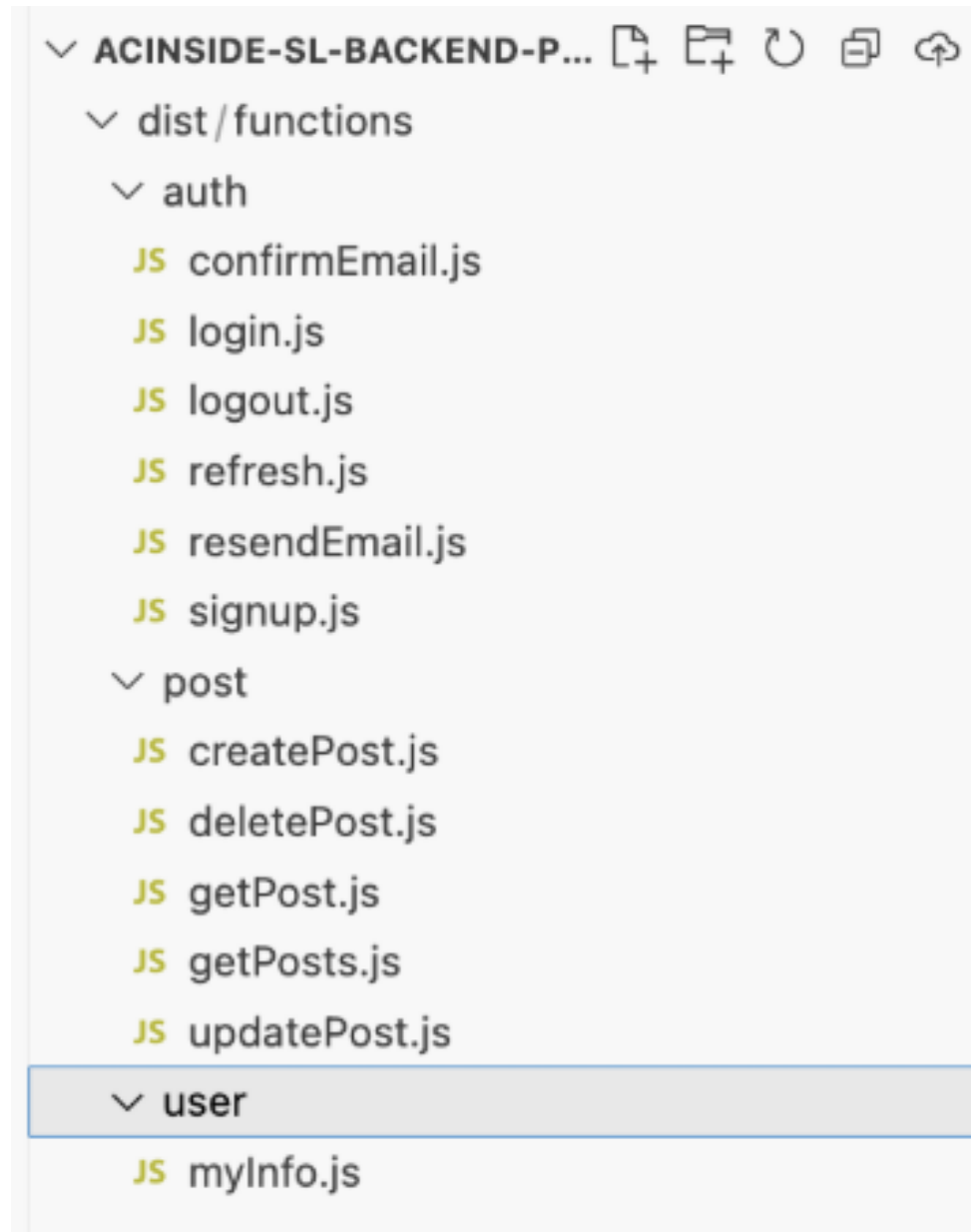
```
package:  
  individually: true
```

람다 함수 배포 시 기본적으로 모든 파일들을 배포함

⇒ 해당 람다에서 사용하지 않는 파일도 같이 올라감 (용량 커짐)

⇒ 해당 람다 핸들러가 사용하는 코드만 올려주는 옵션, esbuild로 의존 코드들을 번들링하였으니 올라가는 파일은 하나 뿐

CI/CD with Github Actions



```
functions:
  acinsideSL_ConfirmEmail: acinside-sl-backend-prod-acinsideSL_ConfirmEmail (1.8 MB)
  acinsideSL_Login: acinside-sl-backend-prod-acinsideSL_Login (1.8 MB)
  acinsideSL_Logout: acinside-sl-backend-prod-acinsideSL_Logout (1.8 MB)
  acinsideSL_ResendEmail: acinside-sl-backend-prod-acinsideSL_ResendEmail (1.8 MB)
  acinsideSL_Signup: acinside-sl-backend-prod-acinsideSL_Signup (1.8 MB)
  acinsideSL_RefreshToken: acinside-sl-backend-prod-acinsideSL_RefreshToken (1.8 MB)
  acinsideSL_CreatePost: acinside-sl-backend-prod-acinsideSL_CreatePost (1.8 MB)
  acinsideSL_DeletePost: acinside-sl-backend-prod-acinsideSL_DeletePost (1.8 MB)
  acinsideSL_UpdatePost: acinside-sl-backend-prod-acinsideSL_UpdatePost (1.8 MB)
  acinsideSL_GetPost: acinside-sl-backend-prod-acinsideSL_GetPost (1.8 MB)
  acinsideSL_GetPosts: acinside-sl-backend-prod-acinsideSL_GetPosts (1.8 MB)
  acinsideSL_MyInfo: acinside-sl-backend-prod-acinsideSL_MyInfo (1.8 MB)
```

```
"FunctionName": "acinside-sl-backend-prod-acinsideSL_CreatePost",
"FunctionArn": "arn:aws:lambda:ap-northeast-2:986129558966:function:acinside-sl-backend-prod-acinsideSL_CreatePost",
"Runtime": "nodejs20.x",
"Role": "arn:aws:iam::986129558966:role/acinside-sl-backend-prod-ap-northeast-2-lambdaRole",
"Handler": "dist/functions/post/createPost.handler",
"CodeSize": 1769349,
```

individually 옵션 적용 전

CI/CD with Github Actions



```
functions:
  acinsideSL_ConfirmEmail: acinside-sl-backend-prod-acinsideSL_ConfirmEmail (144 kB)
  acinsideSL_Login: acinside-sl-backend-prod-acinsideSL_Login (144 kB)
  acinsideSL_Logout: acinside-sl-backend-prod-acinsideSL_Logout (144 kB)
  acinsideSL_ResendEmail: acinside-sl-backend-prod-acinsideSL_ResendEmail (144 kB)
  acinsideSL_Signup: acinside-sl-backend-prod-acinsideSL_Signup (144 kB)
  acinsideSL_RefreshToken: acinside-sl-backend-prod-acinsideSL_RefreshToken (144 kB)
  acinsideSL_CreatePost: acinside-sl-backend-prod-acinsideSL_CreatePost (153 kB)
  acinsideSL_DeletePost: acinside-sl-backend-prod-acinsideSL_DeletePost (152 kB)
  acinsideSL_UpdatePost: acinside-sl-backend-prod-acinsideSL_UpdatePost (153 kB)
  acinsideSL_GetPost: acinside-sl-backend-prod-acinsideSL_GetPost (152 kB)
  acinsideSL_GetPosts: acinside-sl-backend-prod-acinsideSL_GetPosts (152 kB)
  acinsideSL_MyInfo: acinside-sl-backend-prod-acinsideSL_MyInfo (144 kB)
```

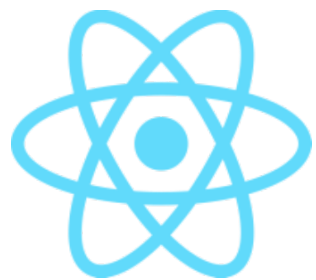


```
"FunctionName": "acinside-sl-backend-prod-acinsideSL_CreatePost",
"FunctionArn": "arn:aws:lambda:ap-northeast-2:986129558966:function:acinside-sl-backend-prod-acinsideSL_CreatePost",
"Runtime": "nodejs20.x",
"Role": "arn:aws:iam::986129558966:role/acinside-sl-backend-prod-ap-northeast-2-lambdaRole",
"Handler": "dist/functions/post/createPost.handler",
"CodeSize": 152586,
```

individually 옵션 적용 후

각 람다 함수 평균 1.8M → 153kB (함수 개수에 비례하여 감소)

Frontend with S3 + CloudFront



React



TypeScript



CloudFront



S3



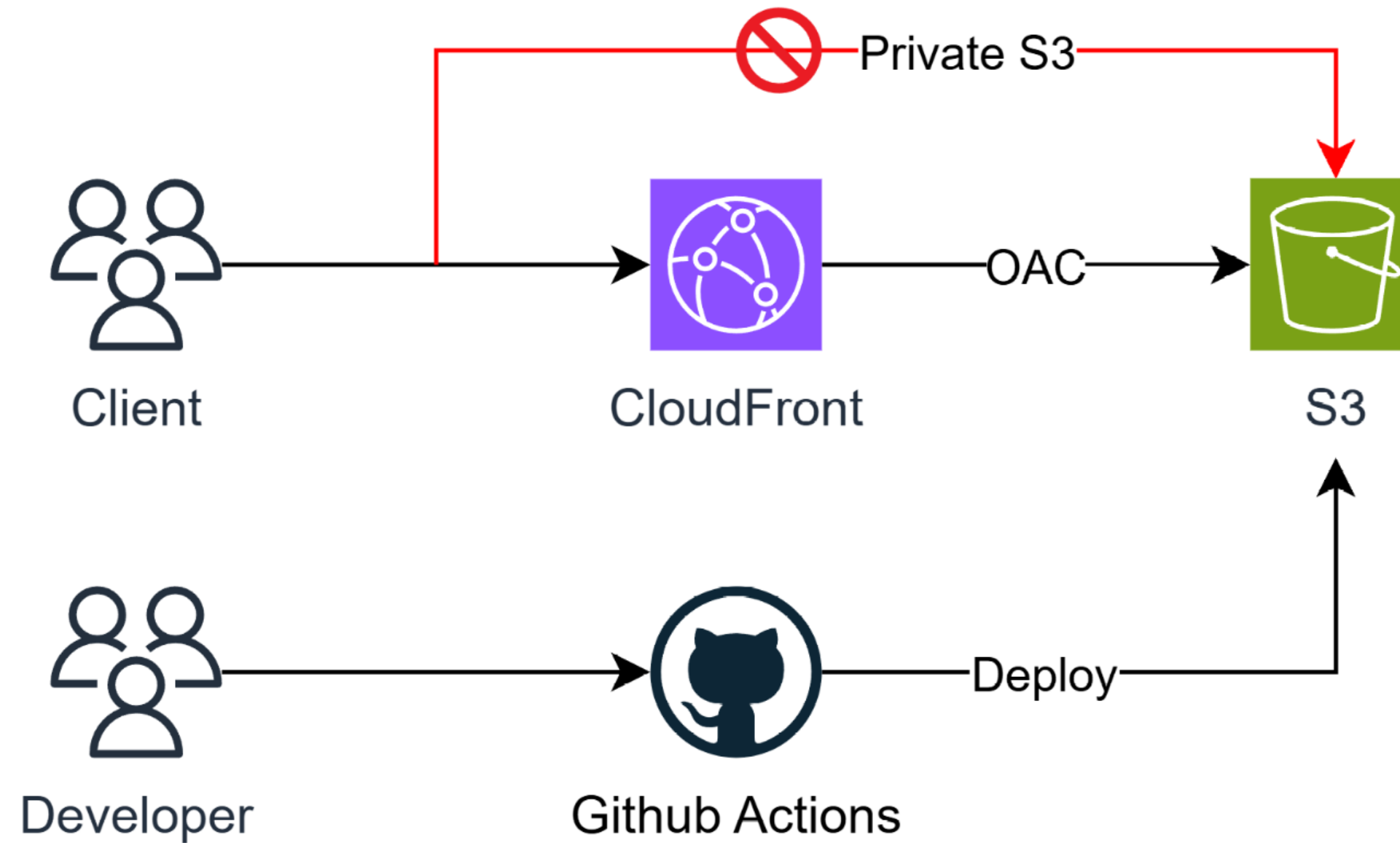
Github Actions



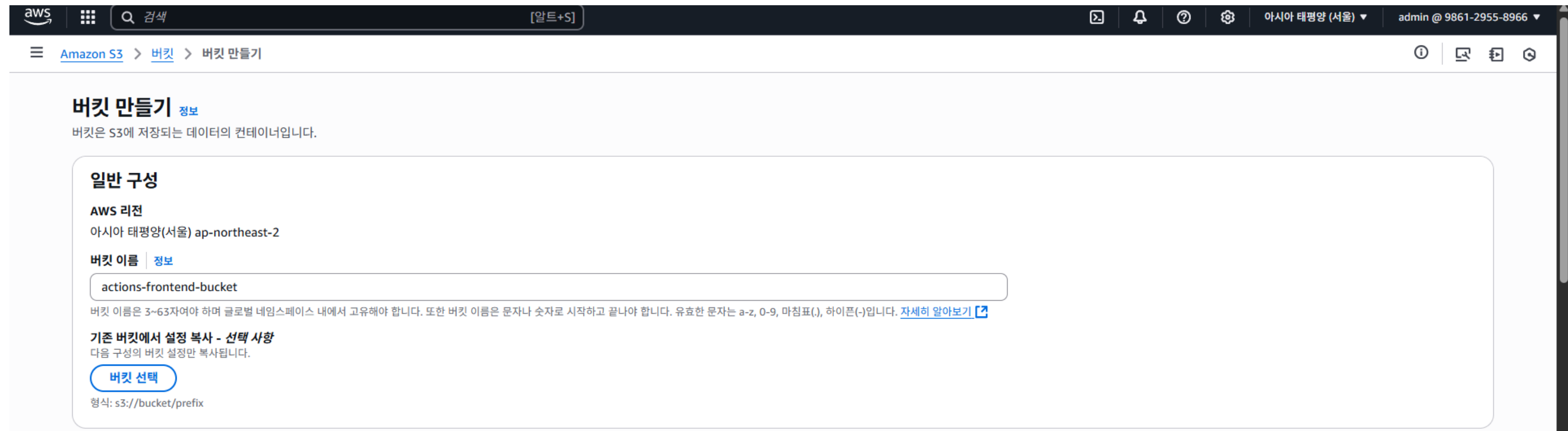
⇒ React Router로 간단하게 클라이언트 측에서 라우팅되는 SPA 페이지

Serverless Framework는 CloudFront 설정 시 복잡하기도 하고
사용할 리소스도 S3, CloudFront 각각 한개 밖에 없어서 Github Actions로 자동화만 하기로 결정

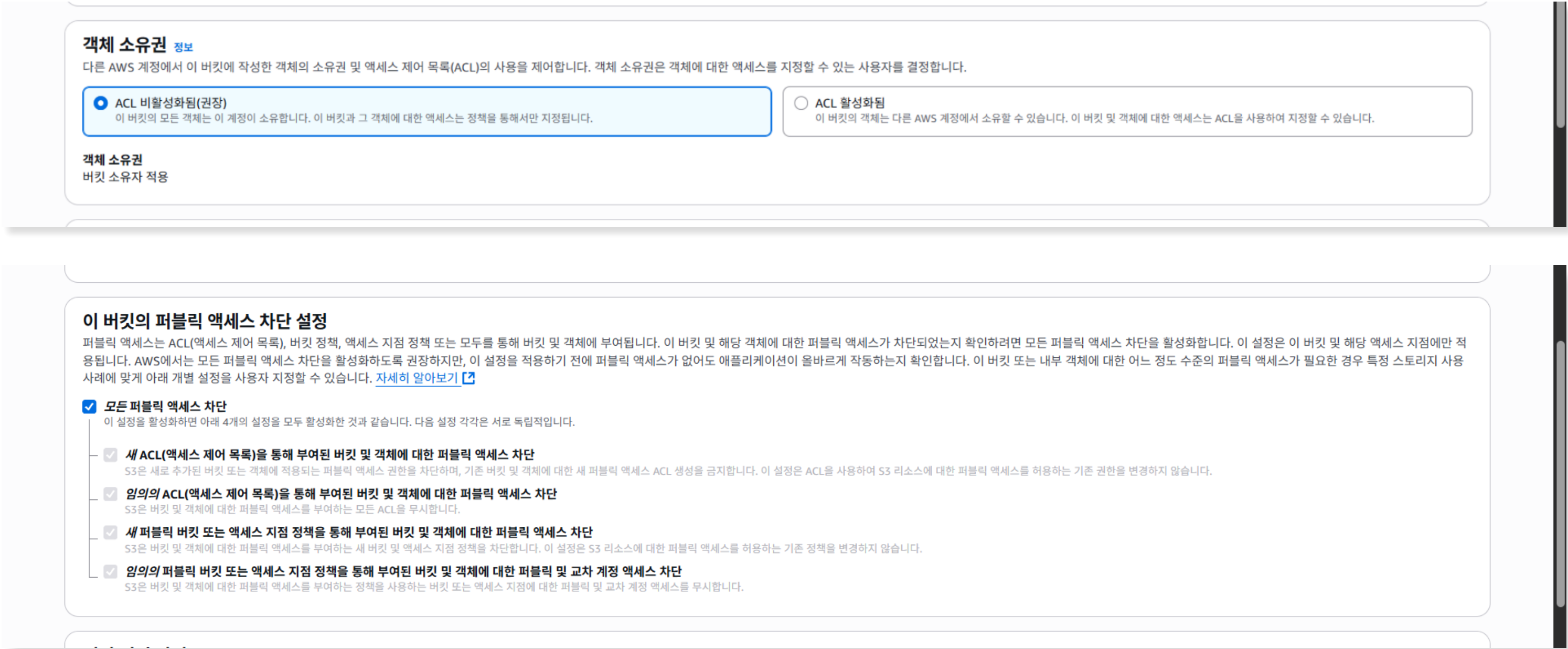
Frontend with S3 + CloudFront



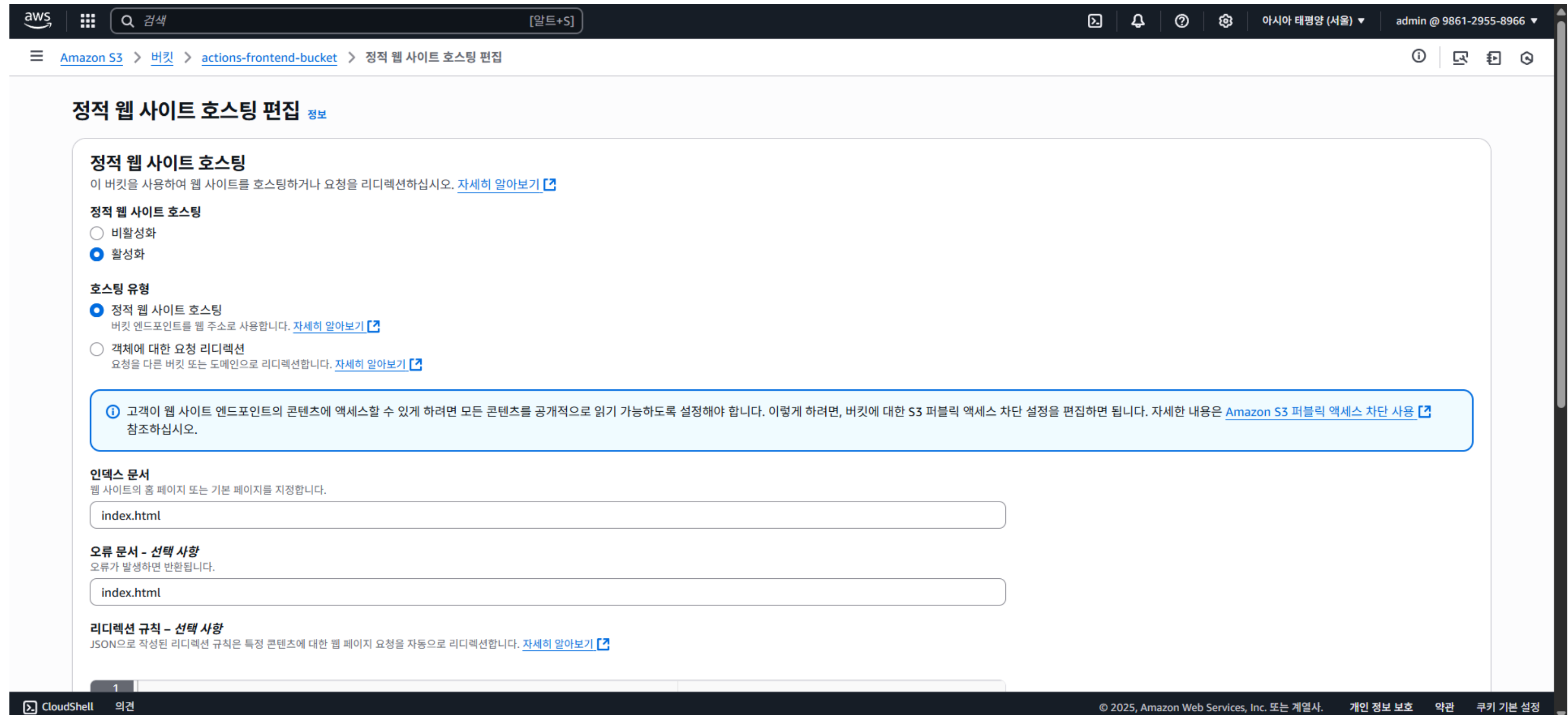
Frontend with S3 + CloudFront



Frontend with S3 + CloudFront



Frontend with S3 + CloudFront



Frontend with S3 + CloudFront

정적 웹 사이트 호스팅

이 버킷을 사용하여 웹 사이트를 호스팅하거나 요청을 리디렉션하십시오. [자세히 알아보기](#)

정적 웹 사이트 호스팅에는 AWS Amplify Hosting을 사용하는 것이 좋습니다.

AWS Amplify Hosting을 사용하여 빠르고 안전하며 안정적인 웹 사이트를 빠르게 배포하세요. [Amplify Hosting](#)에 대해 자세히 알아보거나 [기존 Amplify 앱을 확인하세요](#).

Amplify 앱 생성

S3 정적 웹 사이트 호스팅

활성화됨

호스팅 유형

버킷 호스팅

버킷 웹 사이트 엔드포인트

버킷을 정적 웹 사이트로 구성하면, 해당 웹 사이트를 버킷의 AWS 리전별 웹 사이트 엔드포인트에서 사용할 수 있습니다. [자세히 알아보기](#)

<http://actions-frontend-bucket.s3-website.ap-northeast-2.amazonaws.com>

편집

403 Forbidden

CloudFront OAC(오리진 액세스 제어)로 S3 접근할 예정이니 퍼블릭 액세스 비활성화

- Code: AccessDenied
- Message: Access Denied
- RequestId: C9PCTAKM6ET8YJMY
- HostId: XywR33SyMIGk6TxV+pKC4lKsmmhZMqWmy4sV5sc05gsCZH39KD8+6rGV5KnLlct+0lOtvemMJUnPrKBO4beWq+/h1hnxhWQioNKalwTP1UA=

An Error Occurred While Attempting to Retrieve a Custom Error Document

- Code: AccessDenied
- Message: Access Denied

Null4U 2학년 김준영

Frontend with S3 + CloudFront

aws

검색

[알트+S]

글로벌

admin @ 9861-2955-8966

CloudFront > 배포 > 생성

You're using a previous version of the Create Distribution page. [Go back to the new experience.](#)

배포 생성

☒ Single website or app
Choose if each website or application will have a unique configuration.

☐ Multi-tenant architecture - New
Choose when you have multiple domains that need to share configurations. This is a common architecture for SaaS providers.

원본

Origin domain

Choose an AWS origin, or enter your origin's domain name. [Learn more](#)

actions-frontend-bucket.s3.ap-northeast-2.amazonaws.com

Enter a valid DNS domain name, such as an S3 bucket, HTTP server, or VPC origin ID.

이 S3 버킷은 S3 웹 사이트로 구성됩니다. 이 배포를 웹 사이트로 사용하려는 경우 버킷 엔드포인트 대신 S3 웹 사이트 엔드포인트를 사용하는 것이 좋습니다.

웹 사이트 엔드포인트 사용

Origin path - optional

Enter a URL path to append to the origin domain name for origin requests.

Enter the origin path

이름

이 원본의 이름을 입력합니다.

actions-frontend-bucket.s3.ap-northeast-2.amazonaws.com

원본 액세스 | 정보

☒ 공개
버킷은 공개 액세스를 허용해야 합니다.

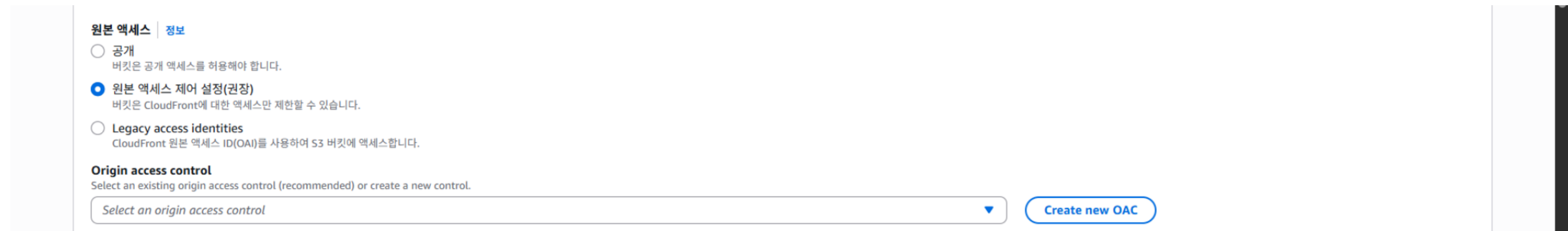
☐ 원본 액세스 제어 설정(권장)

CloudShell

의견

© 2025, Amazon Web Services, Inc. 또는 계열사. [개인 정보 보호](#) [약관](#) [쿠키 기본 설정](#)

Frontend with S3 + CloudFront



CloudFront **OAC(Origin Access Control)** = IAM 권한 기반으로 프라이빗 S3에 CloudFront가 접근 할 수 있게 함

⇒ S3는 프라이빗으로 설정해두고 OAC로 CloudFront와 연동하여 S3의 직접적인 접근 제한 (보안 강화)

Frontend with S3 + CloudFront

⚠ S3 버킷 정책을 업데이트해야 합니다.

정책 설명에서 CloudFront 원본 액세스 제어에 대한 읽기 액세스를 허용하여 배포 구성을 완료하세요. 정책을 업데이트하려면 S3 버킷 권한으로 이동합니다. [\[?\]](#)

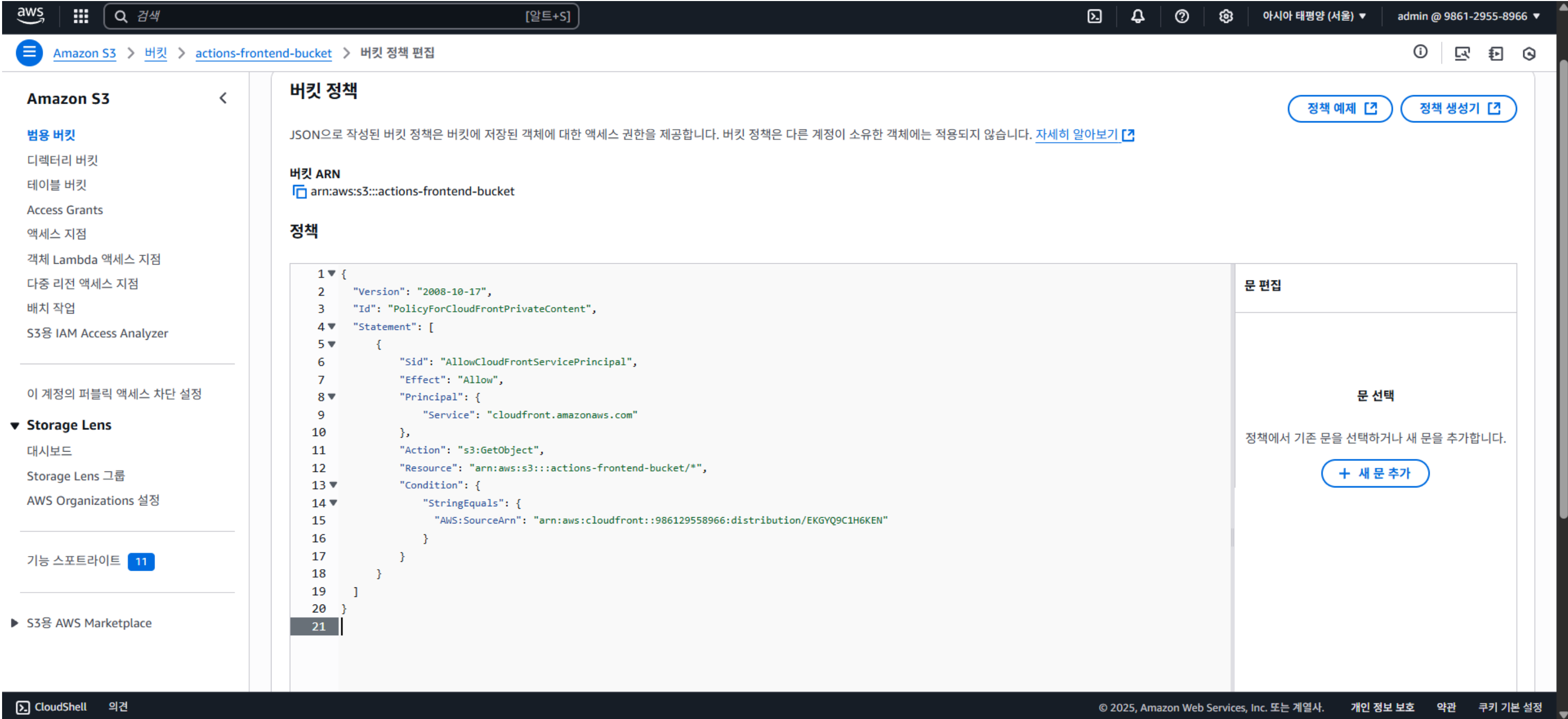
📄 정책 복사



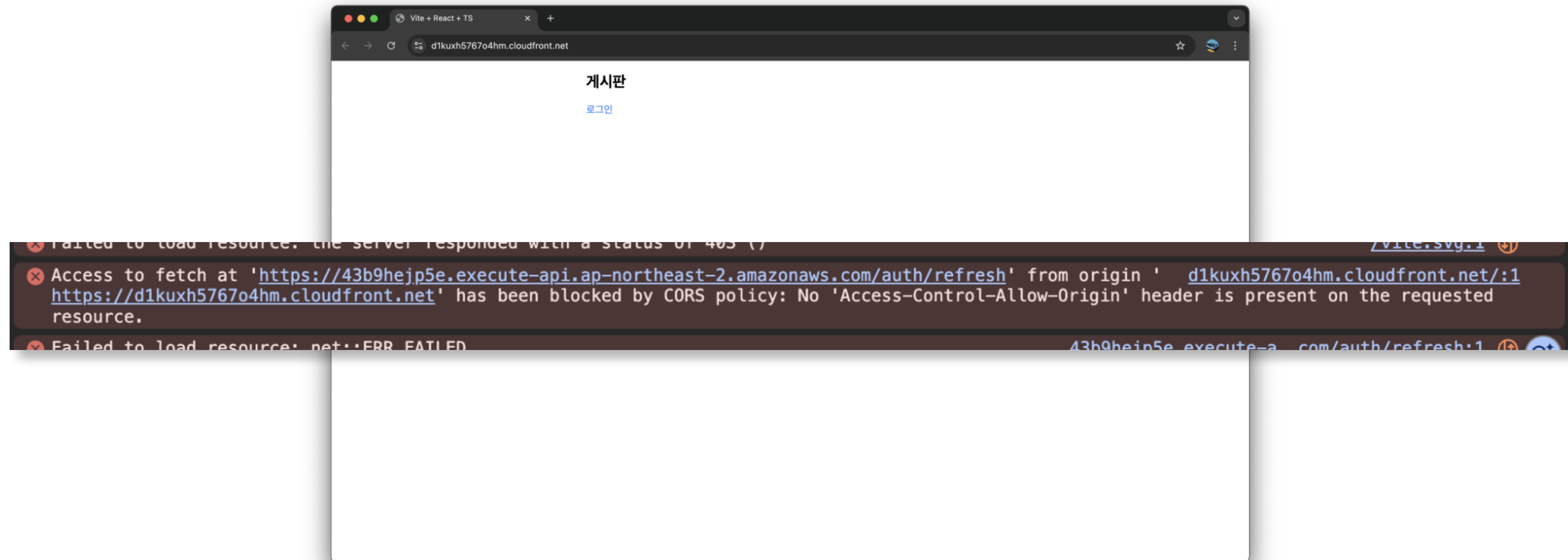
```
{
  "Version": "2008-10-17",
  "Id": "PolicyForCloudFrontPrivateContent",
  "Statement": [
    {
      "Sid": "AllowCloudFrontServicePrincipal",
      "Effect": "Allow",
      "Principal": {
        "Service": "cloudfront.amazonaws.com"
      },
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::actions-frontend-bucket/*",
      "Condition": {
        "StringEquals": {
          "AWS:SourceArn": "[CloudFront_Distribution_ARN]"
        }
      }
    }
  ]
}
```

버킷에 리소스 IAM 정책 적용 (CloudFront 서비스는 허용시키는 IAM)

Frontend with S3 + CloudFront



Frontend with S3 + CloudFront



CORS = 보안상 백엔드 측에서 어떠한 다른 출처(크로스 오리진)에 리소스 제공을 할 것 인지 명시해두는 것

Frontend with S3 + CloudFront

```
14     httpApi:  
15         name: acinside-sl-api  
16         cors:  
17             allowedOrigins:  
18                 - https://d1kuxh5767o4hm.cloudfront.net  
19             allowedHeaders:
```

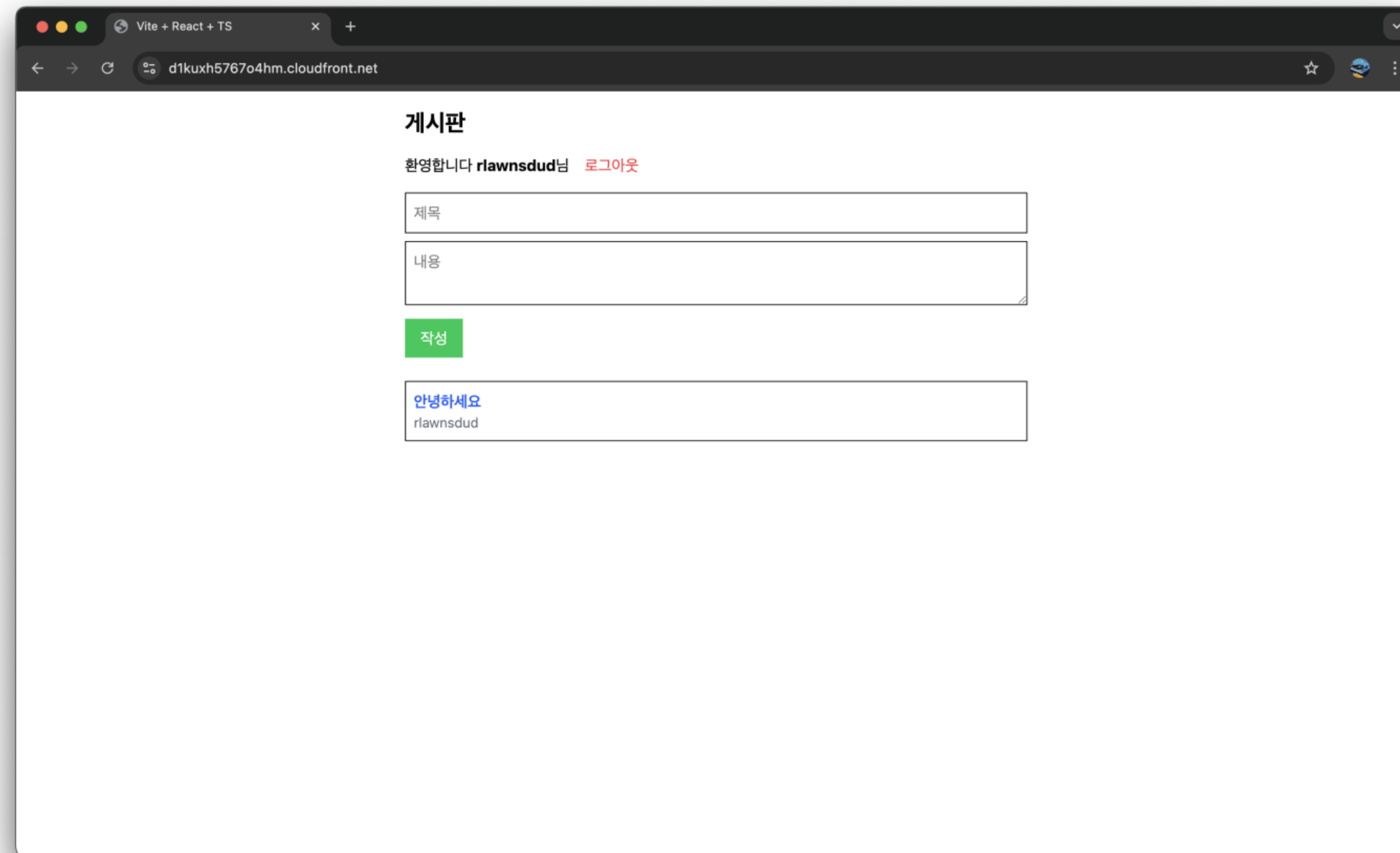
API Gateway에서 **Access-Control-Allow-Origin** 헤더에 해당 CloudFront 주소 추가

쿠키를 전송해야 되므로 **Access-Control-Allow-Credentials** 헤더를 사용

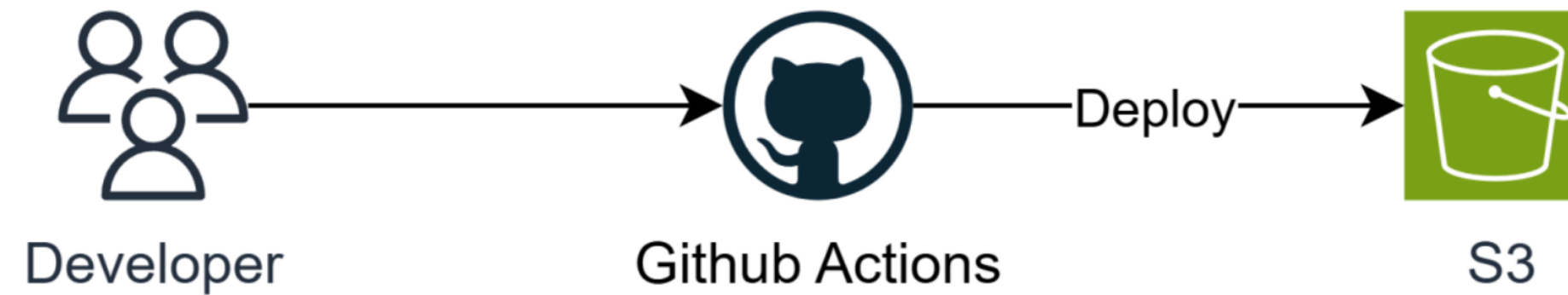
⇒ **Access-Control-Allow-Origin: *** 사용 불가

세션을 저장하지 않고 API 키 등만 제공하는 Public API에선 와일드카드(애스터리스크 *****) 사용 가능

Frontend with S3 + CloudFront



Frontend with S3 + CloudFront



NodeJS 의존성 설치, React 빌드 및 배포 자동화

Frontend with S3 + CloudFront

```
name: Deploy S3 static website (React Build)

on:
  push:
    branches:
      - main
    paths:
      - 'frontend/**'
      - '.github/workflows/aws-deploy-fe.yaml'
```

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

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

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
          aws-region: ap-northeast-2

      - name: Setup NodeJS
        uses: actions/setup-node@v4
        with:
          node-version: '22'
```

(백엔드 Github Actions와 거의 동일)

Frontend with S3 + CloudFront

```
- name: Install dependencies
  run: |
    cd frontend
    npm install

- name: Build React app
  run: |
    cd frontend
    npm run build
```

```
- name: Upload to S3
  run: |
    cd frontend
    aws s3 sync dist/ s3://${ secrets.S3_BUCKET_NAME } --delete
```

React 빌드 및 AWS CLI로 배포

sync = 로컬 파일과 비교하여 변경된 부분만 수정함, **--delete** 옵션 = 기존에 있던 불필요한 파일은 삭제

Frontend with S3 + CloudFront

```
- name: Invalidate CloudFront cache
  run: |
    aws cloudfront create-invalidation \
      --distribution-id ${ secrets.CLOUDFRONT_DISTRIBUTION_ID } \
      --paths "/*"
```

CloudFront(CDN)에 캐싱해뒀기 때문에 캐시 무효화를 통해 캐시된걸 없애고 다시 배포해야함

Frontend with S3 + CloudFront

✓ updated	Deploy S3 static website (React Build) #2: Commit 454ebe8 pushed by yulmwu	main	📅 24 minutes ago 🕒 25s	...
✓ updated	Deploy AWS Backend (Serverless Framework) #2: Commit 454ebe8 pushed by yulmwu	main	📅 24 minutes ago 🕒 2m 7s	...

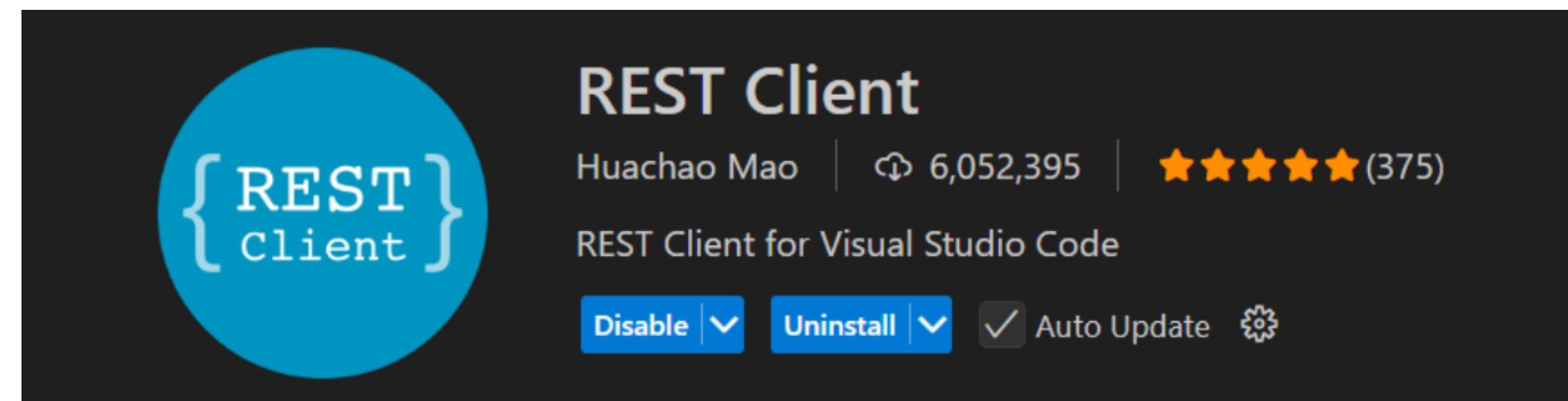


API Testing with Postman



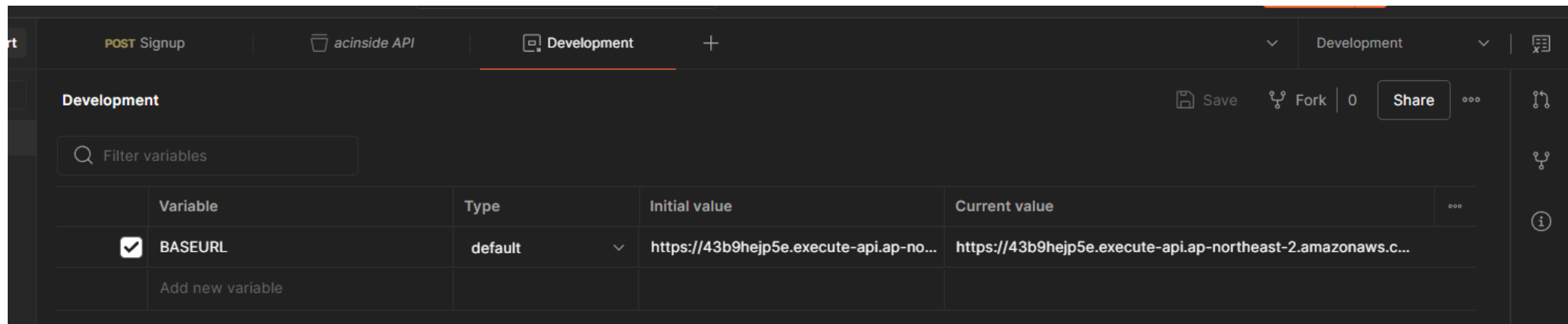
Postman = API 개발 시 (주로) **테스팅**을 위한 툴

⇒ 작은 규모면 Postman 대신 Visual Studio Code의 REST Client 등의 간단한 확장 프로그램도 굳



HTTP 요청 날릴 수 있는 HTTP 클라이언트면 다 됨

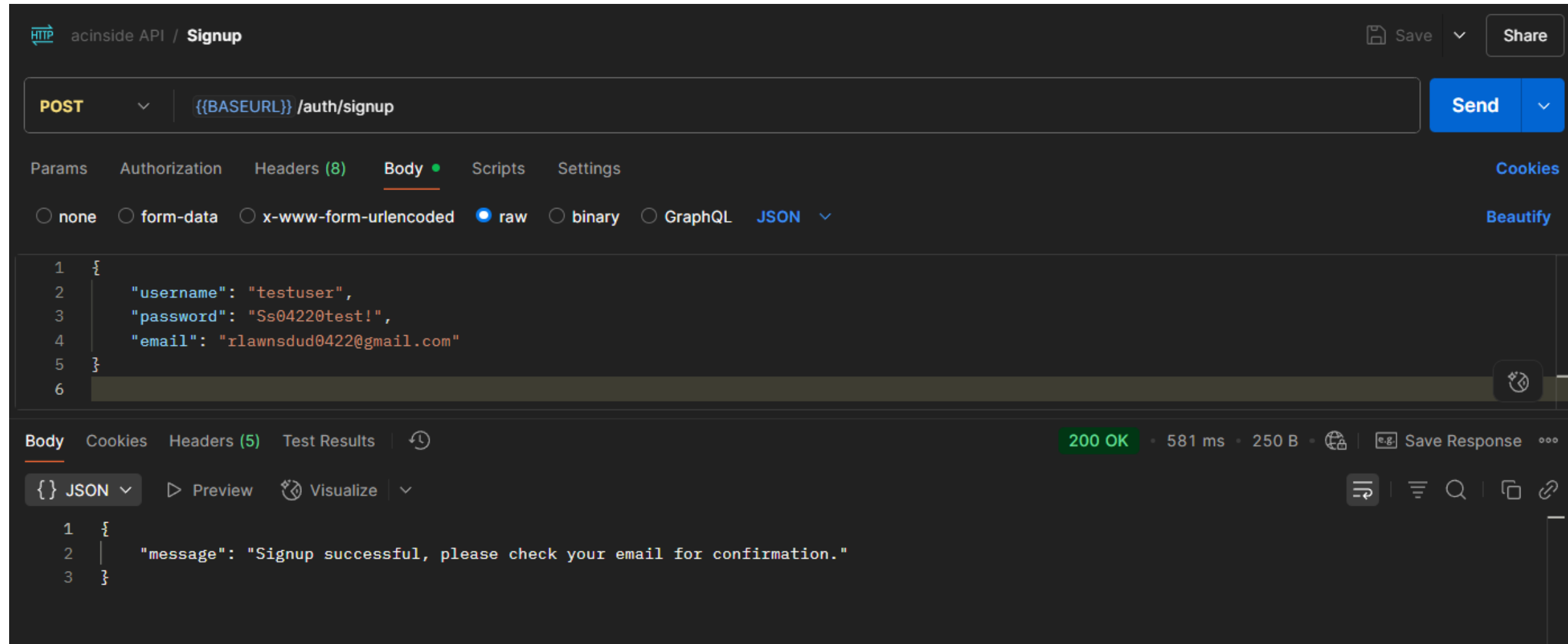
API Testing with Postman



환경 변수에 API Gateway 주소를 **Base URL**로 등록해둬م

Base URL = API 엔드포인트에서 공통으로 사용되는 URL 앞부분 (프로토콜, 호스트, 도메인, 포트 등)

Authorizations API



Authorizations API

Your verification code 받은편지함 x

no-reply@verificationemail.com

나에게 ▼

Your confirmation code is 574842

← 답장

↪ 전달

😊

오후 7:53 (1분 전)

☆

😊

↶

⋮

```
4  "email": "rlawnsdud0422@gmail.com"
5  }
```

사용자 정보

사용자 (2) 정보

사용자 풀에서 사용자를 확인, 편집 및 생성합니다. 사용 설정되고 확인된 사용자는 사용자 풀에 로그인할 수 있습니다.

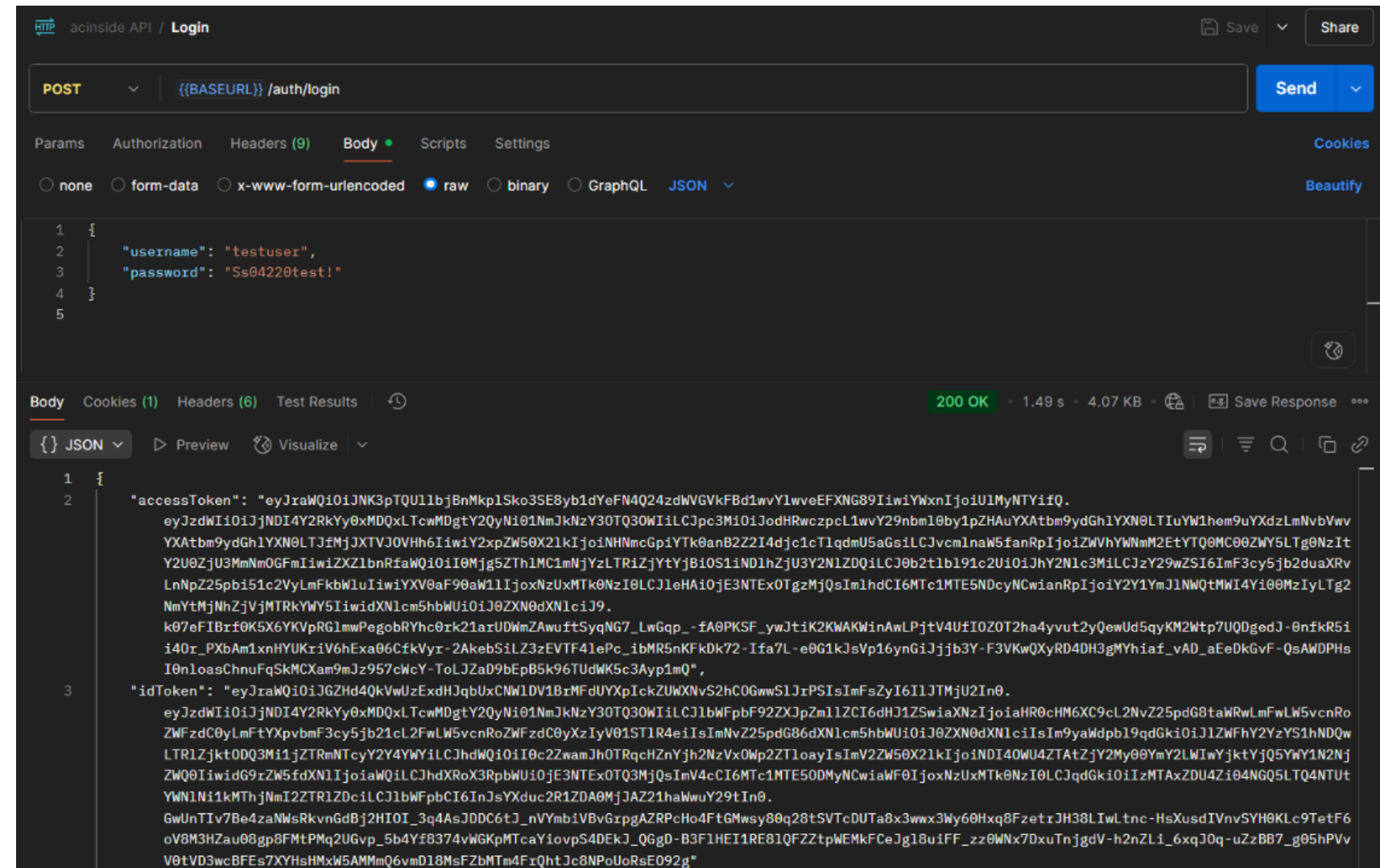
속성: 사용자 이름 ▼

🔍 속성으로 사용자 검색

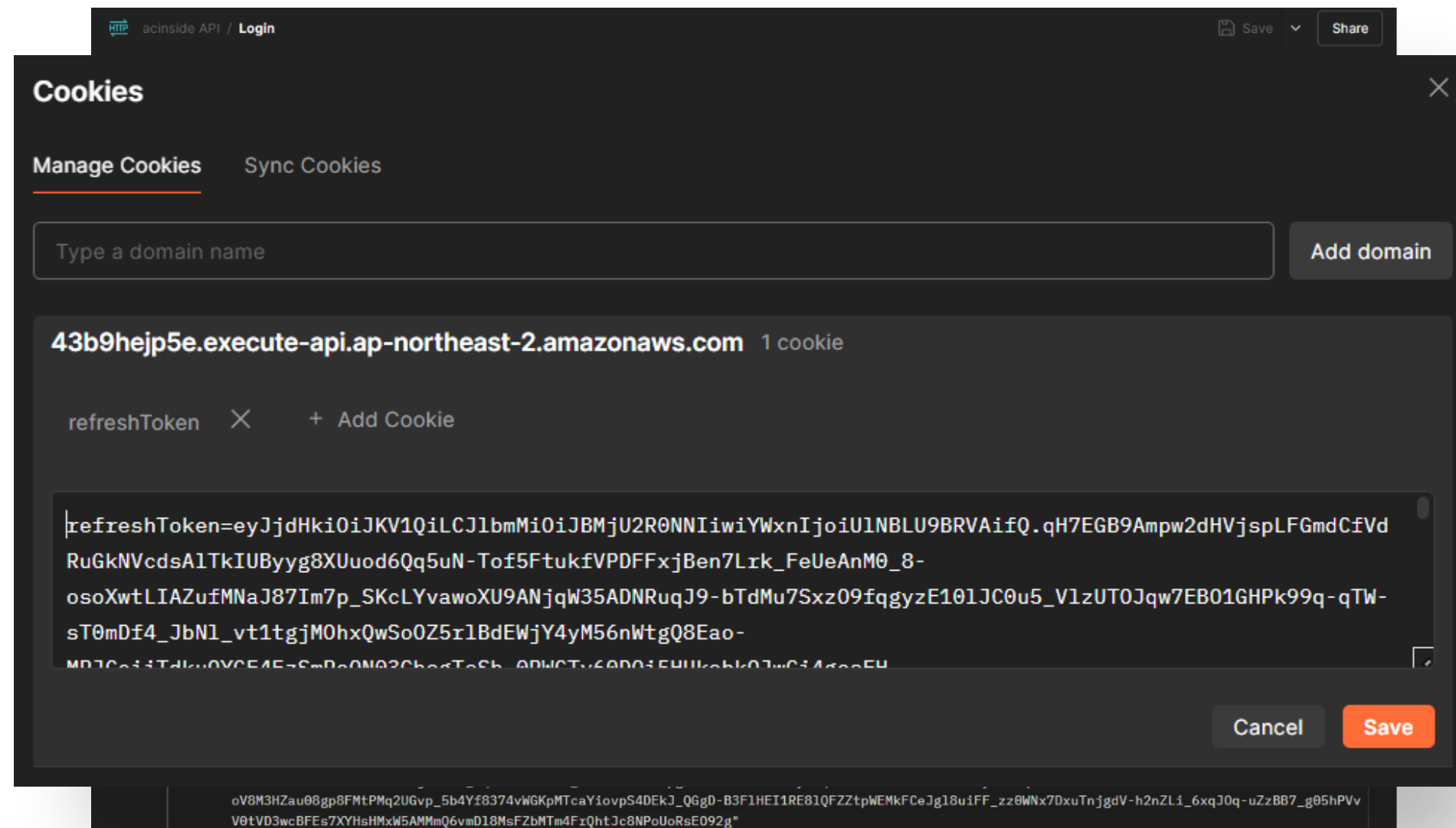
< 1 > ⚙️

	사용자 이름	이메일 주소	이메일 확인됨	확인 상태	상태
☐	rlawnsdud	normal8781@gmail.com	예	확인됨	✅ 활성화됨
☐	testuser	rlawnsdud0422@gmail.com	아니요	확인되지 않음	✅ 활성화됨

Authorizations API

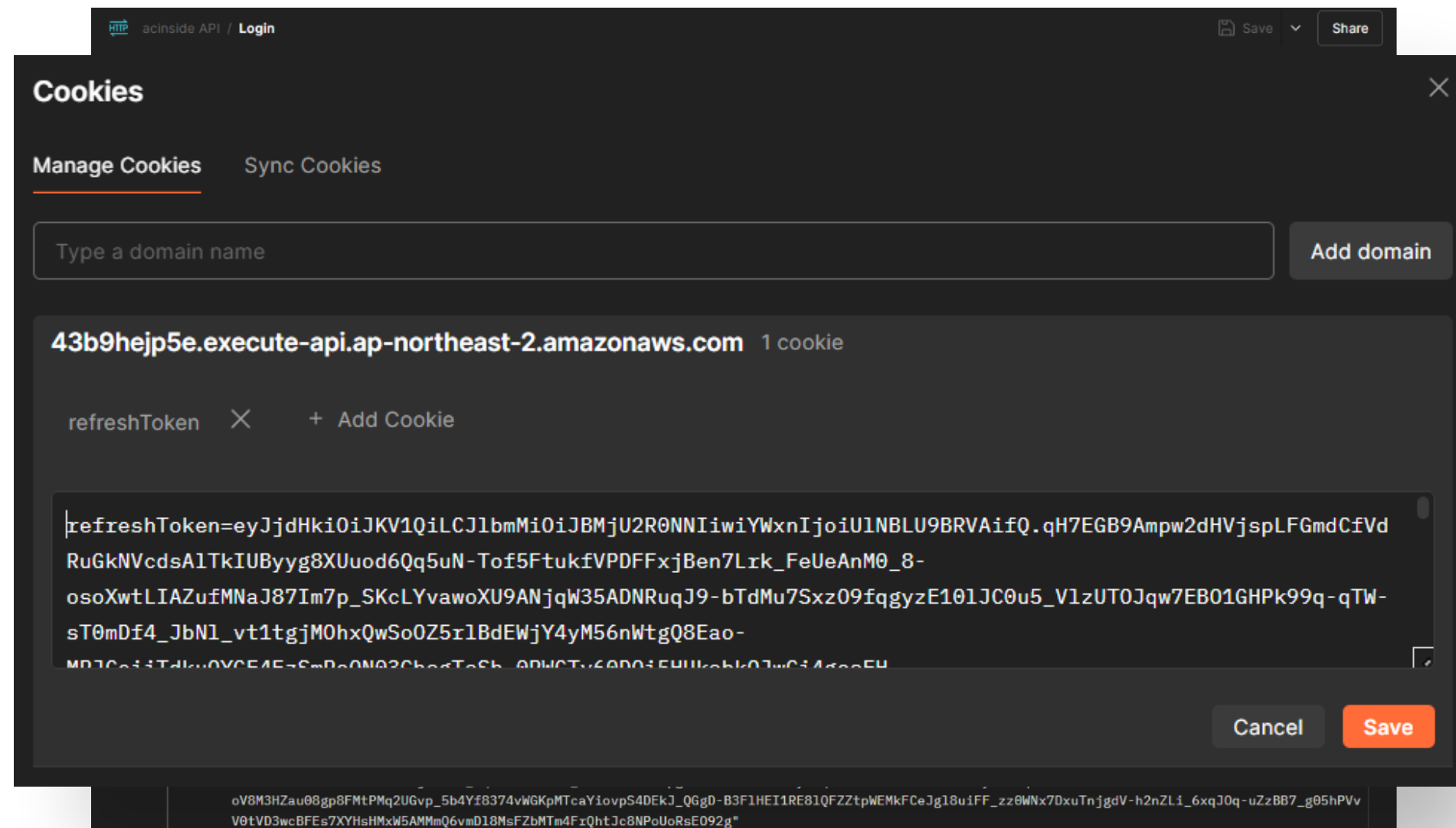


Authorizations API



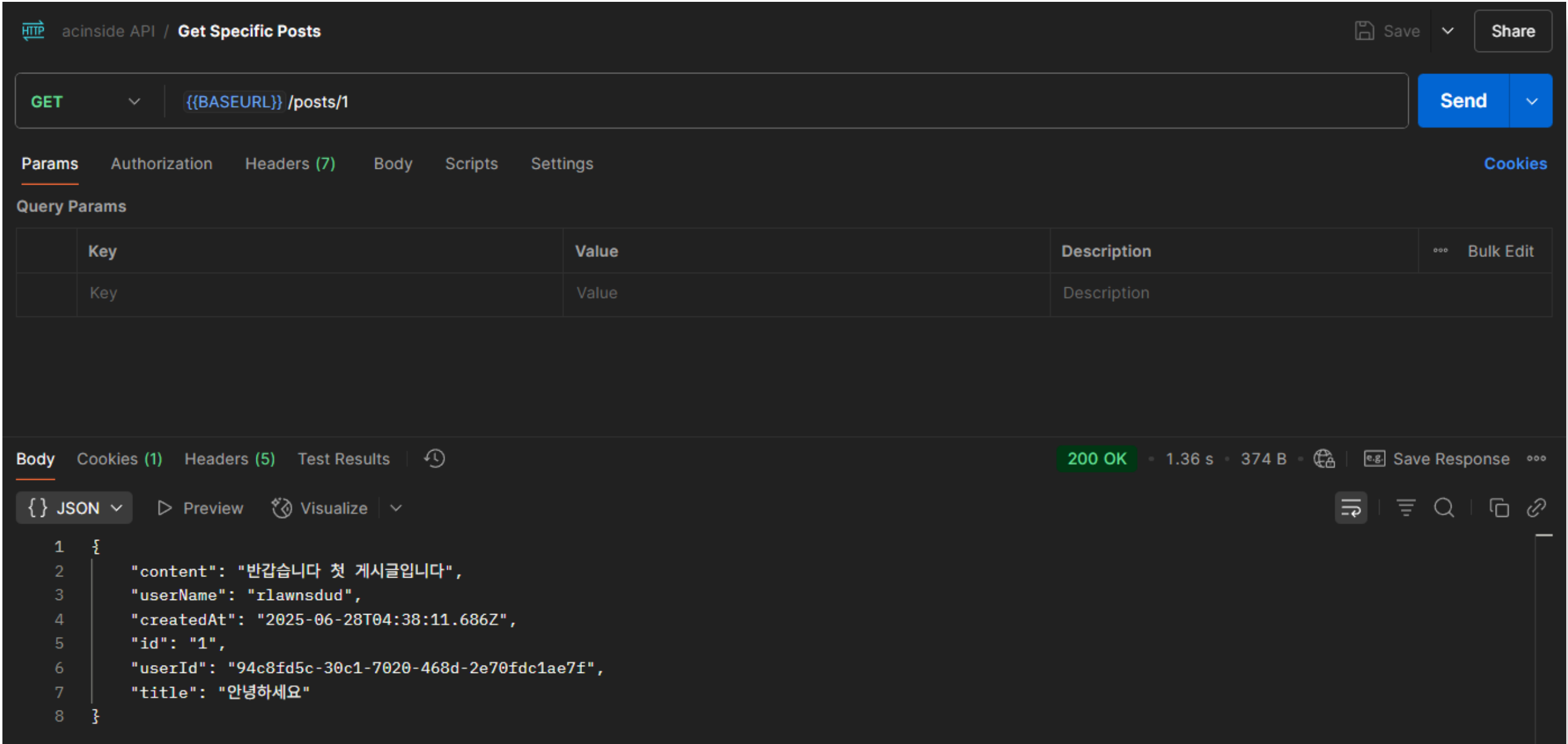
CORS는 브라우저에서만 적용되는 보안 정책이라 Postman 같은 앱에선 적용 안됨
(브라우저에서는 쿠키나 세션 저장 시 다른 사이트가 그걸 이용할 수 있음, CSRF 등)

Authorizations API

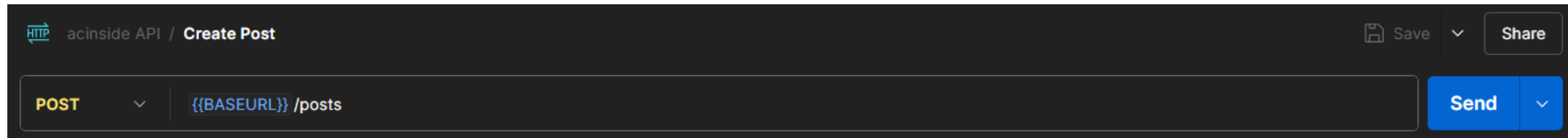


다만 CORS는 요청을 보내고 그 응답을 가지고 확인하기 때문에 CSRF 공격을 막으려면 CSRF 토큰 등이 필요

Posts CRUD API



Posts CRUD API



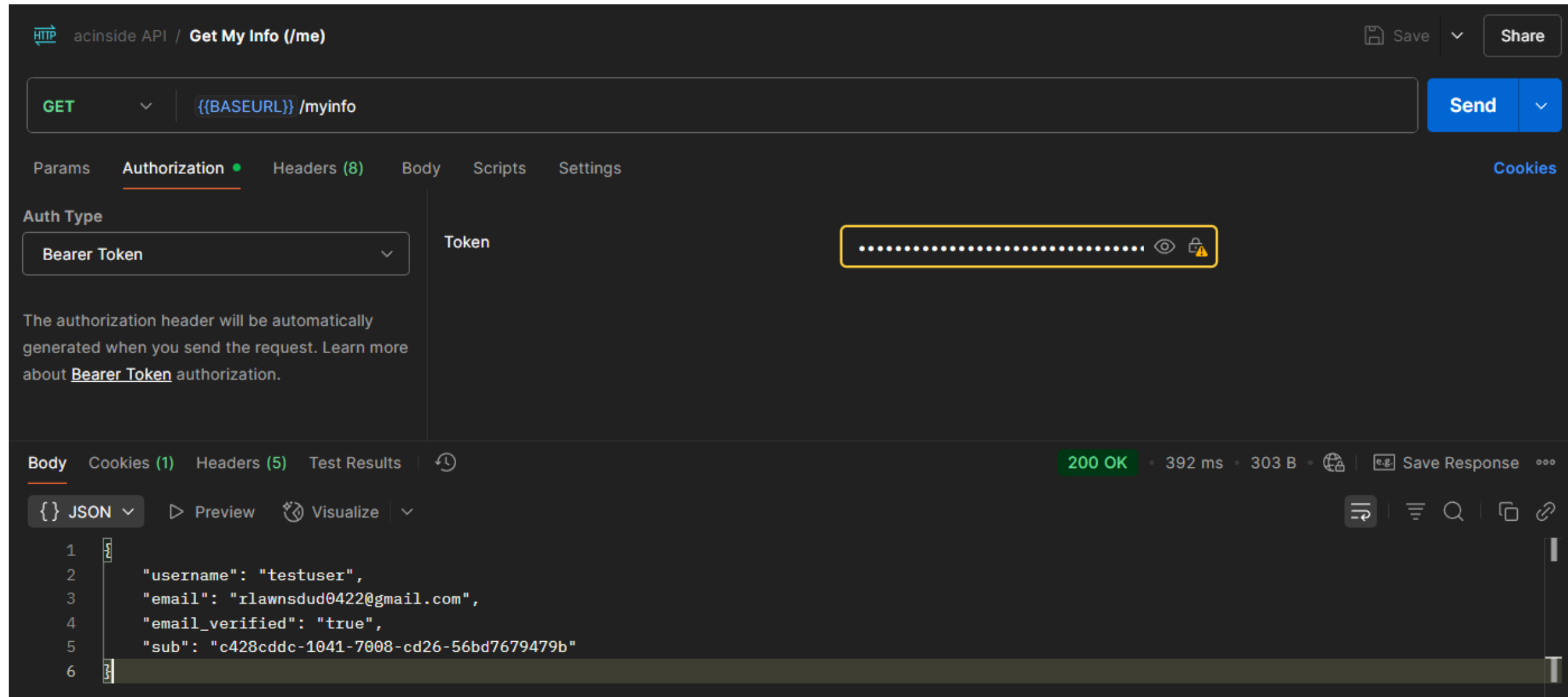
테이블: Posts - 반환된 항목 (2)

스캔 시작 날짜: 6월 29, 2025, 20:04:31

☐	id (문자열)	content	createdAt	title	userId	userName
☐	2	반갑습니다.	2025-06-2...	Postman API 테스트 중 입니다.	c428cddc-1...	testuser
☐	1	반갑습니다 첫 게시글입니다	2025-06-2...	안녕하세요	94c8fd5c-3...	rlawnsdud

```
3  "title": "Postman API 테스트 중 입니다.",
4  "content": "반갑습니다.",
5  "userId": "c428cddc-1041-7008-cd26-56bd7679479b",
6  "userName": "testuser",
7  "createdAt": "2025-06-29T11:03:39.758Z"
8  }
```

Users CRUD API (/me)



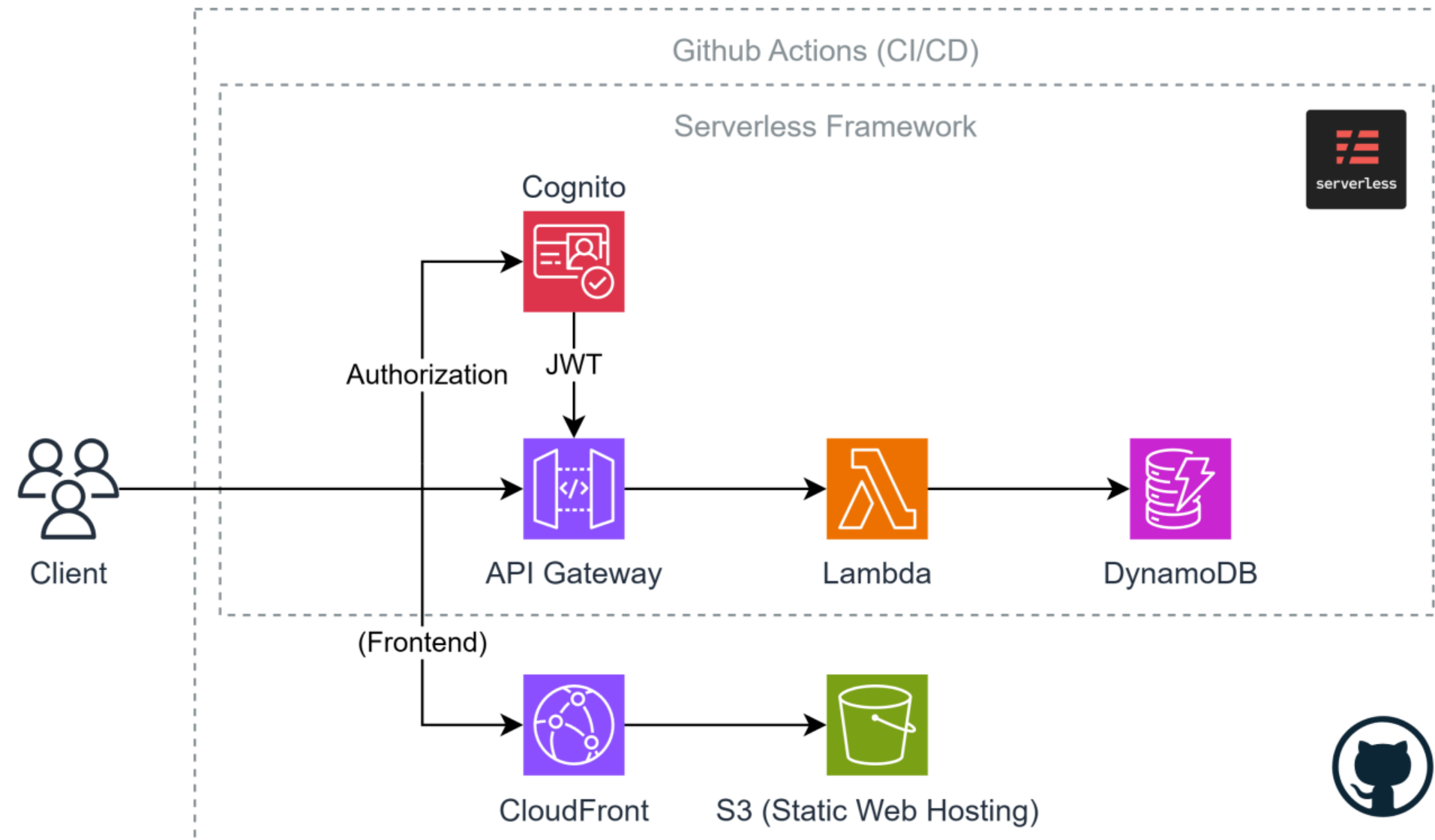
Calculate Price

비용 계산을 수학 공식으로 하기보단 어떠한 상황을 가정하는게 좋을 듯

지표	계산	수치	설명
총 가입자 수		50,000명	누적 회원
MAU		20,000명 (40%)	월간 활성 사용자
월간 방문 수	$MAU \times 10\text{회}$	200,000회	MAU 당 월 10회 방문
월간 회원 기능 API 호출 수	$\text{월간 방문 수} \times 1.5\text{회}$	300,000회	로그인, 리프레시 등
월간 페이지 뷰	$\text{월간 방문 수} \times 10\text{회}$	2,000,000회	방문 시 페이지 확인 수
페이지 당 API 호출 수		3회	게시글, 댓글 쓰기, 불러오기 등
월간 페이지 당 API 호출 수	$\text{월간 페이지 뷰} \times \text{페이지 당 API 호출}$	6,000,000회	
총 API 호출 수	$\text{월간 회원 기능 API 호출 수} + \text{월간 페이지 당 API 호출 수}$	6,300,000회	

간단하게 **MAU(Monthly Active Users)**, 기타 수치만 가정하고 비용 계산을 해보겠음

Calculate Price



지금까지 구축했던 AWS 인프라 아키텍처

Github Actions, CloudFormation (Serverless Framework)는 일반적으로는 무료임

Calculate Price

[Documentation](#) > [AWS Billing and Cost Management](#) > User Guide

Generating estimates with Pricing Calculator

AWS에선 Pricing Calculator(요금 계산기)를 사용하여 요금을 추정해볼 수 있음

Lambda

Lambda function compute

☒ 사용량 계산에 포함
이 사용량을 추정치에 포함하지 않으려면 이 확인란을 선택 취소합니다.

Amount of memory allocated †
Enter the amount between 128 MB and 10 GB

128

MB

Number of requests †

6300000

per month

Duration of each request (in ms) †
Duration is calculated from the time your code begins executing until it returns or otherwise terminates.

1000

Architecture †

x86

사용량

787500

seconds

[▶ 계산 보기](#)

총 API 요청 수는 **6,300,000건** 가정

함수의 평균 실행 시간은 **1초(1000ms)**로 가정

Lambda

☐	▼ AWSLambda							US\$14.39 USD
☐	APN2-Request	Lambda	Asia Pacific (Seoul...	986129558966	6300000 Request	✔ 신규	-	US\$1.26 USD
☐	APN2-Lambda-GB-Second	Lambda	Asia Pacific (Seoul...	986129558966	787500 Lambda-G...	✔ 신규	-	US\$13.13 USD

14.39달러, 한화 2만원 정도 나옴 (매우 저렴)

API Gateway

HTTP APIs

☒ 사용량 계산에 포함
이 사용량을 추정치에 포함하지 않으려면 이 확인란을 선택 취소합니다.

Requests ↑

per month ▼

HTTP API requests units ↑

The unit will apply to the number of requests you will specify below.

Average size of each request ↑

Requests are defined as having 512 KB of data; a request with 513 KB of data is metered as two requests.

KB ▼

사용량

Requests

▶ 계산 보기

API Gateway는 요청 수 + 요청의 크기로 부과됨

요청 수는 람다 함수와 같게 **6,300,000건**으로 가정

평균 요청의 크기는 **20KB**로 가정

API Gateway

☐	▼ AmazonApiGateway							US\$7.75 USD
☐	APN2-ApiGatewayHttpReque...	APIGateway	Asia Pacific (Seoul...	986129558966	6300000 Requests	✔ 신규	-	US\$7.75 USD

7.75달러, 한화 약 만원 정도의 요금 발생 (매우 저렴)

Cognito

Monthly active users (MAU)

☒ 사용량 계산에 포함
이 사용량을 추정치에 포함하지 않으려면 이 확인란을 선택 취소합니다.

Number of monthly active users (MAU) who sign in through SAML or OIDC federation †

Enter the number of users who sign-in through SAML or OIDC federation instead of a Cognito User Pool or with social identity providers.

0

Number of monthly active users (MAU) †

A user is counted as a MAU if, within a calendar month, there is an identity operation related to that user, such as sign-up, sign-in, token refresh or password change.

20000

사용량

10000

CognitoUserPoolsMAU

▶ 계산 보기

Cognito는 MAU(월간 활성 사용자)를 바탕으로 부과

어느 유저에 대해 로그인, 로그아웃 등의 단 한번이라도
작업이 발생하면 1 MAU로 취급함

Lite 플랜 기준으로 설정 (다음 에센셜과 2~3배 차이)

모든 플랜에서 처음 1만명은 무료로 제공해줌

Cognito

☐	▼ AmazonCognito							US\$55.00 USD
☐	APN2-CognitoUserPoolsMAU ...	Cognito	Asia Pacific (Seoul...	986129558966	10000 CognitoUser...	✓ 신규	-	US\$55.00 USD

55달러, 한화 7만 4천원 정도 나옴

직접 구현했다면 유저 DB 관련 로직, 보안 관련 로직, 유지보수 등 모든걸 직접 구현해야하지만

Cognito 사용 시 위 작업들은 기본으로 다 해주고

AWS 서비스에 대한 인가(Authentication) 기능(자격 증명 풀), OAuth 등의 기능 까지 있음

다만 비용은 직접 구현하는 것 보단 좀 더 많이 나갈 순 있음 (Cognito: 다 해줬잖아)

DynamoDB

DynamoDB는 **스토리지 용량 + 읽기 + 쓰기 작업**으로 요금을 계산함

DynamoDB를 사용하는 API 호출 수를 **6,000,000건**으로 가정하여 **읽기는 90%, 쓰기는 10%**로 가정함

DynamoDB

DynamoDB on demand read settings

☒ 사용량 계산에 포함
이 사용량을 추정치에 포함하지 않으려면 이 확인란을 선택 취소합니다.

Table class
Standard

Number of reads †
5400000 per month

Eventually consistent percentage †
95 %

Average item size (all attributes) †
1 KB

Strongly consistent percentage †
5 %

Transactional percentage †
0 %

사용량
2835000 ReadRequestUnits

▶ 계산 보기

Eventually consistent reads와 Strongly consistent reads

- ⇒ 데이터가 쓰여질 시 바로 적용된걸 읽을 수 있는지 아닌지를 나타냄
- ⇒ 예를 들어 게시글 작성 이후: 변경된걸 바로 읽어야함 (후자)
- ⇒ 내부적으로 DynamoDB는 여러 곳에 분산해서 이렇게 있음 (생략)

아이템의 평균 크기는 **1KB**로 가정함

(게시판 API 같은건 그 크기가 엄청 크진 않음)

DynamoDB

DynamoDB on demand write settings

☒ 사용량 계산에 포함
이 사용량을 추정치에 포함하지 않으려면 이 확인란을 선택 취소합니다.

Table class

Standard

Number of writes ↑

600000

per month

Standard writes ↑

100

%

Average item size (all attributes) ↑

1

KB

Transactional writes ↑

0

%

사용량

600000

WriteRequestUnits

▶ 계산 보기

쓰기도 마찬가지로 평균 아이템의 크기는 **1KB**로 가정함

DynamoDB

☐	▼ AmazonS3							US\$0.73 USD
☐	APN2-TimedStorage-ByteHrs	S3	Asia Pacific (Seoul...	986129558966	1 GB-Mo	✔ 신규	-	US\$0.03 USD
☐	APN2-Requests-Tier2	S3	Asia Pacific (Seoul...	986129558966	2000000 Requests	✔ 신규	-	US\$0.70 USD

0.79달러, 한화 1100원 정도 나옴 (매우매우 저렴)

S3

프론트엔드에서 S3 앞에 CloudFront를 붙여줬고, CloudFront와 오리진 사이의 데이터 전송 비용은 발생하지 않음

⇒ 다만 어떠한 이유로 CloudFront에서 캐시 미스가 나서 S3에 접근해야될 땐 S3 비용 발생

캐시 히트율은 넉넉하게 **90%**로 가정 (실제론 더 높음)

S3는 GET 등의 요청 작업과 스토리지 용량으로 비용을 계산함

S3

S3 Standard - GET, SELECT requests

- ☒ 사용량 계산에 포함
이 사용량을 추정치에 포함하지 않으려면 이 확인란을 선택 취소합니다.

GET, SELECT, and all other requests from S3 Standard †

Ongoing monthly number of GET, SELECT and all other requests

사용량

Requests

▶ 계산 보기

S3 Standard - Storage

- ☒ 사용량 계산에 포함
이 사용량을 추정치에 포함하지 않으려면 이 확인란을 선택 취소합니다.

S3 Standard storage †

GB per month ▼

사용량

GB-Mo

▶ 계산 보기

월간 페이지 뷰는 **2,000,000회**로 가정, 프론트엔드 소스코드는 넉넉하게 **1GB**로 가정 (비중 작음)

S3

▼ AmazonS3							US\$0.73 USD
APN2-TimedStorage-ByteHrs	S3	Asia Pacific (Seoul...	986129558966	1 GB-Mo	✔ 신규	-	US\$0.03 USD
APN2-Requests-Tier2	S3	Asia Pacific (Seoul...	986129558966	2000000 Requests	✔ 신규	-	US\$0.70 USD

0.7달러, 한화 약 천원 정도로 매우 저렴함 (S3 → CloudFront 데이터 전송 비용은 없음)

CloudFront

요금 계산 중 가장 복잡함

⇒ 기본적으로 인터넷으로의 데이터 전송 용량(비용은 국가마다 다름)과 HTTPS 요청 수로 계산

월간 페이지 뷰(**2,000,000회**) x 소스코드(10개) 용량(**1MB**) = 약 **2TB**

국가는 대한민국과 일본으로 나뉘었다고 가정함 (중국은 AWS에서 직접 지원하지 않음)

- 서울(대한민국) : Data transfer out to internet: **90% = 1.8TB**, Number of requests = **18,000,000**
- 일본 : Data transfer out to internet: **10% = 0.2TB**, Number of requests = **2,000,000**

CloudFront

데이터를 인터넷으로 전송하는 리전별 요금(GB당)

월별	미국, 멕시코 및 캐나다	유럽, 이스라엘, 튀르키예	남아프리카, 케 냐, 나이지리아, 이집트 및 중동	남아메리카	일본	오스트레일리아 및 뉴질랜드	홍콩, 인도네시 아, 필리핀, 싱가포르, 대한민국, 대만, 태국, 말레 이시아 및 베트남	인도
첫 1TB	무료	무료	무료	무료	무료	무료	무료	무료
다음 9TB	0.085 USD	0.085 USD	0.110 USD	0.110 USD	0.114 USD	0.114 USD	0.120 USD	0.109 USD
다음 40TB	0.080 USD	0.080 USD	0.105 USD	0.105 USD	0.089 USD	0.098 USD	0.100 USD	0.085 USD
다음 100TB	0.060 USD	0.060 USD	0.090 USD	0.090 USD	0.086 USD	0.094 USD	0.095 USD	0.082 USD
다음 350TB	0.040 USD	0.040 USD	0.080 USD	0.080 USD	0.084 USD	0.092 USD	0.090 USD	0.080 USD
다음 524TB	0.030 USD	0.030 USD	0.060 USD	0.060 USD	0.080 USD	0.090 USD	0.080 USD	0.078 USD
다음 4PB	0.025 USD	0.025 USD	0.050 USD	0.050 USD	0.070 USD	0.085 USD	0.070 USD	0.075 USD
5PB 초과	0.020 USD	0.020 USD	0.040 USD	0.040 USD	0.060 USD	0.080 USD	0.060 USD	0.072 USD

데이터 전송 용량이 많아지면 많아질수록 더욱 더 저렴해지는 구조임

CloudFront

☐	▼ AmazonCloudFront							US\$268.53 USD
☐	JP-Requests-HTTPS-Proxy	CloudFront	-	986129558966	2000000 Requests	✓ 신규	-	US\$2.40 USD
☐	AP-DataTransfer-Out-Bytes	CloudFront	-	986129558966	1843.2 GB	✓ 신규	-	US\$221.18 USD
☐	AP-Requests-HTTPS-Proxy	CloudFront	-	986129558966	18000000 Requests	✓ 신규	-	US\$21.60 USD
☐	JP-DataTransfer-Out-Bytes	CloudFront	-	986129558966	204.8 GB	✓ 신규	-	US\$23.35 USD

268.53달러, 한화 **36만 5천원**으로 현재까지 서비스 중 가장 높음 요금

비싸보일 수 있으나 S3 → 인터넷으로 직접 전송하는 비용도 만만치 않고 CloudFront를 사용할수록 요금이 저렴해짐

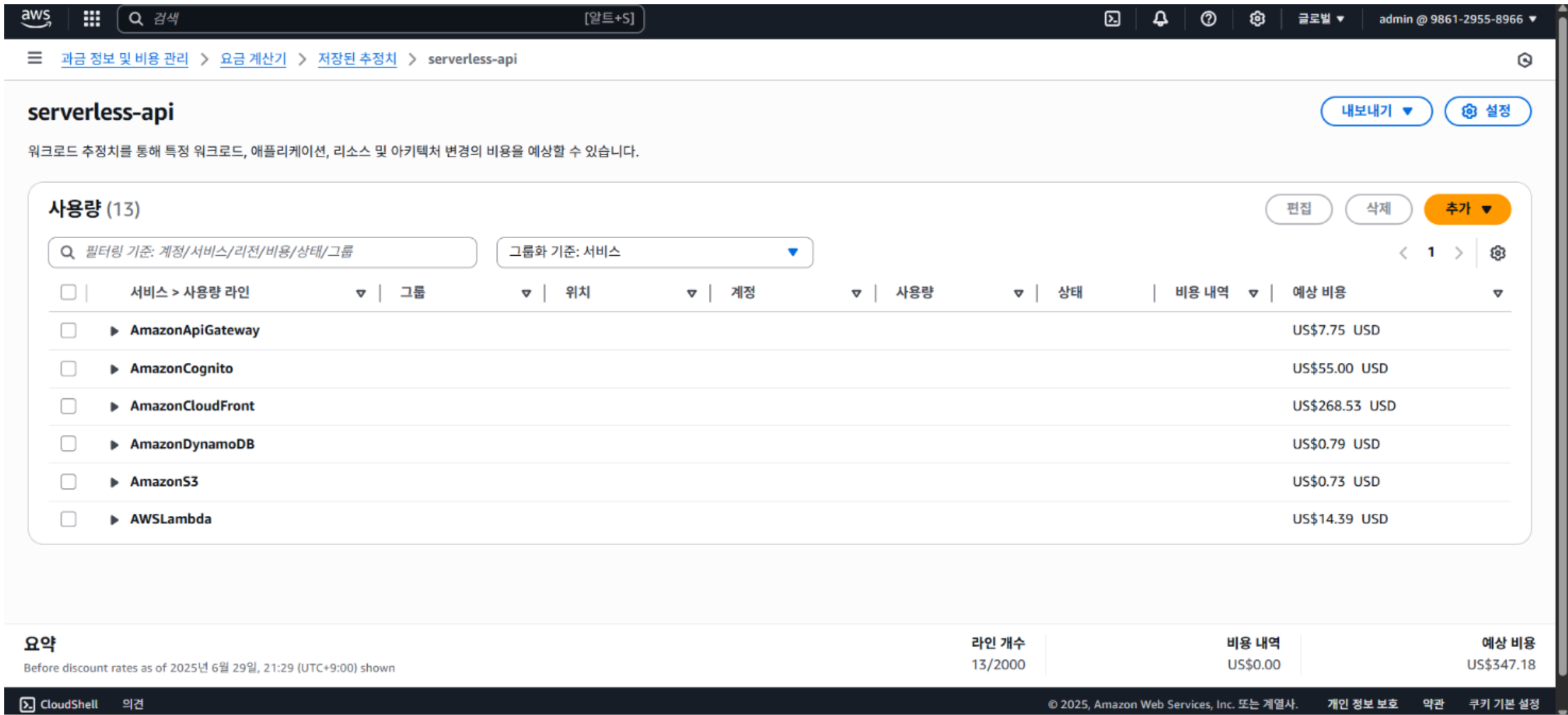
+ CDN, 거기에 WAF(방화벽) 등으로 보안 솔루션 까지 제공

☐	▼ AWSDataTransfer							US\$252.00 USD
☐	APN2-DataTransfer-Out-Bytes	test	Asia Pacifi...	98612955...	2000 GB	✓ 신규	-	US\$252.00 USD

(S3 → 인터넷 직접 전송 시 요금, 현재 상황에선 거의 비슷함)

TCO(Total Cost of Ownership)

TCO(Total Cost of Ownership) = 인프라를 구축하고 운영하는 데 드는 비용 ≈ 총 비용



총 347.18달러, 한화 약 47만 2,506원이 부과됨 (API 백엔드 요금만 계산한다면 77.93달러, 약 10만 5천원 정도)

TCO(Total Cost of Ownership)

물론 기능이 매우 적고 이미지 CDN 등의 기능이 들어가면 더 비싸지긴 할거임

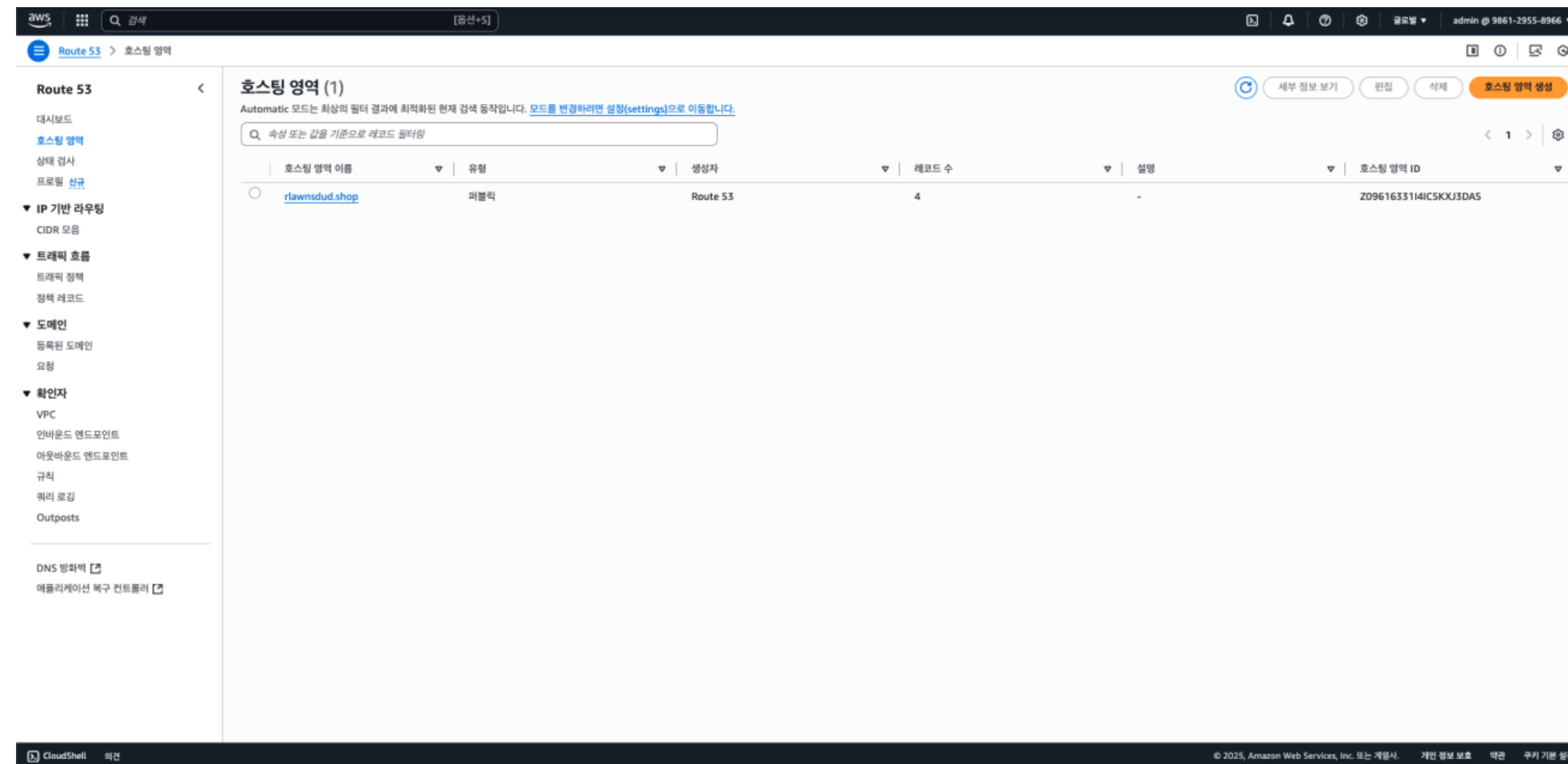
다만 서버리스 FaaS 구조라 요청된 수 + 실행 시간 만큼 과금되고

자주 사용하지 않는 서비스나 스타트업의 확장성 등이 중요한 서비스에선 서버리스가 유리할 수 있다는 것이 핵심

Cleaning AWS Services

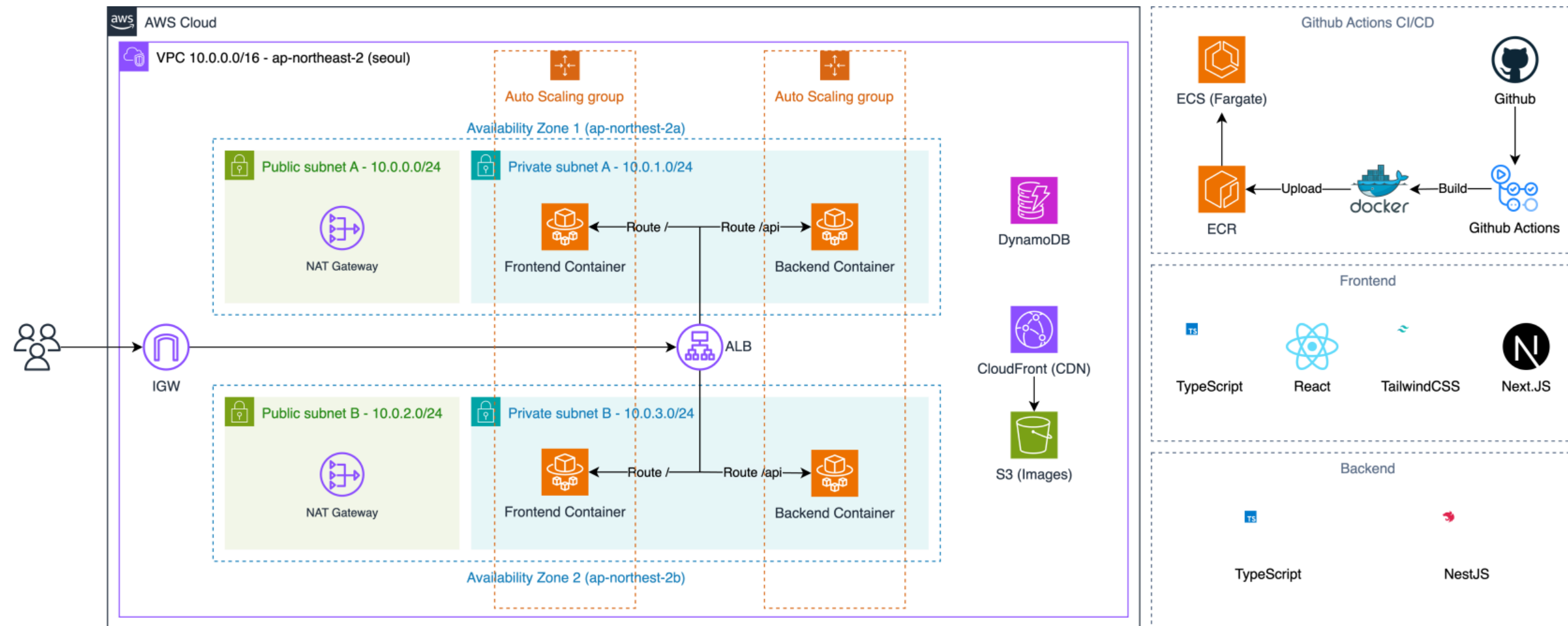
1. Serverless Framework 정리 (Lambda, API Gateway, IAM, CloudWatch, S3 등)
⇒ CloudFormation 스택 삭제
2. **DeletionPolicy**가 적용된 리소스 정리 (Cognito, DynamoDB)
3. 프론트엔드 리소스 정리 (S3, CloudFront)
4. (옵션) 깃허브 레포지토리 삭제

(번외) Route53



도메인 구매 후 Route53에서 설정 가능, 도메인도 깔끔해지고 CORS 문제도 쉽게 해결됨
(도메인도 구매해뒀는데 까먹고 발표 내용에 포함을 안시킴)

(번외) ECS(또는 EKS)



다른 서버리스엔 ECS(또는 EKS)의 Fargate 등이 있음
(위 아키텍처는 유연화 주간에 했던 프로젝트의 배포 구조)

Conclusion & Insights

3줄 요약

- 실험용 또는 자주 안쓰는 API 서비스 구축할 때, 구축만 해두고 필요 시 사용하는 식으로 운용하면 매우 효율적일 듯
- 솔직히 실제 백엔드 구축 시엔 IaaS(EC2) 등의 Serverful 환경에서 구축하는게 정신 건강에 좋을 듯
 1. Cold Start 관리, 서버리스로 인한 아키텍처 설계 등이 오히려 은근 까다로움
 2. 서버리스 개발 프레임워크/라이브러리가 비교적 빈약함
 3. 생태계가 작아서 뭘 검색하려해도 자료가 거의 없음
 4. AWS에 너무 의존하게 됨 (단점?)
- 이번 발표를 준비하면서 서버리스, 람다, AWS SDK 등의 기술에 대해 자세히 공부해보는 계기가 되었음

Thanks For Your Listening

Q&A

Source of the images:

- IaaS vs. PaaS vs. SaaS - RedHat Technology Topics
- <https://github.com/cat-milk/Anime-Girls-Holding-Programming-Books>
- 그 외엔 직접 제작 - 아키텍처: draw.io 사용 (저작권: 김준영)

Links:

- Vlog - <https://velog.io/@yulmwu/aws-serverless>
- Github - <https://github.com/yulmwu/aws-lambda-example>