

SERVERLESS Architectures ON AWS

SECOND EDITION

Peter Sbarski

Yan Cui

Ajay Nair

MANNING



AWS의 서비스 아키텍처

두 번째 에디션

PETER SBARSKI, YAN CUI, AJAY NAIR



매닝
헬퍼 아일랜드

이 책과 다른 Manning 책의 온라인 정보 및 주문에 대해서는 다음을 방문하십시오.

www.manning.com. 발행인은 이 책을 수량으로 주문하면 할인을 제공합니다.

자세한 내용은 문의하십시오

특별영업부
매닝출판사
20 볼드윈 로드
사서함 761
쉘터 앤드, NY 11964
이메일: orders@manning.com

©2022 Manning Publications Co. 판권 소유.

발행인의 사전 서면 승인 없이 이 발행물의 어떤 부분도 전자, 기계, 복사 또는 기타 방법으로 복제하거나 검색 시스템에 저장하거나 전송할 수 없습니다.

제조업체와 판매자가 제품을 구별하기 위해 사용하는 많은 명칭은 상표권을 주장합니다. 이러한 명칭이 책에 나와 있고 Manning Publications가 상표권 주장을 인지한 경우 명칭은 이니셜 대문자 또는 모두 대문자로 인쇄되었습니다.

☞ 기록된 내용을 보존하는 것의 중요성을 인식하고 우리가 출판하는 책을 무산지에 인쇄하는 것이 Manning의 정책이며 우리는 이를 위해 최선을 다하고 있습니다.
지구의 자원을 보존해야 하는 우리의 책임을 인식하여 Manning 책은 원소 염소를 사용하지 않고 최소한 15% 재활용 및 처리된 종이에 인쇄됩니다.

저자와 출판사는 출판 당시 이 책의 정보가 정확한지 확인하기 위해 모든 노력을 기울였습니다. 저자와 발행인은 오류나 누락으로 인한 손실, 손상 또는 중단에 대해 책임을 지지 않으며 이에 따라 그러한 오류나 누락이 과실, 사고 또는 기타 원인 또는 여기에 있는 정보.

 매닝출판사
20 Baldwin Road
PO Box 761 Shelter
Island, NY 11964

개발 편집자: 토니 아리톨라
기술 개발 편집자: Brent Stains
리뷰 편집자: Aleksandar Dragosavljević
프로덕션 에디터: Andy Marinkovich
카피 에디터: Frances Buran
교정자: Jason Everett
기술 교정자: Niek Palm
조판공: Gordan Salinovic
표지 디자이너: Marija Tudor

ISBN 9781617295423

미국에서 인쇄

컴퓨팅에 대한 나의 열정을 항상 지지하고 격려해 주신 엄마, 아빠
에게.

—피터 스바스키

항상 저를 지지하고 격려하고, 심야 코딩 세션을 참아주는 아내에게.

— 얀 추이

제 아내, 제 아이들, 제 남동생, 그리고 제 부모님에게 이 일을 할 수 있는 목적과
시간을 주신 것에 대해 감사드립니다.

—아제이 나이어

간략한 내용

| | |
|-----------------------------------|--|
| 1 부 첫 번째 단계 | 1 |
| 1 ■ 서버리스로 전환 3 | 2 ■ 서버 |
| 리스로의 첫 단계 18 | 3 ■ 아키텍처 및 패턴 |
| 40 | |
| 2 부 사용 사례 | 55 |
| 4 ■ Yubl: 아키텍처 하이라이트, 배운 교훈 57 | 5 ■ A Cloud Guru: 아키텍처 하이라이트, 배운 교훈 70 |
| 6 ■ Yle: 아키텍처 하이라이트, 배운 교훈 84 | |
| 파트 3 실습 | 97 |
| 7 ■ 임시 작업을 위한 스케줄링 서비스 구축 99 | 8 ■ 서비스 병렬 컴퓨팅 설계 132 |
| 9 ■ Code Developer University 146 | |
| 4 부 미래 | 165 |
| 10 ■ 블랙벨트 람다 167 | |
| 11 ■ 새로운 관행 183 | |

내용물

머리말 xiii 감사
의 말 xv 이 책에 대한 xviii

저자에 대해 xx
표지 삽화에 대해 xxii

1 부 첫 번째 단계 1

1 서비스로 전환 3

1.1 이름에 무엇이 들어 있습니까? 4

1.2 서비스 아키텍처의 이해 5 서비스 지향 아키텍처와 마
이크로서비스 7 ■ 기존 방식의 아키텍처 구현 7 ■ 서비스 방식
의 아키텍처 구현 9

1.3 서비스 전환 요청 11 1.4 서비스 장단점 14

1.5 이 두 번째 판의 새로운 기능은 무엇입니까? 16

2 서비스를 위한 첫 번째 단계 18

2.1 비디오 인코딩 파이프라인 구축 19 AWS 사용에 대
한 간략한 설명 19 ■ Amazon Web Services(AWS) 사용 20

2.2 시스템 준비 21

- 시스템 설정 22 ■ IAM(Identity and Access Management) 작업 22
- 버킷을 만들어 봅시다. 25 ■ IAM 역할 생성 26 ■ AWS Elemental MediaConvert 사용 28
- MediaConvert 역할 사용 29

2.3 서비스 프레임워크로 시작하기 29

- Serverless Framework 설정 29 ■ 24시간 비디오로 Serverless Framework 가져오기 31 ■ 첫 번째 Lambda 함수 생성 33

2.4 AWS 36에서 테스트

2.5 로그 보기 37

3 아키텍처 및 패턴 40

3.1 사용 사례 40

- 백엔드 컴퓨팅 41 ■ 사물인터넷(IoT) 41 ■ 데이터 처리 및 조작 42 ■ 실시간 분석 42
- 레거시 API 프록시 43 ■ 예약 서비스 44 ■ 봇 및 기술 44 ■ 하이브리드 44

3.2 패턴 45

- GraphQL 45 ■ 명령 패턴 46 ■ 메시징 패턴 47 ■ 우선 순위 큐 패턴 49 ■ 팬아웃 패턴 50
- 풀로 계산 51 ■ 파이프 및 필터 패턴 52

파트 2 사용 사례 55

4 Yubl: 아키텍처 하이라이트, 배운 교훈 57

4.1 원래 Yubl 아키텍처 58

- 확장성 문제 59 ■ 성능 문제 59 ■ 긴 기능 제공 주기 59 ■ 왜 서비스 인가? 60

4.2 새로운 서비스 Yubl 아키텍처 61

- 재설계 및 재작성 62 ■ 새로운 검색 API 62

4.3 새로운 마이크로서비스로의 순조로운 마이그레이션 64

5 A Cloud Guru: 아키텍처 하이라이트, 교훈 70

5.1 원래 아키텍처 71

- 43개 마이크로서비스로의 여정 75 ■ GraphQL이란 77
- GraphQL로 이동 79 ■ 서비스 검색 80 ■ 보안 BFF 세계 82

5.2 유산의 잔재 82

6 Yle: 아키텍처 하이라이트, 배운 교훈 84

6.1 Fargate 85로 대규모 이벤트 수집

- 비용 고려 사항 85 ■ 성능 고려 사항 85 6.2 실시간 이벤트 처리 86 Kinesis Data Streams 86 ■ SQS 배달 못한 편지 대기열(DLQ) 87 라우터 Lambda 함수 88 ■ Kinesis Data Firehose 88 Kinesis Data Analytics 89 ■ 통합 90

6.3 배운 교훈 91

- 서비스 한계 파악 91 ■ 실패를 염두에 두고 구축 93 일괄처리가 비용과 효율성에 좋다 94 ■ 비용추정이 까다롭다 95

파트 3 실습 97

7 임시 작업을 위한 스케줄링 서비스 구축 99

- 7.1 비기능 요구 사항 정의 101 7.2 EventBridge를 사용한 Cron 작업 102 귀하의 점수 104 ■ 당사 점수 105 ■ 솔루션 조정 107 ■ 최종 생각 109

7.3 다이나모DB TTL 109

- 당신의 점수 110 ■ 우리의 점수 111 ■ 결승 생각 113

7.4 단계 함수 113 당신의 점수

- 115 ■ 우리의 점수 115 ■ 솔루션 조정 116 ■ 최종 생각 119 7.5 SQS 119 당신의 점수 120 ■ 우리의 점수 120 ■ 최종 생각 122

7.6 DynamoDB TTL과 SQS 122 결합

- 당신의 점수 123 ■ 우리의 점수 124 ■ 최종 생각 125

- 7.7 귀하의 애플리케이션에 적합한 솔루션 선택 125 7.8 애플리케이션 125 귀하의 가중치 126 ■ 당사의 가중치 126 ■ 각 애플리케이션에 대한 솔루션 채점 128

8 서버리스 병렬 컴퓨팅 설계 132

8.1 맵리듀스 소개 133

- 비디오를 트랜스코딩하는 방법 134 ■ 아키텍처 개요 135

| | |
|--------------------|-------------|
| 8.2 아키텍처 심층 분석 137 | |
| 상태 유지 138 | ■ 단계 기능 141 |
| 8.3 대체 아키텍처 144 | |

9 코드 개발자 대학 146

| | |
|----------------|--|
| 9.1 솔루션 개요 147 | |
|----------------|--|

| | |
|----------------------|--------------------------------|
| 나열된 요구 사항 147 | ■ 솔루션 아키텍처 148 |
| 9.2 코드 스코어링 서비스 150 | |
| 제출 대기열 152 | ■ 코드 점수 서비스 요약 153 |
| 9.3 학생 프로필 서비스 153 | |
| 학생 점수 업데이트 기능 155 | |
| 9.4 분석 서비스 157 | |
| Kinesis Firehose 158 | ■ AWS Glue 및 Amazon Athena 160 |
| 퀵사이트 163 | |

| | |
|--------------|-----|
| 4 부 미래 | 165 |
|--------------|-----|

10 블랙벨트 람다 167

| | |
|-------------------------|--|
| 10.1 어디에서 최적화할 것인가? 167 | |
|-------------------------|--|

| | |
|----------------------------|-------------------------|
| 10.2 시작하기 전에 169 | |
| Lambda 함수가 요청을 처리하는 방법 169 | ■ 대기 시간: 콜드 vs. 웰 173 |
| 함수 및 애플리케이션의 로드 생성 173 | ■ 성능 및 가용성 추적 174 |
| 10.3 대기 시간 최적화 176 | |
| 배포 아티팩트 크기 최소화 176 | ■ 실행 환경에 충분한 리소스 할당 178 |
| 기능 로직 최적화 179 | |

| | |
|--------------|--|
| 10.4 동시성 180 | |
|--------------|--|

| | |
|------------------------------|--|
| 요청, 대기 시간 및 동시성 간의 상관 관계 181 | |
| 동시성 관리 181 | |

11 새로운 관행 183

| | |
|-------------------------|--|
| 11.1 여러 AWS 계정 사용하기 184 | |
|-------------------------|--|

| | |
|-----------------------------------|---------------------------|
| 보안 위반 격리 184 | ■ 공유 서비스 제한에 대한 경합 제거 185 |
| 네터링 개선 185 | ■ 팀의 자율성 향상 185 |
| AWS Organizations를 위한 코드형 인프라 186 | ■ 비용 모 |

| | |
|---|------------------------|
| 11.2 임시 스택 사용하기 186 | |
| 공통 AWS 계정 구조 186 | ■ 기능 분기에 임시 스택 사용 187 |
| ■ e2e 테스트에 임시 스택 사용 188 | |
| 11.3 환경 변수에서 일반 텍스트의 민감한 데이터를 피하십시오 188 | |
| 공격자는 여전히 침입할 수 있습니다 189 | ■ 민감한 데이터를 안전하게 처리 189 |
| 11.4 이벤트 기반 아키텍처에서 EventBridge 사용 190 | |
| 콘텐츠 기반 필터링 190 | ■ 스키마 검색 191 |
| ■ 이벤트 아카이브 및 재생 191 | ■ 추가 대상 192 |
| ■ 토플로지 192 | |
| 부록 A 서비스 아키텍처를 위한 서비스 195 | |
| 부록 B 클라우드 설정 200 | |
| 부록 C 배포 프레임워크 212 | |
| 인덱스 225 | |

1 부

첫 번째 단계

서비스 전환

이 장에서는 다음을 다룹니다.

기존 시스템 및 애플리케이션 아키텍처
서비스 아키텍처의 주요 특징 및 이점

서비스 아키텍처와 마이크로서비스가 그림에 포함되는 방식

서버에서 서비스로 전환할 때 고려 사항

이 두 번째 판의 새로운 기능은 무엇입니까?

소프트웨어 개발자에게 소프트웨어 아키텍처가 무엇인지 묻는다면 답을 얻을 수 있습니다.
"그것은 청사진 또는 계획"에서 "개념적 모델", "큰 그림"에 이르기까지 다양합니다. 이 책은 최근에 등장한 새로운 아키텍처 접근 방식에 관한 것입니다.
전 세계의 개발자와 회사에서 채택하여 현대적인
애플리케이션 - 서비스 아키텍처.

서비스 아키텍처는

애플리케이션 아키텍처 접근 방식. 개발자에게 다음과 같이 반복할 수 있는 기능을 약속합니다.
비즈니스에 중요한 대기 시간, 가용성, 보안 및
개발자 측에서 최소한의 노력으로 성능을 보장합니다.
이 책은 확장 및 확장할 수 있는 서비스 시스템에 대해 생각하는 방법을 알려줍니다.
프로비저닝하거나

단일 서버를 관리합니다. 중요하게도 이 책은 오늘날의 클라우드 플랫폼이 제공하는 서비스와 아키텍처를 사용하여 높은 수준의 품질과 성능을 유지하면서 개발자가 제품을 시장에 신속하게 제공하는 데 도움이 되는 기술을 설명합니다.

1.1 이름에 무엇이 들어 있습니까?

더 진행하기 전에 서비스라는 단어를 이해하는 것이 중요하다고 생각합니다. AWS의 공식 시도를 포함하여 이미 다양한 시도가 있습니다 (<https://aws.amazon.com/serverless/>). Martin Fowler의 커뮤니티 즐겨 찾기 (<https://martinfowler.com/articles/serverless.html>). 정의하는 방법은 다음과 같습니다.

정의 서비스는 유ти리티 서비스로 사용되어야 하고 사용 시에만 비용이 발생하는 모든 소프트웨어 또는 서비스 오퍼링에 적용할 수 있는 한정자입니다.

충분히 간단하죠? 그러나 그 간단한 정의에는 풀어야 할 것이 많이 있습니다. 서비스를 호출하는 데 필요한 다음 두 가지 기준을 각각 자세히 살펴보겠습니다.

유티리티 서비스로 소비 - "서비스로서의 소프트웨어" 소비 모델은 잘 알려져 있습니다. 이는 소프트웨어를 사용하는 모든 사람이 규정된 API(응용 프로그래밍 인터페이스) 또는 웹 인터페이스를 사용하여 소프트웨어를 사용하고 맞춤화하는 동시에 API에 대한 사용 정책 및 소프트웨어에 대해 게시된 제약 조건을 유지함을 의미합니다. Salesforce, Office365 및 Google Maps는 서비스로 제공되는 잘 알려진 소프트웨어 패키지입니다. 여기서 핵심은 소프트웨어를 호스팅하고 API를 구동하는 실제 인프라(서버, 네트워킹, 스토리지 등)가 소비자로서 완전히 추상화된다는 것입니다. 보이는 모든 것(그리고 중요한 모든 것)은 API가 허용하는 것입니다.

서비스는 일반적으로 서비스 공급자의 가용성, 안정성 및 성능 보증과 함께 제공됩니다. 또한 유티리티 서비스에는 유티리티 컴퓨팅 제품에서 기대할 수 있는 청구 특성이 있습니다. 즉, 예약, 구독 또는 프로비저닝이 아닌 사용량에 대해 비용을 지불합니다. 기존의 모든 퍼블릭 클라우드 제품에는 관련된 유티리티 청구 형식이 있습니다. 예를 들어 Amazon Elastic Compute Cloud(EC2)를 사용하면 가상 머신 임대료를 초 단위로 지불할 수 있습니다.

사용할 때만 비용 발생 - 이는 소프트웨어를 배포하고 사용할 준비를 하는 데 드는 비용이 전혀 없음을 의미합니다. 이것을 전기 및 수도와 같은 공공 시설에서 기대하는 것과 동일한 비용 모델이라고 생각하십시오. 소비자로서 사용하는 경우 세분화된 사용 단위 비용을 지불하지만 사용하지 않으면 0을 지불합니다. 순수한 사용량 기반 가격 책정의 이러한 측면은 이전에 제공된 다른 유티리티 서비스와 서비스 제품을 구별하는 기준입니다.

이 책의 나머지 부분에서는 이러한 기준에 맞는 소프트웨어에 대해서만 "서비스" 한정자를 사용할 것입니다. 예를 들어 Apache 웹 서버와 같은 웹 사이트를 호스팅하기 위해 서버를 제공해야 하는 소프트웨어는 첫 번째 기준을 충족하지 않기 때문에 적합하지 않습니다. 서비스로 제공되지만 구독으로 지불해야 하는 소프트웨어(예: Salesforce)는 두 번째 조건을 충족하지 않기 때문에 자격이 없습니다.

오해를 풀기 위해...

일반적인 오해 중 하나는 "서비스"의 "-less"가 다음을 의미한다는 것입니다.
 "없음 또는 없음"("무설탕", "빼 없는" 등으로 생각)
 애플리케이션 아키텍처가 주장할 수 있는 방법에 대한 소셜 미디어의 다채로운 토론
 서버 없이 실행합니다. 여기서 "-less"는 "사용 상황에서 보이지 않는"을 의미한다고 생각합니다.
 ("무선", "무미"라고 생각하십시오). 분명히 어디기에 서버가 있습니다! 차이점은 이러한 서버가 사용자에게 숨겨져 있다는 것입니다. 할 수 있는 인프라가 없습니다.
 기본 운영 체제 또는 가상 하드웨어를 조정할 방법이 없고 생각할 방법이 없습니다.
 구성. 다른 사람이 인프라의 핵심 세부 사항을 처리합니다.
 관리를 통해 운영 오버헤드를 없애고 고객에게 환원합니다.
 가장 비싼 상품은 시간입니다.

표준. 서비스 아키텍처는 확장하여 전적으로 서비스 아키텍처로 구성된 아키텍처입니다.

구성 요소. 그러나 아키텍처의 어떤 구성 요소가 서비스여야 합니까?

그렇게 불려? 다음은 예를 들어 살펴보겠습니다.

1.2 서비스 아키텍처 이해

시스템과 다르지 않은 일반적인 데이터 기반 웹 애플리케이션의 예를 들어 보겠습니다.

오늘날 대부분의 웹 지원 소프트웨어를 지원합니다. 이들은 일반적으로 백엔드로 구성됩니다.

(서버) 클라이언트의 요청을 수락한 다음 요청을 처리합니다.

백엔드 서버는 다양한 형태의 계산을 수행하고 프론트엔드 클라이언트는 사용자가 브라우저, 모바일 또는 데스크톱을 통해 작동할 수 있는 인터페이스를 제공합니다.

장치. 데이터는 예 저장되기 전에 수많은 애플리케이션 계층을 통해 이동할 수 있습니다.

데이터베이스. 그런 다음 백엔드는 JSON 형식일 수 있는 응답을 생성합니다.

또는 완전히 렌더링된 마크업에서 클라이언트로 다시 전송됩니다(그림 1.1). 이러한 종류의 애플리케이션은 일반적으로 계층 (제어하는 프레젠테이션 계층)으로 설계됩니다.

정보가 캡처되어 사용자에게 제공되는 방식, 애플리케이션 계층

애플리케이션의 비즈니스 로직과 데이터베이스가 있는 데이터 계층을 제어합니다.

해당 액세스 제어).

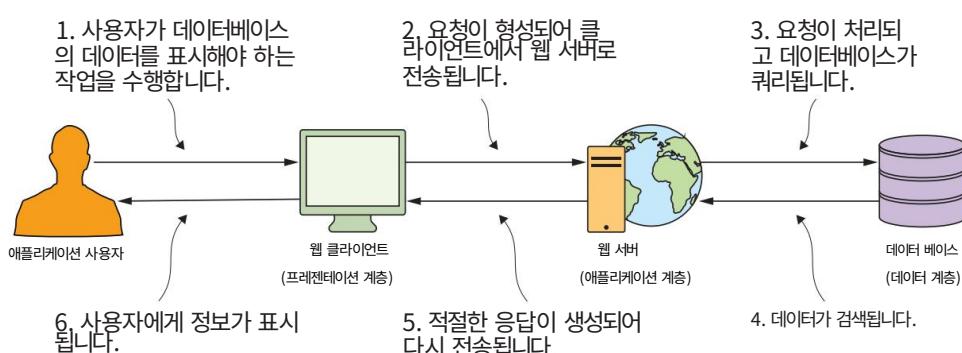


그림 1.1 대부분의 개발자에게 익숙한 기본 요청-응답(클라이언트-서버) 메시지 교환 패턴. 이 그림에는 하나의 웹 서버와 하나의 데이터베이스만 있습니다. 대부분의 시스템은 훨씬 더 복잡합니다.

소프트웨어 아키텍처는 메인프레임에서 실행되는 코드의 시대에서 다음으로 진화했습니다. 프리젠테이션, 데이터 및 애플리케이션/로직 계층이 있는 다중 계층 아키텍처 전통적으로 분리되어 있습니다. 각 계층 내에는 다음을 처리하는 여러 논리적 계층이 있을 수 있습니다. 기능 또는 도메인의 특정 측면. 크로스컷팅도 있고 로깅 또는 예외 처리 시스템과 같은 구성 요소는 여러 영역에 걸쳐 있을 수 있습니다. 레이어링에 대한 선호도는 이해할 수 있습니다. 레이어링을 통해 개발자는 우려를 분리하고 유지 관리가 더 쉬운 응용 프로그램을 갖습니다. 그림 1.2는 API, 비즈니스를 포함한 여러 레이어가 있는 계층화된 아키텍처의 예 논리, 사용자 인증 구성 요소 및 데이터베이스.

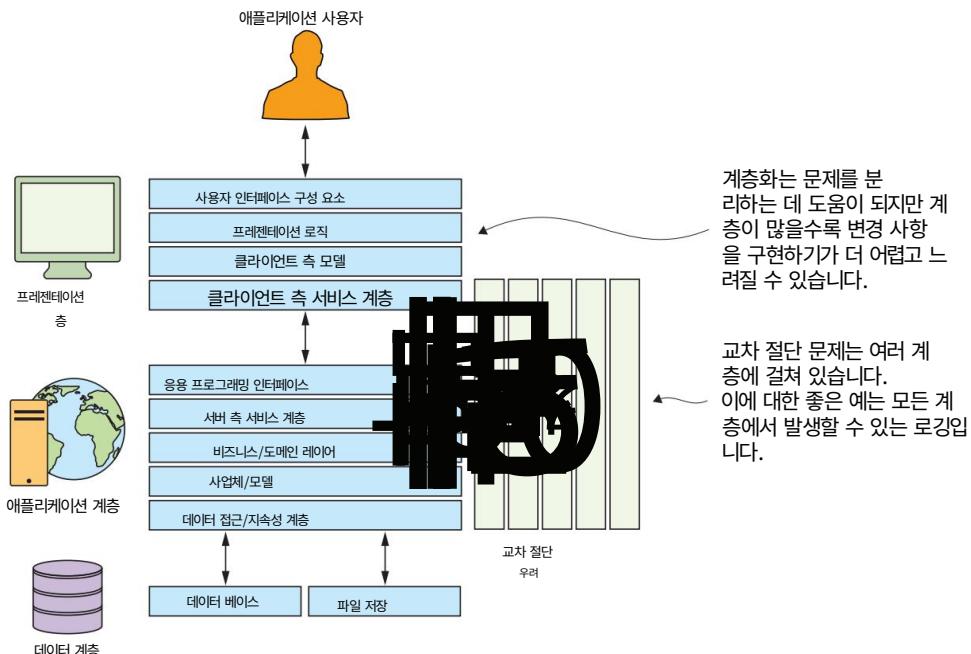


그림 1.2 일반적인 3계층 응용 프로그램은 일반적으로 프레젠테이션, 응용 프로그램 및 데이터 계층으로 구성됩니다. 계층에는 특정 책임이 있는 여러 계층이 있을 수 있습니다.

계층 대 계층

레이어와 레이어의 차이점에 대해 개발자들 사이에 약간의 혼란이 있습니다. 계층. 계층은 주요 구성 요소 간의 경계를 제공하는 모듈 경계입니다. 시스템의 예를 들어 사용자에게 표시되는 프레젠테이션 계층은 별도입니다. 비즈니스 로직을 포함하는 애플리케이션 계층에서. 차례로, 데이터 계층 데이터에 대한 액세스를 관리, 유지 및 제공하는 또 다른 별도의 시스템입니다.

계층으로 그룹화된 구성 요소는 물리적으로 서로 다른 인프라에 상주할 수 있습니다.

계층은 응용 프로그램에서 특정 책임을 수행하는 논리적 조각입니다. 각 계층에는 도메인 서비스와 같은 다양한 기능 요소를 담당하는 여러 계층이 있을 수 있습니다.

1.2.1 서비스 지향 아키텍처와 마이크로서비스

한 가지 무뚝뚝한 접근 방식은 모든 계층(API, 비즈니스 로직, 사용자 인증)을 하나의 모놀리식 코드 기반으로 만듭니다. 다음과 같이 들릴 수 있습니다. 오늘날의 반패턴이지만 그것은 실제로 우리가 초기에 채택한 접근 방식이었습니다. 클라우드 기반 개발. 그러나 대부분의 현대적인 접근 방식에서는 재사용 가능성, 자율성, 구성 가능성 및 검색 가능성을 염두에 두고 설계해야 합니다.

우리 업계의 베테랑들 사이에서 SOA(서비스 지향 아키텍처)는 잘 알려진 유행어입니다. SOA는 개발자가 메시지 전달을 통해 통신하고 종종

메시지가 생성되거나 교환되는 방식을 정의하는 스키마 또는 계약.

서비스 지향 접근 방식의 현대적 구현은 종종 다음과 같이 언급됩니다.

마이크로서비스 아키텍처. 최신 애플리케이션 아키텍처는 이벤트 및 API를 통해 통신하는 서비스와 적절하게 삽입된 비즈니스 로직으로 구성됩니다. 우리는 마이크로서비스를 소규모의 독립형 완전 독립형 서비스로 정의합니다.

특정 비즈니스 목적이나 능력에 관한 것입니다. 이상적으로는 마이크로서비스가 적절한 프레임워크와 언어로 작성된 각 서비스로 쉽게 교체할 수 있습니다.

마이크로서비스가 다른 범용 언어나 DSL(도메인 특정 언어)로 작성될 수 있다는 사실만으로도 많은 개발자에게 그림 카드가 됩니다.

올바른 언어 또는 전문화된 라이브러리 세트를 사용하면 이점을 얻을 수 있습니다.

일을 위해. 각 마이크로 서비스는 상태를 유지하고 데이터를 저장할 수 있습니다. 그리고 만약 마이크로서비스가 올바르게 분리되어 있으면 개발 팀이 서로 독립적으로 마이크로서비스를 작업하고 배포할 수 있습니다. 애플리케이션을 구축하고 배포하는 이러한 접근 방식은

느슨하게 결합된 서비스 모음은 오늘날 클라우드 개발에 대한 기본 접근 방식으로 간주됩니다(원하는 경우 "클라우드 네이티브" 접근 방식).

항상 마이크로서비스를?

마이크로서비스 접근 방식이 장미꽃 침대만 있는 것은 아닙니다. 다양한 언어와 프레임워크는 지원하기 어려울 수 있으며 엄격한 규율 없이는 혼란을 초래할 수 있습니다. 최종 일관성, 조정, 발견 및 복잡한 오류

복구는 마이크로서비스 세계에서 문제를 어렵게 만들 수 있습니다.

소프트웨어 엔지니어링은 항상 트레이드오프의 게임입니다. 뭔가 유행하고 있기 때문에 (마이크로서비스와 같은) 모든 문제와 사용 사례에 보편적으로 적합하지는 않습니다. 중요한 것은 다양한 아키텍처 옵션에 대해 알고 이해하는 것입니다.

그들의 장단점, 그리고 중요하게는 요구 사항과 요구 사항을 이해하는 것
자신의 문제. (그리고 예, 어떤 경우에는 단일체를 갖는 것이 좋습니다.)

1.2.2 기존 방식의 아키텍처 구현

애플리케이션을 어떻게 구성할지 결정하고 모든 각 계층에 필요한 소프트웨어가 준비되어 있으면 가장 어렵다고 생각할 것입니다. 부분이 완료되었습니다. 사실, 그 때 좀 더 복잡한 작업이 시작됩니다. 원하는 서비스를 개발하려면 전통적 으로 데이터 센터 또는

관리, 유지, 패치 및 백업이 필요한 클라우드. 오늘 당신은 몇 가지 옵션 중에서 선택합니다.

VM에 직접 구축 - 각 서비스의 물리적 배포에는 다음이 필요합니다.

다음과 같은 필수 활동을 처리하기 위해 추가 작업이 포함된 인스턴스 집합이 있습니다.

로드밸런싱, 트랜잭션, 클러스터링, 캐싱, 메시징 및 데이터 중복. 이러한 서버의 프로비저닝, 관리 및 패치 작업에는 많은 시간이 소요됨

종종 전담 운영 인력이 필요한 작업입니다.

사소하지 않은 환경은 효과적으로 설정하고 운영하기 어렵습니다. 인프라 구조와 하드웨어는 모든 IT 시스템의 필수 구성 요소이지만

종종 핵심 초점이 되어야 하는 것, 즉 비즈니스 문제를 해결하는 것에서 주의가 산만해집니다. 간단한 웹 애플리케이션 예제에서는 다음을 수행해야 합니다.

분산 시스템 및 클라우드 인프라 구축의 전문가가 되십시오.

관리. 클라우드 환경에서 이러한 형태의 컴퓨팅은 종종 서비스로서의 인프라(IaaS)로.

PaaS 사용 - 지난 몇 년 동안 PaaS(Platform as a Service)와 같은 기술

(PaaS) 및 컨테이너는

일관성 없는 인프라 환경, 총돌 및 서버 관리

간접비. PaaS는 사용자에게 플랫폼을 제공하는 클라우드 컴퓨팅의 한 형태입니다.

기반 인프라의 일부를 숨기면서 소프트웨어를 실행합니다.

PaaS를 효과적으로 사용하려면 개발자는 다음을 대상으로 하는 소프트웨어를 작성해야 합니다.

플랫폼의 기능 및 기능. 레거시 애플리케이션 이동

PaaS 서비스에 대해 독립 실행형 서버에서 실행되도록 설계된 대부분의 PaaS 구현의 일시적인 특성으로 인해 추가 개발 노력이 필요한 경우가 많습니다. 그래도 선택권이 주어지면 많은 개발자가 이해할 수 있습니다.

덕분에 보다 전통적인 수동 솔루션 대신 PaaS를 사용하도록 선택

유지 관리 및 플랫폼 지원 요구 사항 감소.

컨테이너 사용 - 컨테이너화는 마이크로서비스 아키텍처에 이상적인 것으로 간주됩니다.

애플리케이션을 자체 환경과 격리하는 방법이기 때문입니다.

기존 클라우드에서 구현되는 본격적인 가상화에 대한 경량 대안입니다.

서버 사용.

컨테이너는 특히 우수한 배포 및 패키징 솔루션입니다.

증속성이 작동할 때(자신의 하우스 키핑 문제와 복잡성이 발생할 수 있음에도 불구하고). 컨테이너는 격리되어 가볍고,

그러나 공용 또는 사설 클라우드 또는 현장에서.

이러한 각 모델은 완벽하게 유효하고 다양한 수준의 단순성을 제공하지만

서비스 개발 속도와 비용은 여전히 수명주기에 의해 좌우됩니다.

애플리케이션 사용이 아닌 소유한 인프라 또는 서버. 구매하시면

데이터 센터에 랙을 설치하면 연중무휴로 비용을 지불합니다. 클라우드 인스턴스(래핑된)

PaaS 또는 실행 중인 컨테이너 또는 기타) 웹 앱에 대한 트래픽을 제공하는지 여부에 관계없이 실행될 때 비용을 지불합니다.

이는 서버 효율성 향상에 투자하거나 인프라 수명 주기를 애플리케이션 사용 및 서버 크기에 맞추는 데 투자하는 엔지니어의 전체 분야로 이어집니다.

트래픽 패턴에. 이것은 또한 이러한 작업에 들인 모든 노력이 시간이 걸린다는 것을 의미합니다.

기능을 개선하고 애플리케이션의 측면을 차별화하는 것과는 거리가 멀니다.

이것은 기기를 꽂을 장소를 요구하고 비용을 지불해야 하는 것과 같습니다.

유저에게 회사의 발전기 공유 및 원하는 위상, 주파수 및 전력량에 관계없이 전력을 전달하도록 발전기 구성

얼마나 사용하는지. 실제 결과(기기에 연결)

필요한 기본 시설(발전기)에 대한 노력과 비용. 여기는

서비스 접근 방식이 등장합니다. 유저에게 접근 방식과 도덕적으로 동등한 것을 목표로합니다.

우리는 오늘을 알고 사랑합니다. 필요할 때 복잡성이 추상화되고

사용할 때만 비용을 지불하십시오.

1.2.3 서비스 방식의 아키텍처 구현

샘플 애플리케이션을 위한 서비스 아키텍처는 다음과 같이 구성될 수 있습니다.

레이어. 예를 들어 API를 빌드하기 위해 API 호출이 없으면 비용이 전혀 들지 않는 서비스를 사용합니다. 인증 서비스를 구축하기 위해 인증 호출이 없는 경우 비용이 전혀 들지 않는 서비스를 사용합니다. 구축하려면

스토리지 서비스를 사용합니다... 당신은 그림을 얻을.

Lego에 가상 인프라를 제공한 퍼블릭 클라우드 접근 방식과 매우 유사합니다.

초기에 클라우드 스택을 조립하면 서비스 아키텍처는 AWS와 같은 클라우드 공급자의 기존 서비스를 사용하여 아키텍처 구성 요소를 구현합니다. 예를 들어 AWS는 API(Amazon

API Gateway), 워크플로(AWS Step Functions), 대기열(Amazon Simple Queue Service), 데이터베이스(Amazon DynamoDB 및 Amazon Aurora) 등.

기성 서비스를 사용하여 아키텍처의 일부를 구현한다는 아이디어는

새것이 아닌; 실제로 SOA 시대부터 모범 사례였습니다. 예제로 변경된 사항

지난 몇 년 동안 우리 애플리케이션의 맞춤형 측면도 구현할 수 있었습니다.

(비즈니스 로직처럼) 서비스 방식으로. 서비스로 실행하거나 인프라 비용을 지불하기 위해 인프라를 프로비저닝 할 필요 없이 임의의 코드를 실행할 수 있는 이러한 기능을 서비스로서의 기능(FaaS)이라고 합니다.

FaaS를 사용하면 원하는 성능 및 액세스 제어 특성을 나타내는 사용자 지정 코드, 관련 종속성 및 일부 구성은 제공할 수 있습니다.

그런 다음 FaaS는 보이지 않는 컴퓨팅 플랫폼에서 이 단위(함수라고 함)를 실행합니다.

자체 디스크가 있는 격리된 환경을 수신하는 코드를 실행할 때마다

메모리 및 CPU 할당. 코드가 실행된 시간에 대해서만 비용을 지불합니다. 기능은

경량 인스턴스가 아닙니다. 대신 OS의 프로세스와 유사하다고 생각하십시오.

애플리케이션에서 필요로 하는 만큼 생성한 다음

애플리케이션이 실행되고 있지 않습니다.

서비스 아키텍처는 실제로 현재 진행 중인 변화의 정점입니다.

모놀리스에서 서비스로, 인프라 관리에서

점점 더 차별화되지 않은 책임을 위임합니다. 서비스 아키텍처

레이어링 문제와 너무 많은 것을 업데이트해야 하는 문제를 해결하는 데 도움이 될 수 있습니다. 거기 개발자가 시스템을 기능으로 나누고 프론트엔드가 서비스와 안전하게 통신할 수 있도록 하여 계층화를 제거하거나 최소화할 수 있는 여지

직접 데이터베이스. 잘 계획된 서비스 아키텍처는 미래의 변경을 더 쉽게 만들 수 있으며 이는 모든 장기 애플리케이션에 중요한 요소입니다.

요약하자면, 서비스 아키텍처는 각각에 대해 서비스 구현을 활용합니다.

사용자 지정 논리에 FaaS(예: AWS Lambda)를 사용하여 구성 요소 중 이것은 각각의 구성 요소는 사용할 때만 비용이 발생하는 유tility 가격으로 서비스로 구축됩니다.

각 구성 요소는 서비스이며 실행 중인 인프라 구조와 관련된 구성이나 비용을 노출하지 않습니다. 즉, 이러한 아키텍처는 직접 액세스에 의존하지 않습니다.

작동하는 서버에. 다양하고 강력한 단일 목적 API와 웹을 활용하여

서비스를 통해 개발자는 느슨하게 결합되고 확장 가능하며 효율적인 아키텍처를 구축할 수 있습니다. 빠르게. 서버 및 인프라 문제에서 벗어나

개발자가 주로 코드에 집중하는 것이 서비스의 궁극적인 목표입니다.

FaaS에 대한 추가 정보

AWS의 FaaS 오퍼링은 AWS Lambda라고 하며 주요

클라우드 제공업체. Lambda는 이 마을의 유일한 게임이 아닙니다. 마이크로소프트 애저 기능 (<http://bit.ly/2DWx5Gn>), IBM 클라우드 기능 (<http://bit.ly/2l1PWbd>), 그리고

구글 클라우드 기능 (<http://bit.ly/2CbzOem>) 다른 FaaS 서비스입니다. 보고 싶어.

많은 개발자는 서비스를 AWS Lambda와 같은 FaaS 제품과 결합합니다.

종종 컨테이너 또는 서비스 채택과 관련하여 혼란스러운 논쟁으로 이어집니다.

그것들이 정말로 컨테이너나 기능을 의미할 때. Apex 프레임워크의 창시자인 TJ Hallowaychuk이 서비스의 정의를 좋아 합니다. 그는 한때 트위터에

"서비스 != 기능, FaaS == 기능, 서비스 == 온디맨드 확장 및

가격 특성(기능에 국한되지 않음). 우리는 더 이상 동의할 수 없었습니다.

떠오르는 트렌드는 서비스 컨테이너입니다. 즉, 컨테이너를 활용하여

기능 대신 사용자 정의 로직을 구현하고 컨테이너를 유tility 서비스로 사용하고 컨테이너가 실행될 때만 비용이 발생합니다.

AWS Far gate 또는 Google Cloud Run과 같은 서비스가 이 기능을 제공합니다. 둘의 차이점

(함수 대 컨테이너)는 개발자가 전환하려는 정도일 뿐입니다.

공유 책임의 경계. 컨테이너를 사용하면

사용자 공간 라이브러리 및 네트워크 기능. 컨테이너는

기존 서버 기반/VM 모델, 손쉬운 패키징 및 배포 모델 제공

당신의 애플리케이션 스택을 위해. 여전히 운영 체제를 정의해야 합니다.

요구 사항, 원하는 언어 스택 및 코드 배포에 대한 종속성,

인프라 복잡성의 일부를 계속 수행해야 함을 의미합니다. 목적을 위해

이 책에서는 사용자 지정 논리에 FaaS를 사용하는 데 중점을 둘 것입니다.

동일한 서비스 컨테이너의 사용을 탐색할 수 있습니다.

1.3 서비스로 전환하기 위해 호출하기

우리가 살펴본 웹 애플리케이션 예제는 가장 간단한 데모 중 하나입니다.

서비스 아키텍처로 달성할 수 있는 것입니다. 서비스 접근 방식도

혁신하고 빠르게 움직이기를 원하는 조직에 매우 적합합니다.

일반적으로 기능과 서비스 아키텍처는 다양합니다. 당신은 그들을 사용할 수 있습니다

CRUD 애플리케이션, 전자 상거래, 백오피스 시스템, 복잡한 백엔드 구축

웹 앱, 모든 종류의 모바일 및 데스크톱 소프트웨어. 몇 주가 걸리던 작업

기술의 올바른 조합을 선택하기만 하면 며칠 또는 몇 시간 안에 완료할 수 있습니다. Lambda 함수는 상태 비저장 및 확장 가능하므로

병렬 처리의 이점을 얻는 모든 논리 구현.

가장 유연하고 강력한 서비스 설계는 이벤트 기반입니다.

아키텍처의 각 구성 요소는 상태 변경 또는 일부 알림에 반응합니다.

요청에 응답하거나 정보를 풀링하는 대신 친절합니다. 2장에서,

예를 들어 이벤트 기반 푸시 기반 파이프라인을 구축하여

비디오를 다른 비트 전송률과 형식으로 인코딩하는 시스템을 구성합니다.

[참고](#) 이벤트를 통신 메커니즘으로 사용하는 방법을 찾을 수 있습니다.

서비스 아키텍처에서 반복되는 주제가 될 구성 요소 사이;

실제로 AWS Lambda의 초기 출시는 이벤트 기반 컴퓨팅 서비스였습니다. 이벤트 중심의 푸시 기반 시스템을 구축하면 종종 비용이 절감되고

복잡성(변경 사항을 풀링하기 위해 추가 코드를 실행할 필요가 없음) 및

잠재적으로 전반적인 사용자 경험을 더 부드럽게 만듭니다. 그것은 말할 것도 없다

이벤트 중심의 푸시 기반 모델이 좋은 목표이지만

모든 상황에서 적절하거나 달성할 수 있는 것은 아닙니다.

서비스 아키텍처를 통해 개발자는 소프트웨어 디자인과 코드에 집중할 수 있습니다.

인프라보다. 확장성과 고가용성은 더 쉽게 달성할 수 있으며 사용한 만큼만 비용을 지불하기 때문에 가격이 더 공정 한 경우가 많습니다. 더 중요한 것은 당신이

최소화함으로써 시스템의 복잡성을 어느 정도 줄일 수 있는 가능성입니다.

레이어의 수와 필요한 코드의 양.

애플리케이션 개발에 서비스 접근 방식을 채택하면

민첩성, 탄력성 및 비용 효율성 향상. 그러나 모든 애플리케이션에 대해 서비스 접근 방식을 채택하려는 함정에 빠지기 쉽습니다. 몇 가지를 유지하는 것이 좋습니다.

서비스 여정을 시작할 때 염두에 두어야 할 원칙:

리프트 앤 시프트 방지 - 실제로 서비스 아키텍처는 새로운

기존 응용 프로그램을 포팅하는 대신 응용 프로그램. 이는 기존 애플리케이션 코드 베이스에 중복되는 코드가 많기 때문입니다.

서비스 서비스. 예를 들어 Java Spring 앱을 Lambda로 이식하면

기능에 대한 무거운 프레임워크, 대부분은 웹과 상호 작용하기 위해 존재합니다.

서버(Lambda 내부에 존재하지 않음).

서비스 전용 접근 방식이 아닌 서비스 우선 접근 방식을 채택하십시오.

서비스 전용 접근 방식을 채택한 A Cloud Guru와 같은 회사는

애플리케이션의 100%가 서비스 구현으로 실행될수록

Expedia 및 T-Mobile과 같은 회사가 채택한 광범위한 접근 방식은 먼저 서비스로 전환합니다. 이것이 의미하는 바는 개발자가 먼저 빌드를 시도한다는 것입니다.

다음 우선 순위에 따른 모든 새 응용 프로그램: 다음을 사용하여 최대한 많이 빌드

타사 서비스, AWS Lambda와 같은 AWS 서비스 프리미티브를 사용하여 구축된 사용자 지정 서비스로 대체, 마지막으로 사용자 지정을 사용하여 구축된 사용자 지정 서비스로 대체

EC2와 같은 인프라에서 실행되는 소프트웨어. 우리는 당신이 왜 그런지에 대해 이야기합니다.

다음 섹션에서 사용자 지정 서비스 서비스를 넘어 대체해야 할 수도 있습니다.

전부 또는 전무일 필요는 없습니다. 서비스 접근 방식의 한 가지 장점은

기존 애플리케이션을 점차적으로 서비스 아키텍처로 전환할 수 있습니다.

개발자가 모놀리식 코드 기반에 직면하면 점차적으로 이를 애매하게 할 수 있습니다.

개별 구성 요소를 분리하고 서비스 구현으로 변환합니다(

교살자 패턴).

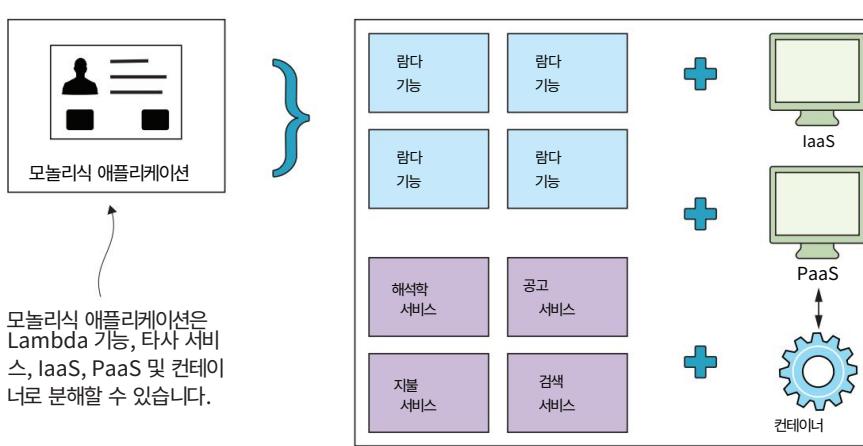
가장 좋은 방법은 초기에 프로토타입을 만들어 시스템이 부분적으로 또는 완전히 서비스를 덜 사용하는 경우 시스템이 어떻게 작동할지에 대한 개발자 가정을 테스트하는 것입니다. 레거시 시스템에는 창의성을 요구하는 흥미로운 제약 조건이 있는 경향이 있습니다.

대규모 아키텍처 리팩터와 마찬가지로

불가피하게 이루어지게 됩니다. 시스템은 결국 하이브리드가 될 수 있지만(그림 1.3에서와 같이), 일부 구성 요소가 Lambda 및

변경되지 않은 레거시 아키텍처를 유지하는 대신 타사 서비스

더 이상 확장되지 않거나 실행하는 데 값비싼 인프라가 필요합니다.



기술 조합은 요구 사항
과 제약 조건에 따라 달라야 합니다.
그러나 기술이 많을수록 더 많은
오버헤드, 시간 및 에너지가 필요합니다.

컨테이너, PaaS, IaaS, Lambda 기능 및 서비스는 서로 통신
할 수 있습니다. 이러한 기술의 조합을 사용하여 시스템을 설계
한 경우 이벤트 오케스트레이션이 발생하는 방식을 고려해야 합니다.

그림 1.3 서비스 아키텍처는 전부 아니면 전무(all-or-nothing) 제안이 아닙니다. 현재 모놀리식 애플리케이션이 있는 경우 점진적으로 구성 요소를 출하하여 격리된 서비스 또는 컴퓨팅 기능에서 실행할 수 있습니다. 도움이 되는 경우 모놀리식 애플리케이션을 다양한 IaaS, PaaS, 컨테이너, 기능 및 타사 서비스로 분리할 수 있습니다.

레거시 서버 기반 애플리케이션에서 확장 가능한 서비스 아키텍처로 전환하는 데 시간이 걸릴 수 있습니다. 신중하고 천천히 접근해야 하며, 개발자는 사전에 훌륭한 테스트 계획과 훌륭한 DevOps 전략을 마련해야 합니다. 그들은 시작합니다.

NoOps는 어떻습니까?

초기에 서비스 기술 및 아키텍처에 대한 첫 번째 회의 즈음 (<https://serverlessconf.io>) 2016년에는 서비스 기술이 NoOps 시대를 예고했다는 이야기가 있었습니다. 어떤 사람들은 덕분에 믿었습니다.

서비스를 사용하면 기업은 더 이상 인프라 운영에 대해 생각할 필요가 없습니다.

클라우드 벤더가 모든 것을 처리해 줄 생각이었다. 그 가정, 그

NoOps는 실제 사건이었고 그렇지 않은 것으로 판명되었습니다.

서비스 애플리케이션을 구축하고 실행할 때 DevOps 엔지니어는

이제 그들은 다른 초점을 가지고 있다는 점을 제외하고는 필수적입니다. 그들의 관심은 배포에 있습니다.

자동화, 테스트 및 선호하는 클라우드 공급자의 운영/지원 팀과의 작업(서버 조정 및 운영 체제 패치 대신).

기업은 더 작고 전문화된 DevOps 팀으로 벗어날 수 있습니다. 하지만,

작업을 완전히 무시하는 것은 재앙의 지름길입니다.

그렇지 않으면). 귀하의 애플리케이션이 실패하면 고객이 귀하에게 책임을 묻고,

클라우드 공급자가 아니므로 준비하고 적절한 사람과 프로세스를 준비하십시오.

서비스 지향 아키텍처에 적합한 애플리케이션 선택 - 서비스 아키텍처

SOA에서 제작된 아이디어의 자연스러운 확장입니다. 서비스 아키텍처에서는 모든

사용자 지정 코드는 AWS Lambda와 같은 컴퓨팅 서비스에서 실행되는 격리되고 독립적이며 종종 세 분화된 기능으로 작성 및 실행됩니다. 왜냐하면

모든 구성 요소는 서비스이며 서비스 아키텍처는 많은 이점을 공유합니다.

이벤트 기반 마이크로서비스 아키텍처의 복잡성. 이것은 또한 의미합니다

이러한 요구 사항을 충족하도록 애플리케이션을 설계해야 할 수 있습니다.

접근 방식(예: 개별 서비스를 상태 비저장으로 만들기).

서비스 접근 방식은 모두 양을 줄이는 것입니다.

더 빠르게 반복하고 혁신할 수 있도록 소유하고 유지해야 하는 코드의 양이 많습니다.

이는 구성 요소의 수를 최소화하기 위해 노력해야 함을 의미합니다.

애플리케이션을 빌드하는 데 필요합니다. 예를 들어 웹을 설계할 수 있습니다.

(복잡한 백엔드 대신) 풍부한 프론트엔드와 대화할 수 있는 애플리케이션

타사 서비스를 직접 제공합니다. 이러한 종류의 아키텍처는 더 나은 사용자 경험에 도움이 될 수 있습니다. 온라인 리소스 간의 흡 감소 및 대기 시간 감소

응용 프로그램의 성능과 유용성을 더 잘 인식하게 됩니다. 즉, FaaS를 통해 모든 것을 라우팅할 필요가 없습니다. 당신의

프론트엔드는 검색 공급자, 데이터베이스 또는 다른 유용한 API와 직접 통신할 수 있습니다.

또한 모놀리식 접근 방식에서 더 많은

분산 서비스 접근 방식은 복잡성을 자동으로 줄이지 않습니다.

기본 시스템의 솔루션의 분산 특성은 다음을 유발할 수 있습니다.
 프로세스 종이 아닌 원격으로 만들어야 하기 때문에 자체적인 문제
 호출 및 네트워크 전반에 걸친 장애 및 지연을 처리해야 하는 필요성,
 애플리케이션은 탄력적이어야 합니다.

맞춤형 코드 최소화 - 서비스의 등장은 많은 표준 애플리케이션을 의미합니다.
 API, 워크플로, 대기열 및 데이터베이스와 같은 구성 요소는 클라우드 공급자 및 타사에서 서버리
 스 제품으로 사용할 수 있습니다. 예 훨씬 더 유용합니다.
 개발자가 자신의 도메인에 고유한 문제를 해결하는데
 다른 사람이 이미 구현한 기능을 다시 만드는 것입니다. 다음을 위해 빌드하지 마십시오.
 실행 가능한 타사 서비스 및 API를 사용할 수 있는지 확인하기 위한 것입니다. 서다
 새로운 높이에 도달하기 위해 개인의 어깨에.

부록 A에는 Amazon Web Services 및 비Amazon Web의 짧은 목록이 있습니다.
 유용하다고 생각한 서비스. 이러한 서비스의 대부분은 다음에서 자세히 살펴보겠습니다.

우리가 책을 통해 이동할 때 세부 사항. 그러나 다음과 같은 경우는 말할 것도 없습니다.
 가격, 기능, 가능성,
 문서 및 지원을 신중하게 평가해야 합니다.

사용자 정의 기능을 구축해야 하는 경우 조언은 간단합니다.
 먼저 기능을 사용하여 문제를 해결하고 작동하지 않는 경우 탐색
 두 번째는 컨테이너 및 보다 전통적인 서버 기반 아키텍처입니다. 개발자
 읽기 및 쓰기와 같은 거의 모든 일반적인 작업을 수행하는 기능을 작성할 수 있습니다.
 데이터 소스에 쓰기, 다른 기능 호출, 계산 수행. 더 복잡한 경우 개발자는 더 정교한 파이프라인을
 설정할 수 있습니다.
 여러 기능의 호출을 조정합니다.

1.4 서비스 장단점

서비스를 신속하게 조합하여 애플리케이션을 구축하는 서비스 접근 방식은 두 가지 중요한 이점을 제공합
 니다.
 우리 애플리케이션에 대한 활동당 가격. 이는 민첩성의 파괴적인 이득으로 해석됩니다.
 및 개발자 생산성, 개발과 재무 간의 훨씬 더 간소화된 조정(애플리케이션의 비효율성 또는 최적화가

직접적이고 실질적인 재정적 영향). 다음은 서비스 아키텍처를 채택하여 실현할 몇 가지 구체적인 이점입니
 다.

서버 관리 없이 높은 확장성과 안정성 - 대규모 분산 시스템을 구축하는 것은 어렵습니다. 서버 구
 성 및 관리 등의 업무,
 패치 및 유지 관리는 공급업체가 관리하고 관리합니다.
 시간과 비용을 절약하는 높은 확장성과 안정성을 위한 인프라 아키텍처
 돈. 예를 들어 Amazon은
 파워 AWS 랍다.

서버를 관리하거나 수정하기 위한 특정 요구 사항이 없는 경우
 그런 다음 Amazon 또는 다른 공급업체가 리소스를 관리하도록 하는 것이 좋습니다.
 해결책. 귀하는 자신의 코드에 대해서만 책임을 지며 운영 및
 다른 유능한 손에 관리 작업을 할당합니다.

경쟁력 있는 가격 - 기존 서버 기반 아키텍처에는 다음과 같은 서버가 필요합니다.
항상 전체 용량으로 실행할 필요는 없습니다. 자동 시스템을 사용하는 경우에도 확장에는 새 서버
가 필요하며 트래픽이나 새 데이터가 일시적으로 급증할 때까지 종종 낭비됩니다.

서비스 시스템은 확장과 관련하여 훨씬 더 세분화되어 있으며 특히 피크 로드가 고르지 않거나
예상치 못한 경우에 비용 효율적입니다. 그들의 때문에
사용량에 따른 유tility 요금 청구, 서비스 서비스는 매우 비용 효율적일 수 있습니다. 그러나 기존
(서버 및 컨테이너) 기술보다 저렴하지 않습니다.
모든 상황. 가장 좋은 것은 시작하기 전에 약간의 모델링을 수행하는 것입니다.
큰 프로젝트.

적은 코드 - 이 장의 시작 부분에서 서비스 아키텍처에 대해 언급했습니다.
보다 전통적인 시스템과 비교하여 복잡성과 코드를 일부 줄일 수 있는 기회를 제공합니다. 서비스
접근 방식을 채택하면
서버 집합을 오케스트레이션하는 데 필요한 것과 같은 미분화된 코드 또는
놀랍게도 구성 요소 간의 라우팅 요청 및 이벤트
현대 코드 베이스의 많은 부분.

그러나 서비스는 모든 상황에서 은총알이 아닙니다. 다음은 몇 가지 이유입니다.
서비스 아키텍처를 피하려는 경우:

퍼블릭 클라우드 기반 아키텍처에 익숙하지 않습니다. 서비스 개발
클라우드 기반 개발로의 이동의 자연스러운 확장입니다.
그리고 더 많은 미분화된 무거운 작업이 공급자에게 이전됩니다. 거기
자체 유지 관리가 필요한 애플리케이션 및 비즈니스 시나리오
데이터 센터; 이러한 경우에는 서비스 아키텍처를 구축할 수 없습니다(하지만
인프라에서 자체 프리미티브를 호스팅하고 사용할 수 있습니다.
응용 프로그램을 빌드하는 것).
서비스가 가용성, 성능, 규정 준수 또는 확장 요구 사항을 충족하지 않습니다.
고객. AWS 서비스 서비스는 가용성 SLA를 제공하지만 임계값
귀하의 비즈니스에 필요한 것보다 낮을 수 있습니다. 또한 다양한 규정 준수 인증을 보유하고 있지
만 비즈니스에 필요한 것이 무엇인지 확인해야 합니다.
필요. AWS Lambda와 같은 서비스도 성능 SLA를 제공하지 않습니다.
원하는 수준에 대해 성능을 평가해야 할 수도 있음을 의미합니다.
비 AWS, 타사 서비스는 동일한 보트에 있습니다. 일부는 강할 수 있습니다
SLA가 있는 반면, SLA가 전혀 없는 경우도 있습니다.
애플리케이션과 비즈니스에 더 많은 제어가 필요하거나 인프라를 사용자 정의해야 합니다. Lambda
와 관련하여 Amazon을 통해 얻은 효율성
플랫폼을 돌보고 기능을 확장하려면
운영 체제를 사용자 지정하거나 기본 인스턴스를 조정합니다. 함수에 할당된 RAM의 양을 수정하고
시간 초과를 변경할 수 있지만 이는
그것에 대해. 유사하게, 다른 제3자 서비스는 다양한 수준의 맞춤화 및 유연성을 갖습니다.

애플리케이션 및 비즈니스 요구 사항에 따라 공급업체에 구애받지 않아야 합니다. 개발자가 AWS를 포함한 타사 API 및 서비스를 사용하기로 결정한 경우 아키텍처가 사용 중인 플랫폼과 강력하게 결합될 가능성이 있습니다.

공급업체 종속의 의미와 회사 생존, 데이터 주권 및 개인 정보 보호, 비용, 지원, 문서 및 사용 가능한 기능 세트를 포함한 타사 서비스 사용의 위험을 철저히 고려해야 합니다.

이 장에서는 서비스 아키텍처가 무엇인지 배우고 그 원칙을 살펴보고 기존 아키텍처와 어떻게 비교하는지 살펴보았습니다. 다음 장에서는 작은 서비스 이벤트 기반 애플리케이션을 만들어 손을 더럽힐 것입니다. 이 접근 방식을 처음 시도하는 경우 서비스에 대한 좋은 취향을 얻는 데 도움이 됩니다. 거기에서 우리는 중요한 아키텍처 패턴을 탐색하고 서비스 아키텍처가 문제를 해결하는 데 사용되는 사용 사례에 대해 논의할 것입니다.

1.5 이 두 번째 판의 새로운 기능은 무엇입니까?

모든 의도와 목적을 위해 이 책은 AWS의 Serverless Architectures 초판과는 완전히 다른 책입니다. 대부분의 챕터는 초판과 완전히 다른 경험을 제공하기 위해 처음부터 작성되었습니다.

이 책의 초판이 2017년에 나왔을 때 서비스는 아직 생소했고 우리 중 많은 사람들이 처음으로 서비스에 대해 배우고 있었습니다. 따라서 초판에서는 서비스에 대해 부드럽게 소개하고 독자에게 서비스 애플리케이션 빌드를 안내했습니다. 그 이후로 AWS에서 서비스 기술을 시작하는 데 도움이 되는 수많은 책과 비디오 과정을 포함하여 많은 새로운 교육 콘텐츠가 우리 책상을 넘었습니다.

AWS의 서비스 아키텍처에 대한 소개를 찾고 계시다면 2장과 부록 A 및 B에 일부 소개 콘텐츠가 포함되어 있습니다. Manning 웹 사이트 (<https://www.manning.com/books/serverless-architectures-on-aws>) 초판의 내용 대부분은 오늘날에도 여전히 관련이 있으며, 이 책을 통해 처음부터 서비스 애플리케이션을 구축하는 방법을 배우게 될 것입니다.

그러나 서비스 기술을 통해 비즈니스를 차별화하는 요소에 집중할 수 있는 것처럼 이 책을 이 2판에서 차별화할 수 있는 요소에 집중하고 싶습니다. 이 책은 서비스에 대한 또 다른 시작 가이드 대신 서비스 사용 사례와 흥미로운 아키텍처에 중점을 둡니다. 이미 서비스 기술에 대한 경험이 있는 개발자를 대상으로 하며 많은 분들이 우리에게 묻는 질문에 답합니다. 초점이 맞춰진 스위치를 감안할 때 이 책에는 실제 코드 샘플이 많지 않습니다. 대신 서비스 아키텍처에 대한 생각에 도전하고 서비스 기술을 최대한 활용할 수 있기를 바랍니다.

AWS에서.

요약

클라우드는 IT 인프라의 게임 체인저였으며 앞으로도 계속 그럴 것입니다.

및 소프트웨어 개발.

소프트웨어 개발자는 사용을 극대화할 수 있는 방법에 대해 생각할 필요가 있습니다.

클라우드 플랫폼을 통해 경쟁 우위를 확보할 수 있습니다.

서버리스 아키텍처는 개발자와 조직이 생각하고 연구하고 채택할 수 있는 최신 단계입니다. 구현의 이 흥미진진한 변화

소프트웨어 개발자가 AWS Lambda와 같은 컴퓨팅 서비스를 수용함에 따라 애플리케이션 아키텍처가 빠르게 성장할 것입니다.

대부분의 경우 서비스 애플리케이션은 실행 비용이 저렴하고 구현 속도가 빠릅니다. 또한 인프라 실행 및 기존 소프트웨어 개발 수행과 관련된 복잡성과 비용을 줄여야 합니다.

시스템.

인프라 유지 관리에 소요되는 비용 및 시간 감소 및

확장성의 이점은 조직과 개발자가 서비스 아키텍처를 고려하는 좋은 이유입니다.

서비스를 위한 첫 단계

이 장에서는 다음을 다룹니다.

AWS Lambda 함수 작성 및 배포

Simple Storage Service와 같은 AWS 서비스
(S3) 및 Elemental MediaConvert

서비스 프레임워크를 사용하여 서비스 구성 및 배포

서비스 아키텍처에 대한 이해를 돋기 위해

소규모 이벤트 기반 서비스 애플리케이션: 특히 비디오 인코딩 파이프라인.

귀하의 서비스는 S3 버킷에 업로드된 동영상을 기준 동영상에서 트랜스코딩합니다.

형식, 해상도 또는 비트 전송률을 다른 형식 또는 비트 전송률(예: YouTube
프론트엔드 웹사이트 없이만).

이 비디오 인코딩 파이프라인을 구축하려면 AWS Lambda, S3 및 Elemental MediaConvert
를 사용합니다. 나중에 원하는 경우 이를 중심으로 프론트엔드를 구축할 수 있습니다.

그러나 우리는 그것을 연습으로 남겨 둘 것입니다. 우리가 어떻게 해왔는지 알고 싶다면 프론트엔드를 자세히 다룬
초판을 참조하세요.

2.1 비디오 인코딩 파이프라인 구축 이 섹션에서는 작은

이벤트 기반 서비스 애플리케이션 구축을 시작합니다. 이 장에서 높은 수준에서 다음 내용을 배웁니다. Lambda를 포함한 세 가지 AWS 서비스를 사용하여 기본적인 서비스 아키텍처를

구성하는 방법 서비스 프레임워크를 사용하여 서비스 애플리케이션을 구성 및 배포하는 방법 실행, 디버그 및 테스트 방법 구축한 서비스 파이프라인

서비스용 프레임워크

서비스 애플리케이션을 구성하고 배포하는 데 사용할 수 있는 몇 가지 프레임워크가 있다는 것을 들어보셨을 것입니 다. 여기에는 AWS 서비스 애플리케이션 모델 (<https://github.com/awslabs/serverless-application-model>) 이 포함됩니다. 서비스 프레임워크 (<https://serverless.com>), 성배 (<https://github.com/aws/chalice>), 그리고 몇 가지 다른 이 장에서는 서비스 프레임워크를 사용하여 서비스 애플리케이션 배포를 구성하고 자동화합니다.

우리의 조언은 항상 서비스 프레임워크 또는 서비스 애플리케이션 모델(SAM)과 같은 프레임워크를 사용하는 것입니다. 서비스 아키텍처의 원칙을 이해하면 프레임워크가 수행하는 모든 작업을 비약적으로 가속화합니다. 부록 C에는 유용하다고 생각한 다양한 프레임워크에 대한 추가 정보가 포함되어 있습니다. 이 장에서 사용할 Serverless Framework에 대한 소개와 약간의 입문서도 있습니다. 기회가 되면 부록 C를 살펴보십시오.

구축하려는 이벤트 기반 파이프라인에 대해 이야기해 보겠습니다(24시간 비디오라고 함). 파이프라인은 지정된 S3 버킷에 업로드된 비디오를 다양한 형식, 해상도 및 비트 전송률로 인코딩합니다. 전체 프로세스가 이벤트 기반이기 때문에 파일이 S3에 업로드되면 시스템이 자동으로 트리거하여 파일을 처리하고 별도의 버킷에 인코딩이 다른 새 버전을 생성합니다. 모든 것이 자동으로 수행되기 때문에 사용자를 대신하여 개입할 필요가 없습니다.

2.1.1 AWS 비용에 대한 간단한 참고 사항

대부분의 AWS 서비스에는 프리 티어가 있습니다. 이 예를 따르면 대부분의 AWS 서비스의 프리 티어 내에 있어야 합니다. 그러나 AWS Elemental MediaConvert는 결국 약간의 비용이 들 수 있는 서비스 중 하나입니다. MediaConvert를 사용하여 비디오 파일을 트랜스코딩합니다. 이 서비스는 선불 비용 없이 사용한 만큼만 지불합니다. 가격은 MediaConvert가 생성하는 새 비디오의 지속 시간만을 기준으로 하며 10 초 단위로 요금이 청구됩니다.

MediaConvert는 주문형 서비스에 대해 Basic 및 Professional의 두 가지 가격 계층을 제공합니다. 이 책의 기본 계층을 사용하게 되지만 응용 프로그램을 다음 단계로 끌어올리려는 경우 전문가 계층을 조사하도록 초대합니다(다음 YouTube를 구축하게 된다면 저희를 기억하세요!). 기본 계층은 다음과 같은 기능을 지원합니다.

단일 패스 인코딩, 클리핑, 스티칭 및 오버레이. 전문가 계층은 다음을 지원합니다.
몇 가지 기능이 더 있습니다.

기본 계층의 분당 속도는 해상도와 프레임 속도에 따라 다릅니다.
원하는 출력의. 기본 SD 품질 출력의 경우 분당 \$0.0075부터
UHD 출력의 경우 분당 \$0.0450입니다. 이 비율은 또한 귀하가 거주하는 지역에 따라 다릅니다.
예를 들어, 미국 동부 1(버지니아 북부)은 미국 서부 1(캘리포니아 북부)보다 저렴하지만 이 책 전체에서 사용할
지역은 미국 동부 1입니다. 당신은 볼 수 있습니다
계층 및 가격 정보는 <https://aws.amazon.com/mediaconvert/pricing/>에서 확인할 수 있습니다.
MediaConvert에는 무료 등급이 없으므로 거의 즉시 비용을 지불하게 됩니다.

S3 프리 티어를 통해 사용자는 표준 스토리지로 5GB의 데이터를 저장할 수 있습니다.
20,000개의 GET 요청과 2,000개의 PUT 요청이 있으며 매월 15GB의 데이터를 전송합니다.
Lambda는 1백만 개의 무료 요청과 400,000GB초의 컴퓨팅 시간이 있는 프리 티어를 제공합니다. 해당 서비스
의 무료 계층 제한 내에 있어야 합니다. 다음은 24시간 비디오에 대한 높은 수준의 요구 사항을 나열합니다 . 트
랜스코딩 프로세스는 업로드된 소스 비디오를 세 가지 다른 해상도 및 비트 전송률로 변환합니다.

- 16 × 9 종횡비 및 1920 × 1080 해상도의 6Mbps
 - 16 × 9 종횡비 및 1280 × 720 해상도의 4.5Mbps
 - 4 × 3 종횡비 및 640 × 480 해상도의 1.5Mbps
- 두 개의 S3 버킷이 있습니다.
- 원본 파일은 업로드 버킷으로 이동합니다.
 - AWS MediaConvert에서 생성된 파일은 트랜스코딩된 비디오에 저장됩니다.
버킷.

일을 더 쉽게 관리할 수 있도록 다음을 사용하여 빌드 및 배포 시스템을 설정합니다.
노드 패키지 관리자(npm) 및 서비스 프레임워크. 먼저, 여기
이 예에서 사용할 AWS 서비스에 대한 개요입니다.

2.1.2 아마존 웹 서비스(AWS) 사용

서비스 백엔드를 생성하려면 AWS에서 제공하는 여러 서비스를 사용합니다. 이것들
파일 저장을 위한 S3(Simple Storage Service), 비디오 변환을 위한 MediaConvert, 시스템의 핵심 부분
을 조정하고 사용자 지정 코드를 실행하기 위한 Lambda가 포함됩니다.
이 장에서는 MediaConvert 작업을 시작하는 첫 번째 Lambda 함수를 생성합니다.
다음은 우리가 사용할 각 AWS 서비스에 대한 간략한 설명입니다.

S3는 스토리지 서비스를 제공합니다. Amazon S3는 업로드된 파일과 새로
트랜스코딩된 비디오.

Lambda는 조정이 필요하거나 직접 수행할 수 없는 시스템 부분을 처리합니다.
다른 서비스로. 이 기능은 파일이 S3에 업로드될 때 자동으로 실행됩니다.
버킷.

MediaConvert는 비디오를 다양한 해상도와 비트 전송률로 인코딩합니다. 기본 사전 설정
사용자 지정 인코딩 프로필을 만들 필요가 없습니다.

그림 2.1은 제안된 접근 방식의 세부 흐름을 보여줍니다. 사용자가 시스템과 상호 작용해야 하는 유일한 지점은 초기 업로드 단계입니다. 이 그림과 아키텍처가 복잡해 보일 수 있지만 이 장의 과정에서 시스템을 관리 가능한 덩어리로 나누고 하나씩 다룰 것입니다.

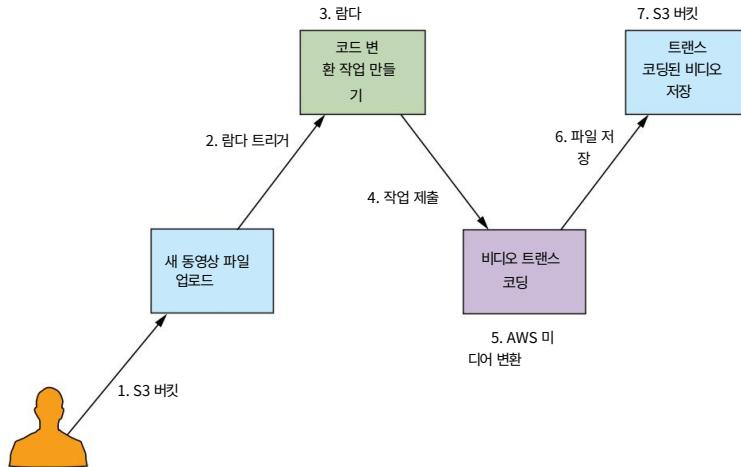


그림 2.1 24시간 비디오 밴드는 AWS S3, MediaConvert 및 Lambda로 구축되었습니다. 이 파이프라인은 처음에는 많은 단계가 있는 것처럼 보일 수 있지만 이 장에서는 이 부분을 세분화하여 즉시 확장 가능한 서비스 시스템을 구축할 것입니다.

2.2 시스템 준비

이제 AWS 서비스를 설정하고 컴퓨터에 소프트웨어를 설치할 차례입니다. 컴퓨터에 설치할 항목은 다음과 같습니다.

Lambda 기능을 관리하고 종속성을 추적하는 데 도움이 되는 Node.js 및 해당 패키지 관리자(npm) 배포 및 향후 사용 사례 및 예제를 지원하는 AWS 명령줄 인터페이스(CLI) 구성에 도움이 되는 서비스 프레임워크(npm 패키지) 애플리케이션을 AWS에 배포

AWS에서는 다음을 생성합니다.

- IAM(Identity and Access Management) 사용자 및 역할
- 비디오 파일을 저장할 S3 버킷
- 첫 번째 람다 함수

이 섹션은 길어 보일 수 있지만 책 전체에서 도움이 될 여러 가지를 설명합니다. 이미 AWS를 사용한 적이 있다면 이 섹션을 빠르게 진행할 수 있습니다.

2.2.1 시스템 설정

시작하려면 AWS 계정을 만들고 컴퓨터에 여러 소프트웨어 패키지와 도구를 설치해야 합니다. 다음을 순서대로 살펴보겠습니다.

1. AWS 계정을 생성합니다. 무료지만 신용카드를 제시해야 합니다.

프리 티어 할당량을 초과하는 경우 요금이 부과되는 경우에 대한 세부 정보.

<https://aws.amazon.com>에서 계정을 생성할 수 있습니다. 최대한 빨리 계정에 2단계 인증(2FA)을 설정하는 것이 좋습니다.

가능한 2FA에 대한 지침은 <https://amzn.to/2ZASm33>에 있습니다.

2. 계정이 생성된 후 적절한 버전의 다운로드 및 설치

여기에서 시스템용 AWS CLI:

<http://docs.aws.amazon.com/cli/latest/userguide/installing.html> 다음과 같은 경우

MSI 설치 프로그램을 포함하여 CLI를 설치하는 다양한 방법이 있습니다.

Windows, pip(Python 기반 도구) 또는 번들 설치 프로그램을 사용하는 경우

Mac 또는 Linux를 사용 중입니다.

3. Node.js와 npm을 설치합니다. <https://nodejs.org/>에서 Node.js를 다운로드할 수 있습니다.

<en/download/> (npm은 Node.js와 함께 번들로 제공됨).

최신 버전의 Node.js를 설치할 수 있지만 작성 당시에는

Lambda에서 지원하는 가장 최신 버전은 14입니다. Node 14.x는

코드를 배포할 때 대상을 지정합니다.

주의: 잠시 후 Serverless Framework를 설치해야 합니다. 하지만,

지금 할 필요가 없습니다. 시간이 되면 다른도록 하겠습니다.

2.2.2 ID 및 액세스 관리(IAM) 작업

AWS 계정을 갖는 것은 좋지만 아직 그것으로 너무 많은 것을 할 수는 없습니다. 을 위한

예를 들어 방금 설치한 AWS CLI가 작동하지 않습니다. 당신은되지 않을 것입니다

리소스를 만들고 배포하거나 무엇이든 할 수 있습니다. AWS가 작동하도록 하려면

IAM 사용자를 생성하고 사용자에게 권한을 할당한 다음 구성해야 합니다.

IAM 사용자의 자격 증명을 사용하는 CLI. 지금 해보자:

1. AWS 콘솔에서 IAM(Identity and Access Management), 사용자, 사용자 추가를 클릭합니다.

2. IAM 사용자에게 이름을 지정합니다(그림 2.2에서는

이름)을 선택하고 프로그래밍 방식 액세스 확인란을 선택합니다. 이 확인란을 선택하면

액세스 키 ID와 보안 액세스 키를 생성할 수 있습니다. (너는 필요할거야

몇 단계로 aws configure 를 실행하는 키입니다.)

3. 계속하려면 다음: 권한을 클릭합니다.

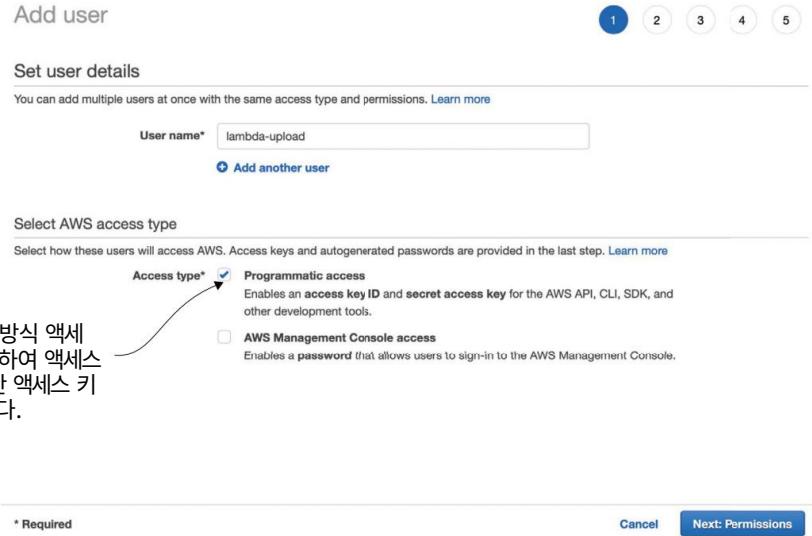


그림 2.2 IAM 콘솔을 사용할 때 새 IAM 사용자를 만드는 것은 간단합니다.

4. 기존 정책 직접 연결을 선택한 다음 옆에 있는 확인란을 클릭합니다.
관리자 액세스(그림 2.3). 계속하려면 다음: 태그를 선택합니다.

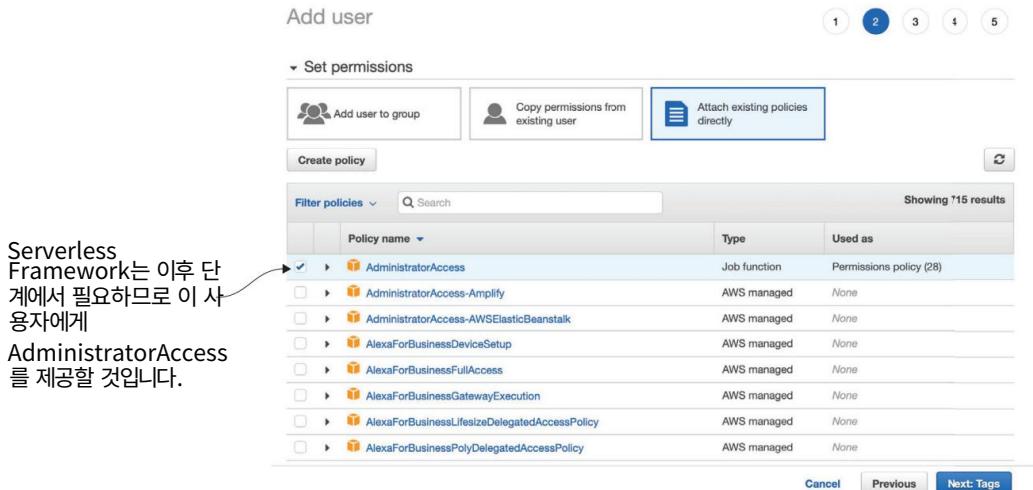


그림 2.3 AdministratorAccess 정책을 선택했는지 확인하십시오. 기능을 업로드하고 다른 서비스를 배포하는 데 필요합니다.

5. 태그는 인벤토리 및 메타데이터를 유지하는 데 유용하지만 이 예에서는 당신은 아무것도 할 필요가 없습니다. 다음: 검토를 클릭하여 계속 진행합니다.
6. 마지막 페이지에서 사용자 세부 정보와 권한 요약을 검토할 수 있습니다. 계속하려면 사용자 생성을 선택하십시오.

이제 사용자 이름, 액세스 키 ID 및 보안 액세스 권한이 있는 테이블이 표시되어야 합니다.

열쇠. 이 정보가 포함된 CSV 파일을 다운로드할 수도 있습니다. 다운로드하세요.

이제 컴퓨터에 키 복사본을 유지하고 닫기를 클릭하여 종료합니다(그림 2.4).

Add user

1 2 3 4 5

Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: <https://peteykins.signin.aws.amazon.com/console>

| User | Access key ID | Secret access key |
|---------------|---------------------|-------------------|
| lambda-upload | AKIAQRZRRW3PZ3NLD5K | ***** Show |

Close

CSV 파일을 다운로드
보기클릭하여 비밀 액세스 키.

당신의 컴퓨터. 액세스 키 ID
와 보안 액세스 키가 있습니다.

그림 2.4 액세스 키 ID와 보안 액세스 키를 저장하는 것을 잊지 마십시오. 이 창을 닫으면 보안 액세스 키를 다시 얻을 수 없습니다.

시스템의 터미널에서 aws configure 를 실행 합니다. AWS CLI는 몇 가지 메시지를 표시합니다.

1. 사용

자 자격 증명을 묻는 프롬프트에서 생성된 액세스 및 비밀 키를 입력합니다.

람다 업로드 사용자 이름 또는 이전에 선택한 사용자 이름의 경우.

2. 지역을 입력하라는 메시지도 표시됩니다. us-east-1 을 입력하고 Enter 키를 누릅니다. 우리 모든 서비스에 대해 동일한 지역을 사용하는 것이 좋습니다.
저렴하고 구성하기가 더 쉽습니다.) 버지니아 북부(us-east-1)

지역은 이 책의 기간 동안 사용할 모든 것을 지원하므로 항상 us-east-1을 사용합니다.

3. 기본 출력 형식을 선택하라는 메시지가 한 번 더 표시됩니다.

json으로 설정합니다.

이제 AWS CLI 구성이 완료되었습니다. IAM 사용자를 만들고 사용했습니다. 해당 사용자의 자격 증명을 사용하여 시스템에서 CLI를 구성합니다. 잘했어요!

세분화된 권한

AWS에서 권한과 관련하여 가장 좋은 방법은 권한을 세분화하는 것입니다.

즉, IAM 사용자와 역할에는 특정 권한만 있어야 합니다.

그들의 목적을 수행하는 데 필요합니다. 예를 들어 합당한 이유가 없는 한 모든 관리자 수준 권한이 있어서는 안 됩니다.

방금 관리자 수준 권한이 있는 IAM 사용자를 생성했습니다. 이것은 날아간다

방금 우리가 준 조언의 얼굴. 그 이유는 서비스

곧 사용할 프레임워크에는 관리자 수준 액세스 권한이 필요합니다.

프레임워크는 많은 API를 호출하며 다음으로 IAM 사용자를 구성하기 어렵습니다.

권한만 있으면 됩니다. Serverless Framework를 사용하지 않고

대신 AWS CLI를 사용하여 함수를 배포하려는 경우 다음을 생성하는 것이 좋습니다.

IAM 사용자를 지정하고 함수를 업로드하는 데 필요한 몇 가지 특정 권한을 할당합니다.

2.2.3 버킷을 만들자

다음 단계는 S3에서 버킷을 생성하는 것입니다. 이 버킷에는 트랜스코딩된 동영상이 포함됩니다.

Elemental MediaConvert에 의해 거기에 넣습니다. S3의 모든 사용자는 동일한 버킷 이름 공간을 공유하므로 사용하지 않는 버킷 이름을 가져와야 합니다.

이 책에서는 이 버킷의 이름이 서비스 비디오 트랜스코딩과 같은 이름이라고 가정합니다.

버킷 이름

버킷 이름은 S3 전역 리소스 공간 전체에서 고유해야 합니다. 우리는

이미 서비스 비디오 트랜스코딩을 수행했으므로 다른

이름. 여기에 이니셜(또는 임의의 문자열)을 추가하는 것이 좋습니다.

책 전체에서 식별하는 데 도움이 되는 버킷 이름(예: serverless video-upload-ps 및 serverless-video-transcoded-ps).

버킷을 생성하려면

1. AWS 콘솔에서 S3를 선택한 다음 버킷 생성을 클릭합니다(그림 2.5).
2. 버킷 이름을 입력하고 리전으로 미국 동부(버지니아 북부)를 선택합니다.
3. 페이지 하단으로 스크롤하고 버킷 생성을 클릭하여 확인합니다. 당신의 버킷은 콘솔에 즉시 나타나야 합니다.

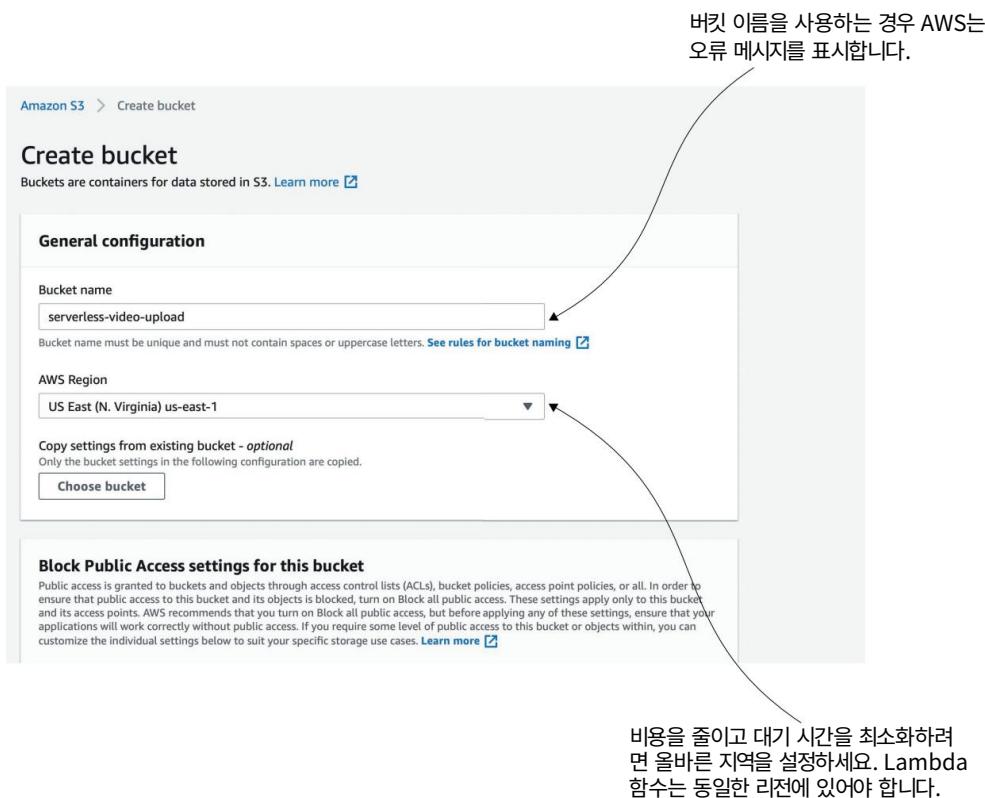


그림 2.5 AWS S3 콘솔에서 버킷 생성. 버킷 이름은 전역적으로 고유하므로 고유한 새 이름을 만들어야 합니다.

참고 결국 다른 S3 버킷이 필요하게 됩니다.

먼저 동영상을 업로드합니다. Serverless Framework는 이 버킷을 생성합니다.

다음 섹션에서 자동으로 제공되므로 아무 것도 할 필요가 없습니다.

아직. 내부에서 CloudFormation을 사용하여 트랜스코딩된 비디오 버킷을 생성할 수 있습니다.

Serverless Framework의 serverless.yml 파일도 있지만 이에 대해 설명하는 것은 이 장의 범위를 벗어납니다(좋은 연습이지만).

2.2.4 IAM 역할 생성

이제 첫 번째 Lambda 함수에 대한 IAM 역할을 생성해야 합니다.

이 기능은 잠시 후). 역할을 통해 함수가 S3 및 Elemental MediaConvert. 이 역할에 두 가지 정책을 추가합니다.

AWSLambdaFullAccess

AWSElementalMediaConvertFullAccess

AWSLambdaExecute 정책은 Lambda가 S3 및 CloudWatch와 상호 작용하도록 허용합니다 . CloudWatch는 로그 파일 수집, 지표 추적 및 설정을 위한 AWS 서비스입니다.

알람. AWSElementalMediaConvertFullAccess 정책은 Lambda가 제출하도록 허용합니다 . Elemental MediaConvert에 대한 새로운 트랜스코딩 작업.

1. AWS 콘솔에서 IAM을 찾아 클릭합니다.
2. 역할을 선택합니다.
3. 역할 만들기 버튼을 클릭하여 시작합니다. AWS 서비스 아래에 다양한 AWS 기술 목록이 표시됩니다. Lambda를 선택하고 다음: 권한 버튼을 선택합니다.
4. 이 보기에서는 미리 만들어진 정책을 검색하고 첨부할 수 있습니다. 찾기 및 첨부 (왼쪽의 확인란을 클릭하여) 다음 두 가지 정책:
 - AWSLambda실행
 - AWSElementalMediaConvertFullAccess
5. 다음: 태그를 클릭하여 진행합니다.
6. 다음: 김토를 클릭하여 김토 페이지로 진행합니다.
7. 역할 이름을 transcode-video로 지정하고 역할 생성을 클릭합니다.

역할이 생성되면 기존 역할 목록이 다시 표시됩니다. 선택하다
트랜스코딩 비디오를 사용하여 내부에 무엇이 있는지 확인합니다. 그림 2.6과 같아야 합니다.

Roles > transcode-video

Summary

[Delete role](#)

| | |
|--------------------------|---|
| Role ARN | arn:aws:iam::038221756127:role/transcode-video |
| Role description | Allows Lambda functions to call AWS services on your behalf. Edit |
| Instance Profile ARNs | [None] |
| Path | / |
| Creation time | 2018-06-28 16:00 UTC+1000 |
| Last activity | 2021-06-14 12:42 UTC+1000 (Today) |
| Maximum session duration | 1 hour Edit |

[Permissions](#) [Trust relationships](#) [Tags](#) [Access Advisor](#) [Revoke sessions](#)

▼ Permissions policies (2 policies applied)

[Attach policies](#) [Add inline policy](#)

| Policy name | Policy type |
|---------------------------------------|--------------------|
| ▶ AWSElementalMediaConvertFullAccess | AWS managed policy |
| ▶ AWSLambdaExecute | AWS managed policy |

두 가지 정책이 역할에 추가되었습니다.
권한은 정책에 포함됩니다.

그림 2.6 트랜스코딩-비디오 역할이 S3에 액세스하고 Elemental MediaConvert 작업을 생성하려면 두 가지 관리형 정책이 필요합니다.

2.2.5 AWS Elemental MediaConvert 사용

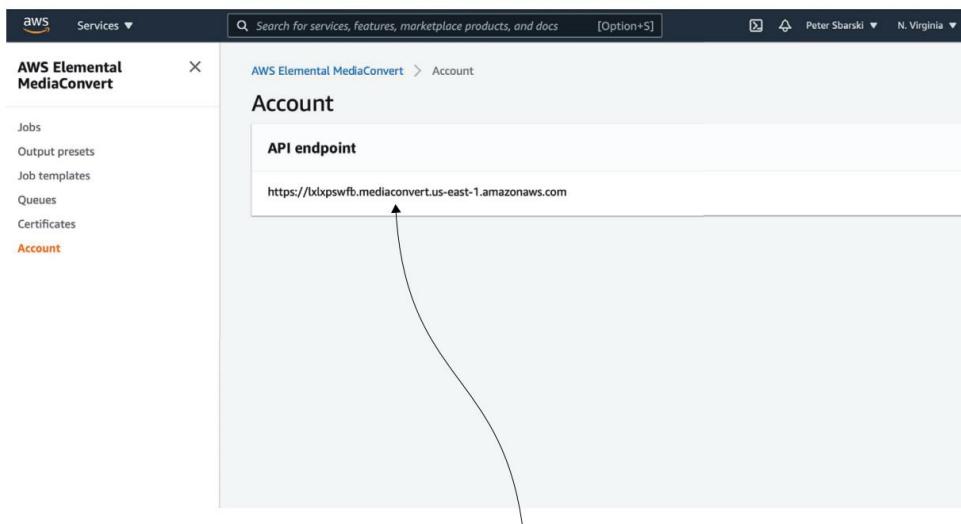
업로드된 비디오 파일을 한 형식에서 다른 형식으로 변환하기 위해 Elemental MediaConvert를 사용할 것입니다. 높은 수준에서 MediaConvert는 S3 버킷에 업로드된 파일을 가져와서 파일을 하나 이상의 다른 버전으로 트랜스코딩한 다음 이러한 버전을 다른 S3 버킷에 배치하는 방식으로 작동합니다.

MediaConvert 작업을 생성할 때 입력 및 출력 버킷과 수행하려는 변환을 포함하여 이 정보를 지정해야 합니다. MediaConvert 끝점도 지정해야 합니다. 각 사용자는 사용자 정의 엔드포인트를 가지고 있으며 귀하는 엔드포인트가 어디에 있는지 알아야 합니다. 어디를 봐야 하는지 알 수 있도록 지금 찾아봅시다.

1. AWS 콘솔에서 MediaConvert를 선택합니다(Media Services 아래에 있음).
범주).
2. 왼쪽 상단에 있는 험버거 아이콘을 클릭합니다.
윤곽).
3. 메뉴에서 계정을 선택합니다.

이 장에서 사용해야 하는 API 엔드포인트가 표시되어야 합니다(그림 2.7).

위치를 잊어버린 경우 언제든지 이 지침을 참조하여 MediaConvert API 끝점을 찾을 수 있습니다.



이 API 엔드포인트는 첫 번째
Lambda 함수에 필요합니다.

그림 2.7 이 장에서 사용할 API 끝점 보기

거의 다 왔습니다! 나중에 작업을 더 쉽게 하기 위해 지금 생성해야 하는 IAM 역할이 하나 더 있습니다.

2.2.6 MediaConvert 역할 사용

MediaConvert 서비스에 대한 역할을 생성해야 합니다. MediaConvert에는 다음이 필요합니다.

S3 및 API 게이트웨이에 대한 액세스. 이 역할이 없으면 Lambda에서 MediaConvert를 호출하려고 할 때 MediaConvert가 실행되지 않습니다. 역할을 만들려면 다음 단계를 따르세요(또는 직접 시도해 보세요!).

1. AWS 콘솔에서 IAM을 클릭한 다음 역할을 클릭합니다.
2. 역할 생성 버튼을 클릭합니다. AWS 서비스 아래에 다양한 AWS 기술 목록이 표시됩니다. MediaConvert 및 다음: 권한 버튼을 선택합니다.
AWS는 이 역할에 필요한 정책을 이미 사전 정의했습니다. S3 전체 액세스 및 API 게이트웨이 전체 호출입니다.
3. 다음: 태그를 선택하고 다음: 검토를 클릭하여 다음 페이지로 진행합니다.
4. 역할 이름을 media-convert-role로 지정하고 역할 생성을 클릭합니다.
5. 역할의 ARN을 메모장이나 쉽게 검색할 수 있는 곳에 복사합니다. 나중에 API 엔드포인트와 역할 ARN이 필요합니다(목록 2.3 참조).

2.3 서버리스 프레임워크 시작 서버 리스 프레임워크는 기능을 구

성하고 AWS에 배포하는 데 도움이 될 것입니다. 이 프레임워크는 강력하므로 코드를 패키징하는 방법, 사용할 변수 및 배포할 환경 측면에서 많은 유연성을 얻을 수 있습니다.

Serverless Framework가 막히면 부록 C를 보거나 <https://serverless.com/framework/docs>를 확인하십시오. 부록 C는 Serverless Framework에 대한 철저한 설명과 유용한 힌트 및 팁을 제공합니다. 그러나 거기에서 답을 찾지 못한다면 온라인 설명서를 참조하십시오.

GitHub의 소스 코드

이 장의 소스 코드는 <https://github.com/sbarski/serverless-architectures-aws-2>에서 찾을 수 있습니다.

2.3.1 서버리스 프레임워크 설정

터미널에서 `npm install -g serverless` 를 실행하여 Serverless Framework를 설치합니다. 작성 당시에는 Serverless Framework 2.63을 사용했습니다. 최신 버전의 프레임워크를 사용 중이고 무언가가 작동하지 않는 경우, 무엇이 잘못되었는지 파악하고 수정(또는 2.63으로 다운그레이드)하기 위해 최고의 탐정 기술을 적용해야 할 수 있습니다.

신임장

Serverless Framework는 사용자를 대신하여 리소스를 생성하고 관리할 수 있도록 AWS 계정에 대한 액세스 권한이 필요합니다. 기본적으로 Serverless Framework는 AWS CLI를 사용하여 마シン에 이미 구성한 AWS pro 파일을 사용합니다.

안녕하세요 세계입니다!

Serverless Framework를 설치하고 자격 증명을 구성했으면 다음을 테스트해 보겠습니다.
공장. 터미널에서 다음 명령을 실행합니다.

서비스 생성 --template aws-nodejs --path hello-world

cd hello-world 를 입력하여 새로 생성된 디렉토리로 변경합니다. 2개를 보셔야 합니다
이 디렉토리의 파일: serverless.yml 및 handler.js. 첫 번째 파일인 serverless.yml은
기능이 사용할 수 있는 기능, 이벤트 및 리소스를 설명하는 프로젝트 파일(서비스)입니다. 두 번째 파일인 handler.js
는 다음을 수행할 수 있는 예제 Lambda 함수입니다.
변화! handler.js를 열고 다음 목록과 같이 함수 구현을 수정합니다.

목록 2.1 새로운 hello-world Lambda 함수

```
'엄격한 사용';

module.exports.hello = 비동기(이벤트) => {
    반품{
        상태 코드: 200,
        본문: JSON.stringify(
            {
                메시지: 'Hello Serverless World!',
                입력: 이벤트,
            },
            없는,
            2
        ),
    };
};
```

함수 수정이 끝나면 파일을 저장하는 것을 잊지 마십시오. 너는
이제 배포할 준비가 되었습니다. 터미널에서 서비스 배포를 실행하고 Enter 키를 누릅니다.
(배포하기 전에 serverless.yml 파일과 동일한 디렉토리에 있는지 확인하십시오.
그렇지 않으면 오류 메시지가 나타납니다).

Serverless Framework 패키지 파일이 표시되고 CloudFormation을 준비합니다.
스택하고 함수를 AWS에 배포합니다. 배포가 완료되는 즉시
기능(dev)에 사용되는 스테이지, 지역 등의 유용한 정보
(us-east-1) 및 서비스 이름(hello-world). 귀하의 기능이 호출됩니다
hello-world-dev-hello 서비스 이름, 단계 및 기능의 조합
내보내다.

마지막으로 다음을 열어 함수가 성공적으로 배포되었는지 확인할 수 있습니다.
AWS의 Lambda 콘솔 및 거기에서 함수 실행. 또 다른 옵션은
터미널에서 배포된 기능을 호출합니다.

AWS에서 함수를 실행하고 응답을 반환하려면 터미널에서 serverless invoke --function hello 명령을 실행합니다. 서비스 프레임
작업은 클라우드 환경에서 이 기능을 호출하는 것을 알게 됩니다.

서비스 배포

원활 때마다 서비스 (서비스 배포에서와 같이) 를 입력할 필요가 없습니다 . 배포하거나 작업을 수행합니다. sls 약어를 사용할 수 있습니다 . 다음 명령은 완전히 유효합니다. sls deploy 또는 sls invoke --function hello.

2.3.2 서비스 프레임워크를 24시간 비디오로 가져오기

이제 Serverless Framework가 작동하도록 하였으므로 The 24-

시간 비디오. 새 함수를 만들고 serverless.yml에서 참조할 것입니다.

단일 작업으로 기능을 배포(및 추가 기능 추가)할 수 있습니다.

명령을 수행하고 나중에 전체 서비스 애플리케이션을 지속적으로 확장하고 구성할 수 있습니다.

- 터미널 창에서 다음 명령을 실행합니다.

```
sls 생성 --템플릿 aws-nodejs --경로 24시간 비디오
```

24 대신 24 를 사용한 이유는 서비스 이름이

알파벳 문자로 시작합니다.

- 방금 만든 새로운 24시간 비디오 디렉토리로 변경합니다.

- handler.js를 삭제하지만 serverless.yml은 그대로 둡니다.

- transcode-video라는 새 하위 폴더를 만듭니다.

잠시 후 serverless.yml을 변경하기 시작합니다. 우리의 구현(목록 2.2)을 고수할 수 있지만 올바르게 변경해야 하는 5개의 매개변수가 있습니다.

당신의 환경을 반영합니다. 이러한 매개변수는 목록 2.2에서 굵게 표시됩니다.

업로드 버킷의 이름

트랜스코딩된 비디오 버킷의 이름

기능에 대한 비디오 역할 ARN

MediaConvert 엔드포인트

MediaConvert 역할

목록 2.2 함수에 대한 serverless.yml 변경

공급자는 AWS이지만 Serverless Framework는 Azure 및 Google Cloud와 같은 다른 클라우드 공급자도 지원합니다.

서비스: 24시간짜리 비디오

아직 설정하지 않은 경우 nodejs14.x로 설정합니다.

공급자:

▶ 이름: aws

배포할 지역을 정의합니다. 이 설정을 재정의하고 다른 지역에 배포할 수 있습니다.

런타임: nodejs14.x 지역: us-east-1

이 사용자 정의 변수를 업로드 버킷의 이름으로 설정하십시오.

커스텀:

업로드 버킷: 업로드 비디오 버킷 트랜스코드 버킷: 트랜스코딩 비디오 버킷

transcode-bucket을 트랜스코딩된 비디오 버킷의 이름으로 설정합니다. 섹션 2.2.3에서 이 버킷을 생성했습니다.

코드 변환 비디오 역할:

arn:aws:iam::038221756127:role/transcode-video

▶ 세션 2.2.4에서 생성한 트랜스코드 비디오 역할 ARN을 설정합니다. ARN을 원하는 값으로 업데이트합니다.

```

미디어 앤드포인트:
https://u4ac0ytu.mediaconvert.us-east-1.amazonaws.com

미디어 역할:
arn:aws:iam::038221756127:role/media-convert-role

기능:
트랜스코딩 비디오:
    핸들러: transcode-video/index.handler
    역할: ${self:custom.transcode-video-role}
    폐기자:
        개별적으로: 사실
    환경:
        MEDIA_ENDPOINT: ${self:custom.media-endpoint}
        MEDIA_ROLE: ${self:custom.media-role}
        TRANSCODED_VIDEO_BUCKET: ${self:custom.transcode-bucket}

이벤트:
    - s3: ${self:custom.upload-bucket}

```

← 서비스가 작동하도록 개인 MediaConvert 끝점을 설정합니다.
URL이 다르므로 반드시 변경 하십시오.

← 세션에서 생성한 MediaConvert 역할 ARN을 설정합니다.
2.2.6. 동일한 이름을 유지한 경우 계정 번호(예:
038221751234)를 계정 번호로 변경하면 모든 것이 작동합니다.

← S3 업로드 버킷의 Lambda 함수에 대한 이벤트 트리거를 지정합니다.

다음은 목록 2.2를 만들기 위해 업데이트해야 하는 모든 것에 대한 간략한 설명입니다.

당신을 위해 일하십시오. 업로드 버킷부터 시작하겠습니다.

버킷 업로드

목록 2.2에서 업로드 버킷의 이름을 지정해야 합니다. 이것은 새 양동이입니다.

그것은 아직 존재하지 않습니다. 전역적으로 버킷 이름을 사용해야 함을 기억하십시오.

고유한. 이를 수행하는 한 가지 방법은 전체 이름의 접두사 또는 접미사를 사용하는 것입니다.

일반 이름 또는 몇 개의 임의의 문자와 숫자를 추가합니다.

CloudFormation을 통한 Serverless Framework는 자동으로 버킷을 생성합니다. upload-bucket-firstname-lastname 과 같은 항목으로 이동할 수 있습니다. 버킷의 경우

이름이 이미 사용 중이면 배포 중에 Serverless Framework에서 알려줍니다. 당신은 변경하고 다시 시도할 수 있습니다.

트랜스코딩된 비디오 버킷

목록 2.2에는 transcode-bucket이라는 사용자 지정 속성이 있습니다. 이 속성에는 트랜스코딩된 비디오 버킷의 이름이 포함됩니다. 이 속성을 다음 이름으로 업데이트합니다.

섹션 2.2.3에서 수동으로 생성한 버킷.

람다 역할 ARN

함수에 대한 IAM 역할을 지정해야 합니다. 운 좋게도 세션에서 역할을 만들었습니다.

2.2.4. 해당 역할의 ARN을 찾아 복사한 다음 파라미터를 업데이트해야 합니다.

트랜스코드-비디오-역할이라고 합니다. 역할 ARN을 가져오고 serverless.yml을 업데이트하려면 다음을 따르십시오.

이 쉬운 단계:

1. IAM 콘솔에서 역할을 선택합니다.
2. 트랜스코딩-비디오 역할을 찾아 선택합니다.
3. 역할 ARN의 값을 복사합니다.
4. 값을 transcode-video-role에 대한 serverless.yml 파일에 붙여넣습니다.

미디어 변환 앤드포인트

목록 2.2에서 media-endpoint 변수를 생성하는 라인을 찾을 수 있습니다. 이 끝점을 얻으려면 섹션 2.2.5를 참조하거나 다음 단계를 따르세요.

1. AWS 콘솔에서 MediaConvert를 선택합니다(Media Services 아래에 있음).
범주).
 2. 왼쪽 상단 모서리에 있는 햄버거 아이콘을 클릭합니다.
세 개의 평행선).
 3. 메뉴에서 계정을 선택합니다. 해야 하는 API 엔드포인트가 표시됩니다.
- 목록에 복사 2.2.

미디어변환 역할

섹션 2.2.6에서 요소 MediaConvert 서비스에 대한 IAM 역할을 생성했습니다. 목록 2.2에서는 해당 역할에 대한 ARN을 지정해야 했습니다. 꼭 찾아보고 복사하세요
제대로 끝났습니다. 가지고 있는 두 IAM 역할을 혼동하지 않도록 주의하십시오. IAM
섹션 2.2.4에서 생성된 역할은 트랜스코딩-비디오 Lambda 함수를 위한 것입니다. 그만큼
2.2.6에서 생성된 역할은 MediaConvert용이며 사용해야 하는 역할입니다.

2.3.3 첫 번째 Lambda 함수 생성

이제 serverless.yml 파일을 만들었으므로 transcode-video 폴더로 변경합니다.
터미널 창에서 npm init를 실행합니다. 를 눌러 모든 옵션에 동의합니다.
입력하다. 원하는 것은 무엇이든 변경할 수 있습니다. 그것은 당신의 기능에 영향을 미치지 않을 것입니다.
package.json이라는 새 파일이 생성됩니다. 원하는 경우 이 파일을 나중에 사용할 수 있습니다.
함수에 추가 종속성 또는 라이브러리를 추가합니다. 이제 방법에 대해 논의해 보겠습니다.
새 기능이 작동하고 수행할 작업:

새 파일이 S3 버킷에 업로드되는 즉시 함수가 호출됩니다.
업로드된 비디오에 대한 정보는 다음을 통해 Lambda 함수로 전달됩니다.
이벤트 개체. 버킷 이름과 파일 이름(키)이 포함됩니다.
업로드 중입니다.
Lambda 함수는 AWS MediaConvert에 대한 트랜스코딩 작업을 준비합니다.
이 함수는 작업을 MediaConvert에 제출하고 메시지를 씁니다.
Amazon CloudWatch 로그 스트림.

index.js라는 새 파일을 만들고 즐겨 사용하는 텍스트 편집기에서 엽니다. 이 파일에는 첫 번째 기능이 포함되어 있습니다. 주의해야 할 중요한 점은 함수를 정의해야 한다는 것입니다.
Lambda 런타임에 의해 호출될 핸들러.

Listing 2.3은 이 함수의 구현을 보여준다. 이 목록을 index.js에 복사합니다.
하지만 이 코드를 배포하고 실행하기 전에 몇 가지 작은
코드 목록 뒤의 텍스트에 자세히 설명된 대로 변경됩니다.

목록 2.3 트랜스코딩 비디오 생성하기 Lambda

```
'엄격한 사용';

const AWS = 요구('aws-sdk');
const mediaConvert = 새로운 AWS.MediaConvert({
    끝점: process.env.MEDIA_ENDPOINT
});
```

설정된 MediaConvert 엔드포인트
환경 변수를 가져옵니다.
serverless.yml에서(목록 2.2)

2 장 서버리스를 위한 첫 번째 단계

```

const 출력 버킷 이름 =
  process.env.TRANSCODED_VIDEO_BUCKET;

export.handler = 비동기(이벤트, 컨텍스트) => {
  const 키 = event.Records[0].s3.object.key; const sourceKey =
    decodeURIComponent(key.replace(/\+/g, ' ')); const outputKey = sourceKey.split('.')[0];

  const 입력 = 's3://' + event.Records[0].s3.bucket.name + '/' +
    event.Records[0].s3.object.key; const 출력 = 's3://' +
    outputBucketName + '/' + outputKey + '/';

  노력하다 {

    const 작업 ={
      "역할": process.env.MEDIA_ROLE, "설정": {
        "입력": [[
          "FileInput": 입력, "AudioSelectors": {
            "오디오 선택기 1": {
              "선택기 유형": "추적",
              "트랙": [1]
            }
          }
        ]],
        "출력 그룹": [
          {"이름": "파일 그룹",
           "출력": [
             {"사전 설정": "시스템-
               일반_Hd_Mp4_Avc_16x9_1920x1080p_24Hz_6Mbps",
              "확장자": "mp4",
              "이름 수정자": "_16x9_1920x1080p_24Hz_6Mbps"
            },
            {"사전 설정": "시스템-
               일반_Hd_Mp4_Avc_16x9_1280x720p_24Hz_4.5Mbps",
              "확장자": "mp4",
              "이름 수정자": "_16x9_1280x720p_24Hz_4.5Mbps"
            },
            {"사전 설정": "시스템-
               일반_Sd_Mp4_Avc_Aac_4x3_640x480p_24Hz_1.5Mbps",
              "확장자": "mp4",
              "이름 수정자": "_4x3_640x480p_24Hz_1.5Mbps"
            }
          ],
          "출력 그룹 설정": {
            "유형": "FILE_GROUP_SETTINGS",
            "파일 그룹 설정": {
              "목적지": 출력
            }
          }
        ]
      };
    };

    const mediaConvertResult = 대기
    mediaConvert.createJob(작업).약속();
  }
}

```

serverless.yml에 지정된 트랜스코딩된 비디오 버킷 이름을 가져옵니다.

serverless.yml에 지정된 MediaConvert 역할 ARN을 가져옵니다.

MediaConvert 작업 정의에 대한 오디오 선택기를 지정합니다. 기본적으로 비디오에서 단일 오디오 트랙의 이름을 지정합니다.

새 비디오 파일의 출력 버킷을 설정합니다.

노력하다 {

작업 정의

의 위치를 설정합니다.

에 대한 입력 비디오 미디어 변환

```

console.log(미디어변환결과);

} 잡기(오류) {
    console.error(오류);
}
};


```

미디어 변환 출력

목록 2.3의 함수는 다음 형식을 정의하는 세 가지 새로운 출력을 선언합니다.

새로 트랜스코딩된 비디오(비트 전송률, 해상도 등 포함). 목록 2.3에 지정된 템플릿은 MediaConvert에 내장된 일반 템플릿입니다. 운 좋게,

우리가 선택한 것을 강제로 사용하지 않아도 됩니다. 다른 템플릿에서 선택하거나 직접 만들 수도 있습니다. MediaConvert에서 사용 가능한 다른 사전 설정을 보려면

다음과 같은:

1. AWS 콘솔에서 MediaConvert를 선택합니다.
2. 왼쪽 상단 모서리에 있는 햄버거 아이콘을 클릭합니다.
3. 출력 사전 설정을 선택합니다.
4. 사용자 정의 사전 설정 드롭다운에서 시스템 사전 설정을 선택합니다.

사용할 수 있는 다양한 사전 설정 그리드가 표시됩니다(그림 2.8). 그리드에는 여러 페이지가 있으며 다른 범주(MP4, HLS, Broadcast XDCAM 등)를 표시하도록 선택할 수 있습니다.

The screenshot shows the AWS Elemental MediaConvert interface. On the left, there's a sidebar with 'Jobs', 'Output presets' (which is highlighted in orange), 'Job templates', 'Queues', 'Certificates', and 'Account'. The main area is titled 'Output presets' and shows a table of 'System Presets'. The columns are 'Name', 'Description', 'Category', and 'Type'. There are 15 entries listed, all under the 'SYSTEM' category. The 'Category' column has a dropdown arrow pointing to a menu. A callout bubble from the top right points to this arrow with the Korean text: '카테고리별로 필터링하여 선택 할 수 있는 다양한 옵션을 확인하세요.' (Check out the various options available by filtering by category).

| Name | Description | Category | Type |
|---|---|----------|------|
| System-Avc_16x9_1080p_29_97fps_8500kbps | Wifi, 1920x1080, 16:9, 29.97fps, 8500kbps | SYSTEM | |
| System-Avc_16x9_270p_14_99fps_400kbps | Cell, 480x270, 16:9, 14.99fps, 400kbps | SYSTEM | |
| System-Avc_16x9_360p_29_97fps_1200kbps | Wifi, 640x360, 16:9, 29.97fps, 1200kbps | SYSTEM | |
| System-Avc_16x9_360p_29_97fps_600kbps | Cell, 640x360, 16:9, 29.97fps, 600kbps | SYSTEM | |
| System-Avc_16x9_540p_23_97fps_3500kbps | Wifi, 960x540, 16:9, 29.97fps, 3500kbps | SYSTEM | |
| System-Avc_16x9_720p_29_97fps_3500kbps | Wifi, 1280x720, 16:9, 29.97fps, 3500kbps | SYSTEM | |
| System-Avc_16x9_720p_29_97fps_5000kbps | Wifi, 1280x720, 16:9, 29.97fps, 5000kbps | SYSTEM | |
| System-Avc_16x9_720p_29_97fps_6500kbps | Wifi, 1280x720, 16:9, 29.97fps, 6500kbps | SYSTEM | |
| System-Avc_4x3_1200p_29_97fps_8500kbps | Wifi, 1600x1200, 4:3, 29.97fps, 8500kbps | SYSTEM | |
| System-Avc_4x3_360p_14_99fps_400kbps | Cell, 480x360, 4:3, 14.99fps, 400kbps | SYSTEM | |
| System-Avc_4x3_480p_29_97fps_600kbps | Cell, 640x480, 4:3, 29.97fps, 600 kbps | SYSTEM | |

그림 2.8 MediaConvert 출력 사전 설정 페이지에서 시스템 사전 설정을 선택하거나 소유하다.

목록 2.3의 코드를 사용하여 다른 유형의 비디오를 만들려면 원하는 사전 설정의 이름을 지정하고 다른 사전 설정과 마찬가지로 함수에 복사합니다. (확장자와 수정된 이름을 지정하는 것을 기억하십시오). 예를 들어, 당신이 HLS 출력을 추가하려면 출력 에 이와 같은 것을 포함해야 합니다.

함수의 배열:

```
{
  "사전 설정": "시스템-Ott_Hls_Ts_Avc_Aac_16x9_1280x720p_30Hz_3.5Mbps",
  "확장자": "hls",
  "이름 수정자": "_Hls_Ts_Avc_Aac_16x9_1280x720p_30Hz_3.5Mbps"
}
```

전개

`sls deploy` 를 입력하여 터미널에서 첫 번째 기능을 배포 합니다 . `serverless.yml` 이 있는 디렉토리에서 `sls 배포` 를 실행합니다. 배포 성공해야 하며 AWS에서 기능을 확인해야 합니다. 첫 번째 기능은 24시간-비디오-개발-트랜스코드-비디오 와 같은 이름으로 지정 됩니다. 나중에 만약 원하는 경우 `sls remove` 를 실행하여 AWS에서 이 함수를 제거할 수 있습니다. 단말기.

다른 참고 사항: 배포 프로세스에서 이름이 추가된 버킷이 생성될 수 있습니다. 24시간 비디오 서비스 배포 벽-sq06y6wjku9z와 같은 것입니다. 이것은 정상입니다. Serverless Framework는 이 버킷을 생성하여 생성하는 Cloud Formation 템플릿을 업로드합니다. 이 버킷을 안전하게 무시할 수 있지만 수동으로 삭제하지 마십시오! `sls remove` 명령 은 이를 제거합니다 (배포된 람다 함수).

2.4 AWS에서 테스트

첫 번째 기능을 테스트하려면 업로드 버킷에 비디오를 업로드하십시오. 이 단계를 따르세요:

1. S3 콘솔로 이동합니다.
2. 업로드 버킷을 클릭한 다음 업로드를 선택하여 업로드 페이지를 엽니다. (그림 2.9).
3. 파일 추가를 클릭하고 컴퓨터에서 비디오 파일을 선택한 다음 업로드 단추. 다른 모든 설정은 그대로 둘 수 있습니다. 동영상 파일이 없다면 테스트, <https://sample-videos.com> 으로 이동 MP4 비디오 중 하나를 가져옵니다.

시간이 지나면 트랜스코딩된 비디오 버킷에 세 개의 새 비디오가 표시됩니다. 이것들

파일은 버킷의 루트가 아닌 폴더에 나타나야 합니다(그림 2.10). 그만큼

새 비디오를 제작하는 데 걸리는 시간은 파일의 길이에 따라 다릅니다.

업로드했습니다. 새 파일을 생성하는 데 5분(또는 그 이상)이 걸릴 수 있으므로 기다리는 동안 차 한잔.

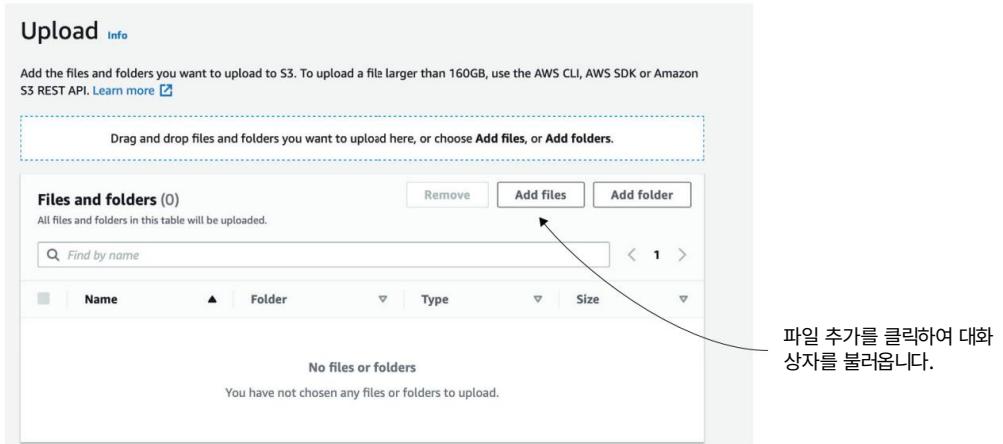


그림 2.9 ASW에서 테스트하려면 업로드 및 트랜스코딩이 훨씬 빨라지기 때문에 처음에 작은 파일을 업로드하는 것이 좋습니다.

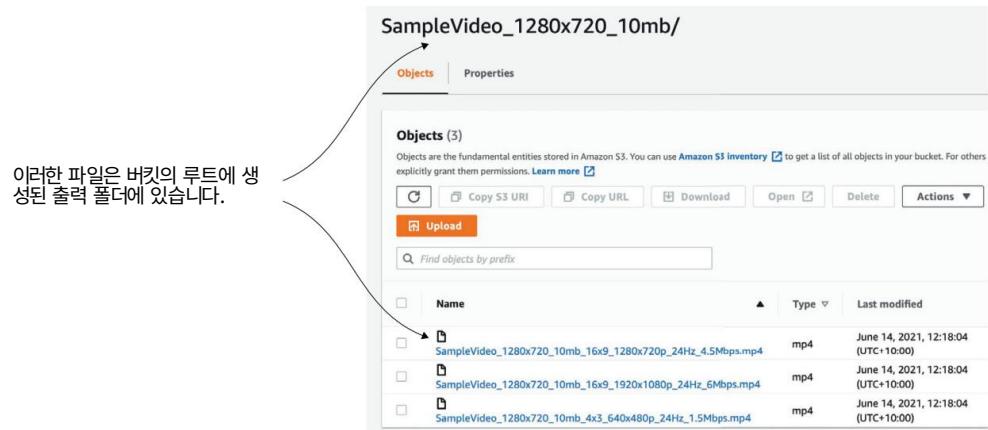


그림 2.10 MediaConvert는 3개의 새 파일을 생성하여 트랜스코딩된 비디오 S3 버킷의 폴더에 넣습니다.

2.5 로그 보기

이전 섹션에서 테스트를 수행한 후 파일에 세 개의 새 파일이 표시되어야 합니다. 트랜스코딩된 비디오 버킷. 그러나 일이 항상 순조롭게 진행되지는 않을 수도 있습니다(비록 우리는 그들이하다)! 새 파일이 나타나지 않는 등의 문제가 있는 경우 두 개의 서로 다른 로그에서 오류를 확인할 수 있습니다. 첫 번째이자 가장 중요한 것은 Cloud Watch에서 Lambda의 로그입니다. 로그를 보려면 다음 단계를 수행하십시오.

1. AWS 콘솔에서 Lambda를 선택한 다음 함수 이름을 클릭합니다.
2. 모니터 탭을 선택합니다. 숫자가 있는 다른 그래프가 표시되어야 합니다. 중 하나 이러한 그래프에는 오류 수 및 성공률이라는 레이블이 지정됩니다. 스파이크가 있는 경우 (즉, 개수가 0보다 크면) 문제가 있음을 의미합니다.
3. CloudWatch에서 로그 보기 버튼을 클릭하여 CloudWatch를 엽니다. 모든 로그를 볼 수 있습니다 날짜순으로 정렬된 항목. 오른쪽에는 그들이 속한 스트림이 표시됩니다.
4. 각 로그 항목을 클릭하면 오류 메시지를 포함한 자세한 내용을 볼 수 있습니다.
5. 이전에 호출 오류율이 0보다 큰 것을 확인한 경우 오류가 있는 항목을 기록하고 문제를 수정합니다.

Lambda 로그에 특별한 것이 없으면 AWS Media Convert 로그를 살펴보십시오. 이 로그를 보려면:

1. AWS 콘솔에서 MediaConvert를 클릭합니다.
2. 왼쪽에 있는 햄버거 아이콘을 선택하여 사이드바를 엽니다.
3. 메뉴에서 작업을 선택합니다. 오른쪽에 작업 목록이 표시됩니다.
4. 작업을 클릭하면(실패한 경우) 추가 정보를 볼 수 있습니다(그림 2.11).

The screenshot shows the AWS Elemental MediaConvert interface. On the left, a sidebar lists 'Jobs', 'Output presets', 'Job templates', 'Queues', 'Certificates', and 'Account'. The main area is titled 'AWS Elemental MediaConvert > Jobs'. It displays a table of jobs with columns: Job ID, First input file name, Submit time, Finish time, Status, Job percent complete, and Queue. Three jobs are listed: one completed successfully and two that failed ('ERROR'). A callout arrow points from the text '오류에 대한 세부 정보를 보려면 작업 ID를 선택하십시오.' to the 'Job ID' column of the failed jobs table. Another callout arrow points from the same text to the 'Job summary' section below. The 'Job summary' section is expanded, showing an 'Overview' table with details like Status (ERROR), Queue (Default), Job ID (1623636792788-z06gwf), Start time (2021-06-14 12:13:15), Duration in queue (00:00:03), and Error message (Unable to open input file [s3://upload-video-bucket-ps/Video+Of+Flower+Field.mp4]: [Failed to probe/open: [Can't read input stream: [Failed to read data: HeadObject failed]]]).

그림 2.11 MediaConvert 실패는 작업이 시작되기 전에 삭제되는 소스 파일, Lambda 함수의 코드 오류 또는 잘못된 구성을 포함하여 다양한 이유로 발생할 수 있습니다.

문제가 발생 하는 경우 우리

의 경험에 따르면 IAM 권한이 올바르게 구성되지 않았거나 함수 코드의 어딘가에 오타가 있기 때문에 문제가 자주 발생합니다. AWS에 항상 가장 설명적인 오류 메시지가 있는 것은 아니므로 때때로 CloudWatch에 대한 약간의 조사 및 조사 작업이 필요합니다.

요약

서비스 애플리케이션을 구성하는 가장 좋은 방법은 서비스 프레임워크와 같은 IaC(Infrastructure as Code) 프레임워크를 사용하는 것입니다.

기능을 배포하고 서비스를 수동으로 설정하는 것은 학습에 좋지만 장기적으로 지속 가능하지 않습니다. Serverless Framework는 가장 복잡한 서비스 애플리케이션을 구성하고 배포하는 데 도움이 될 수 있습니다.

서비스 애플리케이션과 파이프라인은 일반적으로 서로 연결된 서로 다른 서비스로 구성됩니다. 24시간 비디오 예제에서는 AWS Lambda, S3 및 Elemental MediaConvert를 사용합니다. 대부분의 서비스 애플리케이션은 서비스 조합을 사용합니다.

AWS CloudWatch는 AWS Lambda 함수 내에서 일어나는 일을 기록하는 중요한 서비스입니다. 당신이 그것을 가장 확실히 필요로 할 것이기 때문에 그것을 사용하는 방법을 배우는 것이 중요합니다.

AWS의 보안은 몇 가지 예외가 있지만 주로 IAM(Identity and Access Management)을 통해 제어됩니다. AWS 및 서비스 애플리케이션의 전문가가 되고 싶다면 IAM의 작동 방식을 아는 것이 필수적입니다.

AWS에서 비용을 추정하는 것은 까다로울 수 있습니다. 많은 서비스에는 관대한 무료 계층이 있지만 잘못 사용하면 많은 비용이 들 수 있습니다. 사용하려는 모든 서비스의 비용을 검토하고 잠재적 비용이 얼마인지 이해하십시오.

아키텍처 및 패턴

이 장에서는 다음을 다룹니다.

서비스 아키텍처의 사용 사례

패턴 및 아키텍처의 예

서비스 아키텍처의 사용 사례는 무엇이며 아키텍처의 종류는 무엇입니까?

패턴이 유용합니까? 사람들이 시스템 설계에 대한 서비스 접근 방식에 대해 배울 때 종종 이러한 질문을 받고 사용 사례에 대해 질문을 받습니다. 우리는 다른 사람들이 이 기술을 어떻게 적용했고 그들이 어떤 종류의 사용 사례, 디자인 및 아키텍처를 생산했는지 살펴보는 것이 도움이 된다는 것을 알게 되었습니다.

이 장에서는 서비스 아키텍처가 적합한 위치와 서비스 시스템 설계에 대해 생각하는 방법에 대한 확실한 소개를 제공합니다. 이 책의 나머지 부분은 실제 사용 사례에 초점을 맞추고 우리가 특히 매력적으로 발견한 여러 서비스 아키텍처에 대해 자세히 설명합니다.

3.1 사용 사례

서비스 기술 및 아키텍처를 사용하여 전체 시스템을 구축하거나 격리된 구성 요소를 생성하거나 특정 세부 작업을 구현할 수 있습니다. 사용 범위

서비스 디자인은 광범위하며 장점 중 하나는 다음과 같은 용도로 사용할 수 있다는 것입니다.
 크고 작은 작업도 마찬가지입니다. 우리는 웹 및
 수만 명의 사용자를 위한 모바일 애플리케이션과
 특정 문제를 해결합니다.

서비스가 컴퓨팅에서 코드를 실행하는 것만이 아니라는 점을 기억할 가치가 있습니다.
 램다와 같은 서비스. 또한 타사 서비스 및 API를 사용하여
 당신이해야 할 일의 양에. 이를 염두에 두고 몇 가지 기본적인 사용 사례를 살펴보겠습니다.

3.1.1 백엔드 컴퓨팅

AWS Lambda와 같은 기술은 몇 년 전이지만 이미
 전체 비즈니스를 지원하는 서비스 백엔드. 클라우드 전문가 (<https://acloudguru.com>), 예를 들어 수천 명의 사용자가 실시간으로 협업하고
 수백 기가바이트의 비디오를 스트리밍합니다. 또 다른 예로는 보험사,
 처음부터 서비스 우선 접근 방식을 채택한 Branch (<https://amzn.to/3vRumYU>).

실제로 서비스를 최우선으로 생각하면서 전체 비즈니스를 만들고 운영하는 것이 가능합니다. 기술
 에 대한 그런 종류의 철학적 접근 방식을 스스로 명확하게 표현하면 어떤 서비스를 채택할지, 어떻게 하면
 가장 좋은지 등의 질문에 답하는 데 도움이 될 것입니다.

특정 아키텍처 문제를 해결합니다.

민첩성과 효율성을 추구하는 조직은 스타트업만이 아닙니다.
 서비스. 오랜 역사를 지닌 기존 기업들도 서비스 기술과 아키텍처를 사용하여 고객에게 가치를 제공하고 있
 습니다. 이러한 더 큰 회사 중 일부에는 Comcast, Coinbase, Fender, Nordstrom 및

넷플릭스 (<https://aws.amazon.com/serverless/customers/>).

3.1.2 사물 인터넷(IoT)

웹 및 모바일 애플리케이션을 제쳐두고 서비스는 인터넷에 매우 적합합니다.
 사물(IoT) 애플리케이션. Amazon Web Services(AWS)에는 유용한 IoT 플랫폼이 있습니다.
 (<https://aws.amazon.com/iot-platform/how-it-works/>) 결합하는

인증 및 권한 부여

통신 게이트웨이

레지스트리(각 장치에 고유한 ID를 할당하는 방법)

장치 색도잉(장치 상태 유지)

규칙 엔진(장치 메시지를 AWS 서비스로 변환 및 라우팅)

예를 들어 규칙 엔진은 파일을 Amazon의 Simple Storage Service(S3)에 저장할 수 있습니다.
 Amazon Simple Queue Service(SQS) 대기열에 데이터 퓨시 및 AWS 호출
 램다 함수. Amazon의 IoT 플랫폼을 사용하면 서버를 실행할 필요 없이 장치용으로 확장 가능한 IoT 백엔드
 를 쉽게 구축할 수 있습니다. 서비스 애플리케이션 백엔드는
 많은 인프라 관리를 제거하고 세분화되고
 예측 가능한 청구(특히 Lambda와 같은 서비스 컴퓨팅 서비스가
 사용됨), 고르지 않은 요구 사항을 충족하도록 확장할 수 있습니다.

3.1.3 데이터 처리 및 조작

서비스 기술의 일반적인 용도는 데이터 처리, 변환, 조작 및 트랜스코딩입니다. CSV, JSON 및 XML 파일을 처리하기 위해 다른 개발자가 구축한 Lambda 함수를 보았습니다. 데이터의 대조 및 집계; 이미지 크기 조정;

및 형식 변환. Lambda 및 AWS 서비스는 데이터 처리 작업을 위한 이벤트 기반 파이프라인을 구축하는데 적합합니다.

2장에서는 한 형식의 비디오를 변환하기 위한 강력한 파이프라인을 구축했습니다.

다른 사람에게. 이 파이프라인은 지정된 S3에 새 비디오 파일이 추가될 때만 실행됩니다.

버킷, 즉 해야 할 일이 있을 때만 Lambda 실행 비용을 지불하고 시스템이 유휴 상태일 때는 비용을 지불하지 않습니다. 그러나 더 광범위하게는 데이터

특히 서비스 기술의 탁월한 사용 사례로 처리

다른 서비스와 함께 Lambda를 사용합니다.

3.1.4 실시간 분석

로그, 시스템 이벤트, 트랜잭션 또는 사용자 클릭과 같은 데이터 수집은 Amazon Kinesis Data Streams 및 Amazon Kinesis Fire 호스와 같은 서비스를 사용하여 수행할 수 있습니다. Kinesis Data Streams 및 Lambda 함수는 다음과 같은 애플리케이션에 적합합니다.

분석, 집계 및 저장해야 하는 많은 데이터를 생성합니다. 언제

스트림에서 메시지를 처리하기 위해 생성된 함수의 수인 Kinesis에 옵니다.

샤드 수와 동일합니다(따라서 하나의 Lambda 함수가

그림 3.1과 같이 샤드).

Lambda 함수가 배치 처리에 실패하면 작업을 재시도합니다. 이것은 할 수 있습니다

최대 24시간 동안 계속 진행합니다(Kinesis가 이전에 데이터를 유지하는 시간

만료됨) 처리가 실패할 때마다.

Kinesis Streams는 Lambda 함수로 처리
할 수 있는 많은 메시지를 수집할 수 있습니다.
실시간 보고 및 분석을 수행하는 데이터 집약적 애플리케이션은 이 아키텍처의 이점을 누릴 수 있습니다.

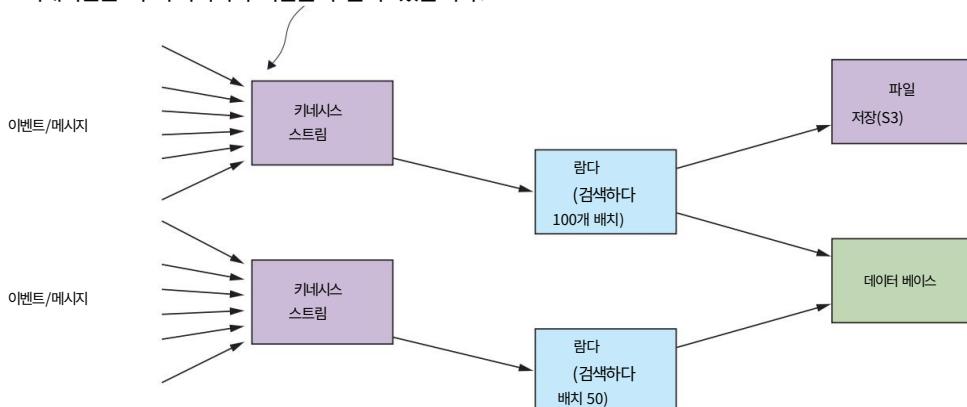


그림 3.1 Lambda는 거의 실시간으로 데이터를 처리하는 완벽한 도구입니다.

Amazon Kinesis Firehose는 기가바이트의 데이터를 수집하도록 설계된 또 다른 Kinesis 서비스입니다. 스트리밍 데이터를 S3, RedShift 또는 Elasticsearch와 같은 다른 서비스로 푸시 추가 분석을 위해. Firehose는 완전히 관리되는 진정한 서비스입니다. 들어오는 데이터의 양에 따라 자동으로 확장되며 필요하지 않습니다. Kinesis Data Streams의 경우처럼 샤퍼드에 대해 생각하는 것입니다.

Kinesis Firehose의 가장 큰 특징은 Lambda 함수를 추가할 수 있다는 것입니다. 데이터가 추가될 때와 최종 대상으로 전송되기 전에 데이터를 원활하게 처리하기 위한 스트림입니다. 프로비저닝할 필요 없이 전송 중인 데이터를 변환하는 데 사용할 수 있습니다. 다른 모든 인프라. 우리는 지금 훨씬 더 깊이 들어가지 않을 것입니다. 6장과 9장에서 Kinesis의 사용 사례와 애플리케이션에 대해 논의하기 때문입니다. 제품을 더 자세히.

3.1.5 레거시 API 프록시

우리가 본 Amazon API Gateway 및 Lambda의 혁신적인 사용 사례 몇 번은 우리가 레거시 API 프록시라고 부르는 것입니다. 여기에서 개발자는 API Gate 방식과 Lambda를 사용하여 레거시 API 및 서비스 위에 새로운 API 계층을 생성합니다. 사용하기 더 쉽습니다.

API Gateway는 RESTful 인터페이스를 생성하고 Lambda 함수는 수정 레거시 서비스가 이해할 수 있는 형식으로 데이터를 요청/응답하고 마샬링합니다. 그만큼 API Gateway 및 Lambda 함수는 클라이언트의 요청을 변환하고 그림 3.2와 같이 레거시 서비스를 직접 호출합니다. 이 접근 방식을 사용하면 이전 프로토콜을 지원하지 않을 수 있는 최신 클라이언트가 레거시 서비스를 더 쉽게 사용할 수 있으며 데이터 형식.

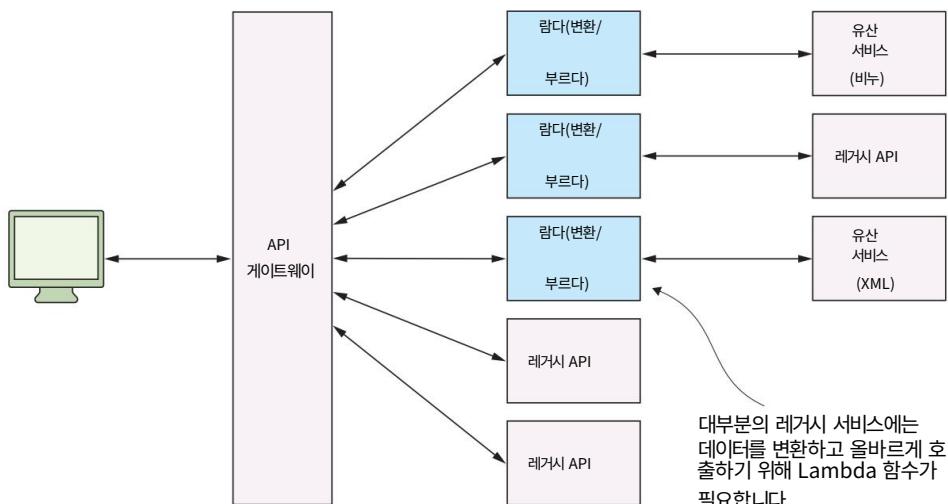


그림 3.2 API 프록시 아키텍처를 사용하여 이전 서비스 및 API 위에 최신 API 인터페이스를 구축할 수 있습니다.

API Gateway가 (어느 정도) 변환하고 문제를 일으킬 수 있다는 점에 유의하는 것이 중요합니다.

다른 HTTP 엔드포인트에 대한 요청 그러나 그것은 상당히 기본적인 숫자에서만 작동합니다.

JSON 변환이 필요한 제한된 사용 사례. 그러나 보다 복잡한 시나리오에서는 데이터를 변환하고 요청을 발행하고 처리하기 위해 Lambda 함수가 필요합니다.
응답.

SOAP(Simple Object Access Protocol) 서비스를 예로 들어 보겠습니다. 당신은 필요합니다

SOAP 서비스에 연결하는 Lambda 함수를 작성한 다음 응답을 매핑하기 위해

JSON. 고맙게도, 예 있는 무거운 작업의 많은 부분을 처리할 수 있는 라이브러리가 있습니다.

람다 함수; 예를 들어 다운로드할 수 있는 SOAP 클라이언트가 있습니다.

이 목적을 위해 npm 레지스트리에서 가져옵니다(<https://www.npmjs.com/package/soap> 참조).

3.1.6 정기 서비스

Lambda 함수는 일정에 따라 실행할 수 있으므로 반복적인 작업에 효과적입니다.

데이터 백업, 가져오기 및 내보내기, 미리 알림 및 경고와 같은 작업. 우리는 개발자가 일정에 따라 Lambda 함수를 사용하여 주기적으로 웹사이트를 ping하여 다음을 확인하는 것을 보았습니다.

그들은 온라인 상태이고 그렇지 않은 경우 이메일이나 문자 메시지를 보냅니다. 당신은 람다를 찾을 수 있습니다

이에 사용할 수 있는 청사진(청사진은 다음을 수행할 수 있는 샘플 코드가 있는 템플릿입니다.

새 Lambda 함수를 생성할 때 선택됨).

또한 개발자가 야간 다운로드를 수행하기 위해 Lambda 함수를 작성하는 것을 보았습니다.

서버에서 파일을 제거하고 사용자에게 일일 계정 명서서를 보냅니다. 반복 작업

파일 백업 및 파일 유효성 검사와 같은 작업도 Lambda를 통해 쉽게 수행할 수 있습니다.

설정하고 잊어버릴 수 있는 스케줄링 기능. 스케줄링 서비스를 생각하고 구축하는 방법에 대한 심층 분석은 7장을 확인하십시오.

3.1.7 봇 및 기술

Lambda 함수 및 서비스 기술의 또 다른 인기 있는 용도는 봇을 구축하는 것입니다.

(봇은 자동화된 작업을 실행하는 앱 또는 스크립트임) Slack과 같은 서비스를 위한 것입니다. 봇

made for Slack은 명령에 응답하고, 작은 작업을 수행하고, 보고서를 보내고

알림. 예를 들어 우리는 Lambda에 Slack 봇을 구축하여 숫자를 보고했습니다.

메일 온라인 판매가 이루어집니다. 그리고 우리는 개발자들이 Telegram용 봇을 구축하는 것을 보았습니다.

Skype 및 Facebook의 메신저 플랫폼.

마찬가지로 개발자는 Amazon용 Alexa 기술을 강화하기 위해 Lambda 함수를 작성합니다.

에코. Amazon Echo는 음성 명령에 응답하는 핸즈프리 스피커입니다. 그것은 작동한다

Alexa라는 가상 비서. 개발자는 Alexa의 기능을 더욱 확장 하는 기술을 구현할 수 있습니다 (스킬은 본질적으로 사람의 음성에 응답할 수 있는 앱입니다).

자세한 내용은 <http://amzn.to/2b5NMFj> 참조). 기술을 작성하여 주문할 수 있습니다.

피자를 먹거나 지리에 대한 퀴즈를 풀 수 있습니다. Alexa는 전적으로 음성으로 구동되며 기술은 람다에 의해 구동.

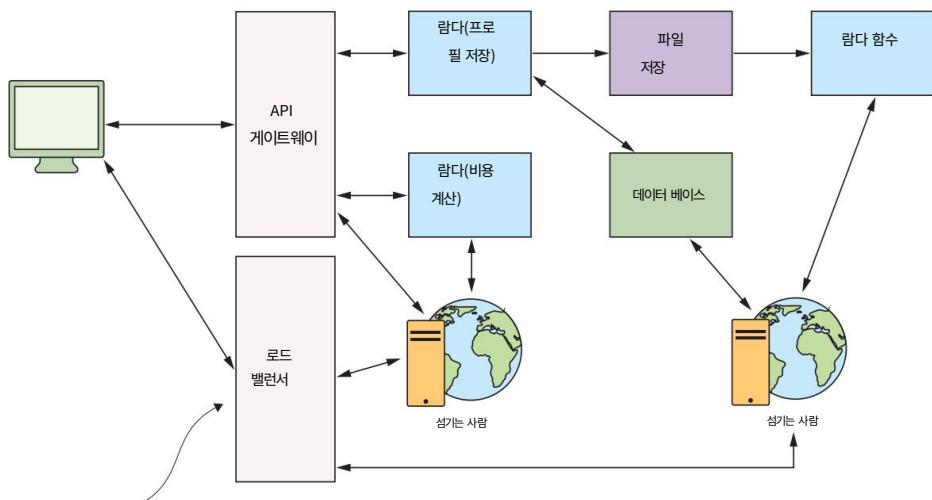
3.1.8 하이브리드

1장에서 언급했듯이 서비스 기술과 아키텍처는 '전부 아니면 전무'가 아닙니다. 기존 시스템과 함께 채택 및 사용할 수 있습니다.

이 하이브리드 접근 방식은 기존 인프라의 일부가 다음과 같은 경우에 특히 효과적일 수 있습니다.

이미 AWS에 있습니다. 우리는 또한 서비스 기술 및 아키텍처의 채택을 보았습니다.

개발자가 있는 조직에서 초기에 독립형 구성 요소를 만들고(종종 추가 데이터 처리, 데이터베이스 백업 및 기본 경고를 수행하기 위해) 시간이 지남에 따라 이러한 구성 요소를 기본 시스템에 통합합니다(그림 3.3).



모든 레거시 시스템은 기능과 서비스를 사용할 수 있습니다.
다. 이를 통해 세계 질서를 너무 많이 방해하지 않으면서 서
버리스 기술을 천천히 도입할 수 있습니다.

그림 3.3 하이브리드 접근 방식은 서버를 사용하는 레거시 시스템이 있는 경우에 유용합니다.

3.2 패턴

패턴은 소프트웨어 설계의 문제에 대한 아키텍처 솔루션입니다. 소프트웨어 개발에서 발견되는 일반적인 문제를 해결하도록 설계되었습니다. 또한 솔루션에 대해 함께 작업하는 개발자를 위한 탁월한 통신 도구이기도 합니다. 방에 있는 모든 사람이 적용 가능한 패턴, 작동 방식, 장점 및 단점을 이해한다면 문제에 대한 답을 훨씬 쉽게 찾을 수 있습니다.

이 섹션에 제시된 패턴은 서비스 아키텍처의 설계 문제를 해결하는 데 유용합니다. 그러나 이러한 패턴은 서비스에만 국한되지 않습니다. 서비스 기술이 실행되기 훨씬 전에 분산 시스템에서 사용되었습니다.

이 장에서 제시된 패턴 외에도 인증, 데이터 관리(예: CQRS, 이벤트 소싱, 구체화된 보기) 및 오류 처리(예: 재시도 패턴)와 관련된 패턴에 익숙해지는 것이 좋습니다.

이러한 패턴을 배우고 적용하면 사용하기로 선택한 플랫폼에 관계없이 더 나은 소프트웨어 엔지니어가 될 수 있습니다. 이러한 패턴 중 몇 가지를 살펴보겠습니다.

3.2.1 GraphQL

GraphQL (<http://graphql.org>) 2012년 Facebook에서 개발하고 2015년에 공개적으로 발표한 인기 있는 데이터 쿼리 언어입니다. 인식된 약점(다중 왕복, 오버 폐칭 및 버전 관리 문제) 때문에 REST(Representational State Transfer)의 대안으로 설계되었습니다.). GraphQL은 해결을 시도합니다.

단일 끝점(예: api/graphql)에서 쿼리를 수행하는 계층적, 선언적 방법을 제공하여 이러한 문제를 해결합니다. 그림 3.4는 GraphQL 및 AWS Lambda 구현의 예를 보여줍니다.

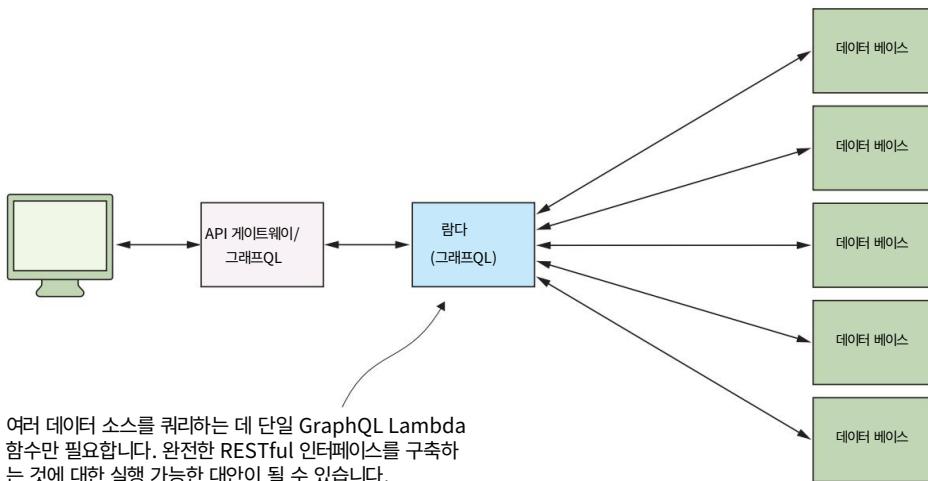


그림 3.4 GraphQL 및 Lambda 아키텍처는 서비스 커뮤니티에서 대중화되었습니다.

GraphQL은 클라이언트에 전원을 제공합니다. 서버에서 응답의 구조를 지정하는 대신 클라이언트에서 정의됩니다 (<http://bit.ly/2aTjh5>). 클라이언트는 반환할 속성과 관계를 지정할 수 있습니다. GraphQL은 여러 소스의 데이터를 집계하여 단일 왕복으로 클라이언트에 반환하므로 데이터 검색을 위한 효율적인 시스템이 됩니다. Facebook에 따르면 GraphQL은 거의 1,000개에 가까운 다양한 버전의 애플리케이션에서 초당 수백만 건의 요청을 처리합니다.

GraphQL 라이브러리(서버)는 Lambda 함수에서 호스팅 및 실행할 수 있습니다. 또한 <https://aws.amazon.com/appsync/>에서 항상 인기 있는 AWS AppSync와 같은 GraphQL의 관리형 솔루션을 찾을 수 있습니다.

이것을 사용할 때

GraphQL은 여러 위치에서 데이터를 집계할 수 있는 복합 패턴 유형입니다. 여러 데이터 소스에서 데이터를 읽고 수화하는 것은 웹 애플리케이션, 특히 마이크로서비스 접근 방식을 채택하는 애플리케이션에서 일반적입니다. 더 작은 페이로드, 데이터 모델을 재구축할 필요가 없고 더 이상 버전이 지정된 API(REST에 비해)를 포함하는 다른 이점도 있습니다. 이것은 GraphQL이 지난 몇 년 동안 인기를 얻은 이유 중 일부일 뿐입니다.

3.2.2 명령 패턴

이전 섹션에서 단일 엔드포인트를 사용하여 서로 다른 데이터가 있는 다양한 요청을 처리할 수 있다는 사실을 언급했습니다(예를 들어 단일 GraphQL 엔드포인트는 클라이언트의 필드 조합을 수락하고 일치하는 응답을 생성할 수 있습니다. 요구). 같은 생각이 더 일반적으로 적용될 수 있습니다. 당신은 디자인 할 수 있습니다

특정 Lambda 함수가 다른 함수를 제어하고 호출하는 시스템. 너 API Gateway에 연결하거나 수동으로 호출하고 메시지를 전달할 수 있습니다. 다른 Lambda 함수를 호출합니다.

소프트웨어 엔지니어링에서 명령 패턴 (그림 3.5)은 요청을 객체로 사용하여 다른 요청으로 클라이언트를 매개변수화 할 수 있습니다. 요청을 대기열에 넣거나 기록하고 실행 취소 할 수 있는 작업을 지원합니다." 요청 중인 작업 또는 요청 수신자" (<http://bit.ly/29ZaoWt>). 명령 패턴을 사용하면 필요한 처리를 수행하는 엔터티에서 작업 호출자를 분리할 수 있습니다.

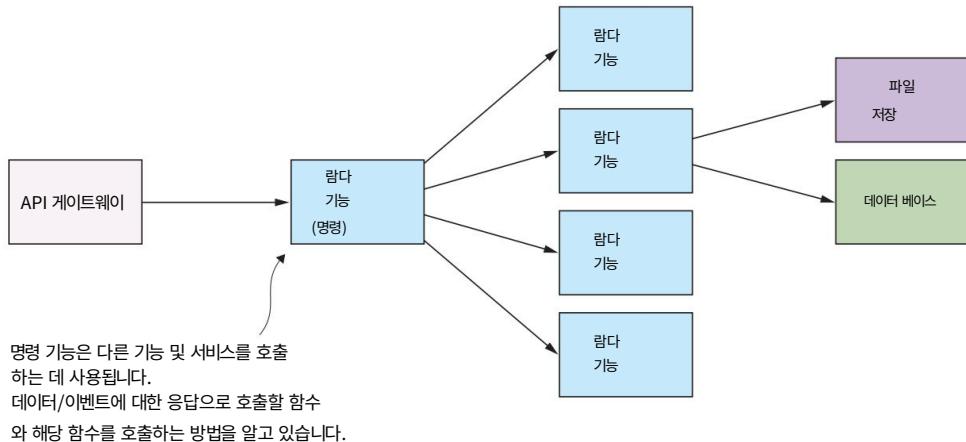


그림 3.5 명령 패턴은 단일 기능에서 기능과 서비스를 호출하고 제어합니다.

실제로 이 패턴은 API Gateway 구현을 단순화 할 수 있습니다. 모든 요청에 대해 RESTful URI를 생성하는 것을 원하지 않거나 생성할 필요가 없을 수 있습니다. 또한 버전 관리를 더 간단하게 만들 수 있습니다. 명령 Lambda 함수는 다른 버전에서 작동할 수 있습니다. 클라이언트에 필요한 올바른 Lambda 함수를 호출합니다.

이것을 사용할 때

이 패턴은 호출자와 수신자를 분리하려는 경우에 유용합니다. 방법을 가지고 인수를 객체로 전달하고 클라이언트가 다른 매개변수로 매개변수화 할 수 있도록 합니다. 요청은 구성 요소 간의 결합을 줄이고 시스템을 더 많이 만들 수 있습니다. 확장 가능.

3.2.3 메시징 패턴

메시징 패턴(그림 3.6)은 분산 시스템에서 널리 사용됩니다. 기능과 서비스를 분리하여 확장 가능하고 강력한 시스템을 구축하는 개발자 서로에 대한 직접적인 의존과 이벤트/기록/ 대기열의 요청. 신뢰성은 소비하는 서비스가 오프라인이 되면 큐는 메시지를 (일정 기간 동안) 유지하며 나중에 계속 처리할 수 있습니다.

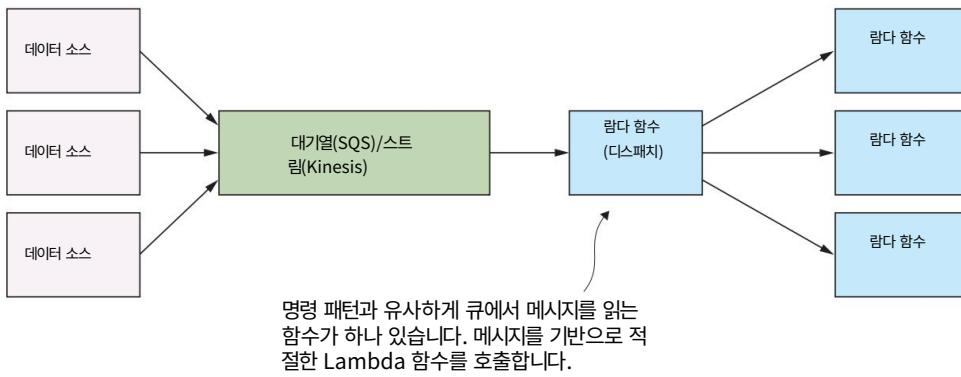
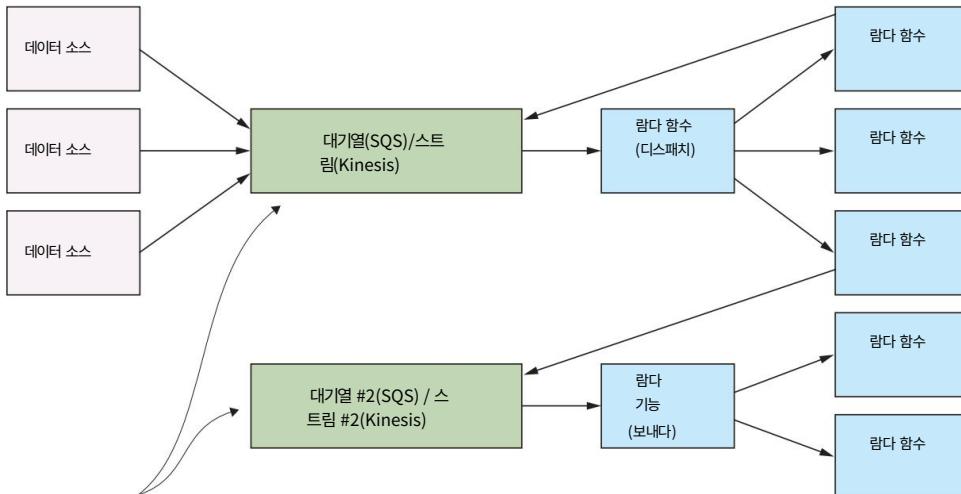


그림 3.6 메시징 패턴과 그 다양한 변형은 분산 환경에서 널리 사용됩니다.

이 패턴은 대기열에 게시할 수 있는 발신자와 대기열에서 메시지를 검색할 수 있는 수신자가 있는 메시지 대기열을 특징으로 합니다. AWS에서의 구현 측면에서 SQS 위에 이 패턴을 구축할 수 있습니다.

시스템 설계 방식에 따라 메시지 대기열에는 단일 발신자/수신자 또는 여러 발신자/수신자가 있을 수 있습니다. SQS 대기열에는 일반적으로 대기열당 하나의 수신자가 있습니다. 여러 소비자가 필요한 경우 이를 수행하는 간단한 방법은 시스템에 여러 대기열을 도입하는 것입니다(그림 3.7). 적용할 수 있는 전략은 SQS를 Amazon SNS와 결합하는 것입니다. SQS 대기열은 SNS 주제를 구독할 수 있으므로 주제에 메시지를 푸시하면 구독된 모든 대기열에 메시지가 자동으로 푸시됩니다.



여러 대기열/스트림을 사용하여 시스템의 여러
구성 요소를 분리합니다.

그림 3.7 시스템에는 들어오는 모든 데이터를 처리하기 위해 여러 대기열 또는 스트림과 Lambda 함수가 있을 수 있습니다.

이것을 사용할 때

메시징 패턴은 워크로드 및 데이터 처리를 처리합니다. 대기열은 다음과 같은 역할을 합니다.
버퍼를 사용하므로 소비하는 서비스가 충돌하더라도 데이터가 손실되지 않습니다. 까지 대기열에 남아 있습니다.
서비스를 다시 시작하고 다시 처리를 시작할 수 있습니다.

메시지 대기열은 기능 간의 결합이 적기 때문에 향후 변경도 더 쉽게 만들 수 있습니다. 데이터 처리, 메시지, 요청이 많은 환경에서는 직접 처리되는 기능의 수를 최소화하도록 노력합니다.

다른 기능에 의존하고 대신 메시징 패턴을 사용합니다.

3.2.4 우선순위 큐 패턴

AWS 및 서비스 아키텍처와 같은 플랫폼을 사용할 때의 큰 이점은

용량 계획 및 확장성은 Amazon 엔지니어에게

당신을 위한. 그러나 경우에 따라 메시지가 처리되는 방법과 시기를 제어할 수 있습니다.

귀하의 시스템에 의해. 여기에 다른 대기열, 주제 또는

스트림을 사용하여 함수에 메시지를 제공합니다.

시스템은 한 단계 더 나아가 다른 우선 순위(우선 순위 대기열 패턴)의 메시지에 대해 완전히 다른 워크플로를 가질 수 있습니다. 즉시 필요한 메시지
더 비싼 것을 사용하여 프로세스를 촉진하는 흐름을 통해 주의를 끌 수 있습니다.
더 많은 용량의 서비스 및 API. 처리할 필요가 없는 메시지
그림 3.8과 같이 다른 워크플로를 빠르게 진행할 수 있습니다.

다른 우선 순위를 가진 메시지는 다른
워크플로와 다른 Lambda 함수로 처
리할 수 있습니다.

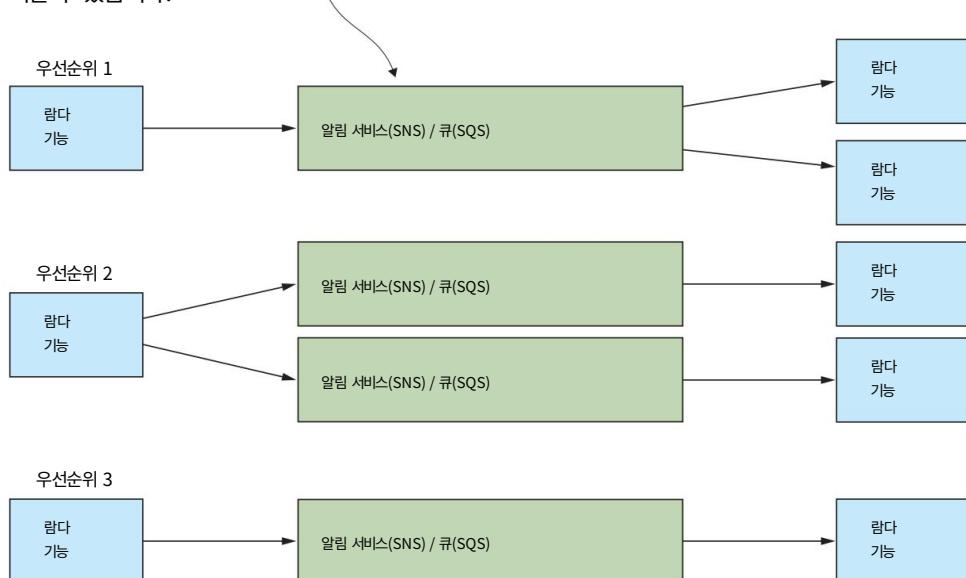


그림 3.8 우선순위 큐 패턴은 메시징 패턴의 진화입니다.

우선순위 큐 패턴은 완전히 다른 생성 및 사용을 포함할 수 있습니다.

SNS 주제, SQS 대기열, Lambda 함수 및 타사 서비스까지. 그러나 추가 구성 요소, 종속성 및 작업 흐름으로 인해 더 복잡해지기 때문에 이 패턴을 드물게 사용하십시오.

[이것을 사용할 때](#)

이 패턴은 메시지 처리에 대해 다른 우선 순위를 가져야 할 때 작동합니다. 시스템은 워크플로를 구현하고 다양한 서비스와 API를 사용하여

다양한 유형의 요구 사항과 사용자를 충족합니다(예: 유료 사용자와 무료 사용자).

3.2.5 팬아웃 패턴

팬아웃은 많은 AWS 사용자에게 익숙한 메시징 패턴 유형입니다. 일반적으로,

팬아웃 패턴은 특정 클라이언트의 모든 수신/구독 클라이언트에 메시지를 푸시합니다.

큐 또는 메시지 파이프라인. AWS에서 이 패턴은 일반적으로 SNS를 사용하여 구현됩니다.

새 메시지가 추가될 때 여러 구독자를 호출할 수 있는 주제

주제.

S3를 예로 들어보겠습니다. 버킷에 새 파일이 추가되면 S3는 파일에 대한 정보를 사용하여 단일

Lambda 함수를 호출할 수 있습니다. 하지만 호출해야 하는 경우

동시에 2개, 3개 또는 그 이상의 Lambda 함수? 원래 기능은

다른 기능(예: 명령 패턴)을 호출하도록 수정되지만,

기능을 병렬로 실행하기만 하면 됩니다. 해결책은 팬아웃을 사용하는 것입니다.

SNS를 통한 패턴(그림 3.9 참조).

SNS 주제에 추가된 메시지는 여러 Lambda 함수를 병렬로 강제 호출할 수 있습니다.

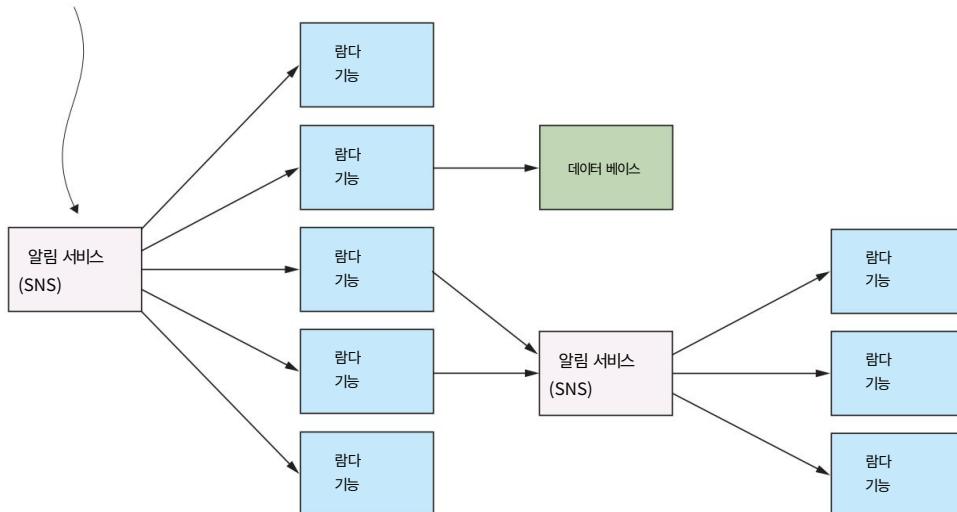


그림 3.9 팬아웃 패턴은 이벤트가 발생할 때 많은 AWS 서비스(예: S3)가 한 번에 둘 이상의 Lambda 함수를 호출할 수 없기 때문에 유용합니다.

SNS 주제는 여러 게시자와 구독자(Lambda 기능 포함)를 가질 수 있는 통신 또는 메시징 채널입니다. 새 메시지가 추가되면 주제에 대해 모든 구독자를 병렬로 강제 호출하여 이벤트를 발생시킵니다.

팬 아웃 .

단일 항목을 호출하는 대신 앞서 설명한 S3 예제로 돌아가서 Lambda 함수를 사용하면 메시지를 SNS 주제로 푸시하도록 S3를 구성할 수 있습니다. 구독한 모든 기능을 동시에 호출합니다. 이벤트 기반 아키텍처를 만들고 작업을 병렬로 수행하는 효과적인 방법입니다. 8장 사용 방법을 보여줍니다.

이 패턴을 사용하여 대규모로 비디오 인코딩을 수행합니다.

이것을 사용할 때

이 패턴은 동시에 여러 Lambda 함수를 호출해야 하는 경우에 유용합니다.

시각. SNS 주제는 전달에 실패할 경우 Lambda 함수를 호출하여 재시도합니다.

메시지 또는 함수 실행에 실패한 경우(<https://go.aws/3DTdCEK> 참조).

또한 팬아웃 패턴은

여러 람다 함수. SNS 주제는 이메일,

SQS 대기열. 주제에 새 메시지를 추가하면 Lambda 함수를 호출하고

이메일을 보내거나 SQS 대기열에 메시지를 동시에 푸시합니다.

SQS 대 SNS 대 EventBridge

어떤 상황에서 어떤 AWS 서비스를 사용해야 하는지 알기 어려운 경우가 있습니다. 언제

이벤트 메시징에 관해서는 SQS와 SNS에 대해 이미 논의했지만

Amazon EventBridge는 제품군을 완성합니다. 다음 Lumigo 블로그 게시물 기능

이러한 서비스에 대한 훌륭한 요약 및 비교: <https://bit.ly/3AYdJga>. 우리

차이점과 사용 사례를 이해하려는 경우 살펴보는 것이 좋습니다. 또 다른 좋은 설명은 AWS 자체에서 나온 것입니다.

<https://go.aws/3phPOWW>.

3.2.6 플로 계산하기

Compute-as-glue 아키텍처(그림 3.10)는 우리가 사용할 수 있는 아이디어를 설명합니다.

강력한 실행 파이프라인 및 워크플로를 생성하는 Lambda 함수. 이 자주

Lambda를 서로 다른 서비스 간의 접착제로 사용, 조정 및 호출 포함

그들을. 이러한 스타일의 아키텍처에서 개발자의 초점은 디자인에 있습니다.

파이프라인, 조정 및 데이터 흐름. Lambda와 같은 서비스 컴퓨팅 서비스의 병렬 처리는 이러한 아키텍처를 매력적으로 만드는 데 도움이 됩니다.

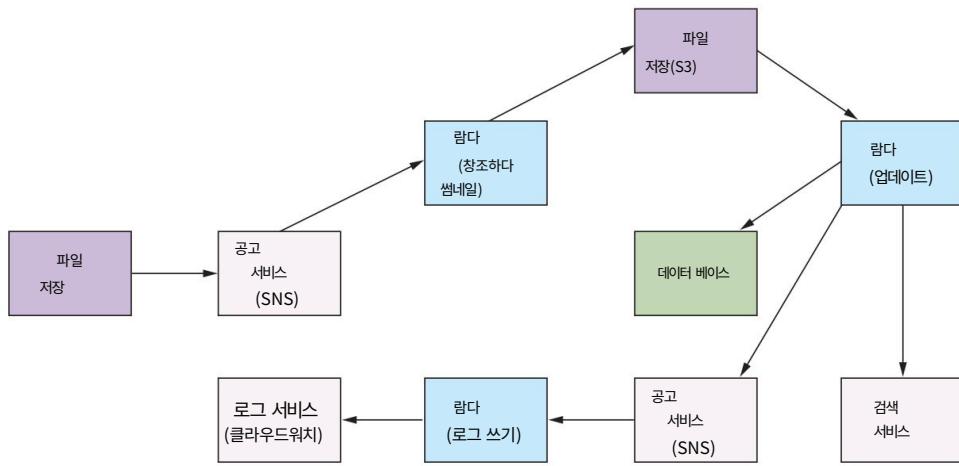


그림 3.10 접착제로서의 컴퓨팅 아키텍처는 Lambda 함수를 사용하여 다양한 서비스와 API를 연결하여 작업을 수행합니다. 이 파이프라인에서 간단한 이미지 변환은 새 파일, 데이터베이스 업데이트, 검색 서비스 업데이트 및 로그 서비스에 대한 새 항목을 생성합니다.

3.2.7 파이프 및 필터 패턴

파이프 및 필터의 목적은 복잡한 처리를 분해하는 것입니다.

파이프라인으로 구성된 일련의 관리 가능한 개별 서비스로 작업을 수행합니다(그림 3.11). 데이터를 변환하도록 설계된 구성 요소를 전통적으로 필터라고 합니다. 한 구성 요소에서 다음 구성 요소로 데이터를 전달하는 커넥터는 파이프라고 합니다. 서비스 아키텍처는 이러한 종류의 패턴에 적합합니다. 이것 결과를 얻기 위해 여러 단계가 필요한 모든 종류의 작업에 유용합니다.

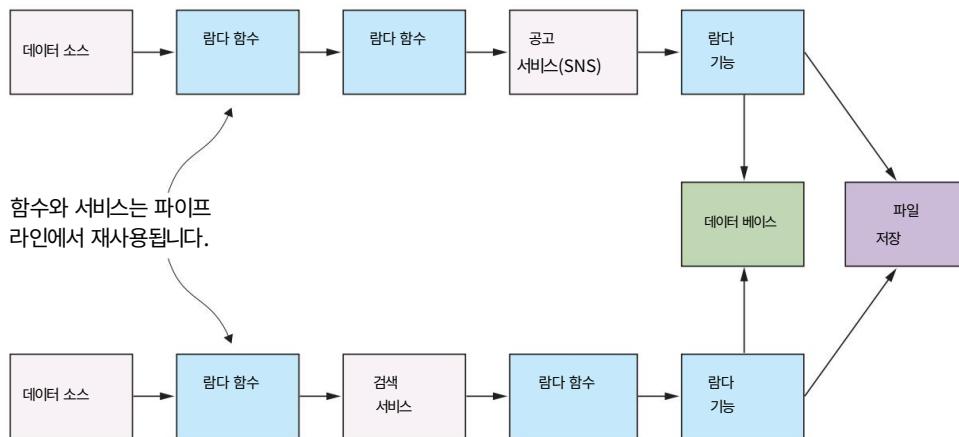


그림 3.11 파이프 및 필터 패턴은 파이프라인 구성이 데이터를 원본(펌프)에서 대상(싱크)으로 전달하고 변환하도록 권장합니다.

이 패턴을 사용하면 모든 Lambda 함수를 세부적으로 작성하는 것이 좋습니다.
 단일 책임 원칙을 염두에 둔 서비스 또는 작업. 입력 및 출력
 명확하게 정의되어야 하고(명확한 인터페이스가 있어야 함) 부작용을 최소화해야 합니다. 이 조언을 따르면 다음에서 재사용할 수 있는 함수를 만들 수 있습니다.
 파이프라인, 더 넓게는 서비스 시스템 내에서.
 이 패턴이 우리가 사용하는 접착제로 계산 아키텍처와 유사하다는 것을 알 수 있습니다.
 이전에 설명했습니다. 당신이 옳습니다. 접착제로 계산하고 이 패턴은 밀접하게
 관련이 있으며 단순히 동일한 개념의 변형입니다.

이것을 사용할 때

복잡한 작업이 있는 경우 일련의 기능(파이프 라인)으로 나누고 다음 규칙을 적용하십시오.

귀하의 기능이 단일 책임 원칙을 따르는지 확인하십시오.
 함수에 대한 인터페이스를 명확하게 정의하십시오. 입력 및 출력이 다음과 같은지 확인하십시오.
 분명히 밝혔습니다.
 블랙박스를 생성합니다. 함수의 소비자는 함수가 어떻게 작동하는지 알 필요가 없습니다.
 작동하지만 사용하려면 어떤 종류의 출력이 예상되는지 알아야 합니다.

이 책의 나머지 부분에서 패턴에 대해 논의하고 더 많은 컨텍스트를 제공할 것입니다.
 그리고 우리가 여기에서 탐구한 건축. 이를 염두에 두고 다음 챕터로 넘어가 Yubi이라는 소셜 네트워크에 대한 이야기를 읽어 보겠습니다.

요약

서비스 아키텍처는 구축을 포함한 다양한 사용 사례를 지원할 수 있습니다.
 웹, 모바일 및 IoT 애플리케이션을 위한 백엔드는 물론 데이터 처리 및
 해석학.
 AWS Lambda와 같은 서비스 기술은 유연합니다. 그들은 결합 될 수 있습니다
 컨테이너 또는 가상 머신을 사용하여 하이브리드 아키텍처로. 당신은 할 필요가 없습니다
 훌륭한 결과를 달성하기 위해 서비스 순수주의자가 되십시오.
 GraphQL과 같은 특정 패턴과 접근 방식은 서비스에 적합합니다.
 AppSync와 같은 AWS 서비스가 준비되어 있고 나머지 아키텍처와 원활하게 통합될 수 있기 때문입니다.
 메시징 패턴과 같은 고전적인 소프트웨어 엔지니어링 패턴은 서비스 아키텍처 및 SQS와 같은 AWS
 제품과 매우 잘 작동합니다.
 팬아웃 패턴은 보다 일반적인 패턴 중 하나입니다. 설정하는 방법을 알고
 AWS를 효과적으로 사용하려면 Amazon SNS를 사용하는 것이 중요합니다.
 AWS에는 겹치는 다양한 서비스와 제품이 많이 있습니다. 철저한
 각 서비스를 언제 사용해야 하는지 이해하면 더 나은 결정을 내리는 데 도움이 됩니다.

2 부

사용 사례

1부를 읽었으며 이제 이해가 잘 되었기를 바랍니다.

서버리스의 모든 것. 세 회사가 어떻게 되는지 살펴보는 시간입니다.

서비스 아키텍처를 사용하여 문제를 해결하고 고객을 기쁘게 합니다. 예
2부에서는 Yubl, A Cloud Guru 및 Yle의 세 가지 사용 사례 연구를 제시합니다.

Yubl: 아키텍처 하이 라이트, 배운 교훈

이 장에서는 다음을 다룹니다.

원래 Yubl 아키텍처와 문제

새로운 서비스 아키텍처와 그에 따른 결정

모놀리스 애플리케이션을 서비스로 전환하기 위한 전략 및 패턴

이 마이그레이션에서 얻은 교훈

2016년 4월에 나는 Yubl이라는 런던에 기반을 둔 소셜 네트워크에 가입했습니다. 거기에서 Node.js로 작성되고 소수의 EC2(Elastic Compute Cloud) 인스턴스에서 실행되는 모놀리식 백엔드 시스템을 상속했습니다. 원래 시스템은 구현하는데 2.5년이 걸렸으며 일단 가동되면 성능 및 확장성 문제로 인해 많은 문제가 발생했습니다. 6명의 엔지니어로 구성된 소규모 팀과 함께 6개월 동안 플랫폼을 서비스로 전환하는 데 성공했습니다. 그 과정에서 많은 새로운 기능을 추가하고 기존 성능 및 확장성 문제를 해결했습니다. 기능 제공 시간을 몇 개월에서 며칠, 경우에 따라 몇 시간으로 단축했습니다. 비용이 이 변화를 수행하는 주요 동기는 아니었지만 그 과정에서 AWS 청구서를 95% 절감했습니다. 원래 Yubl 아키텍처를 살펴보겠습니다.

4.1 원래 Yubl 아키텍처

Yubl("소셜 버블"의 약자)은 모바일 우선의 소셜 네트워크입니다.

17~25세 인구통계. 포함 된 사용자 생성 게시물(yubls라고 함)

비디오뿐만 아니라 애니메이션 및 인터랙티브 요소. 앱에는 다른 소셜 네트워크에서 찾을 수 있는 모든 소셜 기능이 있습니다. 사용자 팔로우, 개인 및 그룹 채팅, 좋아요 표시

콘텐츠를 다시 공유하는 등.

원래 아키텍처(그림 4.1)는 다음으로 구성됩니다.

Node.js로 작성되고 EC2에서 실행되는 모놀리식 REST API

Node.js로 작성되고 EC2에서 실행되는 WebSockets API

MongoLab에서 호스팅되는 모놀리식 MongoDB 데이터베이스

CloudAMQP 메시지 대기열

Node.js로 작성되고 EC2에서 실행되는 백그라운드 작업자 클러스터

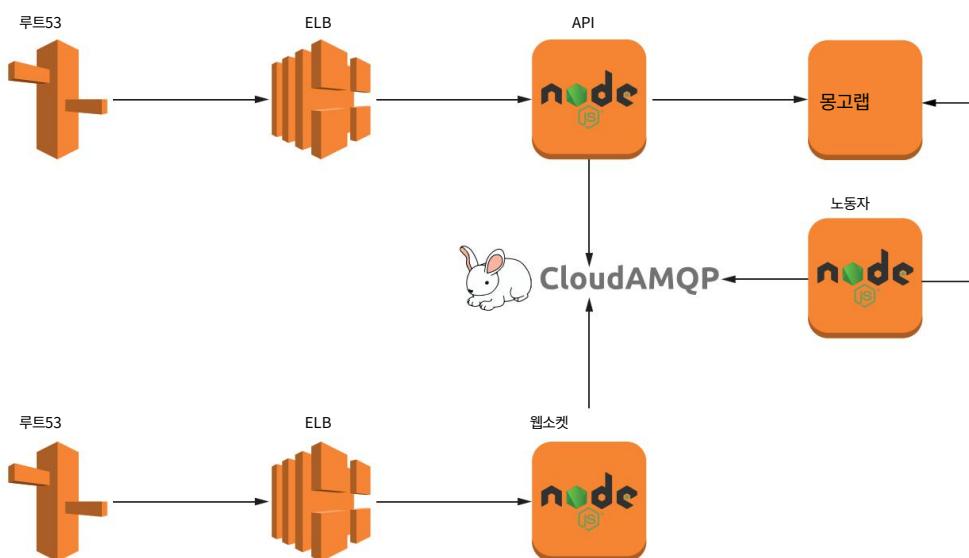


그림 4.1 원래 Yubl 아키텍처의 상위 수준 개요

MongoDB와 MongoLab이란 무엇입니까?

MongoDB는 다음을 저장할 수 있는 인기 있는 문서 지향 NoSQL 데이터베이스입니다.

JSON 문서. <https://www.mongodb.com> 에서 자세히 알아볼 수 있습니다.

MongoLab은 MongoDB 호스팅을 서비스로 제공하는 온라인 서비스입니다. 당신은 할 수 있습니다

몇 번의 클릭으로 MongoDB 클러스터를 생성하면 MongoLab이 기본 인프라를 관리합니다. <https://mlab.com> 에서 자세히 알아볼 수 있습니다. 뒤

2016년에 MongoLab은 기본 인프라를 직접 관리할 필요 없이 MongoDB를 실행할 수 있는 실행 가능한 서비스였습니다.

RabbitMQ 및 CloudAMQP란 무엇입니까?

AMQP(Advanced Message Queuing Protocol)는 메시지 지향 미들웨어를 위한 애플리케이션 프로토콜입니다. 메시지 큐링 및 라우팅을 지원하며 종종 발행 및 구독 시스템에서 사용됩니다.

RabbitMQ는 AMQP 프로토콜을 구현하는 오픈 소스 메시지 브로커입니다. <https://www.rabbitmq.com/>에서 RabbitMQ에 대해 자세히 알아볼 수 있습니다.

CloudAMQP는 RabbitMQ 호스팅을 제공하는 온라인 서비스입니다. 당신은 배울 수 있습니다 <https://cloudamqp.com>에서 자세히 알아보십시오.

4.1.1 확장성 문제

초기 단계의 소셜 네트워크이기 때문에 Yubi의 기본 트래픽은 낮았지만 여러 유명 Instagram 인플루언서를 플랫폼으로 끌어들이는 데 성공했습니다. 이 인플루언서는 많은 Instagram 팔로어와 함께 수만 명의

팔로워, 이러한 인플루언서는 시스템을 통해 예측할 수 없고 급증하는 트래픽을 유도했습니다.

그들이 새로운 콘텐츠를 게시할 때마다.

수천 명의 사용자가 한 번에 몰려들면서 트래픽이 100배 급증하는 것을 종종 보았습니다.

좋아하는 인플루언서의 새로운 콘텐츠를 확인하세요. 이러한 트래픽 급증은 일반적으로 수명이 짧았으며 EC2 AutoScaling 때문에 EC2 기반 시스템에 문제가 되었습니다.

충분히 빠르게 반응하지 못했습니다. 일반적으로 EC2 인스턴스를 가동하는 데 몇 분이 걸립니다. 예 의해 사용자 요청을 처리할 준비가 되었을 때는 이미 늦었습니다. 트래픽 급증은 와다가 사라지고 많은 사용자가 자연을 경험한 후 떠났을 것입니다.

응답 시간.

해결 방법으로 훨씬 더 큰 EC2 클러스터를 실행하여 이전보다 훨씬 일찍 확장했습니다.

우리는 원했다. 이로 인해 많은 EC2 비용을 지불해야 했기 때문에 많은 비용이 낭비되었습니다.

우리가 사용하지 않은 자원. API 웹 서버 클러스터의 평균 사용률은 2~5%였습니다.

4.1.2 성능 문제

모놀리식 MongoDB 데이터베이스는 또한 성능의 지속적인 소스였으며 확장성 문제. 모든 읽기 및 쓰기 작업은 데이터베이스에 직접 영향을 미칩니다. 일부 API 작업은 MongoDB 서버에 막대한 피해를 줄 수 있습니다. 이것의 한 예는 자주 사용되는 API 호출이며 복잡한 정규식 쿼리를 실행하는 사용자 검색 MongoDB에 대해. 또 다른 예에는 사용자 권장 사항이 포함되었는데, 이는 현재에 대한 2차 및 3차 연결을 찾기 위해 복잡한 쿼리를 실행했습니다.

사용자(팔로워를 팔로우하는 사람 또는 내가 팔로우하는 사용자가 팔로우하는 사람).

4.1.3 긴 기능 제공 주기

코드베이스는 복잡했고 많은 기능이 공유를 통해 얹혀 있었습니다.

공유 라이브러리를 통한 MongoDB 컬렉션 및 암시적 결합. 비록 거기에

합당한 코드 범위를 가진 많은 단위 테스트가 있었지만 유용하지 않았습니다.

코드 변경은 종종 모든 테스트를 통과했지만 AWS에 배포될 때만 실패하기 때문입니다.

환경. 외부 서비스와의 상호 작용은 철저히 조롱되었으므로 테스트에서 다루지 않았습니다. 많은 경우에 테스트에서는 MongoDB 쿼리에 구문 오류가 포함된 경우에도 모의가 작동하고 요청한 내용을 반환했음을 단순히 확인했습니다. 테스트가 너무 많은 위양성을 제공하기 때문에 테스트에 대한 믿음이 거의 없었습니다.

설상가상으로 모든 배포는 30분 이상 전체 시스템을 중단해야 했으며 이 시간 동안 사용자는 피드백을 받지 못하고 앱이 손상된 것처럼 보였습니다. 기능을 생산하는 데 몇 달이 걸렸습니다. 간단한 변경도 완료하는 데 몇 주가 소요되는 경우가 많았으며 이는 관련된 모든 사람에게 실망스러웠습니다.

4.1.4 왜 서비스인가?

우리 시스템의 요구 사항과 현재 구현이 경험한 문제를 기반으로 하면 다음과 같은 이유로 서비스가 매우 적합했습니다.

AWS Lambda는 로드를 기반으로 동시 실행 수를 자동 조정합니다. 이것은 즉시 발생하며 우리가 경험하는 예측할 수 없는 급증을 손쉽게 처리합니다.

AWS Lambda는 기본적으로 3개의 가용 영역에 기능을 배포하므로 추가 비용 없이 상당한 중복성을 제공합니다. 우리는 기능이 실행될 때만 비용을 지불하는 반면, EC2에서는 다중 AZ 설정에서 중복성에 대해 비용을 지불했는데, 이로 인해 트래픽이 희석되고 리소스 사용률이 더욱 감소했습니다.

AWS는 기본 물리적 인프라와 코드가 실행되는 운영 체제를 관리합니다. AWS는 패치와 보안 업데이트를 정기적으로 적용하고 우리가 할 수 있는 것보다 훨씬 더 나은 운영 체제 보안을 유지합니다. 이것은 전 세계의 수많은 소프트웨어 시스템을 괴롭히는 모든 종류의 취약점을 제거합니다.

Serverless 프레임워크와 같은 도구를 사용하면 애플리케이션의 배포 파이프라인이 대폭 간소화됩니다. AWS Lambda가 자동으로 요청을 새 코드로 라우팅하기 때문에 일반적인 배포는 1분 미만이 소요되며 가동 중지 시간이 없습니다.

API Gateway, Lambda 및 DynamoDB와 같은 서비스 기술을 사용할 때 기본 인프라에 대해 걱정할 필요가 없습니다. 이를 통해 핵심 비즈니스 요구 사항을 해결하는 데 집중할 수 있습니다. 우리 코드의 거의 모든 라인은 비즈니스 로직입니다! 또한 개발 팀은 기본적으로 확장 가능하고 복원력이 있는 빌드를 알고 있기 때문에 빠르게 이동할 수 있습니다.

같은 규모의 팀에서 프로덕션 배포 횟수는 월 4건에서 6건으로 평균 80건 이상으로 늘어났습니다. 더 빨리 작업하기 위해 더 많은 사람을 고용할 필요가 없었습니다. 대신 각 개발자의 생산성을 높일 수 있었습니다.

점점 더 많은 시스템을 서비스로 마이그레이션하면서 확장성, 비용 및 안정성이 모두 향상되었습니다. 프로덕션 문제가 훨씬 적었고 이전에 EC2에 지출한 금액의 일부를 지출하고 있었습니다.

4.2 새로운 서비스 Yubl 아키텍처

내가 회사에 합류하고 서비스로의 여정을 시작한 지 8개월도 채 되지 않은 2016년 11월까지 거의 전체 백엔드 시스템이 서비스로 마이그레이션되었습니다. API Gateway, Lambda, DynamoDB, Kinesis 등과 같은 서비스 조합을 사용합니다. 그 과정에서 우리는 기존 기능을 개선하고 수많은 새로운 기능을 구현했습니다. 또한 이전 시스템에서 많은 보안 문제를 해결했습니다.

전반적으로 시스템의 신뢰성이 크게 향상되었습니다. 짧은 S3(Simple Storage Service) 중단으로 인해 프로덕션 환경에 경미한 중단이 한 번만 발생했습니다. 다음은 AWS의 새로운 서비스 아키텍처의 주요 하이라이트입니다.

모듈리는 많은 마이크로서비스로 분할되었습니다.

모든 마이크로 서비스에는 자체 GitHub 리포지토리와 하나의 CI/CD(지속적 통합/지속적 전달) 파이프라인이 있습니다. 이 마이크로 서비스를 구성하는 모든 구성 요소(API 게이트웨이, Lambda 함수, DynamoDB 테이블 등)는 Serverless Framework를 사용하여 하나의 CloudFormation 스택으로 함께 배포됩니다.

대부분의 마이크로서비스에는 search.yubl.com과 같은 자체 하위 도메인에서 실행되는 API Gateway REST API가 있습니다.

모든 마이크로 서비스에는 필요한 데이터에 대한 자체 데이터베이스가 있습니다. 대부분 DynamoDB를 사용하지만 마이크로 서비스마다 데이터 요구 사항이 다르기 때문에 보편적이지 않습니다.

시스템의 모든 상태 변경은 이벤트로 캡처되어 Kinesis Data Stream에 게시됩니다(예: 사용자가 새 콘텐츠를 생성하고 사용자가 새 콘텐츠를 게시하는 등).

대부분의 경우 런타임에 동기 API 호출보다 이벤트를 통해 마이크로서비스 간에 데이터를 동기화하는 것을 선호합니다. 이렇게 하면 하나의 마이크로 서비스가 프로덕션 중단을 경험할 때 계단식 오류를 방지하는 데 도움이 됩니다.

대신 마이크로서비스는 관련 Kinesis Data Stream을 구독하고 적절한 이벤트에서 필요한 데이터를 복사합니다.

이 다이어그램 (<https://d2qt42rcwzspd6.cloudfront.net/overall.png>)이 새로운 아키텍처의 조감도를 보여줍니다. 그림의 모든 것을 이해하는 것에 대해 걱정하지 마십시오. 서비스 구성 요소를 사용하여 복잡한 시스템도 구축할 수 있다는 사실을 보여줄 뿐입니다.

서비스로의 전환이 우리의 목표 중 하나가 아니었음을 언급할 가치가 있습니다. 목표는 다운타임을 줄이고 응답성을 높이고 확장성을 높여 더 나은 사용자 경험을 제공하는 것이었습니다. Lambda, API Gateway 및 DynamoDB와 같은 서비스 기술은 우리의 삶을 훨씬 더 쉽게 만들고 기능을 더 빨리 제공할 수 있게 하는 동시에 목표를 달성하는 좋은 방법입니다.

4.2.1 재설계 및 재작성

우리의 목표를 완전히 실현하기 위해 우리는 시스템의 많은 부분을 재설계하고 다시 작성해야 했습니다.

그러나 우리는 마이그레이션을 위해 모든 것을 서비스로 마이그레이션하고 싶지 않았습니다.

기능 개발을 가속화하고 사용자에게 더 빠르게 가치를 제공하고 싶었습니다.

전보다 이를 위해 우리는 실용적인 접근 방식을 취하여 사례별로

작업이 필요할 때 기능을 다시 설계할지 여부를 결정합니다.

위험을 완화하기 위해 우리는 최소한의 비용으로 기능을 재설계하고 마이그레이션했습니다.

비즈니스 영향이 먼저입니다. 타임라인과 같은 비즈니스 크리티컬 기능(첫 번째)

앱에서 볼 수 있는 것)은 우리가 충분한 자신감을 얻었을 때만 해결되었습니다.

그리고 노하우. 대규모 시스템을 하나씩 마이그레이션하는 이러한 접근 방식은 일반적으로

스트랭글러 패턴이라고 합니다.

Yubl 앱에서 이름, 성 및 사용자 이름으로 다른 사용자를 검색할 수 있습니다. 이것은 단순한 기능이었지만

사용자 수가 늘어남에 따라 모놀리스. 검색이 구현되었기 때문입니다.

MongoDB에 대한 정규식 쿼리 사용. 이전 구현도 허용하지 않았습니다.

더 정교한 순위, 그래서 사용자는 종종 그들이 찾고 있는 사람을 찾을 수 없습니다.

마케팅 팀에서 영향력 있는 사람을 더 높은 곳으로 끌어내라는 압력이 있었습니다.

많은 사용자가 플랫폼에서 이러한 인플루언서를 팔로우했기 때문에 검색 결과가 표시됩니다.

이것은 우리가 서비스로 재설계하고 마이그레이션한 첫 번째 기능이었습니다.

위험도가 낮으면서도 큰 영향을 미칠 수 있습니다. 추출한 방법에 대해 알아보겠습니다.

모놀리스에서 검색 기능을 찾고 자체적으로 마이크로서비스를 구축했습니다.

REST API.

4.2.2 새로운 검색 API

첫 번째이자 가장 중요한 단계 중 하나는 레거시 모노리스가

상태 변경 사항을 Kinesis Data Streams에 게시합니다. 이것은 우리에게 기초를 주었다

이러한 이벤트를 기반으로 새로운 마이크로서비스를 구축합니다. 검색을 추출하려면

모놀리스에서 기능을 활용하여 새로운 검색 마이크로서비스를 만들었습니다. 그림 4.2는 보여줍니다

이 검색 마이크로 서비스의 상위 수준 아키텍처입니다.

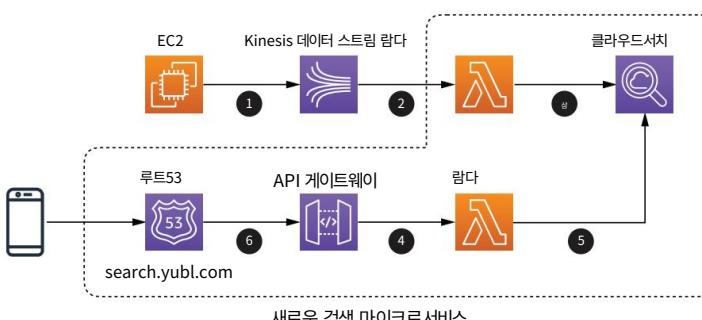


그림 4.2 서비스 구성 요소에서 실행되는 새로운 아키텍처의 개괄적인 개요

그림 4.2에서 번호가 매겨진 화살표를 따라가면 다음과 같이 모든 조각이 맞춰집니다. 1. 레거시 모노리스는 모든 사용자 관련 이벤트를 users라는 Kinesis Data Stream에 게시합니다. 여기에는 새로운 사용자가 가입하거나 사용자가 프로필을 업데이트했을 때 알려주는 사용자 생성 및 사용자 프로필 업데이트 이벤트가 포함됩니다.

2. Lambda 함수는 사용자 스트림을 구독합니다.
3. Lambda 함수는 이러한 이벤트를 사용하여 Amazon CloudSearch의 사용자 인덱스에서 사용자 문서를 삽입, 업데이트 또는 삭제합니다.
4. POST /?query={string} 앤드포인트가 있는 API Gateway의 새 API는 HTTP 요청을 처리하는 다른 Lambda 함수.
5. Lambda 함수는 사용자의 쿼리 문자열을 검색 요청으로 변환합니다.
Amazon CloudSearch의 사용자 인덱스에 대해
6. 새로운 REST API에 대한 사용자 친화적인 하위 도메인을 생성하기 위해 API Gateway의 search.yubl.com에 대한 사용자 지정 도메인 이름이 Route53에 등록됩니다.

이 마이크로 서비스의 경우 Amazon Elastic Search 대신 Amazon CloudSearch를 선택했습니다. 당시 Amazon ElasticSearch에서는 쓰기 처리량에 대한 확장성 문제인 Elasticsearch 클러스터의 쓰기 노드 수를 변경할 수 없었기 때문입니다. 그러나 Amazon CloudSearch에 문제가 없었던 것은 아닙니다.

읽기 및 쓰기 노드를 독립적으로 자동 확장할 수 있지만 CloudSearch 클러스터를 확장하는 데 30분이 걸립니다. 이는 급증하는 워크로드와 잘 맞지 않아 결과적으로 읽기 클러스터를 과도하게 프로비저닝해야 했습니다. 오늘 이 서비스를 다시 구현한다면 Amazon ElasticSearch나 Algolia (<https://algolia.com>) 와 같은 타사 서비스를 반드시 사용할 것입니다. 대신에.

새로운 서비스를 출시하기 전에 CloudSearch 인덱스에서 기존의 모든 사용자 데이터를 사용할 수 있는지도 확인해야 합니다. 이를 위해 가장 최근의 사용자 프로필 업데이트를 추적하면서 모든 기존 사용자 데이터(최대 800,000명의 사용자)를 MongoDB에서 CloudSearch로 복사하는 일회성 작업을 실행했습니다. 이것이 완료된 후에야 사용자 업데이트 처리를 시작하기 위해 2단계(그림 4.2)의 기능을 활성화했습니다.

여기서 주목해야 할 또 다른 중요한 세부 사항은 함수의 Kinesis 구독을 활성화할 때 일회성 작업이 시작된 시점부터 이벤트를 처리했다는 것입니다. Kinesis를 사용하면 구독의 StartingPosition을 지정할 수 있습니다. 이를 AT_TIMESTAMP로 구성하여 특정 타임스탬프에서 이벤트 처리를 시작할 수 있습니다. 일회성 작업이 시작된 이후의 이벤트를 처리하여 Yubl이 실행되는 동안 발생한 업데이트를 놓치지 않았습니다.

일단 라이브되면 새 검색 서비스의 성능이 이전 검색에 비해 크게 향상되었습니다. 또한 프로세스에서 모놀리식 MongoDB 데이터베이스의 많은 부하를 제거하여 앱의 일반적인 응답성에 긍정적인 영향을 미쳤습니다.

또한 API Gateway 및 Lambda와 같은 서비스 기술을 사용하여 다른 마이크로서비스를 구축하는 방법에 대한 템플릿도 제공했습니다.

4.3 새로운 마이크로서비스로 원활하게 마이그레이션

새로운 마이크로서비스를 구축하는 것은 쉬운 일이었습니다. 어려웠던 부분은 안전하게 마이그레이션하고 예상치 못한 문제가 발생한 경우 신속하게 롤백할 수 있습니다. 문제. 또 다른 관심사는 가동 중지 시간과 시스템에 영향을 미치지 않고 우아하게 수행하는 방법이었습니다. 우리 사용자. 시작 위치가 모든 기능이 있는 단일체라고 가정합니다. 공유 데이터베이스에 직접 액세스(그림 4.3). 어디서 시작할까요?

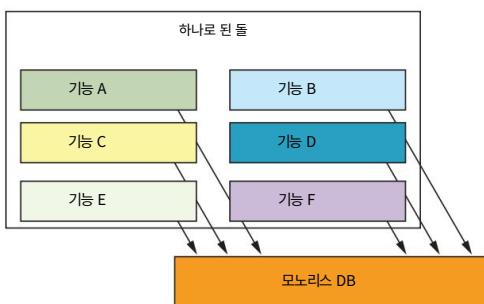


그림 4.3 모든 것이 공유 데이터베이스에 직접 액세스할 수 있는 모놀리식 시스템

우리는 이 모놀리스를 API Gateway, Lambda 및 DynamoDB와 같은 서비스 구성 요소로 구축된 마이크로서비스로 분해하기 시작했습니다. 기능을 모놀리스 자체 마이크로 서비스로 만들기 위해 마이크로 서비스가 사용자 프로필, 제품 카탈로그 또는 고객과 같은 시스템의 일부에 대한 권한이 되기를 원했습니다.

명령. 마이크로 서비스에는 자체 데이터베이스가 있으며 다른 마이크로 서비스(또는 모놀리스)는 데이터베이스에 접근하여 데이터에 액세스하거나 조작할 수 없어야 합니다.

곧장.

대신 상태 변경을 유발하려면 다른 마이크로서비스가 API를 통해 이 마이크로서비스와 통신해야 합니다. 이것은 다음과 같은 형태의 HTTP 기반일 수 있습니다.

이벤트/메시지를 게시하는 형태의 REST API 호출 또는 메시지 기반

대기줄. 중요한 것은 데이터에 대한 직접적인 접근과 조작을 차단하는 것입니다.

마이크로 서비스는 (그림 4.4)의 권한으로 가정됩니다. 이거 어떻게 해요

사용자에게 큰 지장을 주지 않으면서 우아하게?

여기서 문제는 일반적으로 대규모 마이그레이션을 수행하는 것이 위험하다는 것입니다. 다운타임이 필요합니다. 그렇다고 해서 빅뱅 마이그레이션에 대한 생각을 절대로 해서는 안 된다는 말은 아닙니다. 소규모 스타트업이고 현재 플랫폼에 사용자가 거의 없다면 그렇다면 다운타임이 있는 대규모 마이그레이션이 가장 빠르고 효율적인 접근 방식일 수 있습니다. 그러나 이러한 마이그레이션을 겪고 있는 많은 조직의 경우 새로운 마이그레이션으로 인한 위험과 중단을 최소화하는 것이 중요합니다
마이크로 서비스.

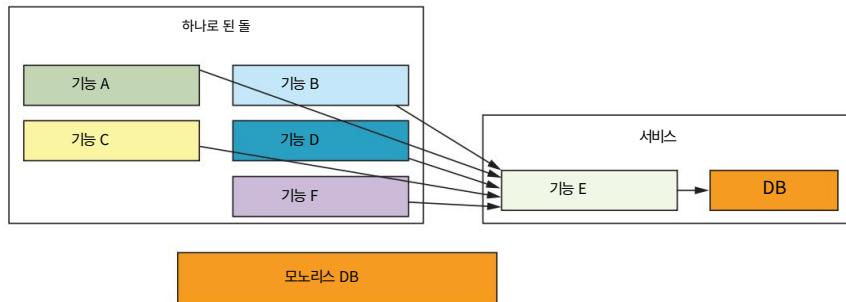


그림 4.4 모든 것이 공유 데이터베이스에 직접 액세스할 수 있는 모놀리식 시스템

일반적인 전략은 안전을 최대화하기 위해 여러 단계로 마이그레이션을 수행하는 것입니다.

예를 들어, 다음 프로세스는 그림 4.5에서 설명하는 것처럼 몇 가지 가능한 단계를 설명합니다. 1. 특정 기능에

대한 비즈니스 논리를 별도의 서비스로 이동하고 자체 API를 만듭니다. 새 서비스는 다음이 될 때까지 모놀리식 데이터베이스를 계속 사용합니다.

데이터에 대한 권한.

2. 모놀리스가 이 기능의 데이터에 직접 액세스하는 위치를 찾고 해당 액세스 포인트를 리디렉션하여 대신 새 서비스의 API를 통과하도록 합니다. 예상치 못한 문제 또는 영향의 폭발 반경을 최소화하려면 가장 덜 중요한 구성 요소부터 시작하십시오.

3. 다른 모든 직접 액세스 지점을 새 서비스의 데이터로 이동하여 API(아마도 한 번에 하나씩).

4. 이제 새 서비스가 데이터에 대한 권한이 있으므로 모놀리식 데이터베이스에서 자체 데이터베이스로 데이터를 마이그레이션하는 과정을 계획할 수 있습니다. 이 새 서비스에 대한 요구 사항에 따라 다른 데이터베이스를 사용할 수 있습니다. 액세스 패턴이 단순하고 대부분 키 조회라면 DynamoDB가 아마도 좋은 선택일 것입니다.

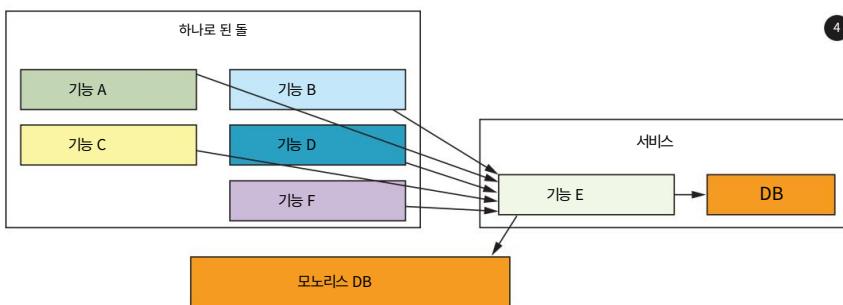
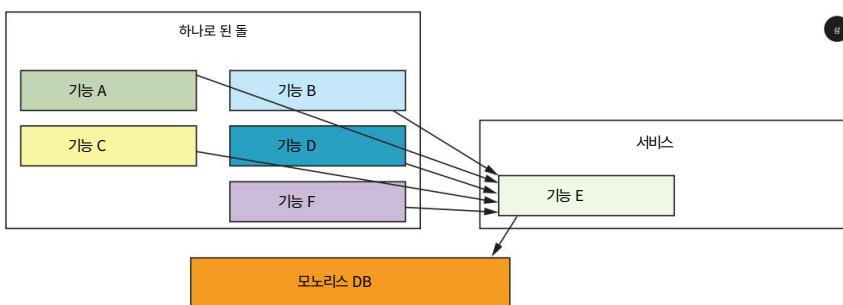
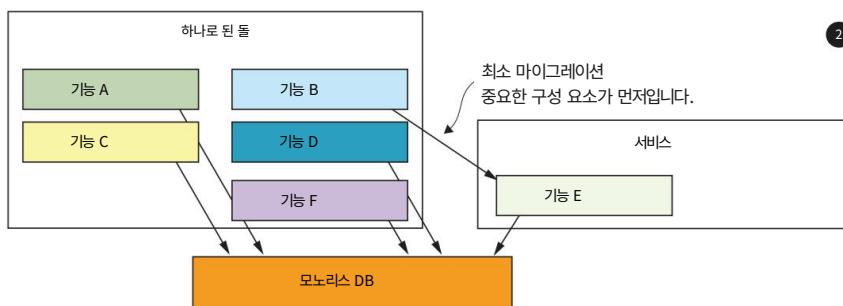
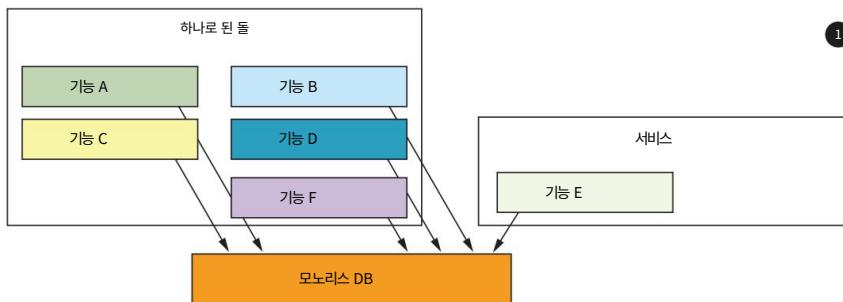


그림 4.5 대신 새 마이크로서비스의 API를 통하여도록 액세스를 이동하여 공유 모놀리식 데이터베이스에 대한 직접 액세스를 점차 차단합니다.

5. 새 데이터베이스를 만든 후에는 모놀리식 데이터베이스에서 데이터를 마이그레이션해야 합니다. 다운 타임 없이 그렇게 하려면 새 데이터베이스를 연속 읽기 및 연속 기입 캐시로 취급할 수 있습니다. 모든 업데이트 및 삽입은 모놀리식 데이터베이스에 작성된 다음 새 데이터베이스에 복사됩니다(그림 4.6).

읽기를 시도할 때 먼저 새 데이터베이스에서 읽습니다. 데이터를 찾을 수 없으면 모놀리식 데이터베이스에서 읽고 데이터를 새 데이터베이스에 저장합니다.

6. 백그라운드에서 일회성 작업을 실행하여 기존 데이터를 모두 복사합니다(그림 4.6).

최신 업데이트를 덮어쓰지 않도록 주의하십시오. (DynamoDB, <https://amzn.to/2lbE818>에서는 조건부 쓰기를 사용하여 이 작업을 수행할 수 있습니다.)

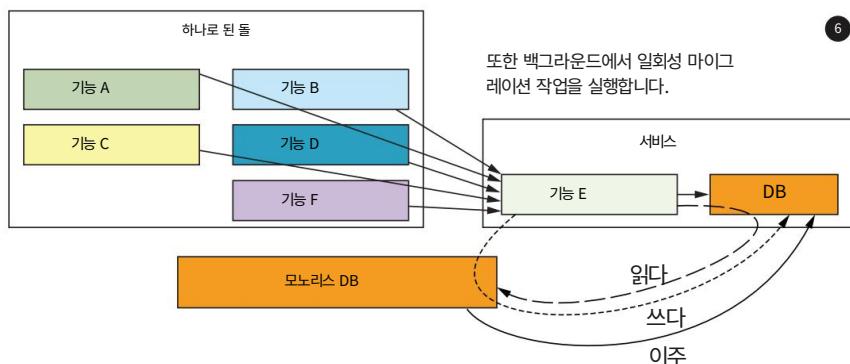
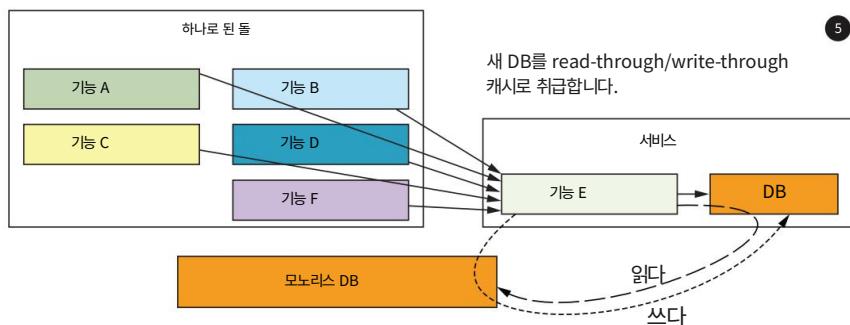


그림 4.6 다운타임 없이 점진적으로 데이터를 새 데이터베이스로 마이그레이션합니다.

이것은 모놀리스에서 기능을 추출하고 독립적으로 확장 및 실패할 수 있는 마이크로서비스로 이동하는 데 유용한 패턴입니다. 사용자에 대한 잠재적인 영향을 최소화하기 위해 안전하고 적절하게 수행할 수 있는 방법이 더 있습니다. 예를 들어, 새로운 마이크로서비스가 처음 가동될 때 소량의 트래픽만 새 마이크로서비스로 라우팅할 수 있습니다. 이것은 새로운 마이크로서비스에 대한 예상치 못한 문제의 폭발 반경을 제한합니다. 사용자를 대변하는 마이크로서비스에 특히 중요합니다.

큰 영향을 미칠 수 있으므로 모바일/웹 클라이언트의 요청을 직접 처리 사용자 경험에.

이 접근 방식은 일반적으로 카나리아 패턴으로 알려져 있으며 시스템 마이그레이션에 국한되지 않습니다. 카나리아 배포라는 용어는 배포 전략을 의미합니다.

적은 비율의 트래픽이

예상치 못한 폭발 반응을 제한하는 새로운 버전의 애플리케이션

문제. 애플리케이션 앞에서 ALB(Application Load Balancer)를 사용하는 경우 여기에서 이 라우팅 동작을 구성할 수 있습니다(그림 4.7).

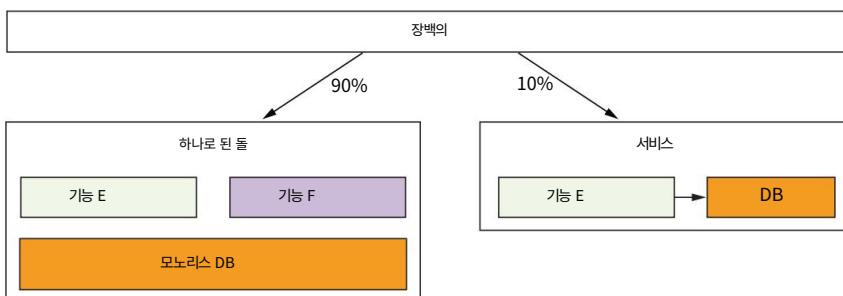


그림 4.7 ALB(Application Load Balancer)를 사용하여 모놀리스와 새 마이크로서비스 간에 트래픽을 분산할 수 있습니다.
이를 통해 예상치 못한 문제가 일부 사용자에게 미치는 영향을 최소화할 수 있습니다.

이 접근이 불가능한 경우(또는 ALB가 존재하지 않는 Yubi의 경우)

시간), 구성 가능한 요청 기간(%)에 대해 모놀리스의 요청을 프록시할 수도 있습니다. 그림 4.8은 이 접근 방식을 보여줍니다.

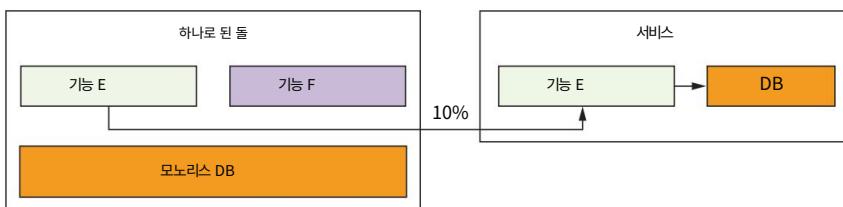


그림 4.8 ALB가 없어도 모놀리를 수정하여 요청을 프록시할 수 있습니다.

요약

서비스의 이점을 최대한 활용하려면 대부분의 애플리케이션을 다시 설계해야 합니다.

건축물. 기존 애플리케이션을 리프트 앤 시프트하는 솔루션이 있지만

서비스로 전환하면 최적의 성능과 확장성을 제공하지 않습니다.

서비스 아키텍처의 이점을 최대한 활용하려면 소규모의 자율성이 필요합니다.

자신의 아키텍처 결정을 내릴 수 있는 팀. 개발자

코드 그 이상을 책임져야 하고 코드를 소유할 권한이 있어야 합니다.

체계. 아마존의 좌우명은 "당신이 만들고, 당신이 실행합니다."

DevOps는 서비스로 더 간단합니다. 즉시 많은 자동화를 얻을 수 있으며 Serverless Framework와 같은 도구가 나머지를 처리합니다. 당신은 여전히 필요합니다.

그러나 프로덕션 시스템 실행에 대한 운영 경험은 여전히 중요하므로 주의해야 할 메트릭과 추가해야 할 경고를 알고 있습니다.

단위 테스트는 서비스 아키텍처의 경우 투자 수익이 낮습니다.

체스. 대부분의 기능은 단순하며 종종 Amazon의 DynamoDB 및 SQS(Simple Queueing Service)와 같은 다른 서비스와 통합됩니다. 이러한 통합 지점을 조롱하는 단위 테스트는 이러한 서비스 상호 작용을 테스트하지 않으며 잘못된 보안 감각을 제공합니다.

행복한 경로를 위해 실제 AWS 서비스를 실행하는 통합 테스트를 선호하고 그렇지 않으면 시뮬레이션하기 어려운 실패 사례에만 모의를 사용합니다. 예를 들어, 함수 코드를 로컬에서 실행하되 실제 DynamoDB 테이블과 통신하도록 합니다. 그런 다음 DynamoDB의 처리량 초과 오류에 대한 오류 처리를 테스트해야 할 때 모의를 사용하십시오.

서비스는 종종 마이크로서비스 아키텍처에서 서로를 호출해야 합니다. 인터페이스를 위해 (AWS 서비스에 비해) 개발 환경에서 손상되기 쉬운 API는 모의를 사용하여 실패를 격리합니다. 마지막으로 원하는 것은 한 서비스의 오류가 해당 서비스에 의존하는 다른 모든 서비스에 대한 테스트에 실패하는 것입니다.

AWS 서비스(예: DynamoDB, SNS, SQS)를 로컬에서 시뮬레이션하는 것은 노력할 가치가 없습니다. 로컬 시뮬레이션 도구를 사용하는 것보다 임시 스택을 배포하는 것이 더 쉽고 빠릅니다.

Kinesis 및 SQS와 같은 일괄 이벤트 소스를 처리할 때 부분 오류를 처리하는 방법에 대해 생각해야 합니다. 작업이 면 등성이고 문제 없이 재시도할 수 있는지 확인하거나 실패한 배치에서 성공적으로 처리된 항목이 배치를 재시도할 때 다시 처리되지 않도록 해야 합니다.

클라우드 전문가: 아키텍처 하이라이트, 배운 교훈

이 장에서는 다음을 다룹니다.

Cloud Guru의 원래 REST 아키텍처
팀이 다음에서 마이그레이션하기로 결정한 이유
마이크로서비스 및 GraphQL에 대한 REST
마이그레이션을 통해 얻은 교훈

이 책의 초판에서는 A Cloud Guru (<https://acloudguru.com>)가 구축한 서비스 LMS(Learning Management System)에 대해 설명했습니다. 당시 A Cloud Guru는 Amazon API Gateway, AWS Lambda, Google Firebase를 기본 데이터베이스로 사용하여 RESTful API 백엔드를 구축했습니다. 첫 번째 에디션을 출판한 이후로 A Cloud Guru는 큰 변화를 겪었습니다. 회사는 RESTful 모놀리식 설계에서 GraphQL 기반 마이크로서비스 아키텍처로 전환했습니다. 이 장에서는 이 여정을 설명합니다. 원래 RESTful 디자인, 마이크로서비스로의 전환, GraphQL이 어떻게 중요한 역할을 하는지, 그 과정에서 배운 교훈을 살펴보겠습니다.

하지만 한 가지 분명한 사실은 서비스 기술을 통해 A Cloud Guru가 최소한의 소란으로 신속하게 플랫폼을 재설계할 수 있다는 것입니다. 개발자는 기존의 3계층 거대 기업보다 서비스 애플리케이션을 사용하여 더 민첩하게 대처할 수 있습니다.

이는 서비스 접근 방식에서 주요 초점이 시스템 아키텍처, 데이터 및 코드에 있기 때문입니다. Cloud Guru 개발자는 서버 프로비저닝, 서버 소프트웨어 업데이트 또는 Kubernetes 클러스터 관리에 대해 걱정하는 데 시간과 에너지를 소비할 필요가 없었습니다. 그것만으로도 시간을 절약하고 비즈니스에 중요한 플랫폼 요소에 집중할 수 있는 기회를 얻었습니다.

5.1 원본 아키텍처 Cloud Guru는 Amazon

Web Services(AWS), Microsoft Azure, Google Cloud Platform 및 클라우드 관련 기술을 배우고자 하는 모든 사람을 위한 온라인 교육 플랫폼입니다. 플랫폼의 핵심 기능은 다음과 같습니다.

주문형 비디오 과정

연습 시험 및 퀴즈

실시간 토론 포럼

대시보드 및 보고 사용자 프로필 및 게

임화 학습 경로와 같은 교육 기능 자신의 기술

을 테스트하려는 학생들을 위한 대화형 샌드박스 환경

Cloud Guru는 또한 학생들이 월간 또는 연간 구독료를 지불하고 콘텐츠 및 기능에 액세스할 수 있는 전자상거래 플랫폼입니다. A Cloud Guru를 위한 과정을 생성하는 교육 설계자는 S3에 직접 비디오를 업로드할 수 있습니다. 이러한 비디오는 다양한 형식과 해상도(1080p, 720p, HLS 등)로 즉시 트랜스코딩됩니다.

2017~2018년에 A Cloud Guru 플랫폼은 Firebase를 기본 데이터베이스로 사용했습니다. 이 데이터베이스의 좋은 기능은 클라이언트 장치(컴퓨터 또는 전화의 브라우저)가 새로 고침이나 폴링 없이 거의 실시간으로 업데이트를 수신할 수 있다는 것입니다.

(Firebase는 웹 소켓을 사용하여 연결된 모든 기기에 동시에 업데이트를 푸시합니다.)

다른 주요 구성 요소는 API Gateway와 AWS Lambda였습니다. 그림 5.1은 초기 REST 아키텍처가 어떻게 생겼는지에 대한 기본적인 상위 수준 보기를 보여줍니다.

이 책의 초판에서는 RESTful 인터페이스를 사용하여 서비스 시스템을 구축하는 방법을 설명했습니다. AWS와 Google Cloud Platform에서 제공하는 기능과 서비스를 사용하여 정교하고 확장 가능하며 가용성이 높은 플랫폼을 만들 수 있다는 사실을 보여주고 싶었습니다. A Cloud Guru 팀은 그 이상을 달성할 수 있었습니다. 그들은 수만 명의 동시 사용자에게 서비스를 제공할 시스템을 구축했습니다. 그림 5.2는 이 책의 초판에서 제시된 것과 동일한 아키텍처의 약간 더 발전된 버전을 보여줍니다.

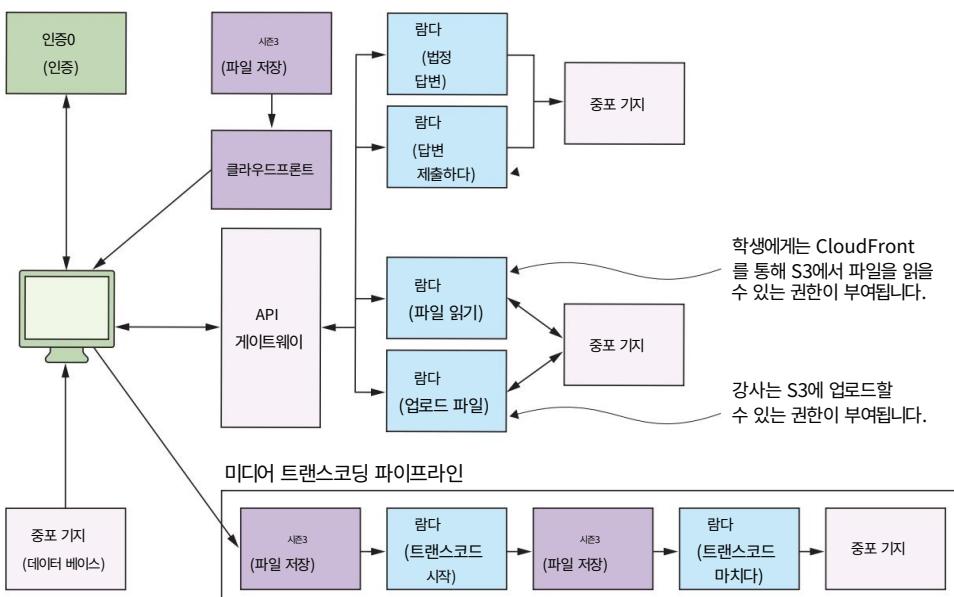
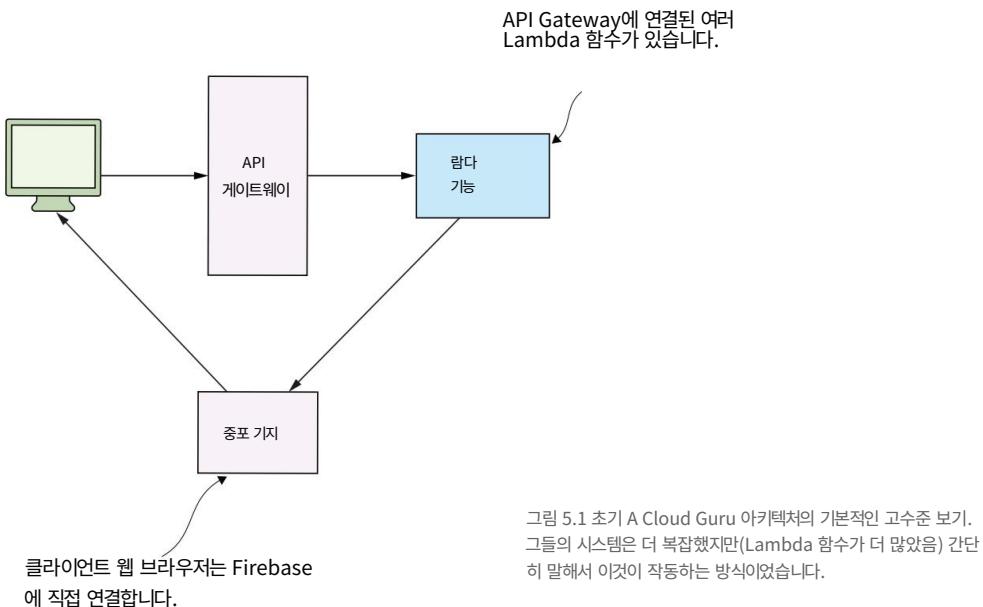


그림 5.2 이것은 A Cloud Guru 아키텍처의 약간 더 발전된 버전입니다. 실제 프로덕션 아키텍처에는 결제, 관리, 게임, 보고 및 분석을 수행하기 위한 Lambda 기능과 서비스가 있었습니다.

원래 시스템은 잘 작동했고 개발 팀이 예상한 대로 확장되었습니다. 또한 AWS 청구서가 수천 달러에 불과하여 실행하는 것도 저렴했습니다(Lambda 및 API Gateway 청구서는 \$1,000 미만이었습니다). 원래 A Cloud Guru 아키텍처에 대해 다음 사항에 유의하십시오(그림 5.2).

프런트엔드는 AngularJS를 사용하여 구축되었으며 Netlify (<https://netlify.com>)에서 호스팅되었습니다.

Auth0은 등록 및 인증을 제공하는 데 사용되었습니다. AngularJS 웹사이트가 Firebase와 같은 다른 서비스와 직접 안전하게 통신할 수 있도록 위임 토큰을 생성합니다.

모든 클라이언트는 웹 소켓을 사용하여 Firebase에 대한 연결을 생성하고 거의 실시간으로 업데이트를 받았습니다. 이는 클라이언트가 폴링 없이 업데이트를 수신했음을 의미합니다(더 나은 사용자 경험으로 이어짐).

플랫폼용 콘텐츠를 생성한 교육 설계자는 브라우저를 통해 파일(일반적으로 동영상)을 S3 버킷에 직접 업로드했습니다.

이를 위해 웹 애플리케이션은 먼저 필요한 업로드 자격 증명을 요청하는 Lambda 함수를 호출했습니다. 자격 증명이 검색되자마자 클라이언트 웹 애플리케이션은 HTTP를 통해 파일을 S3에 업로드했습니다. 이 모든 것은 무대 뒤에서 발생했으며 교육 설계자에게는 보이지 않았습니다.

파일이 S3에 업로드되면 시스템은 비디오를 트랜스코딩하고, 새 파일을 다른 버킷에 저장하고, 데이터베이스를 업데이트하고, 트랜스코딩된 비디오를 즉시 다른 사용자가 사용할 수 있도록 하는 일련의 이벤트를 자동으로 시작했습니다.

비디오를 보기 위해 학생들은 다른 Lambda 함수의 하가를 받았습니다. 권한은 24시간 동안 유효했으며 이후에는 갱신되어야 했습니다.

CloudFront를 통해 파일에 액세스했습니다. CloudFront는 사용자가 낮은 비디오가 어디에 있든 대기 시간에 액세스할 수 있습니다.

시간이 지남에 따라 A Cloud Guru 개발 팀은 서비스 REST 아키텍처의 미래를 고려하기 시작했습니다. 회사는 플랫폼 개발을 더욱 가속화하고 차단 요소를 줄이며 독립적인 팀이 다양한 고부가가치 기능에 집중할 수 있도록 하고 싶었습니다. 아키텍처 변경을 결정하게 된 몇 가지 고려 사항은 다음과 같습니다. 생성된 기존 아키텍처는 어떤 의미에서는 서비스 모놀리스였습니다.

많은 Lambda 함수가 있었지만 동일한 Firebase 데이터베이스에 연결되었습니다. 데이터베이스를 변경하면 거의 모든 Lambda 함수와 해당 함수에서 작업하는 개발자에게 영향을 미칩니다. 이를 통해 기존 시스템에서 개발자가 서로의 발을 쉽게 밟을 수 있었습니다.

이 회사는 제품의 다른 부분을 소유하는 별도의 개발 팀을 원했습니다. 예를 들어, 학생 경험 팀은 청구 및 보고를 담당하는 팀에 영향을 주지 않고 데이터베이스를 업데이트하고 Lambda 함수를 배포할 수 있어야 합니다.

진정한 마이크로서비스 접근 방식으로 전환(각 마이크로서비스가 데이터와 세계에 대한 자체 관점)을 통해 팀은 플랫폼을 병렬로 개발할 수 있습니다. 각 개발 팀은 여러 마이크로 서비스를 관리하고 필요에 따라 반복합니다. 이것은

단일 Firebase 데이터베이스를 여러 데이터베이스에 연결하면서도 여전히 읽을 수 있는 방법을 제공합니다. 필요에 따라 데이터를 수화합니다.

마이크로서비스 접근 방식으로 전환하면 팀에 더 높은 수준의 격리가 제공됩니다. 이것은 서로 다른 하위 시스템과 구성 요소를 의미합니다.

코드 기반은 소유권과 느슨한 측면에서 더 명확한 경계를 갖습니다.

커플 링.

팀은 백엔드로의 왕복을 최소화하고

필요한 데이터만 가져옵니다. Devs는 또한 모바일 및 웹과 같은 여러 클라이언트에 서비스를 제공할 수 있기를 원했습니다. 이것은 REST를 사용하여 수행할 수 있지만

팀은 GraphQL이 더 적합하다고 판단했습니다.

마침내 회사는 Firebase가 너무 비싸지고 있다고 느꼈습니다.

플랫폼의 액세스 사용 패턴을 감안할 때 Amazon의 DynamoDB는

이동할 데이터베이스입니다. DynamoDB로 마이그레이션하면 팀이 더 잘 관리할 수 있습니다.

CloudFormation을 사용하는 인프라 및 이벤트와 같은 내장 DynamoDB 기능 사용

방아쇠. 그리고 이를 통해 팀은 전적으로 AWS 환경 내에 머물 수 있습니다.

적절한 마이크로서비스 접근 방식으로 리팩토링하고 DynamoDB로 이동

기본 데이터베이스는 전체 아키텍처를 재고할 필요가 있었습니다. 주요 중 하나

고려해야 할 질문은 서로 다른 마이크로서비스에서 데이터를 가져와 다음과 같이 수행하는 방법이었습니다.

가능한 한 효율적으로(클라이언트에서 여러 번 왕복하거나 데이터 수화 없이)

사용자가 요청했을 때. 여기에서 GraphQL이 그림에 들어가게 되었습니다.

새로운 아키텍처의 초점. 그러나 GraphQL에 도달하기 전에 A가 어떻게

Cloud Guru 팀은 모놀리스를 분할하고 먼저 마이크로서비스를 만들었습니다.

서비스 모놀리스

A Cloud Guru가 원래 만든 RESTful API 디자인은 서비스 모노리스였습니다. 연결된 데이터를 저장하거나 로드하는 데 필요한 단일 데이터베이스와 기능이 있었습니다. 서비스 모놀리스를 구축하는 데는 아무런 문제가 없습니다. 클라우드를 위해 전문가, 오랫동안 잘 확장되어 회사를 구축하는 데 도움이 되었습니다. 핵심 이유 마이크로서비스 설계로 전환하기 위해서는 여러 팀이 함께 작업해야 했습니다. 평행한.

오늘 시작하는 경우 모놀리식 접근 방식을 사용하는 것이 좋습니다.

필요한 경우 마이크로서비스로 마이그레이션할 수 있습니다. 기억하십시오, 당신은 할 필요가 없습니다 추세를 따르고 자신에게 적합하지 않은 경우 마이크로서비스 방식을 따르십시오.

5.1.1 43개 마이크로서비스로의 여정

회사에서 제안한 서비스 마이크로서비스 접근 방식을 살펴보겠습니다.

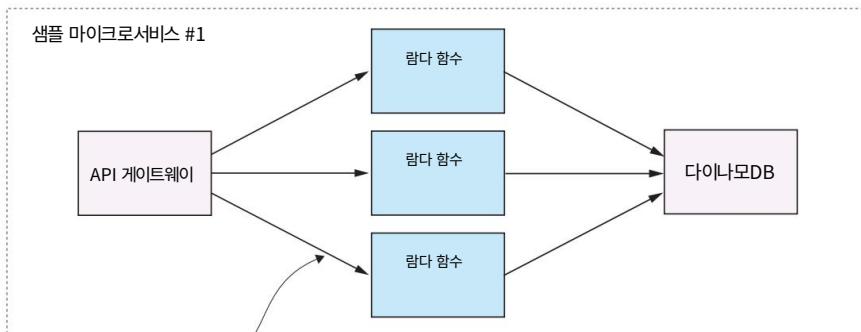
먼저, 2020년 초에 새로운 GraphQL이 재설계된 A Cloud Guru 플랫폼에 대한 몇 가지 통계가 있습니다.

매월 2억 4천만 개의 Lambda 호출(초당 100개) 매월 1억 8천만 개의 API 게

이트웨이 호출(초당 70개)

매월 CloudFront에서 90TB의 데이터 전송(초당 274MB)

팀은 2018년에 모놀리스를 분해하고 마이크로서비스 아키텍처로 이전하기 시작했습니다. API 게이트웨이와 램다는 각자의 책임과 세계관을 가진 별개의 마이크로 서비스로 분리되었습니다. 마이크로서비스의 새로운 세계에서 각 서비스는 단일 DynamoDB 테이블, 몇 가지 Lambda 함수 및 API 게이트웨이 만큼 간단할 수 있습니다. 그림 5.3은 몇 가지 기본 마이크로서비스가 어떻게 보이는지 보여주는 예를 보여줍니다.



Lambda, API Gateway, DynamoDB 및 S3를 사용하여 마이크로 서비스를 어떻게 구성할 수 있는지에 대한 두 가지 기본 예.

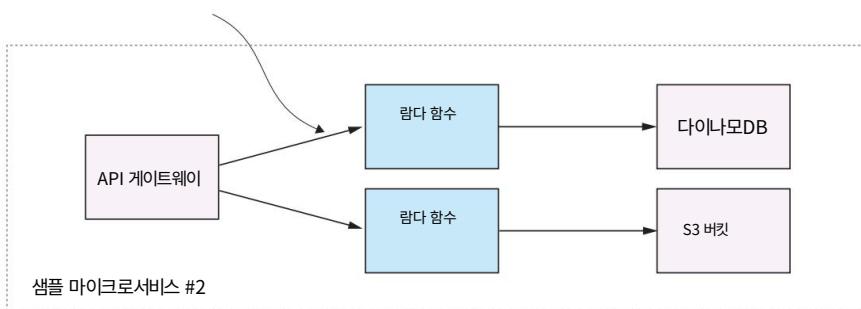


그림 5.3 여기에 있는 두 개의 마이크로서비스는 이전에 논의한 단순화된 RESTful 아키텍처와 유사합니다.

마이크로서비스의 패키징도 흥미롭습니다. Cloud Guru는 Serverless Framework 및 CloudFormation을 사용하여 마이크로서비스를 구성하고 배포합니다.

마이크로 서비스의 일부 서비스는 상태를 저장 하지만 다른 서비스는 상태를 저장하지 않습니다. 램다

기능은 상태 비저장이므로 각 배포에서 덮어쓸 수 있습니다. 그것은 항상 일시적입니다. DynamoDB 테이블 또는 S3 버킷은 스테이트풀(Stateful)입니다. 당신은 조심해야합니다 이미 있는 데이터를 보존합니다. 배포 프로세스를 덮어쓸 수 없습니다. 그것. 또한 특정 서비스에 속하지 않는 글로벌 서비스 및 리소스가 있을 수 있습니다. 마이크로 서비스. 어떻게 배포하고 관리해야 한다고 생각하십니까?

A Cloud Guru 팀은 마이크로서비스를 설계하여 서로 다른 상태 비저장 및 상태 저장 리소스를 위한 CloudFormation 스택은 물론 구성 및 핵심 종속성을 위한 스택입니다. 그림 5.4는 그 모습을 보여줍니다.



그림 5.4 각 마이크로서비스는 자체 CloudFormation 스택에 있으므로 쉽게 배포할 수 있습니다.

다양한 종류의 리소스에 대한 다양한 CloudFormation 스택에 대한 이 접근 방식 개발 팀이 상태 비저장 리소스와 함께 스택을 배포할 수 있도록 허용합니다. 상태 저장 리소스를 건드릴 필요 없이 업데이트해야 합니다. 도 마찬가지 구성 및 핵심 종속성 스택. 마이크로 서비스 내에서 다른 것을 수정하지 않고 업데이트할 수 있습니다. 이러한 종류의 관심사 분리는

상태 저장 리소스의 우발적인 수정을 방지할 수 있기 때문에 유리합니다.

또한 존재하는 몇 가지 다른 전역 종속성이 있지만 이들은 그렇지 않습니다. 모든 마이크로 서비스 내에서. 여기에는 Amazon과 같은 인프라 구성 요소가 포함됩니다. RedShift(데이터 워어하우스), AWS WAF(방화벽) 및 VPC(가상 사설 클라우드). 마이크로서비스는 이러한 글로벌 자원. 사실, 그들 사이에는 상당히 느슨한 결합이 있습니다.

예를 들어 마이크로서비스가 RedShift에 데이터를 푸시해야 하는 경우에는 수행하지 않습니다. 곧장. 대신 일반 ETL 작업이 마이크로서비스에서 데이터를 가져와 Red Shift에 씁니다. 즉, 마이크로서비스는 RedShift에 대해 알 필요가 없습니다. 마이크로서비스

별도의 ETL 작업이 자체 작업을 수행하는 동안 스스로 살고 숨을 쉴 수 있습니다. 그림 5.5 일부 리소스가 특정 마이크로 서비스 외부에 있어야 함을 보여줍니다.



그림 5.5 전역 종속성은 각 개별 마이크로서비스 외부에 있습니다. 모든 것이 마이크로서비스 내에 존재할 필요는 없습니다.

A Cloud Guru 팀은 처음에 생성된 서비스 모놀리스를 점진적으로 분리하고 마이크로서비스로 다시 구현했습니다. 하나에서 이동

다른 아키텍처에 대한 아키텍처는 항상 시간이 걸리지만 여기서 좋은 이점은

팀은 주로 코드에 집중할 수 있습니다. 하드웨어, 서버 또는 컨테이너가 없었습니다.

걱정할 오케스트레이션 엔진(예: Kubernetes). 팀은 사용자가 영향을 받지 않도록 신중하고 점진적으로 다양한 구성 요소를 다시 구현했습니다.

변경 중.

マイクロサービスへの 전환의 일환으로, GraphQL은 클라이언트가

요구. 결국 각 마이크로 서비스에는 자체 데이터베이스와 자체 보기이 있을 수 있습니다.

세계. 사용자가 데이터를 가져와야 할 때 모든 일이 어떻게 이루어집니까? 어떤 마이크로 서비스가 그것을 요구? 그리고 여려 마이크로 서비스에 필요한 정보가 있으면 어떻게 될까요?

클라이언트에 집계 응답이 필요합니까? GraphQL은 마이크로서비스를 쿼리하고 스키마 스티칭을 통해 클라이언트에 필요한 응답을 생성하는 도구가 되었습니다.

5.1.2 GraphQL이란?

우리는 이미 3장에서 GraphQL에 대해 언급했지만, 그것이 무엇인지에 대해 간단히 요약해 보겠습니다. 이다. GraphQL은 2012년 Facebook에서 개발한 인기 있는 데이터 쿼리 언어입니다.

2015년에 공개적으로 출시되었습니다. REST의 약점(다중 왕복, 오버페칭, 버전 관리 문제)으로 인해 REST의 대안으로 설계되었습니다. GraphQL은 단일 엔드포인트(예: api/graphqql)에서 쿼리를 수행하는 계층적이고 선언적인 방법을 제공하여 이러한 문제를 해결하려고 했습니다.

그림 5.6은 이것이 어떻게 보이는지 보여줍니다.

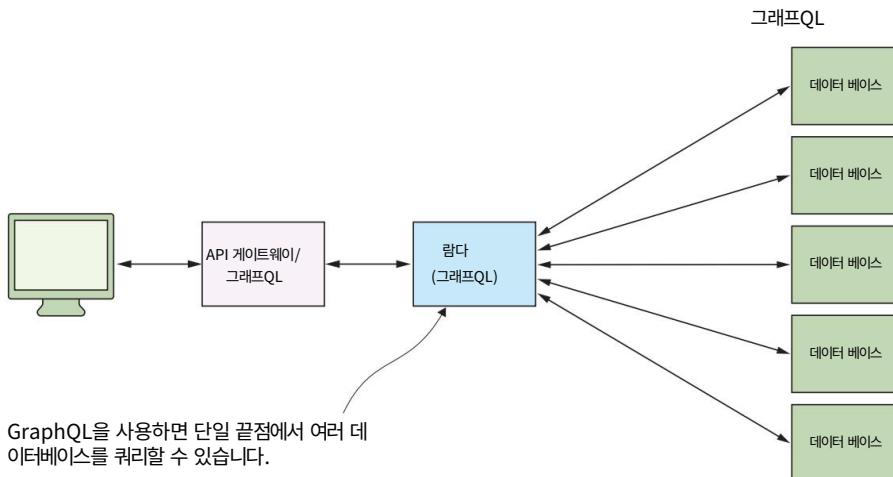


그림 5.6 Lambda 함수에서 실행되는 GraphQL 라이브러리는 여러 데이터베이스를 쿼리하고 스마트 스키마를 사용하여 각 개별 클라이언트와 관련된 결과를 생성할 수 있습니다.

GraphQL은 클라이언트에 전원을 제공합니다. 서버에서 응답의 구조를 지정하는 대신 클라이언트에서 정의됩니다. 클라이언트는 반환할 속성과 관계를 지정할 수 있습니다. GraphQL은 여러 소스의 데이터를 집계하여 단일 왕복으로 클라이언트에 반환하므로 데이터 검색을 위한 효율적인 시스템이 됩니다.

Facebook에 따르면 GraphQL은 거의 1,000개에 가까운 다양한 버전의 애플리케이션에서 초당 수백만 건의 요청을 처리합니다. GraphQL이 어떻게 생겼는지 자세히 설명하기 위해 다음은 <https://graphql.org/learn/queries/>에서 가져온 간단한 쿼리입니다.

```
{
  영웅 {
    이름
  }
}
```

그리고 해당 쿼리에 대한 한 가지 가능한 응답은 다음과 같습니다.

```
{
  "데이터": {
    "영웅": {
      "이름": "R2-D2"
    }
  }
}
```

서버리스 아키텍처에서 GraphQL은 API 게이트웨이에 연결된 단일 Lambda 함수에서 실행되거나(A Cloud Guru가 수행한 작업) AWS AppSync와 같은 서비스를 통해 사용할 수 있습니다. GraphQL은 DynamoDB 테이블과 같은 여러 데이터 소스를 쿼리하고 쓸 수 있으며 스마트 스키마를 사용하여 요청과 일치하는 응답을 어셈블할 수 있습니다.

5.1.3 GraphQL로 이동

A Cloud Guru가 GraphQL로 이전하기 시작했을 때 AWS AppSync는 아직 릴리스되지 않았거나 해당 문제에 대해 발표되지도 않았습니다. 팀에는 Apollo GraphQL 라이브러리 (<https://www.apollographql.com>)를 사용하여 Lambda 함수에서 GraphQL을 실행하는 옵션이 하나 있었습니다.

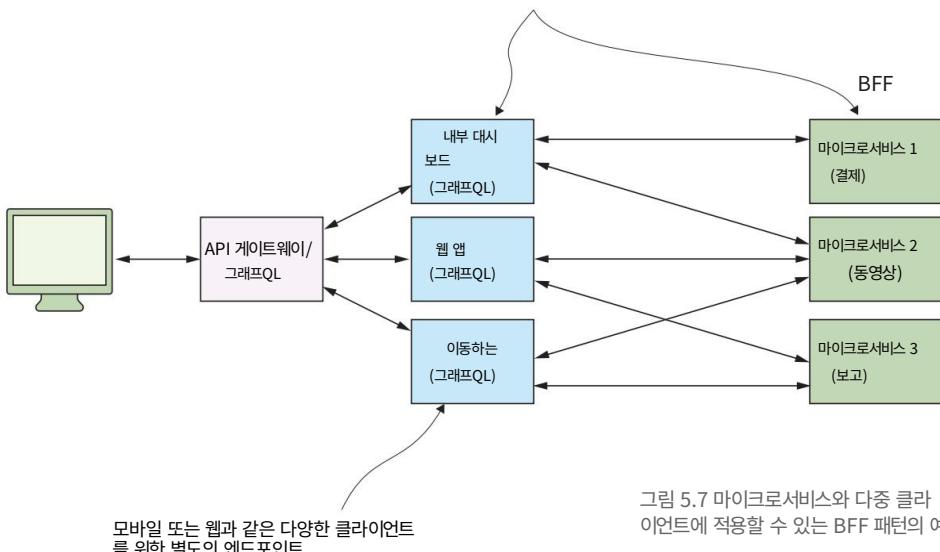
처음에 Apollo GraphQL 구현이 Lambda 함수에서 작동하도록 하는 것은 몇 가지 흥미로운 과제를 제시했습니다. Apollo는 원래 서버와 컨테이너에서 장기간 실행되는 프로세스를 위해 설계되었습니다. 팀은 Lambda에 최적화하기 위해 특정 수의 조정 작업을 수행해야 했습니다.

A Cloud Guru 팀은 GraphQL과 BFF(Backends for Frontends)라는 디자인 패턴을 사용하기 시작했습니다. BFF의 이면에 있는 아이디어는 각 클라이언트에 고유한 API 또는 특정 요구 사항을 처리하는 엔드포인트가 있다는 것입니다(예: 모바일 전용 엔드포인트와 웹용 엔드포인트가 있음). 각 엔드포인트는 적절한 마이크로서비스를 쿼리하여 필요에 따라 데이터를 저장하거나 로드할 수 있습니다. 클라이언트는 알 필요가 없다

다양한 마이크로서비스에 대해 쿼리할 끝점만 알면 됩니다.

이 패턴은 많은 시스템에 존재하는 분리 문제를 해결합니다. 그림 5.7은 BFF 아키텍처의 예와 A Cloud Guru 팀이 추구하는 바를 보여줍니다.

서로 다른 엔드포인트는 서로 다른 마이크로서비스와 데이터베이스를 쿼리하여 주어진 클라이언트에 필요한 데이터를 얻을 수 있습니다.



이 장 전체에서 우리는 다음을 포함하는 Lambda 함수를 호출했습니다.

GraphQL JavaScript 라이브러리는 GraphQL 끝점입니다. 하지만 이라고 하는 것이 더 나을 것 같다.

이제 이 패턴을 이해하는 BFF 끝점입니다. A Cloud Guru의 구현이 높은 수준에서 작동하는 방식은 다음과 같습니다(서비스 검색과 같은 몇 가지 세부 정보 제외).

사용자의 요청이 API 게이트웨이에 도달합니다.

API 게이트웨이는 Apollo GraphQL 라이브러리를 사용하여 Lambda 함수를 호출합니다.

이것은 우리가 논의한 BFF 끝점입니다.

Apollo GraphQL 라이브러리는 알고 있는 마이크로서비스를 쿼리합니다(자세한 내용은 서비스 검색 섹션에서 대상으로 삼을 마이크로서비스에 대해 아는 방법).

각 마이크로 서비스에는 Lambda 기능이 있는 API 게이트웨이인 엔드포인트가 있습니다(각 마이크로 서비스에 /graphql 엔드포인트가 있음).

Lambda 함수는 여러 쪐 스키마로 GraphQL 라이브러리를 실행합니다.

해결사. 마이크로 서비스에 포함된 데이터베이스를 쿼리하고 결과를 생성하여 BFF 끝점으로 다시 보냅니다.

BFF 엔드포인트는 쿼리한 다양한 마이크로서비스로부터 응답을 받습니다. 스키마 스티칭을 사용하여 최종 응답을 조합합니다.

이 최종 응답은 API Gateway를 통해 클라이언트로 다시 전송됩니다.

GraphQL Lambda 함수는 수많은 마이크로서비스를 인식합니다(자세한 내용은

잠시 후) 클라이언트 요청을 받으면 쿼리할 수 있습니다. 수치

5.8은 이 아키텍처의 높은 수준의 개요를 보여줍니다.

그레프QL

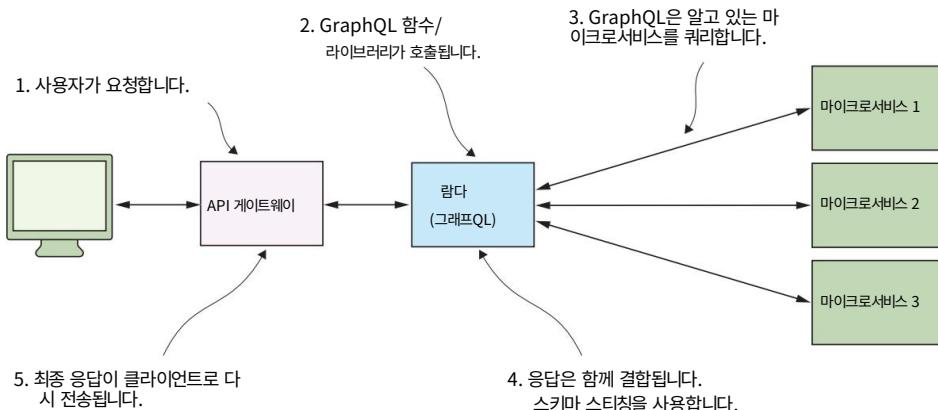


그림 5.8 GraphQL 끝점은 데이터가 필요한 클라이언트의 중심점 역할을 합니다.

5.1.4 서비스 검색

시스템에 43개의 마이크로서비스가 있는 경우 GraphQL은 쿼리할 서비스를 어떻게 알 수 있습니까?

클라이언트 요청이 들어올 때? A Cloud Guru 팀은 Sputnik이라는 내부 서비스 검색 서비스를 구축했습니다.

공개적으로 사용할 수 없음). Sputnik은 API/URI 정의가 있는 데이터베이스로 구성되며 데이터베이스 스키마. 마이크로서비스는 Sputnik을 업데이트할 시기를 알고 있습니다. 또한, GraphQL Lambda 함수는 각각에 대한 스키마를 얻기 위해 Sputnik을 쿼리할 때를 알고 있습니다. 마이크로 서비스를 사용하고 요청을 라우팅할 위치를 파악합니다.

정의 서비스 검색은 마이크로서비스 아키텍처의 표준 기술로, 어떤 서비스를 사용할 수 있는지, 어떻게 액세스하고 인터페이스가 어떻게 생겼는지 확인합니다.

Sputnik은 Lambda 함수와 스키마가 포함된 DynamoDB 테이블로 구성됩니다. 및 다른 마이크로서비스의 URI. 그것은 실제로 시스템의 다른 마이크로 서비스와 BFF의 통신을 용이하게 하는 마이크로 서비스입니다. 그림 5.9는 Sputnik이 BFF 엔드포인트가 쿼리를 생성할 위치를 알도록 돕는 방법을 보여줍니다.

TIP AWS에는 AWS 자체 서비스 검색인 Cloud Map이라는 서비스가 있습니다. 제품. 심지어 "클라우드를 위한 서비스 검색" 자원." Sputnik과 같은 것을 찾고 있다면 Cloud를 확인하십시오. 지도. 그것은 당신을 위해 일할 수 있습니다. <https://aws.amazon.com/cloud-map/>.

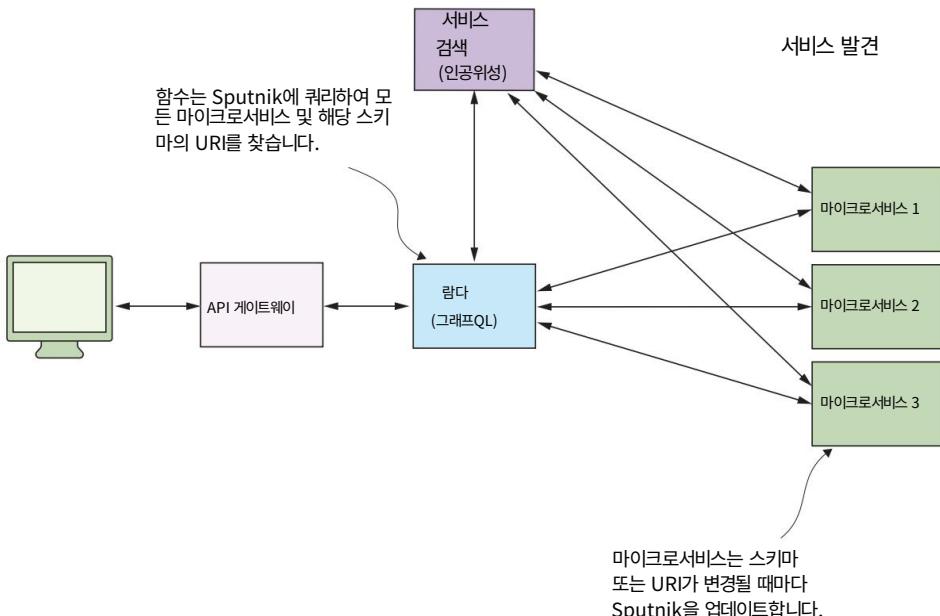


그림 5.9 A Cloud Guru를 위한 서비스 검색(Sputnik) 메커니즘. Cloud Map이라는 AWS 서비스가 있습니다. 비슷한 것을 찾고 있다면 확인하고 싶을 것입니다.

각 마이크로 서비스(URI 및 스키마)에 대한 메타데이터는 BFF 엔드포인트에 캐시됩니다. 또한 모든 요청에 대해 Sputnik을 쿼리하는 함수의 필요성을 부정합니다. 하지만, Sputnik은 캐시를 무효화하고 Lambda 함수가 캐시를 다시 쿼리하도록 할 수 있습니다.

5.1.5 BFF 세계의 보안

Cloud Guru는 여러 계층의 보안을 내장하여 심층적인 보안을 실천합니다.

체계. 사용자 인증/권한 부여의 두 가지 주요 구성 요소에 대해 이야기해 보겠습니다.

マイクロサービス 보안에 대한 BFF.

A Cloud Guru 플랫폼에서 학생은 Auth0 서비스를 사용하여 인증됩니다.

각 사용자에 대해 고유한 JWT 토큰을 생성합니다. AWS에 대한 모든 요청은

API Gateway에서 사용자 지정 권한 부여자를 사용하여 검증된 해당 JWT 토큰. 만약에
JWT 토큰이 유효하면 BFF 끝점으로 요청을 계속할 수 있습니다. 만약

그렇지 않은 경우 응답이 생성되어 인증되지 않았음을 알리는 클라이언트로 다시 전송됩니다. 이것은 원래
REST 디자인에서도 사용된 간단한 메커니즘입니다.

플랫폼의.

두 번째 흥미로운 요소는 BFF의 요청을 인증하는 방법입니다.

マイクロ서비스에. 이 시나리오에서 A Cloud Guru는 API 키를 사용하여 인증합니다.

요청. 각 마이크로 서비스에는 BFF가 요청에 포함하는 고유한 API 키가 있습니다.

헤더에서(X-API-Key 매개변수 사용). 마이크로서비스는 포함된 키를 확인합니다.

모든 것이 정상이면 요청을 승인합니다.

5.2 유산의 잔재

REST에서 GraphQL로 마이그레이션하는 데 시간이 좀 걸렸습니다. 팀이 사용자에게 문제를 일으키지 않도록 주의했기 때문입니다. 이것이 흥미로운 부작용은 재설계 과정에서 시스템이 보이는 방식이었습니다. 팀은 새로운

マイクロ서비스와 BFF가 있었지만 이전 Firebase 데이터베이스가 여전히 사용 중이었습니다.

A Cloud Guru 웹사이트에서 사용자 인터페이스의 일부 요소에 전원을 공급합니다.

그림 5.10은 그 중간 아키텍처가 어떻게 생겼는지 보여줍니다.

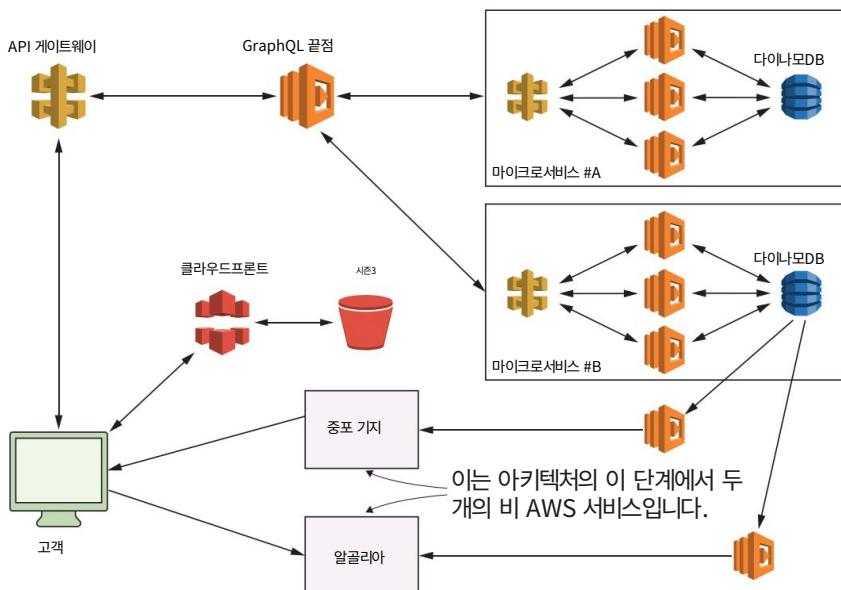


그림 5.10 전환 과정에서 A Cloud Guru 아키텍처의 상위 수준 개요

그림 5.10에 대한 흥미로운 참고 사항: Firebase가 일부 클라이언트 대면 사용자 인터페이스 요소를 구동하는데 여전히 사용된다는 것을 알 수 있습니다. Firebase의 데이터를 최신 상태로 유지하기 위해 팀은 DynamoDB 이벤트 스트림과 Lambda를 사용하여 이를 확인했습니다.

마이크로 서비스에서 테이블이 업데이트되면 DynamoDB에서 해당 변경 사항을 DynamoDB 이벤트 스트림으로 푸시하고, DynamoDB 이벤트 스트림에서 Lambda 함수를 호출합니다. 해당 Lambda 함수는 변경 사항을 분석한 다음 Firebase(및 필요에 따라 Algolia와 같은 기타 서비스)를 업데이트합니다.

이제 Firebase는 기본적으로 인터페이스의 일부를 구동하는 데 사용되는 구체화된 뷰가 됩니다. 직접 쿼리 되지는 않지만 이에 의존하는 이전 구성 요소에 대해 존재합니다. 이것은 팀이 이전 서비스 아키텍처에서 새 아키텍처로 전환하면서 내린 창의적인 결정 중 하나입니다. DynamoDB와 마이크로서비스를 도입하고 모든 것을 옮기는 동안 Fire base를 사용할 수 있었습니다.

결론서.

여기에 마이그레이션에 대한 중요한 교훈도 있습니다. 사물을 더 작은 조각으로 분할하고 하나씩 이동하여 이전 아키텍처를 유지하면서 새로운 아키텍처를 점진적으로 구현할 수 있습니다.

요약 팀은

서로 영향을 주지 않고 플랫폼에서 작업할 수 있습니다. 서로 다른 팀은 서로 다른 마이크로서비스를 담당하며 다른 사람에게 영향을 주지 않고 작업할 수 있습니다.

A Cloud Guru의 성능이 크게 향상되었습니다.

예를 들어, BFF 패턴은 필요한 데이터만 가져오고(모바일에 적합) 이를 수행하기 위해 단 한 번의 왕복만 필요합니다. 이것은 이전에 있었던 것에 대한 최적화입니다.

BFF 패턴은 여러 클라이언트 유형 및 장치를 지원하는 데 도움이 됩니다. 이는 클라이언트의 요구 사항에 따라 다를 수 있습니다.

팀은 또한 Apollo가 Lambda와 잘 작동하도록 추가 리엔지니어링을 수행해야 했습니다. 요즘에는 별 문제가 되지 않지만 얼리 어답터라면 문제가 됩니다.

항상 그렇듯이 보안은 최고의 관심사입니다. 더 많은 마이크로서비스는 공격에 대한 더 넓은 표면적을 생성합니다. 따라서 마이크로서비스와 엔드포인트를 보호하는 것이 중요합니다. 백엔드 구성 요소 간의 통신을 보호하기 위해 기계 키를 사용하는 것은 사용자가 알아야 할 모범 사례 중 하나입니다.



Yle: 건축

하이라이트, 교훈

이 장에서는 다음을 다룹니다.

Yle의 빅 데이터 아키텍처
확장성 및 탄력성, 배운 교훈

Yle는 핀란드의 국영 방송사이며 수백만 가구에서 사용하는 Yle Areena라는 자체 인기 스트리밍 서비스를 운영하고 있습니다. 수년 동안 Yle는 아키텍처에서 서비스 기술을 대규모로 사용했습니다.

AWS Fargate (<https://aws.amazon.com/fargate>) 의 조합을 사용합니다.
Lambda 및 Kinesis를 통해 당 5억 개 이상의 사용자 상호 작용 이벤트 처리
낮. 이러한 이벤트는 Yle의 머신 러닝(ML) 알고리즘을 제공하고 더 나은 콘텐츠 추천, 이미지 개인화, 스마트 알림,
그리고 더.1

¹ 이 기회를 빌어 이 아키텍처에 대한 세부 정보와 그녀와 그녀의 팀이 그 과정에서 배운 교훈을 공유한 Anahit Pogosova에게 감사드립니다.

6.1 Fargate를 사용하여 대규모 이벤트 수집

더 나은 콘텐츠 추천을 제공하기 위해 Yle는 방문자가 가장 많이 상호 작용하는 콘텐츠를 알아야 합니다. Yle는 스트리밍 서비스에서 사용자 상호 작용 데이터를 수집합니다.

HTTP API를 통한 모바일 및 TV 앱뿐만 아니라 이 API의 문제점은 라이브 스포츠 경기와 같이 트래픽이 급증할 수 있습니다. 그리고 가끔 이벤트가 겹친다 (예를 들어, 선거 결과 보도가 하키와 동시에 켜진 경우 핀란드에서 가장 인기 있는 스포츠인 게임)!

언급한 바와 같이 Yle의 API는 1회당 5억 개 이상의 사용자 상호작용 이벤트를 수집합니다. 피크 시간 동안 분당 600,000개 이상의 요청이 있는 날. 라이브 스포츠 이벤트 또는 특별 이벤트(예: 선거 결과)로 인해 최대 트래픽이 훨씬 더 높아질 수 있습니다. 그들이 관찰한 최대 트래픽 처리량은 분당 2,500,000개 요청입니다.

트래픽이 너무 급증하기 때문에 Yle 팀은 AWS의 대신 Fargate를 사용하기로 결정했습니다. API 게이트웨이 및 람다. AWS 서비스이기도 한 Fargate를 사용하면 기본 가상 머신에 대해 걱정할 필요 없이 컨테이너를 실행할 수 있습니다. 떠오르는 트렌드의 일부입니다. 컨테이너를 유틸리티 서비스로 사용하는 서비스 컨테이너의 경우.

6.1.1 비용 고려 사항

일반적으로 가동 시간을 기준으로 요금을 부과하는 AWS 서비스는 규모에 따라 요금을 부과하는 서비스에 비해 대규모로 실행할 때 훨씬 더 저렴한 경향이 있습니다.

요청 횟수. API Gateway 및 Lambda를 사용하면 개별 API 요청에 대해 비용을 지불합니다.

반면 Fargate는 vCPU, 메모리, 컨테이너가 사용하는 스토리지 리소스. 컨테이너가 사용자 트래픽을 제공하지 않더라도 컨테이너가 실행되는 동안 비용이 발생합니다.

가동 시간에 대해 비용을 지불하는 것은 요청을 많이 받지 않는 API의 경우 비효율적일 수 있습니다. 예를 들어, 하루에 수천 개의 요청을 수신하는 API는 API Gateway 및 Lambda를 사용하여 비용을 상당히 절감할 수 있습니다. 고려할 때 특히 그렇습니다.

API가

컨테이너가 실패하거나 컨테이너를 호스팅하는 AWS 가용 영역(AZ) 중 하나가 정전을 경험합니다. 그러나 Yle API와 같은 처리량이 높은 API의 경우 하루에 수억 건의 요청을 처리하므로 Fargate에서 API를 실행하면 API Gateway 및 Lambda를 사용하는 것보다 경제적입니다.

6.1.2 성능 고려 사항

Yle 팀에 대한 더 중요한 고려 사항은 트래픽이 얼마나 급증할 수 있는지를 감안할 때 API Gateway 및 Lambda를 사용하여 제한 제한에 부딪힐 가능성이 있다는 것입니다.

Lambda 함수의 동시성은 다음을 수행하는 해당 함수의 인스턴스 수입니다.

주어진 시간에 요청을 처리합니다. 이를 동시 실행 수라고 합니다.

대부분의 AWS 리전에는 기본적으로 모든 리전에서 동시 실행이 1,000개로 제한되어 있습니다. 해당 지역의 Lambda 함수. 그러나 이것은 소프트 한도이며 다음으로 높일 수 있습니다. 지원 요청. Lambda는 최대에 대한 엄격한 제한을 부과하지 않지만 동시 실행 수, 필요한 동시 실행 수에 도달하는 속도는 두 가지 요소에 의해 제한됩니다.

초기 버스트 제한은 500에서 3,000에 따라 다릅니다.
지역.

초기 버스트 제한 후에 기능의 동시성이 500까지 증가할 수 있습니다.
분당 인스턴스. 이는 제공할 인스턴스가 충분할 때까지 계속됩니다.
모든 요청 또는 동시성 제한에 도달할 때까지

API 트래픽은 종종 초당 요청 수(또는 RPS)로 측정됩니다. 주목할 가치가 있습니다.
RPS는 Lambda의 동시 실행과 동일하지 않습니다. 예를 들어 API가
요청을 처리하는 데 평균 100ms가 걸리고 Lambda의 단일 인스턴스
함수는 초당 최대 10개의 요청을 처리할 수 있습니다. 이 API가 100을 처리해야 하는 경우
RPS가 최고조에 달하면 약 10개의 Lambda 동시 실행이 필요할 것입니다.
이 처리량을 처리하기 위한 피크입니다.
그러나 API의 처리량이 100RPS에서 20,000RPS로 점프하면
30초가 지나면 초기 버스트 제한과 분당 500개 인스턴스의 후속 확장 제한이 소진될 수 있습니다. 결국
Lambda는 확장할 수 있습니다.
이 피크 로드를 처리하기에 충분한 API 함수 인스턴스가 있지만 그 동안 많은 API 요청이 제한되었을 것입니다.

고려해야 할 또 다른 주의 사항은 라이브 이벤트가 미리 예정되어 있기 때문에
Yle 팀은 방송 일정을 사용하여 인프라를 미리 확장할 수 있습니다.
프로비저닝된 동시성을 사용하는 것 외에는 Lambda로 이를 수행하는 쉬운 방법이 없습니다.
(<https://amzn.to/3faBkCU>). 그러나 프로비저닝된 동시성을

사전 크기 조정이 필요한 모든 Lambda 함수 사용 가능한
지역의 동시성.

이와 같이 광범위하게 사용하면 동시성이 충분하지 않을 수 있으므로 트래픽 급증을 흡수하는 능력에 상당한 영향을 미칠 수 있습니다.
대부분이 프로비저닝 동시성에 의해 사용되는 경우 리전입니다. 이러한 확장 제한 때문에 AWS API Gateway
및 Lambda는
트래픽이 극도로 급증하는 API에 적합합니다.
Yle 팀이 선택한 주요 이유입니다.
Fargate를 사용하여 API를 구축한 것은 현명한 결정이었습니다.

6.2 실시간 이벤트 처리

Yle의 API가 사용자 상호작용을 수집하면
이벤트를 Amazon Kinesis에 게시했습니다.
한 번에 500개의 레코드를 일괄 처리하는 데이터 스트림
Amazon Simple Queue Service(SQS) 사용
대기열을 배달 못한 편지 대기열(DLQ)로 사용합니다. 수치
6.1은 이 과정을 보여줍니다.

6.2.1 Kinesis 데이터 스트림

Amazon의 Kinesis Data Streams는 완전히 관리되고 대
규모로 확장 가능한 서비스로 다음을 수행할 수 있습니다.

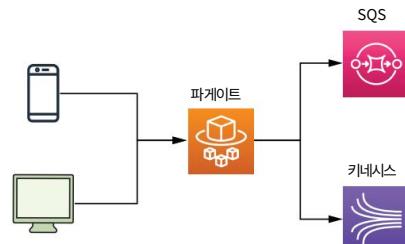


그림 6.1 분당 600,000개 이상의 이벤트의 최대 처리량
으로 하루 500,000,000개 이상의 이벤트를 동화하는 Yle
의 수집 API의 상위 수준 아키텍처. 이벤트는 Kinesis Data
Stream에 전달됩니다.

500개의 레코드 배치. Kinesis라면
데이터 스트림을 사용할 수 없는 경우 이벤트는 나중
에 재처리할 SQS 배달 못한 편지 대기열(DLQ)로 전송
됩니다.

데이터를 스트리밍하고 실시간으로 처리합니다. 데이터는 밀리초 단위로 스트림 소비자가 사용할 수 있으며 기본적으로 24시간 동안 스트림에 저장되지만 구성에 따라 1년으로 확장할 수 있습니다. (스트림 보관 기간을 연장하면 추가 요금이 부과된다는 점에 유의하세요.)

Kinesis 스트림 내에서 병렬 처리의 기본 단위는 샤드입니다. Kinesis 스트림으로 데이터를 전송하면 요청에서 전송한 파티션 키를 기반으로 데이터가 샤드 중 하나로 전송됩니다. 각 샤드는 초당 1MB의 데이터 또는 초당 최대 1,000개의 레코드를 수집할 수 있으며 초당 최대 2MB의 송신 처리량을 지원합니다. 스트림에 샤드가 많을수록 처리할 수 있는 처리량이 늘어납니다.

스트림에 포함할 수 있는 샤드 수에는 상한선이 없으므로 이론적으로 Kinesis 스트림에 샤드를 더 추가하여 Kinesis 스트림을 무기한 확장할 수 있습니다. 그러나 워크로드에 필요한 샤드 수를 결정할 때 고려해야 할 비용 영향이 있습니다.

Kinesis 요금은 샤드 시간과 PUT 페이로드 단위라는 두 가지 핵심 차원을 기준으로 청구됩니다. 하나의 PUT 페이로드 단위는 Kinesis 스트림에 최대 25KB의 레코드를 전송하기 위한 하나의 요청과 동일합니다. 예를 들어 크기가 45KB인 데이터 조각을 보내는 경우 두 개의 PUT 페이로드 단위로 계산됩니다. Amazon의 DynamoDB의 읽기 및 쓰기 요청 단위와 동일한 방식으로 작동합니다.

Kinesis 샤드는 시간당 0.015 USD, PUT 페이로드 단위 백만 개당 0.014 USD입니다. 또한 데이터 보유 기간 연장과 같은 선택적 기능을 활성화하면 추가 요금이 부과됩니다. 이러한 추가 비용 중 일부는 연장된 데이터 보존 및 향상된 팬아웃 비용과 같은 시간당 요금도 부과됩니다.

시간당 비용 때문에 필요한 샤드 수를 과도하게 프로비저닝하는 것은 경제적으로 효율적이지 않습니다. 각 샤드가 지원하는 처리량을 감안할 때 Yle의 데이터 수집 파이프라인과 같은 높은 처리량 시스템을 지원하기 위해 많은 샤드가 필요하지 않습니다.

분당 600,000개 요청의 Yle 황금 시간대 트래픽을 기반으로 트래픽이 1분에 걸쳐 균일하게 분산된다고 가정하면 초당 10,000개 요청에 도달합니다. 그리고 각 이벤트의 크기가 25KB 미만이라고 가정하면 Yle은 이 트래픽 패턴을 수용하기 위해 약 10개의 샤드가 필요합니다. 그러나 우리가 논의한 것처럼 트래픽이 급증하고 Kinesis가 자동 크기 조정을 지원하지 않기 때문에 Yle 팀은 항상 40개의 샤드를 실행하여 스트림을 초과 프로비저닝합니다. 이를 통해 팀은 예상치 못한 급증을 처리하고 데이터 손실 위험을 최소화할 수 있는 충분한 여유 공간을 확보할 수 있습니다.

6.2.2 SQS 배달 못한 편지 대기열(DLQ)

데이터는 Yle의 ML 알고리즘에 대한 혈액 공급이므로 팀은 Yle 지역에서 Kinesis 서비스가 중단될 때 데이터가 손실되지 않도록 하려고 합니다. Kinesis 서비스가 중단된 경우 API는 이벤트를 SQS DLQ로 전송하여 나중에 캡처 및 재처리할 수 있도록 합니다.

6.2.3 라우터 람다 기능

일정한 이벤트 스트림을 처리하기 위해 라우터라는 Lambda 함수가 Kinesis 데이터 스트림을 구독합니다. 이 함수는 이벤트를 다른 Kinesis Firehose로 라우팅합니다.

다른 마이크로 서비스가 사용하는 스트림.

데이터를 보다 효율적으로 저장하고 쿼리할 수 있도록 Yle 팀은 이벤트를 Apache Parquet 형식으로 저장합니다. 이를 위해 AWS Glue Data Catalog(스키마 제공)과 함께 Amazon Kinesis Data Firehose(데이터를 대용량 파일로 일괄 처리하고 S3로 전달)를 사용합니다. 그림 6.2는 이 배열을 보여줍니다.

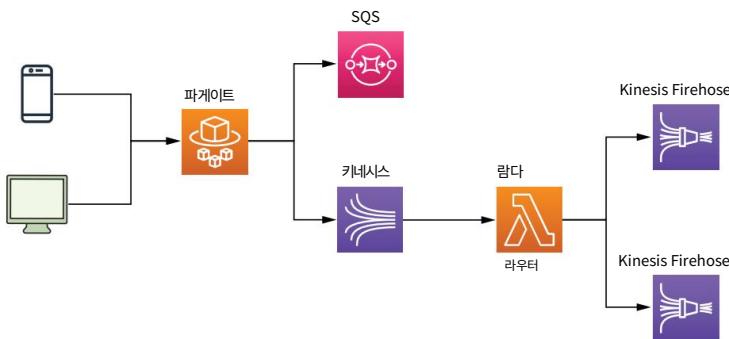


그림 6.2 Lambda 라우터 기능은 이벤트를 다른 Kinesis Firehose 스트림으로 라우팅하여 이벤트를 집계하고 Apache Parquet 파일로 변환할 수 있도록 합니다.

6.2.4 Kinesis Data Firehose

Kinesis Data Firehose는 Amazon Kinesis 서비스 제품군의 또 다른 구성원입니다. 스트리밍 데이터를 대상으로 로드하는 완전관리형 서비스입니다. Kinesis Firehose는 Amazon S3, Amazon Redshift, Amazon Elasticsearch Service(Amazon ES) 및 Datadog, New Relic 및 Splunk와 같은 외부 서비스 공급자가 소유하거나 사용자가 소유한 모든 HTTP 엔드포인트로 데이터를 보낼 수 있습니다.

Firehose 스트림을 사용하면 코드 줄 없이 스트리밍 데이터를 로드할 수 있습니다. Kinesis Data Streams와 달리 Kinesis Firehose 스트림은 자동으로 확장되며 스트림에 수집한 데이터 볼륨에 대해서만 비용을 지불합니다. Kinesis Data Firehose로 데이터를 수집하는 비용은 매월 처음 500TB의 데이터에 대해 GB당 \$0.029부터 시작합니다. 더 많은 데이터를 수집할수록 더 저렴해집니다.

자동화된 조정 외에도 Firehose 스트림은 수신 데이터를 일괄 처리하고 압축하고 선택적으로 Lambda 함수를 사용하여 변환할 수 있습니다. 또한 입력 데이터를 대상으로 로드하기 전에 JSON에서 Apache Parquet 또는 Apache ORC 형식으로 변환할 수 있습니다.

Kinesis Data Streams와 마찬가지로 최대 24시간 동안만 스트림에 데이터를 저장합니다. 최대 레코드 수 또는 일정 기간 동안 배치 크기를 구성할 수 있습니다. 예를 들어, Firehose 스트림에 데이터를 128MB 파일로 일괄 처리하거나 5분 분량의 데이터 중 먼저 제한에 도달하도록 요청할 수 있습니다. 편리하다

확장을 위한 관리 오버헤드가 없는 서비스이며,
데이터를 의도한 대상으로 전송하기 위한 사용자 정의 코드.

데이터를 JSON 형식에서 Apache Parquet 또는 Apache ORC로 변환하려면 다음을 수행합니다.
AWS Glue 데이터 카탈로그 서비스를 사용해야 합니다. Kinesis Firehose 스트림은
대상으로 보내기 전에 Glue 데이터 카탈로그에 캡처된 스키마.

Yle 팀은 S3를 Kinesis Firehose의 데이터 레이크 및 대상으로 사용합니다.
스트림(그림 6.3). 데이터가 S3로 전달되면 인구 통계 예측과 같은 여러 ML 작업을 수행하기 위해 여러 마이크로 서비스에서 추
가 처리 및 소비됩니다.

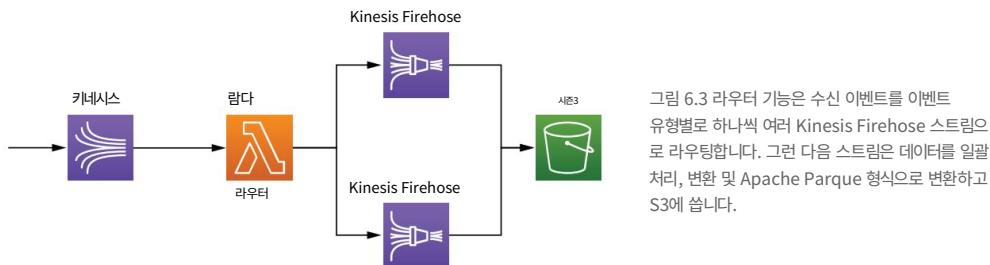


그림 6.3 라우터 기능은 수신 이벤트를 이벤트 유형별로 하나씩 여러 Kinesis Firehose 스트림으로 라우팅합니다. 그런 다음 스트림은 데이터를 일괄 처리, 변환 및 Apache Parque 형식으로 변환하고 S3에 씁니다.

6.2.5 Kinesis 데이터 분석

비디오의 아이콘 이미지를 개인화하기 위해 Yle 팀은 상황별 도적 모델을 사용합니다.

이는 감독되지 않은 ML 모델의 한 형태입니다. 그들은 사용자 상호 작용 이벤트를 사용하여
사용자가 무엇을 좋아하는지 학습할 수 있도록 모델에 보상을 제공합니다. 이를 위해 팀은 Kinesis를 사용합니다.

Firehose 스트림에서 데이터를 필터링 및 집계하여 전달하는 데이터 분석
보상 라우터라는 Lambda 함수에. 그런 다음 이 함수는 여러 보상을 호출합니다.

Yle 팀이 유지 관리하는 개인화 모델을 보상하기 위한 API(그림 6.4).

Kinesis Data Analytics를 사용하면 SQL 또는 Java 및 Apache Flink 프레임워크. SQL 접근 방식을 사용하여 조인, 필터링,
사용자 정의 코드를 작성하거나 실행하지 않고 여러 스트림에 걸쳐 데이터를 집계합니다.

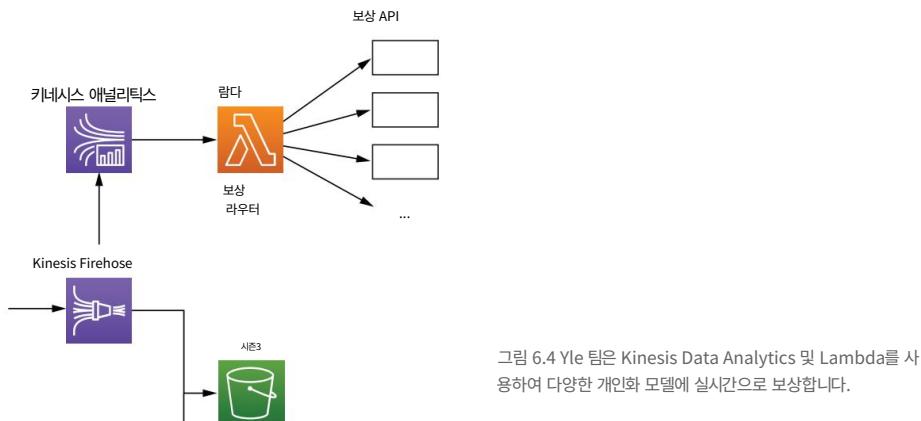


그림 6.4 Yle 팀은 Kinesis Data Analytics 및 Lambda를 사용하여 다양한 개인화 모델에 실시간으로 보상합니다.

모든 인프라. 그러나 Java 접근 방식을 사용하면 애플리케이션이 실행되는 방식과 데이터를 처리하는 방식을 가장 많이 제어할 수 있습니다.

쿼리 결과를 Kinesis Data Stream, Kinesis Firehose 또는 Lambda 함수로 출력할 수 있습니다. 그러나 Lambda 함수를 대상으로 사용하면 많은 유연성을 얻을 수 있습니다. 결과를 추가로 처리하거나 원하는 곳으로 결과를 전달하거나 둘 다 할 수 있습니다. Yle의 경우 보상 라우터 기능을 Kinesis Data Analytics 애플리케이션의 대상으로 사용하고 관련 개인화 모델에 보상합니다.

6.2.6 완전히 넣기

한 걸음 물러나면 그림 6.5에서 Yle의 데이터 파이프라인이 높은 수준에서 어떻게 보이는지 볼 수 있습니다.

Kinesis Fire 호스 스트림이 Lambda 함수를 사용하여 데이터를 변환하고 형식을 지정한다는 사실과 같은 몇 가지 사소한 세부 사항은 생략했습니다.

이것은 많은 사용자 이벤트에 대한 여정의 시작일 뿐입니다. 데이터가 Apache Parquet 형식으로 S3에 저장되면 많은 마이크로서비스가 데이터를 수집, 처리하고 ML 모델을 강화하는 데 사용합니다.

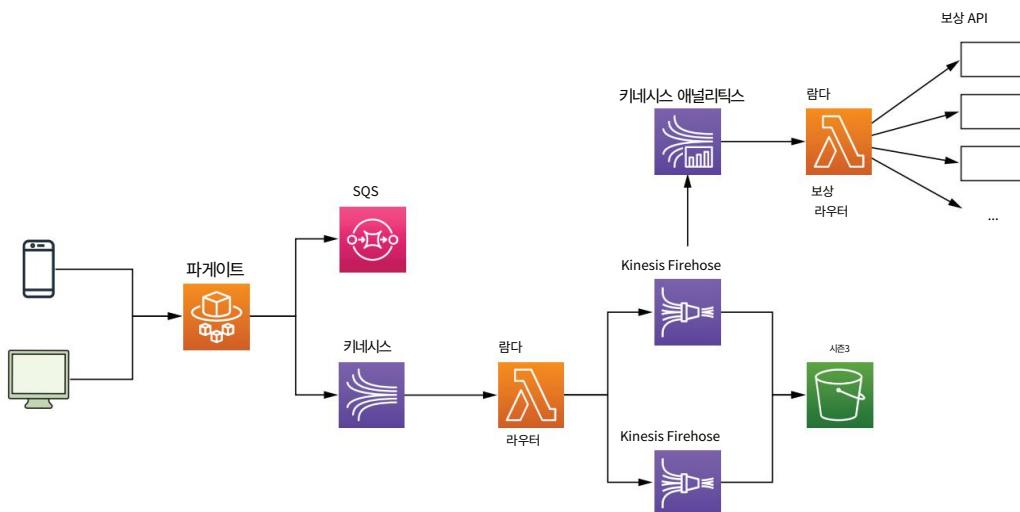


그림 6.5 Yle의 데이터 파이프라인 아키텍처. 비용 및 성능 고려 사항 때문에 Fargate를 사용하여 수집 API를 실행한 다음 Kinesis Data Streams, Kinesis Firehose 및 Lambda를 사용하여 수집된 이벤트를 실시간으로 처리합니다. 데이터는 Apache Parquet 형식으로 변환, 압축 및 변환되어 S3에 데이터 레이크로 저장됩니다. 동시에 Kinesis Data Analytics를 사용하여 실시간 집계를 수행하고 Lambda를 사용하여 관련 개인화 ML 모델에 보상합니다.

이 아키텍처에서 강조하고 싶은 것은 Kinesis와 해당 데이터 분석 기능이 널리 사용되고 있다는 점입니다. 여기에는 다음이 포함됩니다.

대량의 사용자 이벤트를 수집하기 위한 Kinesis Data Streams.

데이터를 일괄 처리, 형식 지정 및 다운스트림 ML 모델에서 더 쉽게 사용할 수 있는 대용량 압축 파일로 출력하기 위한 Kinesis Firehose Streams.

실시간으로 데이터의 라이브 스트림에 대해 집계를 실행하고 개인화 모델에 보상하기 위한 대상으로 Lambda 함수를 사용하기 위한 Kinesis Data Analytics.

6.3 배운 교훈

이러한 Kinesis 기능의 사용 및 결합 방식은 데이터 분석 애플리케이션에서 흔히 볼 수 있습니다. 그러나 Yle는 대부분의 것보다 훨씬 더 큰 규모로 이벤트를 처리하고 있습니다! 이러한 대규모 운영에는 고유한 과제가 따르며 Yle 팀은 그 과정에서 다음을 포함하여 몇 가지 귀중한 교훈을 배웠습니다.

6.3.1 서비스 한도 파악

AWS의 모든 서비스에는 서비스 제한이 있습니다. 이는 일반적으로 세 가지 범주로 나뉩니다. 리소스 제한 - 한 지역에서 보유할 수 있는 X의 수 . 예를 들어 Kinesis Data Streams의 기본 할당량은 us-east-1, us-west-1 및 eu-west-1에서 리전당 500개, 다른 모든 리전에서는 리전당 200개입니다. 마찬가지로 AWS Identity and Access Management(IAM)의 기본 할당량은 리전당 1,000개 역할입니다.

제어 평면 API 제한 - 리소스를 관리하기 위해 제어 평면 API에 보낼 수 있는 초당 요청 수입니다. 예를 들어 Kinesis Data Streams는 CreateStream API에 대한 초당 5개의 요청으로 제한합니다.

데이터 플레인 API 제한 - 데이터에 대해 조치를 취하기 위해 데이터 플레인 API에 보낼 수 있는 초당 요청 수입니다. 예를 들어 Kinesis Data Streams는 샤프당 초당 GetRecords 요청을 5개로 제한합니다.

이러한 제한은 AWS 서비스 할당량 콘솔에 게시됩니다. 콘솔에서 현재 한도와 한도를 높일 수 있는지 여부를 볼 수 있습니다.

소프트 VS. 하드 리미트

올릴 수 있는 제한은 소프트 제한 으로 간주 되고 올릴 수 없는 제한은 하드 제한으로 간주됩니다. 지원 티켓을 통해 소프트 한도 인상을 요청하거나 AWS 서비스 할당량 콘솔에서 요청할 수 있습니다. 그러나 때때로 이러한 소프트 한계를 높일 수 있는 범위에 한계가 있음을 기억할 가치가 있습니다. 예를 들어 한 지역의 IAM 역할 수는 소프트 제한이지만 해당 제한을 지역당 5,000개의 역할로만 늘릴 수 있습니다. 접근 방식이 이러한 소프트 제한을 무기한 높이는 데 의존하는 경우 설계되지 않은 방식으로 서비스를 사용하고 있을 가능성이 높으며 접근 방식을 재고해야 할 수도 있습니다.

사용 수준과 현재 한도를 주시하는 것은 모든 AWS 사용자가 해야 하는 일이지만 대규모로 운영해야 하고 이러한 한도에 도달할 위험이 있는 경우 특히 중요합니다. Yle 팀의 경우 그들이 배운 중요한 교훈 중 하나는

Fargate 작업을 실행할 수 있으며 시간이 오래 걸릴 수 있으므로 충분한 여유 공간을 확보할 수 있습니다.
AWS에서 계정 한도를 높이는 데 며칠이 걸립니다.

현재 기본 제한은 지역당 1,000개의 동시 Fargate 작업입니다. 언제
Yle 팀이 시작했지만 기본 제한은 100개에 불과했고 팀이
그 한도를 200개로 늘리는 데 3일이 걸립니다.

파이프라인을 따라 모든 지점에서 프로젝트 처리

어떤 서비스 제한이 애플리케이션에 영향을 미치는지 이해하려면 함께 모든 서비스를 살펴보십시오.
사용자 트래픽에 따라 처리량이 어떻게 변하는지 예측합니다. Yle's를 가져 가라
사례: 동시 사용자 수가 증가할수록 더 많은 트래픽이 통과합니다.
Fargate에서 실행되는 수집 API.

이 증가는 다음에서 처리해야 하는 처리량에 어떤 영향을 줍니까?

Kinesis와 이에 따라 프로비저닝해야 하는 색드 수는?

현재 BatchSize 및 ParallelizationFactor 구성은 기반으로,
처리하는 데 필요한 동시 Lambda 실행 수
최대 부하에서 이벤트?

많은 동시 Lambda 실행을 감안할 때 얼마나 많은 이벤트가
각 Kinesis Firehose 스트림으로 전송됩니까?

Kinesis Data Firehose에 대한 현재 처리량 제한은 다음을 지원합니까?
초당 많은 이벤트?

항상 테스트를 로드하고 가정 하지 마십시오 .

파이프라인의 모든 서비스는 병목 현상이 될 수 있으며 이를 알 수 있는 가장 좋은 방법은
애플리케이션이 원하는 처리량을 처리할 수 있는 것은 부하 테스트입니다. 당신이 서비스
응용 프로그램을 빌드하는 것은 확장 가능하지만 응용 프로그램이
서비스 제한에 대한 적절한 조정 없이는 아닙니다.

목표가 100,000명의 동시 사용자를 처리하는 것이라면 부하 테스트를 최소한
200,000 동시 사용자. 누가 알겠습니까? 귀하의 신청서가 성공적일 것입니다!
그것이 당신이 바라는 것입니다. 맞죠? 당신의 신청이 이미 편안하더라도
50,000명의 동시 사용자를 처리하지만 어쨌든 200,000명의 동시 사용자에게 로드 테스트합니다. 너
시스템이 무한히 확장 가능하고 성능 특성이 있다고 가정할 수 없습니다.
처리량이 증가함에 따라 완벽하게 일관됩니다. 아무것도 가정하지 마십시오. 찾아.

일부 한계에는 다른 한계보다 더 큰 폭발 반경이 있습니다.

일부 서비스 제한은 다른 제한보다 폭발 반경이 더 크다는 점도 언급할 가치가 있습니다. Lambda의 지역 동시
성 제한이 이에 대한 좋은 예입니다.

Kinesis 색드의 쓰기 처리량 제한을 소진하면
해당 색드에 대해 putRecord 작업을 수행하면 영향이 단일 색드에 국한됩니다.
단일 Kinesis 스트림은 애플리케이션에 큰 영향을 미치지 않습니다. 다른쪽에
한편, Lambda 동시 실행 제한을 소진하면 광범위한 영향을 미칠 수 있습니다.
Lambda 함수를 사용하여 처리할 가능성성이 높기 때문에 애플리케이션에 미치는 영향
다양한 워크로드: API, 실시간 이벤트 처리, 데이터 변환
Kinesis Firehose 등이 있습니다.

이것이 폭발 반경이 큰 서비스 제한에 더욱 주의를 기울여야 하는 이유입니다. Lambda의 경우 ReservedConcurrency 구성은 적절하고 필요한 경우 함수가 가질 수 있는 최대 동시 실행 수를 제한할 수도 있습니다.

MIND CLOUDWATCH의 메트릭 세분성

사용 수준을 모니터링하고 서비스 한도를 높이는 데 적극적으로 대처해야 합니다. 이를 수행하는 한 가지 방법은 관련 지표에 대해 CloudWatch 경보를 설정하는 것입니다. 여기서 유념해야 할 한 가지 주의 사항은 CloudWatch가 종종 분 단위 단위로 사용량 지표를 보고하지만 Kinesis Data Streams와 DynamoDB의 처리량 지표 모두에 적용되는 제한이 초당이라는 점입니다. 이러한 경우 해당 CloudWatch 경보를 설정할 때 임계값을 올바르게 설정했는지 확인하십시오. 예를 들어 초당 처리량 제한이 1인 경우 해당하는 분당 임계값은 60 이어야 합니다.

6.3.2 실패를 염두에 두고 구축

그림 6.1에는 SQS DLQ가 있습니까? Kinesis Data Streams 서비스에 문제가 있는 경우를 위한 백업으로 있습니다. Kinesis Data Streams는 강력하고 확장성이 뛰어난 서비스이지만 오류가 없는 것은 아닙니다.

모든 것은 항상 실패합니다 AWS CTO인

Werner Vogel은 "모든 것은 항상 실패합니다"라는 유명한 말을 했습니다. 가장 안정적이고 강력한 서비스라도 때때로 문제가 발생할 수 있다는 것은 사실입니다.

S3가 다운되었을 때를 기억하십시오 (<https://aws.amazon.com/message/41926>). 2017년에 몇 시간 동안? 아니면 Kinesis Data Streams가 중단되어 CloudWatch 및 EventBridge에도 영향을 미쳤던 때 (<https://aws.amazon.com/message/11201>)? 아니면 Gmail, Google 드라이브 및 YouTube가 다운되었을 때 (<http://mng.bz/ExIO>)?

시스템 수준에서 개별 디스크 드라이브, CPU 코어 또는 메모리 칩은 지속적으로 오류가 발생하고 교체됩니다. AWS 및 Google과 같은 클라우드 제공업체는 이러한 장애가 고객에게 영향을 미치지 않도록 물리적 인프라와 소프트웨어 인프라에 막대한 투자를 하고 있습니다. 실제로 API Gateway, Lambda 및 DynamoDB와 같은 서비스 기술을 사용하면 코드와 데이터가 지정된 AWS 리전 내의 여러 가용 영역(데이터 센터)에 저장되기 때문에 애플리케이션이 데이터 센터 전체의 장애로부터 이미 보호됩니다. 그러나 앞서 언급한 S3 및 Kinesis 중단과 같이 전체 AWS 리전에서 하나 이상의 서비스에 영향을 미치는 리전 전체 중단이 가끔 있습니다.

이것이 의미하는 바는 실패를 염두에 두고 애플리케이션을 구축하고 기본 서비스가 사무실에서 좋지 않은 날을 보낼 경우에 대비하여 계획 B(및 아마도 계획 C, D 및 E)가 있어야 한다는 것입니다. DLQ는 처음 요청했을 때 기본 대상에 전달할 수 없는 트래픽을 캡처하는 좋은 방법입니다. 이제 많은 AWS 서비스에서 즉시 사용 가능한 DLQ 지원을 제공합니다. 예를 들어 SNS, EventBridge 및 SQS는 모두 캡처한 이벤트가 재시도 후 의도한 대상에 전달되지 않는 경우 DLQ를 기본적으로 지원합니다. Lambda 함수로 Kinesis Data Stream의 이벤트를 처리하는 경우 OnFailure 구성은 사용하여 DQL을 지정할 수도 있습니다.

시스템이 처리해야 하는 처리량이 많을수록 더 많은 오류가 발생합니다.
이러한 DLQ가 더욱 중요해집니다. 5,000,000을 처리해야 하는 경우 100만 분의 1 이벤트라도 1분에 5번 발생한다는 것을 기억하십시오.
분을 요청합니다!

구성을 다시 시도하려면 주의하십시오.

아키텍처에 더 많은 움직이는 부분을 도입하고 더 많은 프로세스를 수행함에 따라
또한 시간 초과 및 재시도 구성에 더 많은 주의를 기울여야 합니다. 대규모로 작동하는 애플리케이션을 종종 괴롭히는 두 가지 문제가 있습니다.

Thundering herd - 이벤트를 기다리고 있는 다수의 프로세스가 각각
동시에 요청을 처리할 수 있는 리소스가 충분하지 않습니다.
이 모든 새로 각성 과정에서. 이로 인해 많은 리소스 경합이 발생하여 잠재적으로 시스템이 중단되거나
장애 조치될 수 있습니다.

재시도 폭풍 - 클라이언트-서버 통신의 안티 패턴입니다. 서버가
비정상 상태가 되면 클라이언트가 적극적으로 재시도하여 볼륨을
나머지 정상 서버에 대한 요청의 양을 줄이고 차례로 시간 초과 또는 실패를 일으킵니다. 이것은 더 많은
은 재시도를 유발하고 문제를 악화시킵니다.
더 나아가.

재시도는 대부분의 간헐적인 문제를 처리하는 간단하고 효과적인 방법이지만 이러한 설정은 주의해서 수행해야 합니다. 자주 백오프를 사용하는 것이 좋습니다.

재시도의 위험을 완화하기 위해 재시도와 회로 차단기 패턴 사이
폭풍 (<https://martinfowler.com/bliki/CircuitBreaker.html>).

6.3.3 일괄 처리는 비용 및 효율성 면에서 좋습니다.

비용은 개발자가 아키텍처 결정을 내릴 때 종종 생각하지 않는 것 중 하나이지만 나중에 다시 돌아와 큰 타격을 입을 수 있습니다.

대규모로 운영하고 대량의 데이터를 처리해야 하는 경우 특히 그렇습니다.
이벤트. 섹션 6.1.1에서 논의한 바와 같이 Yle 팀이 결정한 이유 중 하나는
API Gateway 대신 Fargate를 사용하여 사용자 상호 작용 이벤트 수집 및
람다는 비용과 효율성이었습니다.

일반적으로 시간을 기준으로 요금을 부과하는 AWS 서비스는
대규모로 실행할 때 기본으로 청구하는 것과 비교하여 훨씬 더 저렴합니다.
요청 횟수. 규모가 클수록 비용 대비 이벤트를 더 많이 일괄 처리해야 합니다.
효율성. 결국 단일 Lambda 호출로 1,000개의 이벤트를 처리하는 것은
1,000개의 Lambda 호출로 처리하는 것보다 저렴하고 효율적입니다.

또한 배치로 이벤트를 처리하면 실행해야 하는 동시 Lambda 실행 수가 줄어들고 지역 동시 실행이 소진 될 위험이 최소화됩니다.

실행 제한. 그러나 일괄 처리를 사용하면 부분적인 오류가 발생할 가능성이 있습니다.

한 번에 하나의 이벤트를 처리하고 해당 이벤트가 충분히 실패하면
DLQ로 이동하고 다음 이벤트로 이동합니다. 그러나 하나의 이벤트에서 1,000개의 이벤트를 처리하면
단일 호출과 하나의 이벤트가 실패하면 다른 999개의 이벤트는 어떻게 합니까? 하다
오류가 발생하고 호출을 다시 시도하여 잠재적으로 999를 다시 처리할 수 있습니다.

성공적인 이벤트? 실패한 이벤트를 DLQ에 넣고 나중에 처리합니까? 이것들은 당신이 대답해야 하는 종류의 질문입니다.

6.3.4 비용 추정이 까다롭다

비용에 신경을 쓰지 않으면 금세 쌓이고 깜짝 놀라게 할 수 있습니다.

그러나 비용을 미리 정확하게 예측하는 것도 어렵습니다. 생산 비용에 영향을 줄 수 있는 많은 요소가 있습니다. 예를 들어, 그림 6.5의 아키텍처 다이어그램을 보면 Fargate, Lambda 및 Kinesis 서비스 제품군의 비용에 초점을 맞추고 있을 수 있습니다. CloudWatch, X-Ray 비용 및 데이터 전송 비용과 같이 고려해야 할 다른 주변 서비스도 있습니다.

Lambda 비용은 일반적으로 서비스 애플리케이션의 전체 비용에서 작은 부분입니다. 실제로 대부분의 프로덕션 시스템에서 Lambda 비용은 CloudWatch 지표, 로그 및 경보 비용과 비교할 때 종종 하찮습니다.

요약 Yle의

수집 API는 하루에 5억 개 이상의 이벤트를 처리하고 피크 시간에는 분당 600,000개 이상의 이벤트를 처리합니다. 트래픽이 급증하고 라이브 하키 경기 또는 선거 결과와 같은 실제 이벤트의 영향을 많이 받습니다.

Yle 팀은 비용 및 성능 고려 사항 때문에 수집 API에 Fargate를 사용합니다.

일반적으로 가동 시간을 기준으로 요금을 부과하는 AWS 서비스는 사용량(요청 수, 처리된 데이터 양 등)에 따라 요금을 부과하는 서비스에 비해 규모에 따라 훨씬 저렴합니다.

Yle 팀은 Kinesis Data Stream, Kinesis Data Firehose 및 Lambda를 사용하여 수집된 이벤트를 처리, 변환 및 Apache Parquet 형식으로 변환합니다.

수집된 데이터는 S3에 데이터 레이크로 저장됩니다.

Yle 팀은 Kinesis Data Analytics를 사용하여 수집된 이벤트에 대한 실시간 집계를 수행합니다.

집계된 이벤트는 관련 개인화 ML 모델에 대해 보상합니다.

3부

실습

세 가지 흥미로운 문제를 살펴보고 우리가 어떻게

서버리스 아키텍처를 사용하여 문제를 해결할 것입니다. 우리는 소스 코드를 제공하지 않을 것이지만 샘플 아키텍처를 보여주고 세 가지 서로 다른 고유한 시스템을 설계하는 방법에 대해 논의할 것입니다. 이 맛있는 서버리스 아키텍처에 빠져봅시다.

임시 작업을 위한 스케 줄링 서비스 구축

이 장에서는 다음을 다룹니다.

새로운 문제에 직면했을 때 아키텍처 결정에 접근하기

비기능 요구사항 정의

비기능 요구 사항을 충족하는 올바른 AWS 서비스 선택

다양한 AWS 서비스 결합

서비스 기술을 사용하면 인프라 책임을 AWS로 분담하여 확장 가능하고 탄력적인 애플리케이션을 빠르게 구축할 수 있습니다. 그렇게 하면 고객과 비즈니스의 요구 사항에 집중할 수 있습니다. 이상적으로는 귀하가 작성하는 모든 코드가 귀하의 비즈니스를 차별화하고 고객에게 가치를 추가하는 기능에 직접 기인합니다.

이것이 실제로 의미하는 바는 자체 서비스를 구축하고 실행하는 대신 많은 관리 서비스를 사용한다는 것입니다. 예를 들어 EC2에서 RabbitMQ 서버 클러스터를 실행하는 대신 Amazon Simple Queue Service(SQS)를 사용합니다. 이 책을 읽는 동안 DynamoDB 및 Step Functions와 같은 다른 AWS 서비스에 대해서도 읽었습니다.

따라서 중요한 기술은 시스템의 비기능 요구 사항을 분석하고 작업할 올바른 AWS 서비스를 선택할 수 있는 것입니다. 그러나 AWS 에코 시스템은 방대하고 수많은 다양한 서비스로 구성되어 있습니다. 이러한 서비스 중 다수는 사용 사례에서 종복되지만 운영 계약 및 확장 특성이 다릅니다. 예를 들어 두 Lambda 함수 사이에 대기열을 추가하여 분리하려면 다음 서비스 중 하나를 사용할 수 있습니다.

아마존 SQS

Amazon Simple Notification Service(SNS)

Amazon Kinesis 데이터 스트림

Amazon DynamoDB 스트림

아마존 이벤트브리지

AWS IoT 코어

이러한 서비스는 조정 동작, 비용, 서비스 제한 및 Lambda와 통합하는 방식과 관련하여 다른 특성을 가지고 있습니다. 귀하의 요구 사항에 따라 일부는 다른 것보다 귀하에게 더 적합할 수 있습니다.

AWS는 시스템 설계를 위한 다양한 서비스를 제공하지만 언제 어떤 서비스를 사용해야 하는지에 대한 지침이나 의견은 제공하지 않습니다. AWS와 함께 작업하는 개발자 또는 설계자로서 가장 어려운 작업 중 하나는 이를 스스로 파악하는 것입니다.

이 장에서는 임시 작업을 위한 스케줄링 서비스의 설계 프로세스를 안내하여 문제를 조명합니다. 이는 애플리케이션에 대한 일반적인 요구이며 AWS는 아직 이 문제를 해결하기 위한 관리형 서비스를 제공하지 않습니다. AWS에서 가장 가까운 것은 EventBridge의 예약된 이벤트이지만 반복 작업을 예약하는 것(예: Y 초마다 X 수행)은 임시 작업을 예약하는 것과 다릅니다(예: 2021-08-30T23:59:59Z에 X 수행, Y 2021-08-20T08:05:00Z에서).

이러한 일정 서비스에 대한 기능 요구 사항은 간단합니다. 즉, 지정된 날짜 및 시간에 실행되도록 임시 작업을 예약합니다(예: "월요일 9시에 엄마에게 전화하라고 알림"). 여기서 흥미로운 점은 애플리케이션에 따라 다양한 비기능 요구 사항을 처리해야 한다는 것입니다(예: "예약되었지만 아직 실행되지 않은 백만 개의 열린 작업을 처리해야 합니다").

이 장의 나머지 부분에서는 다양한 AWS 서비스를 사용하는 이 일정 서비스에 대한 다섯 가지 솔루션을 보고 평가하는 방법을 배웁니다. 그러나 먼저 솔루션을 평가할 비기능 요구 사항을 정의하겠습니다.

이 장의 계획은 다음과 같습니다.

비기능 요구 사항을 정의합니다. 우리가 고려할 4가지 비기능적 요구 사항은 정밀도, 확장성(열린 작업 수), 확장성(핫스팟) 및 비용입니다. 다음의 모든 솔루션은 이러한 요구 사항에 대해 평가됩니다. – EventBridge가 있는 Cron - 미해결 작업을 찾기 위해 cron 작업을 사용하는 간단한 솔루션

실행합니다.

- DynamoDB TTL - DynamoDB의 TTL(Time-to-Live) 메카를 창의적으로 사용
nism을 사용하여 예약된 임시 작업을 트리거하고 실행합니다.
- Step Functions - Step Function의 Wait 상태를 사용하여 일정을 잡는 솔루션
작업을 실행합니다.

- SQS - SQS의 DelaySeconds 및 VisibilityTimeout 설정을 사용하여 예약된 실행 시간까지 작업을 숨기는 솔루션입니다.

- DynamoDB TTL과 SQS 결합 - DynamoDB 를 결합한 솔루션
서로의 단점을 보완하기 위해 SQS를 사용한 TTL.

애플리케이션에 적합한 솔루션을 선택하십시오. 응용 프로그램마다 요구 사항이 다르며 일부 비기능 요구 사항이 다른 요구 사항보다 더 중요할 수 있습니다. 이 섹션에서는 세 가지 다른 응용 프로그램을 보고 해당 요구 사항을 이해하고 가장 적합한 솔루션을 선택합니다.

7.1 비기능 요구사항 정의

Ad Hoc 스케줄링 서비스는 종종 다른 컨텍스트에서 나타나고 다른 비기능 요구사항을 갖는 흥미로운 문제입니다. 예를 들어 데 이트 앱은 날짜가 다가오고 있음을 사용자에게 알리기 위해 임시 작업이 필요할 수 있습니다. 멀티플레이어 게임은 토너먼트를 시작하거나 중지하기 위해 임시 작업을 예약해야 할 수 있습니다. 뉴스 사이트는 임시 작업을 사용하여 만료된 구독을 취소할 수 있습니다.

이러한 컨텍스트 간에 사용자 행동과 트래픽 패턴이 다르므로 서비스가 충족해야 하는 다양한 비기능 요구 사항이 생성됩니다. 무의식적인 편향(예: 확증 편향)이 스며드는 것을 방지하기 위해 이러한 요구 사항을 미리 정의하는 것이 중요합니다.

너무 자주 우리는 무의식적으로 우리의 솔루션과 일치하는 특성 뒤에 더 많은 가중치를 부여합니다. 비록 그것들이 우리 애플리케이션에 중요하지 않더라도 말입니다. 요구 사항을 미리 정의하면 객관성을 유지하는 데 도움이 됩니다. 특정 시간에 실행되도록 임시 작업을 예약할 수 있는 서비스의 경우 다음은 고려해야 할 몇 가지 비기능 요구 사항을 나열합니다.

정밀도 - 작업 실행이 예정된 시간에 얼마나 근접합니까?

확장성(미결 작업 수) - 서비스가 예약되었지만 아직 처리되지 않은 수백만 개의 작업을 지원할 수 있습니까?

확장성(핫스팟) - 서비스가 수백만 개의 작업을 동시에 실행할 수 있습니까?
비용

이 장 전체에서 이 비기능 요구 사항 집합에 대해 5가지 솔루션을 평가합니다. 그리고 오답은 없다는 것을 기억하십시오! 이 장의 목표는 솔루션을 통해 사고하고 솔루션을 평가하는 기술을 연마하는 데 도움이 되는 것입니다. 이 장의 나머지 부분에서 이러한 다양한 솔루션을 살펴보겠습니다. 각각은 다른 접근 방식을 제공하고 다른 AWS 서비스를 활용합니다. 그러나 모든 솔루션은 서비스 구성 요소만 사용하며 관리할 인프라가 없습니다. 5가지 솔루션은 다음과 같습니다.

EventBridge를 사용한 크론 작업

DynamoDB TTL(수명)

단계 함수

SQS

DynamoDB TTL과 SQS 결합

각 솔루션 후에 앞서 언급한 비기능 요구 사항에 대한 솔루션의 점수를 매기도록 요청합니다. 귀하의 점수를 당사와 비교하고 당사의 점수에 대한 근거를 볼 수 있습니다. EventBridge로 cron 작업을 위한 솔루션부터 시작하겠습니다.

7.2 EventBridge를 사용한 크론 작업

이 솔루션은 EventBridge의 cron 작업을 사용하여 몇 분마다 Lambda 함수를 호출합니다(그림 7.1). 이 솔루션을 사용하려면 다음이 필요합니다.

실행 시기를 포함하여 예약된 모든 작업을 저장할 데이터베이스(예: DynamoDB)

X 분마다 실행되는 EventBridge 일정

데이터베이스에서 기한이 지난 작업을 읽고 실행하는 Lambda 함수

cron 작업을 일정으로 정의합니다
(예: X 분마다).

실행할 기한이 지난 작업을
DynamoDB에 쿼리합니다.

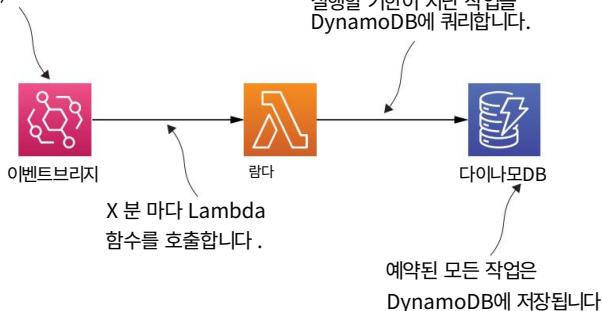


그림 7.1 임시 예약 작업을 실행하기 위해 Lambda를 사용하는 EventBridge 크론 작업을 보여주는 상위 수준 아키텍처.

이 솔루션에 대해 주의해야 할 몇 가지 사항이 있습니다.

EventBridge 일정의 최소 단위는 1분입니다. 서비스가 실행해야 하는 예약된 작업의 속도를 따라갈 수 있다고 가정하면 이 솔루션의 정밀도는 1분 이내입니다.

Lambda 함수는 최대 15분 동안 실행할 수 있습니다. Lambda 함수가 1분 동안 처리할 수 있는 것보다 더 많은 예약된 작업을 가져오면 배치를 완료할 때까지 계속 실행할 수 있습니다. 그 동안 cron 작업은 이 기능의 다른 동시 실행을 시작할 수 있습니다. 따라서 동일한 예약된 작업을 가져와서 두 번 실행하지 않도록 주의해야 합니다.

배치 내 개별 작업의 정밀도는 배치에서의 상대적 위치와 실제로 처리되는 시기에 따라 달라질 수 있습니다. 1분 이내에 처리할 수 없는 대규모 배치의 경우 일부 작업의 정밀도는 1분 이상일 수 있습니다(그림 7.2).

Lambda 함수를 대상으로 여러 번 추가하여 이 솔루션의 처리량을 늘릴 수 있습니다(그림 7.3).

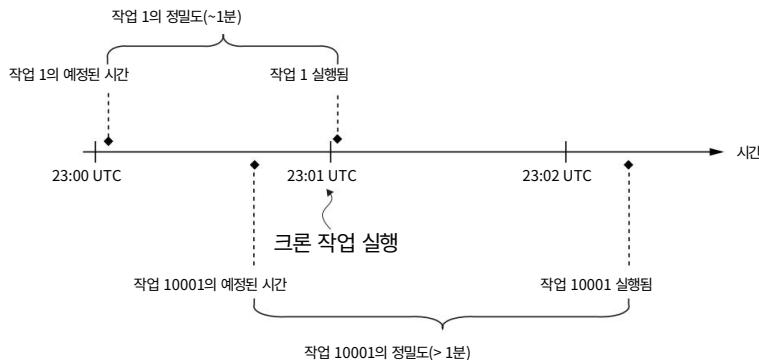


그림 7.2 배치 내 개별 작업의 정밀도는 배치 내 위치에 따라 크게 달라질 수 있습니다.

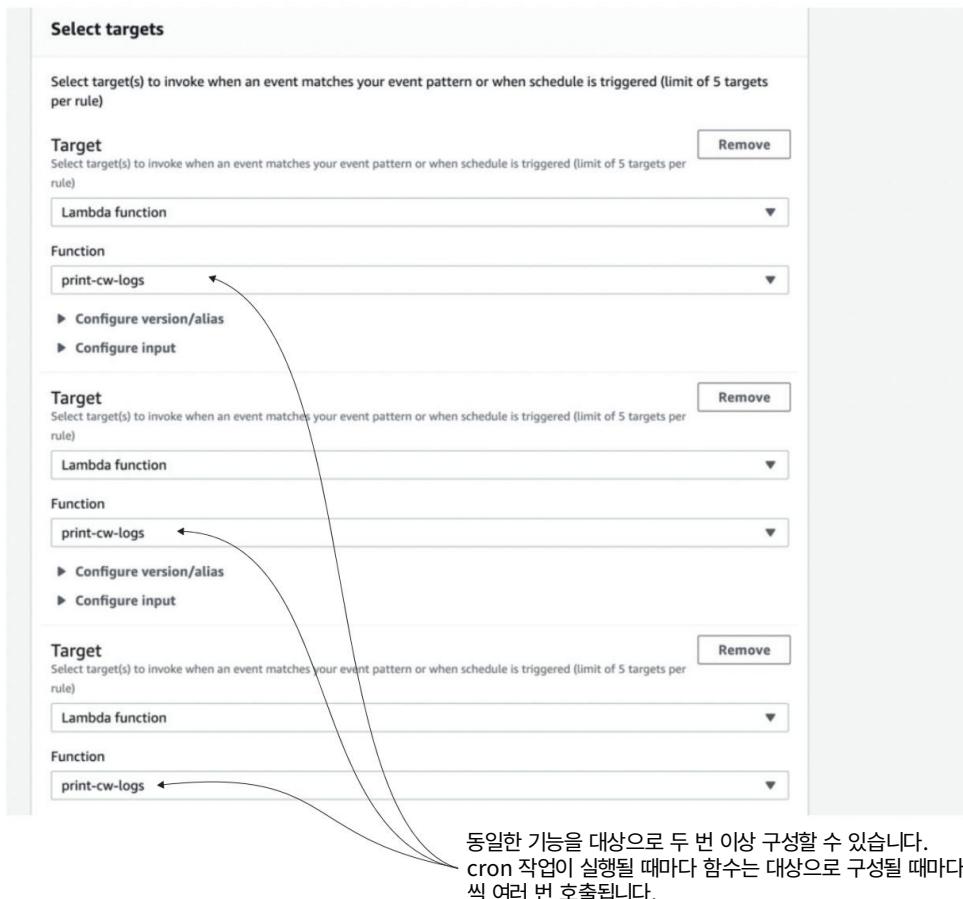
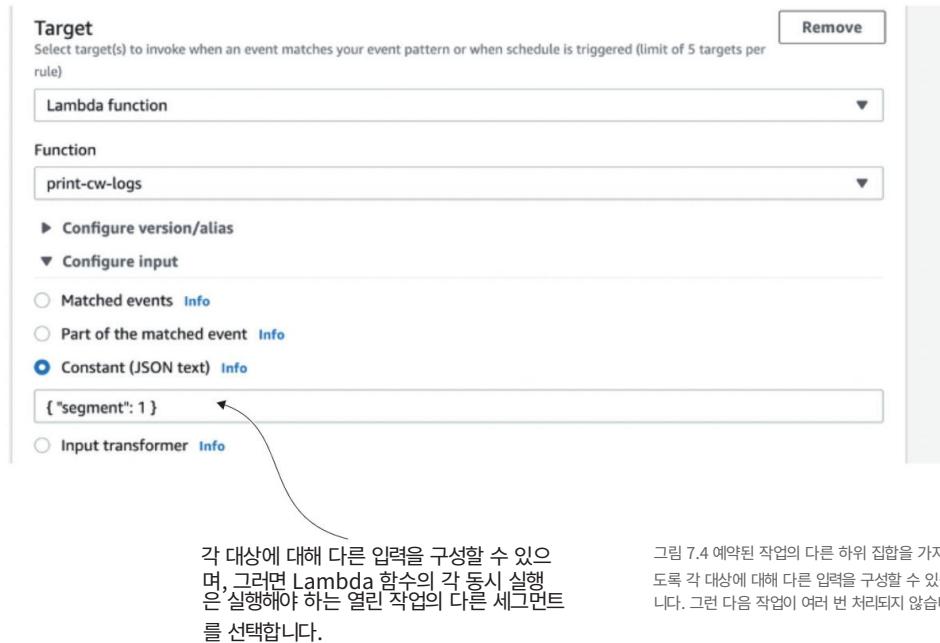


그림 7.3 동일한 Lambda 함수를 EventBridge 규칙의 대상으로 여러 번 추가할 수 있습니다.

EventBridge에는 규칙당 대상이 5개로 제한되어 있으므로 이 기술을 사용하여 처리량을 5배로 늘릴 수 있습니다.
 이것은 cron 작업을 할 때마다
 실행하면 이 Lambda 함수의 5개의 동시 실행이 생성됩니다. 피하기 위해
 모두 동일한 작업을 선택하고 실행하므로 다른 구성은 할 수 있습니다.
 그림 7.4와 같이 각 대상에 대한 입력을 보여줍니다.



The screenshot shows the AWS Lambda function configuration interface. In the 'Target' section, there is a dropdown menu set to 'Lambda function'. Below it, the 'Function' dropdown is set to 'print-cw-logs'. Under the 'Configure input' section, the 'Constant (JSON text)' option is selected, and the input value is '{ "segment": 1 }'. A callout arrow points from the text '각 대상에 대해 다른 입력을 구성할 수 있으며, 그러면 Lambda 함수의 각 동시 실행은 실행해야 하는 열린 작업의 다른 세그먼트를 선택합니다.' to the input field.

Target
 Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule)

Lambda function

Function

print-cw-logs

▶ Configure version/alias
 ▼ Configure input

Matched events [Info](#)
 Part of the matched event [Info](#)
 Constant (JSON text) [Info](#)

{ "segment": 1 }

Input transformer [Info](#)

각 대상에 대해 다른 입력을 구성할 수 있으며, 그러면 Lambda 함수의 각 동시 실행은 실행해야 하는 열린 작업의 다른 세그먼트를 선택합니다.

그림 7.4 예약된 작업의 다른 하위 집합을 가져오도록 각 대상에 대해 다른 입력을 구성할 수 있습니다. 그런 다음 작업이 여러 번 처리되지 않습니다.

7.2.1 점수

이 솔루션에 대해 어떻게 생각하십니까? 1(최악)부터 10까지의 척도로 평가한다면?

(최상) 각각의 비기능적 요구 사항에 대해? 예) 점수를 기록하십시오.

이를 위해 제공된 테이블의 빈 공간(예: 테이블 7.1 참조). 그리고

옳고 그른 답은 없다는 것을 기억하십시오. 최선의 판단을 바탕으로 사용 가능한 정보에.

표 7.1 cron 작업에 대한 솔루션 점수

| | 점수 |
|--------------|----|
| 정도 | |
| 확장성(미결 작업 수) | |
| 확장성(핫스팟) | |
| 비용 | |

7.2.2 우리의 점수

이 솔루션의 가장 큰 장점은 구현이 정말 간단하다는 것입니다. 솔루션의 복잡성은 실제 프로젝트에서 중요한 고려 사항입니다.

항상 자원과 시간 제약에 얹매여 있습니다. 그러나 이 책의 목적을 위해

우리는 이러한 실제 제약 조건을 무시하고 섹션 7.1에 설명된 비기능 요구 사항만 고려합니다. 즉, 이 솔루션에 대한 점수는 다음과 같습니다(표)

7.2). 그런 다음 다음 하위 섹션에서 이러한 점수에 대한 이유를 설명합니다.

표 7.2 EventBridge를 사용한 cron 작업에 대한 솔루션 점수

| | 점수 |
|--------------|----|
| 정도 | 6 |
| 확장성(미결 작업 수) | 10 |
| 확장성(핫스팟) | 2 |
| 비용 | 7 |

정도

EventBridge 크론 작업이 최대로 실행될 수 있기 때문에 이 솔루션에 정밀도 6을 부여했습니다.

분당 한 번. 이것이 우리가 이 솔루션에서 기대할 수 있는 최고의 정밀도입니다. 또한 이 솔루션은 다음에서 처리할 수 있는 작업의 수도 제한됩니다.

각 반복. 동시에 실행해야 하는 작업이 너무 많은 경우,

쌓이고 지연될 수 있습니다. 이러한 지연은 이 솔루션의 가장 큰 문제인 핫스팟 처리의 증상입니다. 다음에 더.

확장성 (오픈 태스크 수)

열린 작업이 함께 클러스터되지 않는 경우 (핫스팟) 이 솔루션은

수백만 개의 미해결 작업으로 확장하는 데 문제가 없습니다. 매번

Lambda 함수가 실행되며 현재 기한이 지난 작업에만 관심이 있습니다. 때문에

이 솔루션에 확장성(열린 작업 수)에 대해 완벽한 10을 부여했습니다.

확장성 (핫스팟)

cron 작업이 처리하지 않기 때문에 이 기준에 대해 이 솔루션에 낮은 2를 부여했습니다.

핫스팟은 전혀. Lambda 함수가 처리할 수 있는 것보다 많은 작업이 있는 경우

한 번의 호출로 이 솔루션은 모든 종류의 문제에 부딪히고 우리를 어려운 절충안으로 몰아넣습니다.

예를 들어, 함수가 1분 이상 실행되도록 허용합니까? 그렇지 않으면

그러면 함수가 시간 초과되고 일부 작업이

호출이 도중에 중단되었기 때문에 처리될 수 있지만 그렇게 표시되지 않습니다. 예약된 작업이 멱등인지 확인하거나

다음 중에서 선택해야 합니다.

데이터베이스에서 처리된 것으로 표시한 경우 일부 작업을 두 번 실행 성공적으로 처리 중입니다.

데이터베이스에서 처리된 것으로 표시하면 일부 작업을 전혀 실행하지 않음 처리를 마치기 전에.

Saga 패턴과 같은 메커니즘 사용 (<http://mng.bz/AOEp>) ~을 위한 트랜잭션을 관리하고 데이터베이스 레코드를 안정적으로 업데이트합니다. 작업이 성공적으로 처리되었습니다. (이는 많은 복잡성과 비용을 추가할 수 있습니다. 해결책.)

반면에 함수가 1분 이상 실행되도록 허용하면

충분히 큰 핫스팟을 볼 때까지 이 문제를 경험할 가능성이 적습니다.

Lambda 함수는 15분 안에 처리할 수 없습니다! 또한 이제 하나 이상의

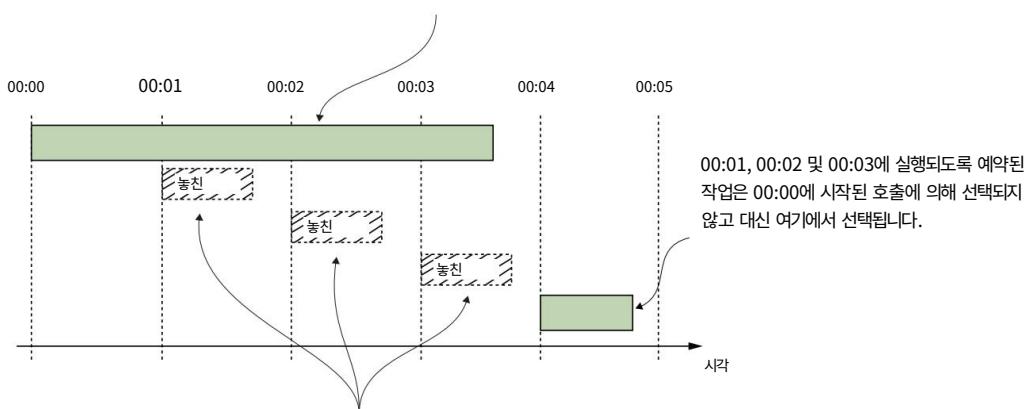
동시에 실행되는 이 기능의 동시 실행. 같은 것을 피하기 위해

작업이 두 번 이상 실행되는 경우 함수의 예약된 동시성 설정을 1로 설정할 수 있습니다. 이렇게 하면 언제든지

Lambda 함수가 실행 중입니다(그림 7.5 참조). 그러나 이것은 시스템의 잠재적 처리량을 심각하게 제한합니다.

00:00 UTC에 실행해야 하는 1,000,000개의 작업을 상상해 보십시오. 그러나 Lambda 함수는 분당 10,000개의 작업만 처리할 수 있습니다. 우리가 아무것도 하지 않는다면, 함수는 시간 초과되고 재시도되며 모든 작업을 완료하는 데 최소 100번의 호출이 필요합니다.

예약된 동시성이 1로 설정되면 함수의 동시 실행 하나만 동시에 실행할 수 있습니다.



cron의 다음 반복에서 작업을 수행하면 함수를 다시 호출하려고 시도합니다. 이러한 호출은 Lambda에 의해 조절되고 cron 작업은 이러한 반복을 놓치게 됩니다.

그림 7.5 동시성 예약을 1로 제한하면 어떤 순간에도 Lambda 함수가 한 번만 동시에 실행됩니다. 이는 일부 cron 작업 주기를 건너뛸 것을 의미합니다.

작업. 그 동안 다른 작업도 지연되어 사용자 경험에 미치는 영향을 더욱 화나게 합니다. 이것이 이 솔루션의 아킬레스건입니다. 그러나 처리량을 높이고 핫스팟에 더 잘 대처하도록 솔루션을 조정할 수 있습니다. 이에 대한 자세한 내용은 나중에 설명합니다.

비용

EventBridge를 사용하면 크론 작업이 무료이지만 실행할 작업이 없는 경우에도 Lambda 호출에 대해 비용을 지불해야 합니다. 크론 작업에 적당한 메모리 크기를 사용하면 Lambda 비용을 최소화할 수 있습니다. 결국 CPU를 많이 사용하는 작업을 수행하지 않으면 많은 메모리(따라서 CPU)가 필요하지 않습니다.

이 시나리오에서 이 솔루션의 주요 비용은 DynamoDB 읽기 및 쓰기 요청입니다. 모든 작업에 대해 하나의 쓰기 요청(작업을 예약할 때)과 하나의 읽기 요청(크론 작업에서 검색할 때)이 필요합니다. 이러한 액세스 패턴은 DynamoDB의 온디맨드 가격 책정에 적합하며 솔루션 비용이 규모에 따라 조기에 성장할 수 있도록 합니다. 백만 쓰기 단위당 1.25 USD 및 백만 읽기 단위당 0.25 USD로 예약된 작업 백만 개당 비용은 1.50 USD만큼 낮을 수 있습니다. 이는 DynamoDB 비용일 뿐이며 DynamoDB 읽기/쓰기 단위가 페이로드 크기를 기반으로 계산되기 때문에 각 작업에 대해 저장해야 하는 항목의 크기에 따라 다릅니다. 또한 메모리 크기 및 실행 기간과 같은 여러 요인에 따라 달라지는 Lambda 비용도 고려해야 합니다.

그럼에도 불구하고 하루에 수백만 개의 예약된 작업으로 확장하더라도 여전히 비용 효율적인 솔루션입니다. 따라서 우리가 7점을 준 이유입니다. 전반적으로 이것은 핫스팟을 처리할 필요가 없는 응용 프로그램에 좋은 솔루션이며 구현하기도 쉽습니다. 앞서 언급했듯이 핫스팟 문제를 해결하기 위해 아키텍처를 약간 조정할 수도 있습니다.

7.2.3 솔루션 조정

앞서 우리는 Lambda의 여러 동시 실행을 병렬로 실행하여 이 솔루션의 처리량을 늘릴 수 있다고 언급했습니다. EventBridge 규칙에서 Lambda 함수 대상을 복제하여 이를 수행할 수 있습니다. EventBridge 규칙당 대상이 5개로 제한되어 있기 때문에 이것 해야 5배의 증가만 기대할 수 있습니다.

그 외에도 EventBridge 규칙 자체를 필요한 만큼 복제할 수도 있습니다.

그러나 이러한 트릭에도 불구하고 Lambda의 15분 실행 시간 제한은 여전히 우리 머리를 어렵듯이 느끼고 있습니다. 또한 동시 실행이 동일한 작업을 처리하지 않도록 읽기를 샤딩해야 합니다. 그렇게 하면 더 높은 운영 비용과 복잡성도 발생합니다. 구성 및 관리 할 리소스가 더 많고 대부분의 경우 필요하지 않은 경우에도 더 많은 Lambda 호출 및 데이터베이스 읽기가 있습니다. 기본적으로 우리는 항상 최대 처리량을 위해 애플리케이션을 "프로비저닝"했습니다(더 나은 용어가 없기 때문에).

이 방법으로 처리량을 늘리는 것은 비효율적입니다. 훨씬 더 나은 대안은 실행해야 하는 작업의 수를 기반으로 처리 논리를 팬아웃(fan-out)하는 것입니다. Lambda의 버스트 용량 제한을 통해 최대 3,000개의 동시 실행을 즉시 생성할 수 있습니다(<https://amzn.to/2BxRuVG> 참조). 이것은 병렬 처리에 대한 엄청난 잠재력을 허용합니다.

우리가 그것의 일부만 사용하더라도, 이것이 작동하려면 비즈니스 로직을 옮겨 다른 Lambda 함수로 작업을 가져와 실행해야 합니다. 여기에서 대규모 작업 배치에 직면했을 때 필요하다고 생각하는 만큼 이 새로운 기능의 인스턴스를 호출할 수 있습니다(그림 7.6은 이 접근 방식을 보여줍니다).

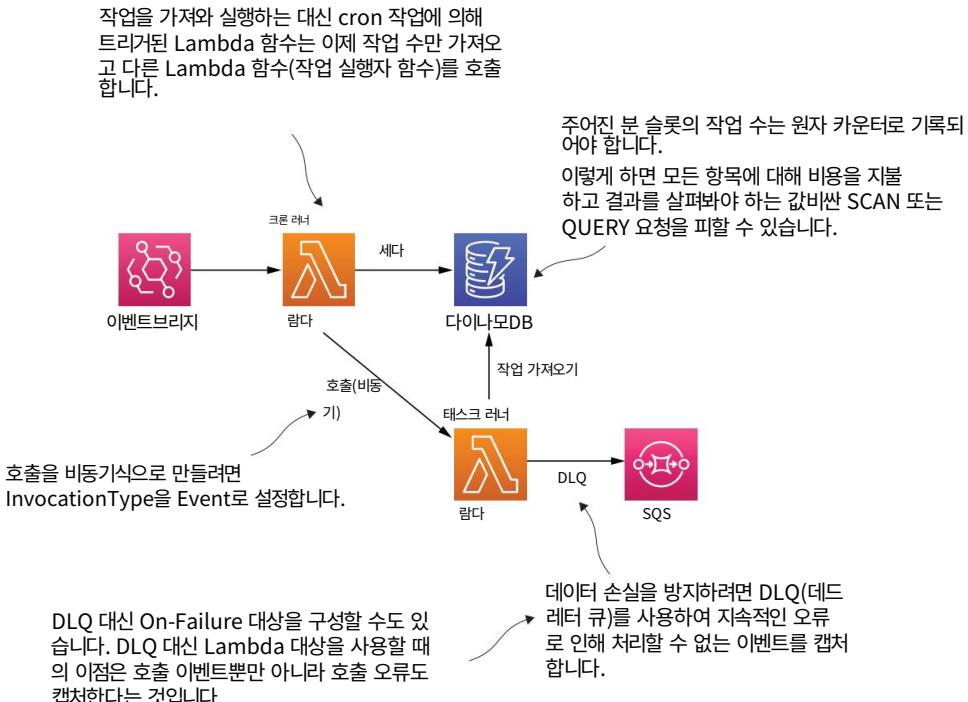


그림 7.6 cron 작업에 대한 솔루션으로 대안 아키텍처. 솔루션은 처리 논리를 다른 기능으로 확장합니다.

실행해야 하는 작업의 수를 알면 필요한 동시 실행 수를 계산할 수 있습니다. 시간 압박을 완화하고 시간 초과의 위험을 최소화하기 위해 계산에 약간의 여유를 추가할 수 있습니다.

예를 들어 처리 기능의 처리량이 분당 10,000개 작업이면 5,000개 작업마다 새로운 동시 실행을 시작할 수 있습니다. 1,000,000개의 작업이 있는 경우 200개의 동시 실행이 필요합니다. 이는 해당 지역에서 동시 실행 3,000건이라는 버스트 용량 제한보다 훨씬 낮은 수치입니다.

연습: 수정된 솔루션에 접수 매기기

제안된 변경 사항이 정밀도, 확장성(열린 작업 수), 확장성(핫스팟) 및 비용과 같은 비기능적 요구 사항에 어떤 영향을 미칠지 고려하십시오. 이 수정된 솔루션을 어떻게 평가하시겠습니까?

연습: 다른 대안

크론 작업을 사용하는 것과 동일한 일반적인 접근 방식을 유지하면서 정밀도의 단점을 보완할 수 있는 기본 설계에 대한 수정 사항이 있습니까?

핫스팟에 대한 확장?

7.2.4 최종 생각

Cron 작업은 간단하면서도 효과적인 솔루션이 될 수 있습니다. 보시다시피, 약간의 조정으로 또한 대규모 핫스팟을 지원하도록 확장할 수도 있습니다. 그러나, 그것은 많은 것을 밀어내는 경향이 있습니다. 데이터베이스에 대한 로드. 앞서 말한 1,000,000개의 작업이 필요한 시나리오에서

1분 안에 실행하려면 DynamoDB에서 1,000,000번을 읽어야 합니다. 운 좋게

DynamoDB는 이러한 수준의 트래픽을 처리할 수 있지만

처리량 제한이 있습니다. 예를 들어 DynamoDB의 기본 제한은

주문형 테이블의 경우 테이블당 읽기 단위 40,000개 (<https://amzn.to/3eH0THZ>).

읽지 않고도 스케줄링 서비스를 구현할 수 있는 방법이 있다면

DynamoDB 테이블에서 전혀? 우리는 그것을 활용함으로써 그것을 할 수 있다는 것이 밝혀졌습니다.

DynamoDB의 TTL(Time-to-Live) 기능 (<https://amzn.to/2NRgARU>).

7.3 다이나모DB TTL

DynamoDB를 사용하면 항목에 TTL 값을 지정할 수 있으며

TTL이 지났습니다. 이것은 완전히 관리되는 프로세스이므로 아무 것도 할 필요가 없습니다.

각 항목에 대한 TTL 값을 지정하는 것 외에

TTL 값을 사용하여 특정 시간에 실행해야 하는 작업을 예약할 수 있습니다.

항목이 테이블에서 삭제되면 REMOVE 이벤트가 해당 DynamoDB 스트림에 게시됩니다. 이 스트림에 대한 Lambda 함수를 구독하고

테이블에서 제거되면 작업을 실행합니다(그림 7.7).

이 테이블에는 다음과 같은 모든 작업이 포함됩니다.

예정되었습니다. 예정된 시간

작업은 TTL로 사용되며 일정 시간이 되면

DynamoDB TTL에 의해 삭제됩니다.

Execute 함수는 다음을 수신합니다.

REMOVE 이벤트 및

해당 작업을 실행합니다.

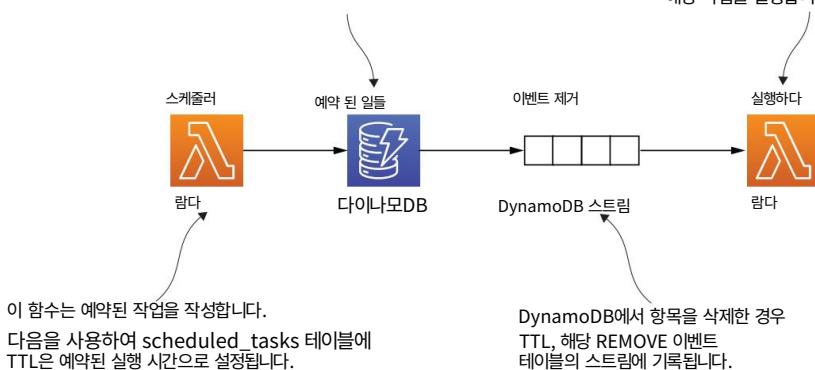


그림 7.7 DynamoDB TTL을 사용하여 임시 예약된 작업을 실행하는 상위 아키텍처.

이 솔루션에 대해 염두에 두어야 할 몇 가지 사항이 있습니다. 첫 번째이자 가장 중요한 것은 DynamoDB TTL이 테이블에서 만료된 항목을 삭제하는 속도에 대한 보장을 제공하지 않는다는 것입니다. 실제로 공식 문서 (<https://amzn.to/2NRgARU>)에는 "TTL은 일반적으로 만료 후 48시간 이내에 만료된 항목을 삭제합니다"(그림 7.8 참조)라고 나와 있습니다. 실제로 실제 타이밍은 일반적으로 그렇게 암울하지 않습니다.

수집한 경험적 데이터에 따르면 항목은 일반적으로 만료 후 30분 이내에 삭제됩니다. 그러나 그림 7.8과 같이 테이블의 크기와 활동 정도에 따라 크게 달라질 수 있다.

⚠ Important

- Depending on the size and activity level of a table, the actual delete operation of an expired item can vary. Because TTL is meant to be a background process, the nature of the capacity used to expire and delete items via TTL is variable (but free of charge). TTL typically deletes expired items within 48 hours of expiration.

그림 7.8 테이블에서 만료된 항목을 삭제하는 기능에 대한 DynamoDB TTL의 알림

두 번째로 고려해야 할 사항은 DynamoDB 스트림의 처리량이 스트림의 색드 수에 따라 제한된다는 것입니다. 색드 수는 DynamoDB 테이블의 파티션 수에 따라 결정됩니다. 그러나 파티션 수를 직접 제어할 수 있는 방법은 없습니다. 테이블의 항목 수와 읽기 및 쓰기 처리량을 기반으로 DynamoDB에서 완전히 관리됩니다.

데이터 분할 방식과 같은 내부 메커니즘을 포함하여 DynamoDB에 대한 많은 정보를 제공하고 있다는 것을 알고 있습니다. 이 모든 것이 처음이라도 걱정하지 마십시오. AWS re:invent 2018에서 이 세션을 시청하여 DynamoDB가 내부적으로 작동하는 방식에 대해 많은 것을 배울 수 있습니다. <https://www.youtube.com/watch?v=yvBR71D0nAQ>.

7.3.1 점수

이 솔루션에 대해 어떻게 생각하십니까? 각각의 비기능 요구 사항에 대해 1에서 10까지의 척도로 평가하시겠습니까? 이전과 같이 표 7.3의 빈 칸에 점수를 적으십시오.

표 7.3 DynamoDB TTL 점수

| | 점수 |
|--------------|----|
| 정도 | |
| 확장성(미결 작업 수) | |
| 확장성(핫스팟) | |
| 비용 | |

7.3.2 우리의 점수

이 솔루션의 가장 큰 문제는 DynamoDB TTL이 예약된 항목을 안정적으로 삭제하지 않는다는 것입니다. 이 제한은 원격으로 시간에 민감한 애플리케이션에는 적합하지 않음을 의미합니다. 다음은 표 7.4의 점수입니다. 다시 말하지만, 다음 하위 섹션에서 이러한 점수에 도달한 방법을 제시합니다.

표 7.4 DynamoDB TTL에 대한 점수

| | 점수 |
|--------------|----|
| 정도 | 1 |
| 확장성(미결 작업 수) | 10 |
| 확장성(핫스팟) | 6 |
| 비용 | 10 |

정도

예약된 작업은 48시간 이내에 실행됩니다.

예정된 시간의 시간. 1점

여기에서 아첨하는 점수로 간주 될 수 있습니다.

확장성 (오픈 태스크 수)

열려 있는 작업의 수가 scheduled_tasks 테이블의 항목 수와 같기 때문에 이 솔루션에 완벽한 10을 주었습니다.

DynamoDB는 테이블에 포함할 수 있는 총 항목 수에 제한이 없으므로 이 솔루션은 수백만 개의 열린 작업으로 확장할 수 있습니다. 데이터베이스가 커질수록 성능이 빠르게 저하될 수 있는 관계형 데이터베이스와 달리 DynamoDB는 규모에 관계없이 일관되고 빠른 성능을 제공합니다. 그림 7.9는 성능에 대한 증언을 제공합니다.

확장성 (핫스팟)

DynamoDB 스트림의 처리량으로 인해 여전히 처리량 관련 문제에 직면할 수 있기 때문에 이 솔루션에 6을 부여했습니다. 그러나 작업은 단순히 스트림에 대기되고 예정보다 약간 늦게 실행됩니다.

이 처리량 제약 조건을 이해하는 데 유용하므로 좀 더 자세히 살펴보겠습니다. 앞서 언급했듯이 DynamoDB 스트림의 사용 수는 DynamoDB에서 관리합니다. 모든 샤크에 대해 Lambda 서비스에는 전용

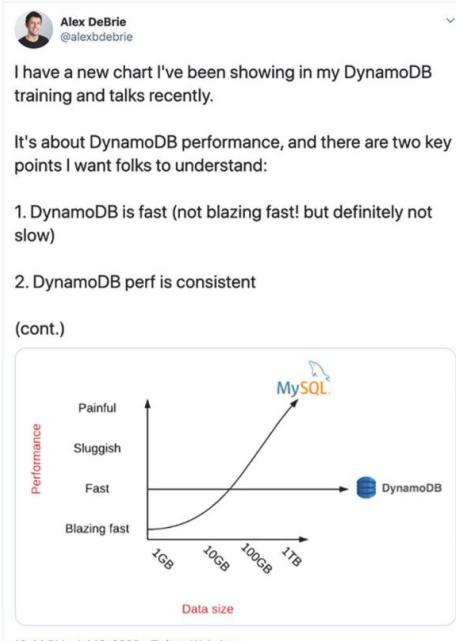


그림 7.9 DynamoDB의 확장성과 열린 작업 수에 대한 만족스러운 고객의 진술

실행 기능의 동시 실행. Lambda가 어떻게 작동하는지 자세히 읽을 수 있습니다.

공식 문서에서 DynamoDB 및 Kinesis 스트림과 함께 작동

<https://amzn.to/2Zlu3Cx>.

많은 수의 작업이 DynamoDB에서 동시에 삭제되면

REMOVE 이벤트는 실행 함수가 다음을 수행할 수 있도록 DynamoDB 스트림에 대기합니다.

프로세스. 이러한 이벤트는 최대 24시간 동안 스트림에 유지됩니다. 실행하는 한

함수가 결국 따라잡을 수 있게 되면 데이터가 손실되지 않습니다.

핫스팟 자체에 대한 확장성 문제는 없지만 이 솔루션의 처리량에 영향을 미치는 요소를 고려해야 합니다. 궁극적으로 이러한 쓰루풋 제한은 이 솔루션의 정밀도에 영향을 미칩니다.

핫스팟이 처리되는 속도는 DynamoDB TTL이 삭제하는 속도에 따라 다릅니다.

그 항목들. DynamoDB TTL은 일괄적으로 항목을 삭제하며 우리는 제어할 수 없습니다.

실행 빈도와 각 배치에서 삭제되는 항목 수.

실행 기능이 핫스팟의 모든 작업을 처리하는 속도는 다음으로 제한됩니다.

병렬로 실행되는 인스턴스 수입니다. 불행히도, 우리는 궁극적으로 결정하는 scheduled_tasks 테이블의 파티션 수를 제어할 수 없습니다.

실행 기능의 동시 실행 수입니다. 그러나 우리는 할 수 있습니다

샤드당 동시 배치 구성 설정을 재정의합니다 (<https://amzn.to/2YUGE59>), 병렬 처리 계수를 10배로 늘릴 수 있습니다.

(그림 7.10 참조).

▼ Additional settings

On-failure destination

Lambda discards records that are expired or fail all retry attempts. You can send discarded records from a stream to an Amazon SQS queue or an Amazon SNS topic.

Queue or topic ARN

Retry attempts

The maximum number of times to retry when the function returns an error.

10000

Maximum age of record

The maximum age of a record that Lambda sends to a function for processing. The age can be up to 604,800 seconds (7 days).

604800

Split batch on error

If the function returns an error, split the batch into two and retry.

Concurrent batches per shard

Process multiple batches from the same shard concurrently.

1

그림 7.10 Kinesis 및 DynamoDB 스트림 기능에 대한 추가 설정에서 샤드당 동시 배치 설정을 찾을 수 있습니다.

비용

이 솔루션에는 DynamoDB 읽기가 필요하지 않습니다. 삭제된 항목은 DynamoDB 스트림의 REMOVE 이벤트에 포함됩니다. DynamoDB 스트림의 이벤트는 일괄 처리로 수신되기 때문에 처리하는 데 효율적이며 더 적은 수의 Lambda 호출이 필요합니다. 또한 DynamoDB 스트림은 일반적으로 읽기 요청 수에 따라 요금이 부과되지만 Lambda로 이벤트를 처리할 때는 무료입니다. 이러한 특성 때문에 이 솔루션은 수백만 개의 예약된 작업으로 확장되는 경우에도 매우 비용 효율적입니다. 따라서 이것이 우리가 비용에 대해 완벽한 10을 준 이유입니다.

7.3.3 최종 생각

이 솔루션은 DynamoDB의 TTL 기능을 창의적으로 사용하고 예약된 작업을 실행하기 위한 매우 비용 효율적인 솔루션을 제공합니다. 그러나 DynamoDB TTL은 작업이 얼마나 빨리 삭제되는지에 대한 합리적인 보장을 제공하지 않기 때문에 많은 애플리케이션에 적합하지 않습니다. 사실, cron 작업이나 DynamoDB TTL은 예약된 시간에서 몇 초 이내에 작업을 실행해야 하는 애플리케이션에 적합하지 않습니다. 이러한 응용 프로그램의 경우 다른 비기능 요구 사항을 희생시키면서 비활 데 없는 정밀도를 제공하는 당사의 다음 솔루션이 가장 적합할 수 있습니다.

7.4 Step Functions Step

Functions는 복잡한 워크플로를 상태 머신으로 모델링할 수 있는 오케스트레이션 서비스입니다. 상태 머신이 새로운 상태로 전환될 때 Lambda 함수를 호출하거나 DynamoDB, SNS 및 SQS와 같은 다른 AWS 서비스와 직접 통합할 수 있습니다.

Step Functions의 절제된 초능력 중 하나는 대기 상태입니다 (<https://amzn.to/38po884>). 최대 1년 동안 워크플로를 일시 중지할 수 있습니다! 일반적으로 유휴 대기는 Lambda에서 수행하기 어렵습니다. 그러나 Step Functions를 사용하면 JSON 몇 줄 만큼 쉽습니다.

```
"wait_ten_seconds": {
    "유형": "기다려",
    "초": 10,
    "다음": "다음 상태"
}
```

특정 UTC 타임스탬프까지 기다릴 수도 있습니다.

```
"wait_until": {
    "유형": "기다려",
    "타임스탬프": "2016-03-14T01:59:00Z",
    "다음": "다음 상태"
}
```

그리고 TimestampPath를 사용하면 실행에 전달되는 데이터를 사용하여 Timestamp 값을 매개변수화할 수 있습니다.

```
"wait_until": {
    "유형": "대기",
    "TimestampPath": "$.scheduledTime",
    "다음": "다음 상태"
}
```

임시 작업을 예약하기 위해 상태 시스템 실행을 시작하고 대기 상태를 사용하여 지정된 날짜 및 시간까지 워크 플로를 일시 중지할 수 있습니다. 이 솔루션은 정확합니다.
수집한 데이터를 기반으로 작업은 90번째 백분위수에서 예약된 시간의 0.01초 이내에 실행됩니다. 그러나 염두에 두어야 할 몇 가지 서비스 제한이 있습니다(<https://amzn.to/2C4fGPD> 참조).

StartExecution API에는 제한이 있습니다. 이 API는 모든 작업에 자체 상태 머신 실행이 있기 때문에 새 작업을 예약할 수 있는 속도를 제한합니다(그림 7.11 참조).

Quotas Related to API Action Throttling

Some Step Functions API actions are throttled using a token bucket scheme to maintain service bandwidth.

 Note

Throttling quotas are per account, per AWS Region. AWS Step Functions may increase both the bucket size and refill rate at any time. Do not rely on these throttling rates to limit your costs.

Quotas In US East (N. Virginia), US West (Oregon), and Europe (Ireland)

| API Name | Bucket Size | Refill Rate per Second |
|----------------|-------------|------------------------|
| StartExecution | 1,300 | 300 |

Quotas In All Other Regions

| API Name | Bucket Size | Refill Rate per Second |
|----------------|-------------|------------------------|
| StartExecution | 800 | 150 |

그림 7.11 AWS Step Functions에 대한 StartExecution API 제한

초당 상태 전환 수에는 제한이 있습니다. 대기 상태가 만료되면 예약된 작업이 실행됩니다. 그러나 많은 작업이 모두 동시에 실행되는 대규모 핫스팟이 있는 경우 이러한 제한으로 인해 제한될 수 있습니다(그림 7.12 참조).

Quotas Related to State Throttling

Step Functions state transitions are throttled using a token bucket scheme to maintain service bandwidth.

 Note

Throttling on the `StateTransition` service metric is reported as `ExecutionThrottled` in Amazon CloudWatch. For more information, see the [ExecutionThrottled CloudWatch metric](#).

| Service Metric | Bucket Size | Refill Rate per Second |
|---|-------------|------------------------|
| <code>StateTransition</code> — In US East (N. Virginia), US West (Oregon), and Europe (Ireland) | 5,000 | 1,500 |
| <code>StateTransition</code> — All other regions | 800 | 500 |

그림 7.12 AWS Step Functions의 상태 전환 제한

기본 제한은 1,000,000개의 열린 실행입니다. 예약된 작업당 하나의 열린 실행이 있기 때문에 이것은 시스템이 지원할 수 있는 열린 작업의 최대 수입니다.

고맙게도 이러한 제한은 모두 소프트 제한이므로 서비스 제한을 높이면 늘릴 수 있습니다. 그러나 이들 중 일부에 대한 기본 제한이 매우 낮다는 점을 감안할 때 단일 핫스팟에서 백만 개의 예약된 작업 실행을 지원할 수 있는 수준으로 높이는 것이 불가능할 수 있습니다.

또한 실행이 1년이라는 엄격한 제한이 있습니다.

이것은 시스템이 1년 이내에 작업을 예약하도록 제한합니다. 대부분의 사용 사례에서는 이것으로 충분할 것입니다. 그렇지 않은 경우 1년 이상 예정된 작업을 지원하도록 솔루션을 조정할 수 있습니다(자세한 내용은 나중에 설명).

7.4.1 점수

이 솔루션에 대해 어떻게 생각하십니까? 각각의 비기능 요구 사항에 대해 1에서 10까지의 척도로 평가하시겠습니까? 이전과 마찬가지로 표 7.5에 제공된 빈 공간에 점수를 기록합니다.

표 7.5 Step Functions에 대한 솔루션 점수

| | 점수 |
|--------------|----|
| 정도 | |
| 확장성(미결 작업 수) | |
| 확장성(핫스팟) | |
| 비용 | |

7.4.2 우리의 점수

Step Functions는 당면한 문제에 대한 간단하고 우아한 솔루션을 제공합니다. 그러나 확장을 어렵게 만드는 여러 서비스 제한으로 인해 방해를 받습니다. 우리는 이러한 제한 사항에 대해 자세히 알아볼 것이지만 먼저 표 7.6은 이 솔루션에 대한 점수를 보여줍니다.

표 7.6 단계 함수에 대한 점수

| | 점수 |
|--------------|----|
| 정도 | 10 |
| 확장성(미결 작업 수) | 7 |
| 확장성(핫스팟) | 4 |
| 비용 | 2 |

정도

앞서 언급했듯이 Step Functions는 90번째 백분위수에서 0.01초 정밀도 내에서 작업을 실행할 수 있습니다. 그것보다 더 정확하지 않습니다!

확장성 (오픈 태스크 수)

StartExecution API 제한이 얼마나 많은

초당 생성할 수 있는 예약된 작업. 예약된 솔루션을 저장하는 반면

DynamoDB의 작업은 초당 수만 개의 작업을 예약하도록 쉽게 확장할 수 있습니다. 여기서 우리는

더 큰 AWS 리전. 운 좋게도, 그것은 소프트 한계이므로 기술적으로 우리는 그것을 무엇이든 올릴 수 있습니다. 우리는 필요합니다. 그러나 현재 한도에 대해 사용량을 지속적으로 모니터링하는 책임은 우리에게 있습니다. 제한되지 않도록 합니다.

열린 실행 횟수 제한에도 동일하게 적용됩니다. 동안

기본 한도인 1,000,000은 관대하지만 사용 수준을 계속 주시해야 합니다.

한도에 도달하면 기존 작업이 실행될 때까지 새 작업을 예약할 수 없습니다.

사용자 행동은 여기에 큰 영향을 미칩니다. 시간이 지남에 따라 작업이 더 균일하게 분산될수록 이 제한이 문제가 될 가능성이 줄어듭니다.

확장성 (핫스팟)

대규모 작업 클러스터가 동시에 실행되어야 하는 경우 초당 StateTransition에 대한 제한이 문제가 되기 때문에 이 솔루션에 4를 부여했습니다. 한계 때문에

모든 상태 전환에 적용되며 초기 대기 상태도 조절되어 영향을 미칠 수 있습니다.

새로운 작업을 예약하는 능력.

버킷 크기(버스트 제한으로 생각)와

초당 재충전 속도. 그러나 이러한 제한을 높이는 것만으로는 이를 확장하기에 충분하지 않을 수 있습니다.

1,000,000개의 작업으로 대규모 핫스팟을 지원하는 솔루션입니다. 다행히도 있습니다

대규모 핫스팟을 더 잘 처리할 수 있도록 이 솔루션을 조정할 수 있지만

약간의 정밀도를 절충해야 합니다(나중에 자세히 설명).

비용

Step Functions가 다음 중 하나이기 때문에 이 솔루션에 2를 주었습니다.

AWS에서 가장 비싼 서비스. 우리는 다음을 기준으로 청구됩니다.

상태 전이의 수. 대기하는 상태 머신의 경우

예약된 시간까지 작업을 실행하기까지 4개의

상태(그림 7.13 참조) 및 이들에 대한 모든 실행 비용

상태 전환 (<http://mng.bz/ZxGm>).

1,000개의 상태 전환당 \$0.025에서 1,000,000개의 작업을 예약하는 비용은 \$100에 작업 실행과 관련된 Lambda 비용을 더한 것입니다. 이것은 거의 두 주문

지금까지 고려한 다른 솔루션보다 훨씬 높습니다.

7.4.3 솔루션 조정

지금까지 이 솔루션과 관련된 몇 가지 문제에 대해 논의했습니다.

1년 이상 작업을 예약할 수 없고 핫스팟에 문제가 있습니다. 다행스럽게도 이러한 문제를 해결할 수 있는 간단한 수정 사항이 있습니다.

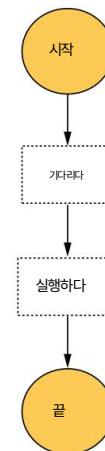


그림 7.13 작업을 실행하기 위해 예약된 시간까지 기다리는 단순 상태 머신

예정된 시간을 1년 이상으로 연장

상태 머신 실행이 실행될 수 있는 최대 시간

1년입니다. 따라서 Wait 상태가 기다릴 수 있는 최대 시간도 1년입니다. 그러나 꼬리 재귀의 아이디어를 빌려 이 제한을 확장할 수 있습니다 (<https://www.geeksforgeeks.org/tail-recursion/>). 함수형 프로그래밍에서, 기본적으로 Wait 상태가 끝나면 더 기다려야 하는지 확인할 수 있습니다.

그렇다면 상태 머신은 자체적으로 또 다른 실행을 시작하고 또 다른 1년을 기다리는 식으로 계속됩니다. 결국 우리는 작업의 예정된 시간에 도착하여 작업을 실행합니다.

이것은 첫 번째 exe가 실행되기 때문에 꼬리 재귀와 유사합니다.
cut은 재귀가 끝날 때까지 기다릴 필요가 없습니다.

단순히 두 번째 실행을 시작한 다음 자체 완료를 진행합니다. 이 수정된 상태 머신이 어떻게 보이는지 그림 7.14를 참조하십시오.

핫스팟을 위한 확장

때로는 초당 StateTransition 수에 대한 소프트 제한을 높이는 것만으로는 충분하지 않습니다. 기본 제한은 버킷 크기가 5,000(초기 버스트 제한)이고 재충전 속도가 초당 1,500이기 때문에 동시에 1,000,000개의 작업 실행을 지원하려면 이러한 제한을 여러 차수만큼 높여야 합니다. 둘. AWS는 그러한 요청을 이행하지 않을 것이며 Step Functions는 그러한 사용 사례를 위해 설계되지 않았음을 정중하게 상기할 것입니다.

다행히도 핫스팟을 처리할 때 솔루션을 훨씬 더 확장할 수 있도록 솔루션을 약간 조정할 수 있습니다. 불행히도, 우리는 새로 발견된 확장성을 위해 이 솔루션의 정밀도 중 일부를 절충해야 합니다.

예를 들어 00:00 UTC에 정확히 00:00 UTC에 예약된 모든 작업을 실행하는 대신 1분 창에 분산할 수 있습니다. 예정된 시간에 임의의 지연을 추가하여 이를 수행할 수 있습니다. 이 간단한 변경에 따라 앞서 언급한 작업 중 일부는 00:00:12 UTC에 실행되고 일부는 00:00:47 UTC에 실행됩니다. 이를 통해 put을 통해 사용 가능한 것을 최대한 활용할 수 있습니다. 기본 제한인 버킷 크기는 5,000개이고 재충전 속도는 초당 1,500개로 분당 최대 상태 전환 수는 93,500개입니다.

처음 1초 동안 5,000개의 모든 상태 전환을 사용합니다.

남은 59초 동안 초당 1,500회 리필 사용

이렇게 하면 정밀도가 "1분 이내에 실행"으로 줄어들지만 기본 제한을 거의 올릴 필요가 없습니다. 작업이 분 단위로 균일하게 분산되도록 예약된 시간에 다양한 지연 시간(0~59초)을 주입하는 것은 사소한 변경입니다. 이 간단한 조정으로 Step Functions는 더 이상

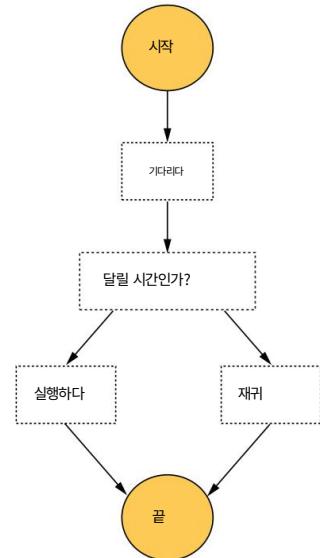


그림 7.14 1년 이상 남은 예정된 작업을 지원할 수 있는 수정된 상태 기계 설계

확장성 병목 현상. 대신, 우리는 요율 제한에 대해 걱정할 필요가 있습니다.
작업을 실행할 Lambda 함수.

또 다른 대안은 각 상태 머신 실행이 모든
동일한 분에 일괄 처리 및 병렬로 예약된 작업. 예를 들어,
작업을 예약할 때 DynamoDB 테이블에 예약된 시간이 있는 작업을 다음과 같이 추가합니다.
HASH 키와 RANGE 키로 고유한 작업 ID. 동시에 원자적으로
이 타임스탬프에 대해 예약된 작업 수에 대한 카운터를 증가시킵니다. 둘다
이러한 업데이트는 단일 DynamoDB 트랜잭션에서 수행할 수 있습니다. 그럼 7.15는
테이블이 어떻게 보일지.

| | timestamp ⓘ | sk | count | task |
|---|---------------------|---|------------------------|------|
| ☐ | 2020-07-04T21:53:22 | item_count | 2 | |
| ☐ | 2020-07-04T21:53:22 | item_3013a975-163e-45f6-89c0-357c161bb76c | { "message": "world" } | |
| ☐ | 2020-07-04T21:53:22 | item_3aa4e344-5faf-4fd7-934a-c11f653e5730 | { "message": "hello" } | |

그림 7.15 동일한 DynamoDB 테이블에서 예약된 작업과 개수를 설정합니다.

타임스탬프를 실행 이름으로 사용하여 상태 머신 실행을 시작합니다.
실행 이름은 고유해야 하므로 기존 실행이 있는 경우 StartExe 절단 요청
이 실패합니다.
이미. 이렇게 하면 해당 분에 예약된 모든 작업을 한 번만 실행할 수 있습니
다.
(2020-07-04T21:53:22).

예약된 작업을 즉시 실행하는 대신
대기 상태 후에는
실행해야 하는 작업. 여기에서 지도를 사용합니다.
실행할 병렬 분기를 동적으로 생성하는 상태
이러한 작업을 병렬로 수행합니다. 이 대안적인 디자인이 어떻게 보이는지 그
림 7.16을 참조하십시오.

이러한 변경은 정밀도에 영향을 미치지 않습니다.
너무 많이, 그러나 그것은 국가의 수를 줄일 것입니다
머신 실행 및 Lambda 호출이 필요합니다.
기본적으로 하나의 상태 머신 실행이 필요합니다.
예약된 일부를 실행해야 하는 1분마다
작업. 365일 calendar year에 총 525,600분이 있으므로 이를 통해 증
가할 필요가 없습니다.
열린 실행 횟수에 대한 제한(다시),
기본 제한은 1,000,000입니다. 이것이 이러한 구성 가능한 아키텍처
구성 요소의 아름다움입니다! 있기 때문에
그것들을 구성하는 많은 방법, 그것은 당신에게 많은 다른 것을 제공합니다
옵션과 절충안.

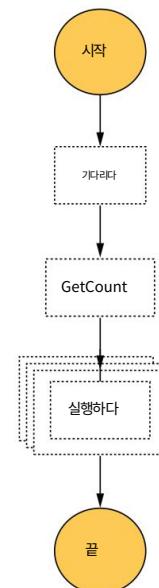


그림 7.16 일괄 작업을 병렬로 실행할
수 있는 상태 머신의 대안 설계

연습: 수정된 솔루션에 점수 매기기

우리가 제안한 수정된 솔루션에 대해 동일한 채점 연습을 반복합니다.

많은 수의 열린 데이터를 처리하는 시스템의 능력에 얼마나 영향을 미칠까요?

작업 또는 핫스팟?

추가 비용 영향이 있습니까(예: 제안된 조정 중 하나가

DynamoDB 테이블)을 고려해야 합니까?

7.4.4 최종 생각

Step Functions는 매우 정밀하게 작업을 실행할 수 있는 간단하고 우아한 솔루션을 제공합니다. 가장 큰 단점은 비용과 살펴봐야 하는 다양한 서비스 제한입니다.

확장성을 방해합니다. 하지만 보시다시피,

정밀도에 대한 절충안이 있는 경우 솔루션을 수정하여 훨씬 더 확장할 수 있습니다.

우리는 몇 가지 요소를 취하는 것을 포함하여 몇 가지 가능한 수정 사항을 살펴보았습니다.

크론 작업 솔루션의 특징을 파악하고 이 솔루션을 보다 유연한 크론 작업으로 전환합니다.

실행해야 하는 작업이 있을 때만 실행됩니다. 우리는 또한

상태에 고리 재귀를 적용하여 1년 제한을 해결할 수 있습니다.

기계 설계. 다음 솔루션에서는 SQS에 동일한 기술을 그대로 적용합니다.

작업이 열린 상태로 유지될 수 있는 기간에 대한 더욱 엄격한 제약에 의해 제한됩니다.

7.5 SQS

Amazon Simple Queue Service(SQS)는 완전 관리형 대기열 서비스입니다. 당신은 할 수 있습니다 큐에 메시지를 보내고 큐에서 메시지를 받습니다. 일단 메시지가

소비자가 수신한 메시지는 다음 기간 동안 다른 모든 소비자에게 숨겨집니다.

가시성 시간 초과로 알려진 기간입니다. 대기열에서 가시성 시간 초과 값을 구성할 수 있지만 개별 메시지에 대해 설정을 재정의할 수도 있습니다.

SQS에 메시지를 보낼 때 DelaySeconds 속성을 사용하여 다음을 수행할 수도 있습니다.

메시지가 적시에 보이도록 합니다. 스케줄링을 구현할 수 있습니다.

이 두 가지 설정을 사용하여 예약된 시간까지 메시지를 숨김으로써 서비스를 제공합니다. 하지만,

최대 DelaySeconds는 겨우 15분이고 최대 가시성 제한 시간은 12시간에 불과합니다. 그러나 모든 것이 손실된 것은 아닙니다.

실행 함수가 초기 DelaySeconds 후에 메시지를 수신하면

메시지를 검사하고 작업을 실행할 시간인지 확인할 수 있습니다(그림 7.17 참조). 그렇지 않은 경우

메시지에서 ChangeMessageVisibility 를 호출하여 메시지를 숨길 수 있습니다.

또 12시간 (<https://amzn.to/3e1GVY6>). 될 때까지 반복적으로 수행할 수 있습니다.

마지막으로 예약된 작업을 실행할 시간입니다.

이 솔루션에 점수를 매기기 전에 120,000개의 기내 메시지 제한이 있다는 점을 고려하십시오. 불행히도 이것은 하드 한도이며 올릴 수 없습니다. 이 제한은 일부 사용 사례에 전혀 적합하지 않다는 것을 의미할 수 있는 전문적인 의미를 가지고 있습니다!

메시지가 전송되면 이 솔루션은 지속적으로 메시지를 전송 상태로 유지합니다.

예정된 시간까지 가시성 제한 시간을 연장합니다. 이 경우, 수

인플라이트 메시지는 열려 있는 작업의 수와 같습니다. 그러나 일단 도달하면

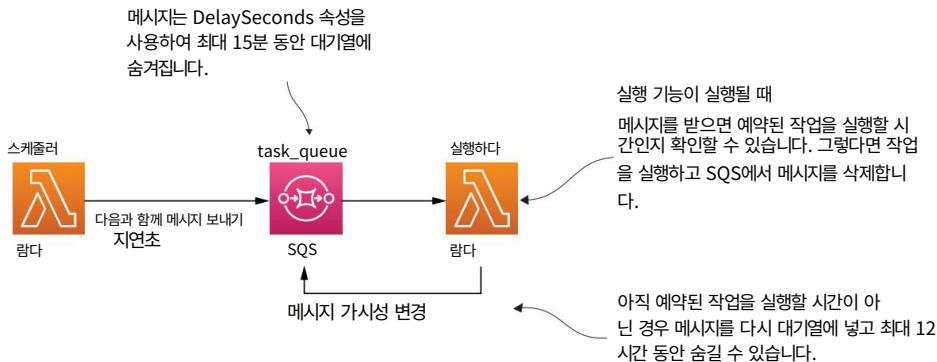


그림 7.17 SQS를 사용하여 임시 작업을 예약하는 상위 수준 아키텍처

전송 중인 메시지가 120,000개로 제한되면 일부 최신 메시지가 이미 전송 중인 메시지보다 더 빨리 실행해야 할 수 있는 경우에도 최신 메시지가 대기열의 백 로그에 남아 있게 됩니다. 실행 시간이 아니라 예약된 시간을 기준으로 작업에 우선 순위가 부여됩니다.

이것은 스케줄링 서비스에 바람직한 특성이 아닙니다. 사실, 그것은 우리가 원하는 것과 반대되는 사이트입니다. 곧 실행되도록 예약된 작업은 제시간에 실행되도록 우선 순위를 지정해야 합니다. 즉, 이것은 120,000개의 기내 메시지 제한에 도달한 경우에만 발생하는 문제입니다. 더 멀리 떨어져 있는 작업을 예약할 수 있을수록 더 많은 작업을 수행하게 되며 이 문제가 발생할 가능성도 높아집니다.

7.5.1 점수

이러한 심각한 제한을 염두에 두고 이 솔루션의 점수를 매길 수 있습니까? 표 7.7의 빈 칸에 점수를 기록하십시오.

표 7.7 SQS에 대한 솔루션 점수

| | 점수 |
|--------------|----|
| 정도 | |
| 확장성(미결 작업 수) | |
| 확장성(핫스팟) | |
| 비용 | |

7.5.2 우리의 점수

이 솔루션은 작업이 너무 먼 시간에 예약되지 않은 시나리오에 가장 적합합니다. 그렇지 않으면 많은 수의 미해결 작업이 누적되고 기내 메시지 제한에 부딪힐 가능성이 있습니다. 또한 오랫동안 12시간마다 메시지에 Change MessageVisibility 를 호출해야 합니다. 작업이 1년에 실행되도록 예약된 경우 총 730번입니다. 1,000,000을 곱한 값

이는 총 7억 3천만 개의 API 요청 또는 1,000,000개 작업을 1년 내내 열어 두는 데 292달러입니다. 이를 염두에 두고 표 7.8은 우리의 점수를 보여줍니다.

표 7.8 SQS에 대한 솔루션 점수

| | 점수 |
|--------------|----|
| 정도 | 9 |
| 확장성(미결 작업 수) | 2 |
| 확장성(핫스팟) | 8 |
| 비용 | 5 |

정도

정상적인 조건에서 지연되거나 숨겨진 SQS 메시지는 예정된 시간 이후 몇 초 이내에 실행됩니다. Step Functions 만큼 정확하지는 않지만 여전히 매우 좋습니다. 이것이 우리가 이 솔루션에 9점을 준 이유입니다.

확장성 (오픈 태스크 수)

120,000개의 인플라이트 메시지라는 엄격한 제한이 많은 미해결 작업을 지원하는 이 솔루션의 기능을 심각하게 제한하기 때문에 이 솔루션에 낮은 점수를 부여했습니다. 작업을 계속 예약할 수는 있지만 진행 중인 메시지 수가 120,000개 미만으로 떨어질 때까지 작업을 실행할 수 없습니다. 이것은 심각한 장애이며 최악의 경우 시스템을 완전히 사용할 수 없게 만들 수 있습니다. 예를 들어 120,000개의 작업이 1년 동안 실행되도록 예약된 경우 처음 120,000개의 작업이 실행될 때까지 그 이후에 예약된 다른 작업은 실행할 수 없습니다.

확장성 (핫스팟)

Lambda 서비스는 긴 폴링을 사용하여 SQS 대기열을 폴링하고 메시지가 있는 경우에만 기능을 호출합니다 (<http://mng.bz/Rqyj>). 이러한 폴리는 SQS와 우리 기능 사이에 보이지 않는 계층이며 우리는 이에 대해 비용을 지불하지 않습니다. 그러나 우리는 SQS ReceiveMessage 요청에 대한 비용을 지불합니다. Randall Hunt 의 이 블로그 게시물 (<https://amzn.to/31MfVtl>) 에 따르면

Lambda 서비스는 인플라이트 메시지 수를 모니터링하고 이 수가 증가하는 추세를 감지하면 폴링 빈도를 분당 20개의 ReceiveMessage 요청만큼 증가시키고 함수 동시성을 분당 60개의 호출만큼 증가시킵니다. 대기열이 계속 사용 중인 한 기능 동시성 제한에 도달할 때까지 계속 확장됩니다. 인플라이트 메시지 수가 감소함에 따라 Lambda는 폴링 빈도를 분당 10개의 ReceiveMessage 요청으로 줄이고 함수를 호출하는데 사용되는 동시성을 분당 30개의 호출로 줄입니다.

많은 수의 기내 메시지로 대기열을 인위적으로 바쁘게 유지함으로써 Lambda의 폴링 빈도와 기능 동시성을 인위적으로 높이고 있습니다. 이것은 핫스팟을 처리하는 데 유용합니다.

이 솔루션이 작동하는 방식 때문에 열려 있는 모든 작업은 진행 중인 메시지로 유지됩니다. 이는 Lambda 서비스가 많은 수의 동시 실행을 실행할 가능성이 있음을 의미합니다.

항상 투표소, 메시지 클러스터를 동시에 사용할 수 있게 되면 병렬 처리가 높은 실행 기능에 의해 처리될 가능성이 높습니다. 그리고 Lambda는 메시지가 많아질수록 동시 실행 수를 확장합니다. 사용 가능. 따라서 Lambda가 제공하는 자동 크기 조정 기능을 사용할 수 있습니다. 이 때문에 우리는 이 솔루션에 매우 좋은 점수를 주었습니다. 그러나 다른 한편으로는, 이 동작은 많은 종복 SQS ReceiveMessage 요청을 생성합니다. 대규모로 실행할 때 비용에 눈에 띠는 영향을 미칩니다.

비용

Lambda가 우리를 대신하여 수행 하는 많은 ReceiveMessage 요청과 12시간마다 모든 메시지에 대해 ChangeMessageVisibility 를 호출하는 비용, 대부분 이 솔루션의 비용은 SQS에 기인할 가능성이 높습니다. SQS는 100만 API 요청당 0.40달러로 비싼 서비스는 아니지만 비용이 빠르게 누적될 수 있습니다. 이 솔루션은 대규모로 수백만 건의 요청을 생성할 수 있습니다. 이와 같이 우리는 이 솔루션에 5를 주었습니다. 즉, 좋지는 않지만 사용자를 유발할 가능성이 낮습니다. 너무 많은 문제.

7.5.3 최종 생각

이 솔루션에 대한 점수를 DynamoDB TTL과 나란히 배치하면 다음을 확인할 수 있습니다.
그들은 서로를 완벽하게 보완합니다. 하나가 강한 곳에서 다른 하나는 약합니다.
표 7.9는 두 서비스에 대한 등급을 보여줍니다.

표 7.9 SQS와 DynamoDB TTL에 대한 평가

| | 점수(SQS) | 점수(DynamoDB TTL) |
|--------------|---------|------------------|
| 정도 | 9 | 1 |
| 확장성(미결 작업 수) | 2 | 10 |
| 확장성(핫스팟) | 8 | 6 |
| 비용 | 5 | 10 |

이 두 가지 솔루션을 결합하여 최고의 기능을 제공하는 솔루션을 만들 수 있다면 어떨까요?
두 세계? 다음으로 살펴보겠습니다.

7.6 DynamoDB TTL과 SQS 결합

지금까지 우리는 DynamoDB TTL 솔루션이 대규모 열려 있는 작업의 수이지만 대부분의 사용 사례에 필요한 정밀도가 부족합니다. 거꾸로, SQS 솔루션은 우수한 정밀도를 제공하고 핫스팟을 처리하는 데 탁월하지만 많은 수의 열린 작업을 처리할 수 없습니다. 둘은 오히려 서로를 보완한다. 결합하여 큰 효과를 얻을 수 있습니다.

예를 들어 장기 작업이 2일 전까지 DynamoDB에 저장되어 있으면 어떻게 될까요?
그들의 예정된 시간은? 왜 이틀? DynamoDB에 대한 유일한 소프트 보증이기 때문에 TTL은 다음을 제공합니다.

TTL은 일반적으로 만료 후 48시간 이내에 만료된 항목을 삭제합니다 (<https://amzn.to/2NRgARU>).

작업이 DynamoDB 테이블에서 삭제되면 SQS로 이동되어 예약된 시간까지 계속 유지됩니다(앞에서 설명한 ChangeMessageVisibility API 사용). 2일 이내에 실행되도록 예약된 작업의 경우 즉시 SQS 대기열에 추가됩니다. 이 솔루션이 어떻게 보이는지 그림 7.18을 참조하십시오.

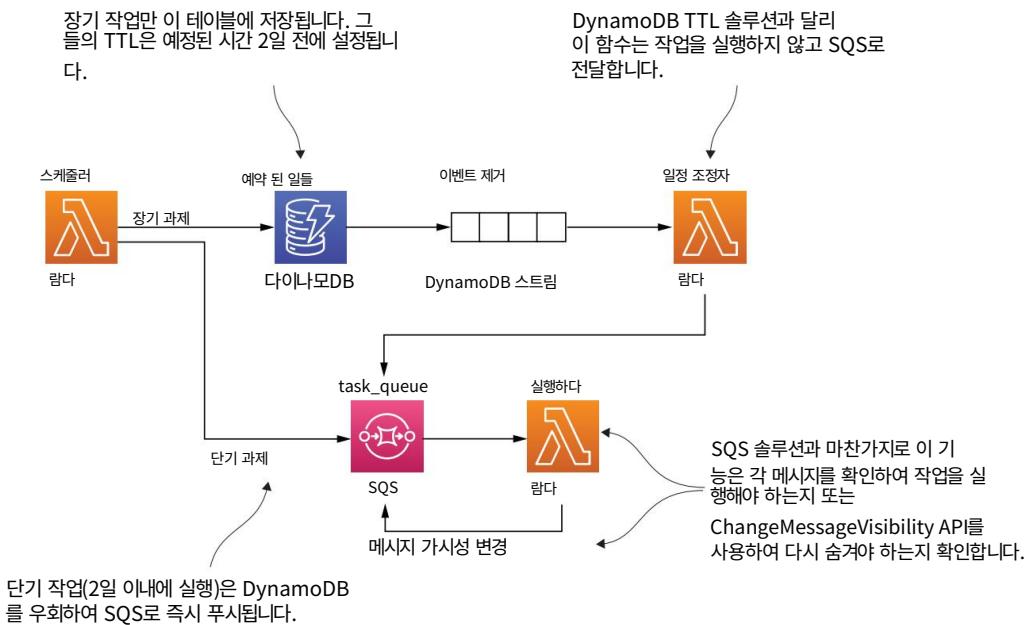


그림 7.18 DynamoDB TTL과 SQS를 결합하는 상위 수준 아키텍처

7.6.1 점수

이 솔루션을 어떻게 평가하시겠습니까? 다시, 표 7.10의 빈 칸에 점수를 쓰십시오.

표 7.10 SQS가 포함된 DynamoDB TTL에 대한 솔루션 점수

| | 점수 |
|--------------|----|
| 정도 | |
| 확장성(미결 작업 수) | |
| 확장성(핫스팟) | |
| 비용 | |

7.6.2 우리의 점수

"소프트웨어 엔지니어링의 기본 정리" (<https://dzone.com/articles/why-fundamental-theorem>)에 따르면:

추가 수준의 간접 참조를 도입하여 모든 문제를 해결할 수 있습니다.

이 장의 앞부분에서 본 다른 대안 솔루션과 마찬가지로 이 솔루션은 추가 수준의 간접 참조를 도입하여 기존 솔루션의 문제를 해결합니다. 서로 다른 서비스를 함께 구성하여 각각의 부족함을 만회함으로써 그렇게 합니다.

이 솔루션에 대한 점수에 대해 표 7.11을 살펴보고 이러한 점수에 도달한 방법에 대해 논의할 것입니다.

표 7.11 SQS가 포함된 DynamoDB TTL에 대한 솔루션 점수

| | 점수 |
|--------------|----|
| 정도 | 9 |
| 확장성(미결 작업 수) | 8 |
| 확장성(핫스팟) | 8 |
| 비용 | 7 |

정도

모든 실행이 SQS를 거치므로 이 솔루션은 SQS 전용 솔루션인 9와 동일한 수준의 정밀도를 갖습니다.

확장성 (오픈 태스크 수)

DynamoDB에 장기 작업을 저장하면 열린 작업 수를 확장하는 SQS의 문제가 크게 해결됩니다. 그러나 단기 작업만으로도 120,000개의 기내 메시지 제한에 도달할 수 있습니다. 가능성은 훨씬 낮지만 여전히 고려해야 할 가능성입니다. 따라서 이 솔루션을 8로 표시했습니다.

확장성 (핫스팟)

모든 실행이 SQS를 거치므로 이 솔루션의 점수는 SQS 전용 솔루션인 8과 동일합니다.

비용

이 솔루션은 모든 장기 작업이 DynamoDB에 저장되기 때문에 대부분의 ChangeMessageVisibility 요청을 제거합니다. 이것은 SQS 솔루션과 관련된 비용의 상당 부분을 줄입니다. 그러나 그 대가로 DynamoDB 사용 및 일정 변경 기능에 대한 Lambda 호출에 대한 추가 비용이 추가됩니다. 전반적으로 이 솔루션이 제거하는 비용은 추가되는 새로운 비용보다 큽니다. 따라서 SQS 솔루션에 대한 원래 점수 5점에서 개선된 7점을 주었습니다.

7.6.3 최종 생각

이것은 다른 솔루션이나 솔루션의 측면을 결합하여 보다 효과적인 답변을 만드는 방법의 한 예일 뿐입니다.
이 조합 효과는 다음 중 하나입니다.

이는 클라우드 아키텍처를 흥미롭고 매력적으로 만들면서도 동시에 너무 복잡하게 만듭니다.

그리고 때때로 혼란스럽습니다. 같은 목표를 달성하는 방법은 다양하지만,

응용 프로그램에 필요한 사항에 따라 일반적으로 모든 응용 프로그램에 최상의 결과를 제공하는 만능 솔루션은 없습니다.

지금까지 우리는 방정식의 공급 측면과 각 솔루션이 제공할 수 있는 것을 살펴보았습니다. 우리는 아직 수요 측면이나 응용 프로그램에 필요한 사항을 살펴보지 않았습니다. 예

뭐라고요. 결국 용도에 따라 다른 가중치를 두는 경우가 있기 때문에

각각의 비기능적 요구사항 뒤에 있습니다. 우리의 솔루션을 다음과 일치시키도록 노력합시다.

다음은 올바른 응용 프로그램입니다.

7.7 애플리케이션에 적합한 솔루션 선택

표 7.12는 이 장에서 고려한 5가지 솔루션에 대한 점수를 보여줍니다.

이 표의 솔루션에는 제안된 조정이 포함되어 있지 않습니다. 예

응용 프로그램에서 이러한 요구 사항 중 일부는 다른 요구 사항보다 더 중요할 수 있습니다.

표 7.12 다섯 가지 솔루션 모두에 대한 점수

| | 크론 작업 | 다이나모DB TTL | 단계 기능 | SQS | SQS + 다이나모DB TTL |
|--------------|-------|------------|-------|-----|---------------------|
| 정도 | 6 | 1 | 10 | 9 | 9 |
| 확장성(미결 작업 수) | 10 | 10 | 7 | 2 | 8 |
| 확장성(핫스팟) | 2 | 6 | 4 | 8 | 8 |
| 비용 | 7 | 10 | 2 | 5 | 7 |

7.8 애플리케이션

Ad Hoc 스케줄링 서비스를 사용할 수 있는 세 가지 애플리케이션을 살펴보겠습니다.

응용 프로그램 1은 알림 앱으로, 이것을 RemindMe라고 합니다.

Application 2는 모바일 게임을 위한 멀티플레이어 앱으로 Tourna라고 부를 것입니다.
멘츠RU.

애플리케이션 3은 귀하의 동의를 디지털화하여 관리하는 헬스케어 앱입니다.

귀하의 의료 데이터를 iConsent라고 하는 의료 제공자와 공유합니다.

미리 알림 앱인 RemindMe에서 사용자는 향후 이벤트에 대한 미리 알림을 생성할 수 있으며,
시스템은 이벤트 10분 전에 사용자에게 SMS/푸시 알림을 보냅니다.

미리 알림은 일반적으로 시간이 지남에 따라 균등하게 배포되지만 주변에 핫스팟이 있습니다.
공휴일 및 슈퍼볼과 같은 주요 스포츠 행사. 이 동안

핫스팟의 경우 애플리케이션이 수백만 명의 사용자에게 알려야 할 수 있습니다. 다행히도

리마인더는 이벤트 10분 전에 전송되며 시스템은 타이밍 면에서 약간의 여유를 줍니다.

애플리케이션 2, TournamentsRUs라는 멀티 플레이어 모바일 앱에서 플레이어는 15-30분 길이의 사용자 생성 토너먼트에서 경쟁합니다. 종료 휴들이 울리면 토너먼트가 종료되고 모든 참가자는 게임 내 팝업을 통해 승자가 발표되기를 기다립니다. TournamentsRUs는 현재 150만 명의 일일 활성 사용자(DAU)를 보유하고 있으며 12개월 내에 이 숫자를 두 배로 늘리기를 희망합니다. 최대 동시 사용자 수는 DAU의 약 5%이며 토너먼트는 일반적으로 각각 10-15명의 플레이어로 구성됩니다.

애플리케이션 3인 의료 앱 iConsent에서 사용자는 의료 제공자가 의료 데이터에 액세스할 수 있는 디지털 양식을 작성합니다. 각 동의에는 만료 날짜가 있으며 만료 날짜가 지나면 앱은 상태를 만료됨으로 변경해야 합니다. iConsent는 현재 수백만 명의 등록 사용자를 보유하고 있으며 사용자는 평균 3번의 동의를 받았습니다.

이러한 각 응용 프로그램은 특정 시간에 임시 작업을 실행하기 위해 일정 서비스를 사용해야 하지만 사용 사례는 크게 다릅니다. 일부는 수명이 짧은 작업을 처리하는 반면 다른 일부는 작업이 미래의 특정 시점으로 예약되도록 허용합니다. 일부는 실제 이벤트 주변에 핫스팟이 발생하는 경향이 있습니다. 다른 사람들은 작업을 예약할 수 있는 거리에 제한이 없기 때문에 많은 수의 미해결 작업을 누적할 수 있습니다.

어떤 솔루션이 각 애플리케이션에 가장 적합한지 더 잘 이해할 수 있도록 각 비기능 요구 사항에 가치를 적용할 수 있습니다. 예를 들어 TournamentsRU는 사용자가 토너먼트가 끝나면 결과를 기다리기 때문에 정밀도에 많은 관심을 기울입니다. 토너먼트를 마무리하는 작업이 지연되면 앱에 대한 사용자 경험에 부정적인 영향을 미칠 수 있습니다.

7.8.1 당신의 무게

각 응용 프로그램에 대해 표 7.13의 각 비기능 요구 사항에 대해 1("관심 없음")에서 10("이것은 중요") 사이의 가중치를 작성하십시오. 여기에는 옳고 그른 답이 없다는 것을 기억하십시오! 각 앱에 대해 알고 있는 제한된 정보를 바탕으로 최선의 판단을 내리세요.

표 7.13 RemindMe, TournamentsRU 및 iConsent에 대한 귀하의 평가

| | 미리 알림 | 토너먼트RU | 동의 |
|--------------|-------|--------|----|
| 정도 | | | |
| 확장성(미결 작업 수) | | | |
| 확장성(핫스팟) | | | |
| 비용 | | | |

7.8.2 우리의 가중치

표 7.14는 가중치를 보여줍니다. 이 점수가 당신과 비슷합니까? 각 응용 프로그램을 살펴보고 표 다음 섹션에서 이러한 가중치에 도달한 방법에 대해 설명합니다.

표 7.14 RemindMe, TournamentsRU 및 iConsent에 대한 점수

| | 미리 알림 | 토너먼트RU | 동의 |
|-----------------|-------|--------|----|
| 정도 | 5 | 10 | 4 |
| 확장성(미결 작업 수) 10 | | 6 | 10 |
| 확장성(핫스팟) | 8 | 삼 | 1 |
| 비용 | 삼 | 삼 | 삼 |

알림

미리 알림이 10분 동안 전송되기 때문에 이 앱에 대해 Precision에 5의 가중치를 부여했습니다. 이벤트 전에. 이것은 우리에게 많은 여유를 줍니다. 5분동안 알림을 보내도 늦었지만 아직 괜찮아.

확장성(열린 작업의 수)은 상위 항목이 없기 때문에 10의 가중치를 얻습니다. 이벤트를 얼마나 멀리 예약할 수 있는지에 대한 경계입니다. 언제든지 수백만 개의 미리 알림이 열려 있을 수 있습니다. 이렇게 하면 열려 있는 작업의 수를 절대적으로 확장할 수 있습니다. 이 응용 프로그램에 대한 중요한 요구 사항입니다. 확장성(핫스팟)에 대해 가중치를 주었습니다. 8 대규모 핫스팟이 공휴일 전후에 형성될 가능성이 높기 때문입니다(예: 어머니의 날) 및 스포츠 이벤트(예: 슈퍼볼 또는 올림픽).

마지막으로 Cost의 경우 가중치를 3으로 지정했습니다. 이는 아마도 우리의 일반적인 태도를 반영하는 것일 수 있습니다. 서비스 기술 비용으로 종량제 가격은 우리의 비용을 확장할 때 선형적으로 성장합니다. 일반적으로 비용 최적화를 원하지 않습니다. 해결책이 집을 불태우지 않는 한!

토너먼트

TournamentsRU의 경우 토너먼트가 종료되면 플레이어가 모두 승자의 발표를 기다리기 때문에 Precision은 가중치 10을 얻습니다. 예정된 경우 (토너먼트를 마무리하고 승자를 계산하기 위해) 작업이 몇 분 동안 지연됩니다. 초, 그것은 나쁜 사용자 경험을 제공합니다.

DAU의 적은 비율만이 한 번에 온라인 상태이고 토너먼트 기간이 짧기 때문에 확장성(진행 중인 작업 수)에 가중치를 6으로 지정했습니다. 150만 DAU에서 최대 동시 사용자 수는 5%이고 평균 10-15명의 플레이어는

각 토너먼트에서 이 숫자는 피크 시간에 약 5,000-7,500개의 오픈 투어 이름으로 변환됩니다.

확장성(핫스팟)의 경우 토너먼트가 사용자 생성이고 시간 길이(15-30분 사이)가 다르기 때문에 낮은 3을 받았습니다. 이를 법하지 않다

이러한 조건에서 큰 핫스팟이 형성됩니다. 그리고, 리마인드미와 마찬가지로 우리는 무게는 3입니다(집을 태우지 마십시오!).

동의

마지막으로 iConsent의 경우 Precision은 4의 가중치를 받았습니다. 동의가 만료되면 올바른 상태로 UI에 표시되어야 합니다. 그러나 사용자는 업데이트를 위해 몇 분마다 앱을 확인하지 않을 가능성이 높으므로 상태가 다음과 같으면 괜찮습니다. 몇 분(또는 한 시간) 후에 업데이트됩니다.

확장성(열린 작업의 수)에 대해 10의 가중치를 주었습니다. 이는 의료 동의가 1년 또는 그보다 더 오래 유효할 수 있기 때문입니다. 이러한 모든 활성 동의는 공개 작업이므로 시스템에는 언제든지 수백만 개의 공개 작업이 있습니다.

반면에 확장성(핫스팟)의 경우 가중치를 1로 지정했습니다.

핫스팟으로 이어질 수 있는 자연스러운 클러스터링이 없습니다. 마지막으로 비용의 가중치는 3입니다.

그것이 우리가 일반적으로 서비스 애플리케이션의 비용에 대해 느끼는 방식이기 때문입니다.

7.8.3 각 애플리케이션에 대한 솔루션 채점

지금까지 우리는 자체 장점을 기반으로 각 솔루션에 점수를 매겼습니다. 그러나 이것은 아무것도 말하지 않는다 솔루션이 애플리케이션에 얼마나 적합한지 살펴보았듯이 애플리케이션은

다른 요구 사항. 솔루션의 점수와 애플리케이션의 가중치를 결합하여

우리는 그들이 각각에 얼마나 잘 맞는지에 대한 지표 점수에 도달할 수 있습니다.

다른. 이 작업을 수행할 수 있는 방법을 보여주고 이 운동을 직접 수행할 수 있습니다.

다음 표는 cron 작업 솔루션에 대한 점수를 보여줍니다.

| | 점수(크론 작업) |
|--------------|-----------|
| 정도 | 6 |
| 확장성(미결 작업 수) | 10 |
| 확장성(핫스팟) | 2 |
| 비용 | 7 |

알림에 대해 다음 가중치를 부여했습니다.

| | 무게(알림) |
|--------------|--------|
| 정도 | 5 |
| 확장성(미결 작업 수) | 10 |
| 확장성(핫스팟) | 8 |
| 비용 | 삼 |

이제 각 비기능 요구 사항에 대한 가중치와 점수를 곱하면 다음 표의 점수에 도달합니다.

| | 가중 점수(Cron 작업 × 알림미) |
|--------------|----------------------|
| 정도 | $6 \times 5 = 30$ |
| 확장성(미결 작업 수) | $10 \times 10 = 100$ |
| 확장성(핫스팟) | $2 \times 8 = 16$ |
| 비용 | $7 \times 3 = 21$ |

총합계는 $30 + 100 + 16 + 21 = 167$ 이 됩니다. 그 자체로 이 점수는 다음을 의미합니다.

아주 작은. 그러나 연습을 반복하고 각 솔루션에 RemindMe에 대한 점수를 매기면

그리면 서로 얼마나 잘 비교되는지 알 수 있습니다. 이것은 우리가 선택하는 데 도움이 될 것입니다
리마인드미에 가장 적합한 솔루션, 솔루션과 다를 수 있음

TournamentsRU 또는 iConsent에 사용할 것입니다.

이를 염두에 두고 표 7.15의 공백을 사용하여 가중치를 계산하십시오.

지금까지 알림에 대해 논의한 5가지 솔루션 각각에 대한 점수입니다. 그 다음에

표 7.16 및 7.17의 TournamentsRU 및 iConsent에 대해 각각 동일한 작업을 수행합니다.

표 7.15 앱 알림에 대한 가중 점수

| | 크론 작업 | 다이나모DB TTL | 단계 기능 | SQS | SQS + 다이나모DB TTL |
|--------------|----------|---------------|----------|-----|---------------------|
| 정도 | | | | | |
| 확장성(미결 작업 수) | | | | | |
| 확장성(핫스팟) | | | | | |
| 비용 | | | | | |
| 총 점수 | | | | | |

표 7.16 앱 TournamentsRU에 대한 가중 점수

| | 크론 작업 | 다이나모DB TTL | 단계 기능 | SQS | SQS + 다이나모DB TTL |
|--------------|----------|---------------|----------|-----|---------------------|
| 정도 | | | | | |
| 확장성(미결 작업 수) | | | | | |
| 확장성(핫스팟) | | | | | |
| 비용 | | | | | |
| 총 점수 | | | | | |

표 7.17 앱 iConsent에 대한 가중 점수

| | 크론 작업 | 다이나모DB TTL | 단계 기능 | SQS | SQS + 다이나모DB TTL |
|--------------|----------|---------------|----------|-----|---------------------|
| 정도 | | | | | |
| 확장성(미결 작업 수) | | | | | |
| 확장성(핫스팟) | | | | | |
| 비용 | | | | | |
| 총 점수 | | | | | |

점수가 각 애플리케이션에 가장 적합한 솔루션에 대한 직관과 일치했습니까? 그 과정에서 뜻밖의 것을 발견 하셨나요? 일부 솔루션이 효과가 없었습니까?

당신이 생각했던 것만큼?

이러한 운동의 결과로 불편한 결과를 발견했다면,
그들의 일을 했습니다. 요구 사항을 미리 정의하고
각 요구 사항에 대한 가중치는 우리가 객관적인 상태를 유지하고 인지 능력과 싸우는 데 도움이 되는 것입니다.
편견. 표 7.18은 각 솔루션 및 애플리케이션에 대한 총 가중치 점수를 보여줍니다.

표 7.18 RemindMe, TournamentsRU 및 iConsent에 대한 총 가중치 점수

| | 미리 알림 | 토너먼트RU | 동의 |
|-----------------------|-------|--------|-----|
| 크론 작업 | 167 | 147 | 147 |
| 다이나모DB TTL | 180 | 115 | 137 |
| 단계 함수 | 158 | 160 | 120 |
| SQS | 144 | 141 | 79 |
| SQS가 포함된 DynamoDB TTL | 210 | 183 | 145 |

이 점수를 통해 각 애플리케이션에 가장 적합한 솔루션을 파악할 수 있습니다. 그러나 그들은 당신에게 확실한 답을 주지 않으며 맹목적으로 따라해서는 안됩니다.

예를 들어, 종종 점수 체계에 포함되지 않은 요소가 있지만

그럼에도 불구하고 고려해야 합니다. 복잡성, 리소스 제약 및 기술에 대한 친숙도와 같은 요소는 일반적으로 실제 환경에서 중요합니다.

프로젝트.

요약

이 장에서는 광고를 실행하는 서비스를 구현하는 다섯 가지 방법을 분석했습니다.

우리는 우리가 설정한 비기능적 요구 사항에 따라 이러한 솔루션을 판단했습니다.

챕터 시작 부분에 나옵니다. 챕터 전반에 걸쳐 우리는 여러분에게 비판적으로 생각하도록 요청했습니다.

이러한 비기능적 요구 사항에 대해 각 솔루션이 얼마나 잘 수행되는지에 대해

그리고 평가해달라고 했습니다. 그리고 우리는 당신과 우리의 근거를 공유했습니다.

이 점수. 이 연습을 통해 다음 사항에 대한 통찰력을 얻으셨기를 바랍니다.

문제 해결에 접근하는 방법과 평가에 필요한 고려 사항

잠재적인 솔루션:

관련 서비스 제한은 무엇이며 확장성에 어떤 영향을 줍니까?

응용 프로그램의 요구 사항?

해당 서비스의 성능 특성은 무엇이며 수행하는 작업은 무엇입니까?

응용 프로그램의 요구 사항과 일치합니까?

서비스 요금은 어떻게 되나요? 생각하여 응용 프로그램 비용을 예측하십시오.

애플리케이션이 이러한 AWS 서비스를 사용하는 방법과 해당 사용 패턴에 서비스의 청구 모델을 적용하는 방법을 통해.

이 요점은 말보다 훨씬 쉽고 능숙해지기 위해서는 연습이 필요합니다. AWS 서비스는 항상 진화하고 새로운 서비스와 기능을 항상 사용할 수 있으므로 새로운 옵션과 기술이 등장할 때마다 자신을 지속적으로 교육해야 합니다. 10년 넘게 AWS와 함께 일했음에도 불구하고 우리는 여전히 스스로를 배우고 있으며 다양한 AWS 서비스가 작동하는 방식에 대한 이해를 지속적으로 업데이트해야 합니다.

또한 AWS는 많은 서비스에 대한 성능 특성을 게시하지 않습니다. 예를 들어, Step Functions는 특정 태임스탬프 이후에 대기 상태를 얼마나 빨리 실행합니까? 솔루션이 이러한 알 수 없는 성능 특성에 대한 가정에 의존하는 경우 가정을 테스트하기 위해 소규모 실험을 설계해야 합니다. 이 장을 작성하는 동안 Step Functions 와 SQS가 지연된 작업을 얼마나 빨리 실행하는지 알아보기 위해 몇 가지 실험을 수행했습니다. 이러한 가정을 조기에 검증하지 못하면 치명적인 결과를 초래할 수 있습니다. 전체 솔루션이 잘못된 가정을 기반으로 구축된 경우 수개월의 엔지니어링 작업이 낭비될 수 있습니다.

장의 끝에서 우리는 또한 주어진 문제에 대한 최상의 솔루션을 찾는 연습을 하도록 요청했고 각각 다른 요구 사항을 가진 세 가지 예제 응용 프로그램을 제공했습니다. 저희가 귀하에게 적용하도록 요청한 점수 매기기 방법은 확실한 것은 아니지만 객관적인 결정을 내리고 확증 편향을 퇴치하는 데 도움이 됩니다.

이 장에서 제시한 솔루션을 브레인스토밍하고 평가하면서 여기에서 훨씬 더 중요한 교훈을 얻었기를 바랍니다. 모든 아키텍처 결정은 트레이드오프를 상속하며 모든 애플리케이션 요구 사항이 동일하게 생성되는 것은 아닙니다. 서로 다른 애플리케이션이 아키텍처의 특성에 대해 서로 다른 정도로 관심을 갖는다는 사실은 우리에게 현명한 절충안을 만들 여지를 많이 줍니다.

선택할 수 있는 다양한 AWS 서비스가 있으며, 각각은 서로 다른 특성과 절충안을 제공합니다. 서로 다른 서비스를 함께 사용하면 종종 흥미로운 시너지 효과를 낼 수 있습니다. 이 모든 것이 서로 다른 아키텍처 접근 방식을 혼합하고 일치시키고 창의적인 문제 해결 프로세스에 참여할 수 있는 풍부한 옵션을 제공합니다. 정말 아름답습니다!

서버리스 병렬 컴퓨팅 설계

이 장에서는 다음을 다룹니다.

맵리듀스의 원리

Step Functions 및 EFS가 포함된 서비스 솔루션

AWS Lambda에 대해 사람들에게 말하고 싶은 비밀이 있습니다. 바로 가장 큰 EC2 인스턴스보다 빠르게 수행할 수 있는 슈퍼컴퓨터입니다. 비결은 병렬 계산의 관점에서 Lambda를 생각하는 것입니다. 문제를 수백 또는 수천 개의 작은 문제로 나누어 병렬로 해결할 수 있다면 동일한 문제를 순차적으로 진행하여 해결하려고 하는 것보다 더 빨리 결과를 얻을 수 있습니다.

병렬 컴퓨팅은 컴퓨터 과학에서 중요한 주제이며 학부 컴퓨터 과학 커리큘럼에서 자주 거론됩니다. 흥미롭게도 람다는 본질적으로 병렬 컴퓨팅에서 개념을 생각하고 적용하도록 합니다. Step Functions 및 DynamoDB와 같은 서비스를 사용하면 병렬 애플리케이션을 더 쉽게 구축할 수 있습니다.

이 장에서는 서비스 비디오 트랜스코더를 구축하는 방법을 설명합니다.
더 크고 더 비싼 EC2 서버를 능가하는 Lambda.

8.1 맵리듀스 소개

MapReduce는 자주 사용되는 유명하고 잘 알려진 프로그래밍 모델입니다. 큰 데이터 세트를 처리합니다. 원래 Google에서 만든 개발자는 잘 알려진 두 가지 함수형 프로그래밍 기본 요소(고차 함수)인 map과 reduce에서 영감을 받았습니다. MapReduce는 큰 데이터 세트를 분할하여 작동합니다.

여러 개의 더 작은 하위 집합으로 나누어 각 하위 집합에 대해 작업을 수행한 다음 결합하거나 합산하여 결과를 얻습니다.

툴스토이의 전쟁과 평화에서 등장인물의 이름이 몇 번이나 언급되는지 알고 싶다고 상상해 보십시오. 모든 페이지를 순차적으로 볼 수 있습니다.

하나씩 계산하고 발생 횟수를 계산합니다(그러나 느림). 맵리듀스를 적용하면 그러나 다음과 같이 훨씬 더 빨리 할 수 있습니다.

1. 데이터를 여러 개의 독립적인 하위 집합으로 분할합니다. 전쟁과 평화의 경우, 개별 페이지 또는 단락이 될 수 있습니다.
2. 각 부분 집합에 지도 기능을 적용합니다. 이 경우 지도 기능은 스캔 페이지(또는 단락) 및 문자 이름이 몇 번인지 방출합니다. 말하는.
3. 여기에 일부 데이터를 결합하는 선택적 단계가 있을 수 있습니다. 그 수 다음 단계에서 계산을 조금 더 쉽게 수행할 수 있도록 도와줍니다.
4. reduce 함수는 요약 연산을 수행합니다. 의 수를 계산합니다.
맵 함수가 캐릭터의 이름을 내보내고 전체 결과.

[참고](#) 전쟁에서 MapReduce의 위력을 이해하는 것이 중요합니다. 그리고 평화의 예는 지도 단계가 병렬로 실행될 수 있다는 사실에서 비롯됩니다. 수천 페이지 또는 단락에. 이것이 사실이 아니라면 이 프로그램은 순차 카운트와 다르지 않을 것입니다.

그림 8.1은 이론적 MapReduce 아키텍처가 어떻게 생겼는지 보여줍니다. 우리는 구축 할 것입니다 곧 이런 것.

이미 짐작하셨겠지만 실제 MapReduce 애플리케이션은 종종 더 복잡한. 많은 경우 맵과 리듀스 사이에 중간 단계가 있습니다. 데이터를 결합하거나 단순화하고 데이터의 지역성과 같은 고려 사항이 오버헤드를 최소화하기 위해 중요합니다. 그럼에도 불구하고 우리는 문제를 더 작은 덩어리로 나누고 병렬로 처리한 다음 결합하는 아이디어를 취할 수 있습니다. 필요한 결과를 달성하기 위해 이를 줄입니다. 랍다.

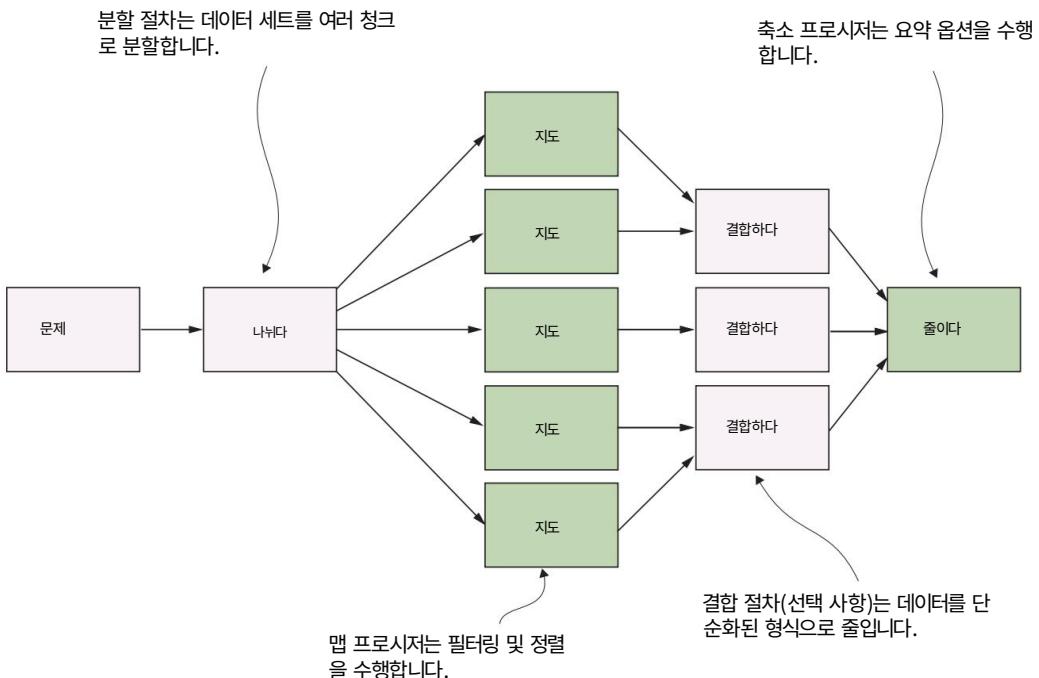


그림 8.1 이것은 가상의 MapReduce 알고리즘이 취할 수 있는 단계입니다.

8.1.1 비디오를 트랜스코딩하는 방법

이 책의 두 번째 장에서는 비디오를 한 형식에서 다른 형식으로 변환하는 서비스 파이프라인을 구축했습니다. 이를 위해 AWS Elemental MediaConvert라는 AWS 서비스를 사용했습니다. 이 서비스는 S3 버킷에 업로드된 비디오를 가져와 사용자가 지정한 다양한 형식으로 트랜스코딩합니다. 이 장의 목표는 미친 짓을 하고 Lambda를 사용하여 자체 비디오 트랜스코더 서비스를 구현하는 것입니다.

이것은 단지 실험일 뿐이며 고도로 병렬화된 서비스 아키텍처를 탐색할 기회입니다. 서비스 인코딩 서비스에 대한 주요 요구 사항은 다음과 같습니다.

비디오 파일을 가져와서 다른 형식이나 비트 전송률로 트랜스코딩된 버전을 생성하는 트랜스코더를 빌드합니다. 우리는 트랜스코딩을 완벽하게 제어하기를 원합니다
프로세스.

Lambda 및 S3와 같은 AWS의 서비스 서비스만 사용하십시오. 분명히 우리는 EC2 또는 관리형 서비스를 사용하여 트랜스코딩을 수행할 수 없습니다.

강력하고 빠른 제품을 만드십시오. 대부분의 경우 대규모 EC2 인스턴스를 능가할 수 있어야 합니다.

병렬 컴퓨팅과 이러한 문제를 해결하는 방법에 대해 알아봅니다.

우리가 제시하려는 솔루션은 작동하지만 프로덕션 환경에서 실행하는 것이 좋습니다. 핵심 비즈니스가 비디오 트랜스코딩이 아니라면

차별화되지 않은 무거운 짐을 가능한 한 많이 아웃소싱해야 합니다.
 자신의 고유한 문제에 집중할 수 있습니다. 대부분의 경우 다음과 같은 관리형 서비스 AWS Elemental MediaConvert가 더 좋습니다. 계속 실행하는 것에 대해 걱정할 필요가 없습니다. MapReduce 및 병렬 계산(언제 큰 문제에 직면할지 알 수 없습니다.
 여기에서 선택할 수 있는 기술이 필요합니다.)

Lambda에서 비디오 트랜스코딩을 어떻게 하시겠습니까?

잠시 시간을 내어 AWS를 사용하여 비디오 트랜스코더를 구축하는 방법에 대해 생각해 보십시오. 람다. 모든 추측이 좋습니다. 귀하가 어떻게 접근할 것인지 알고 싶습니다. 문제 (twitter.com/sbarski). 나머지 부분을 읽지 않고 직접 구축할 수 있습니까? 장?

8.1.2 아키텍처 개요

Lambda를 사용하여 파일을 트랜스코딩하려면 MapReduce의 원칙을 적용해야 합니다. 우리는 여기에 고전적인 MapReduce를 구현하지 않습니다. 대신, 우리는 이 알고리즘을 사용하여 트랜스코더를 구축할 수 있습니다.

Lambda에 대한 흥미로운 점(이전에 언급한)

우리는 병렬로 생각합니다. 다음과 같은 경우 Lambda 함수에서 대용량 비디오 파일을 처리할 수 없습니다. 단일 기능을 기준 컴퓨터처럼 취급합니다. 메모리가 부족하고 빨리 시간 초과. 비디오 파일이 충분히 크면 함수는 15분 동안 처리하거나 사용 가능한 모든 RAM을 소진하고 충돌합니다.

이 문제를 해결하기 위해 문제를 다음과 같은 더 작은 문제로 분해합니다.

Lambda의 제약 조건 내에서 처리됩니다. 의미는 큰 비디오 파일을 처리하기 위해 다음을 시도할 수 있다는 것입니다.

1. 비디오 파일을 많은 작은 부분으로 나눕니다.
2. 이 세그먼트를 병렬로 트랜스코딩합니다.
3. 이 작은 세그먼트를 함께 결합하여 새 비디오 파일을 생성합니다.

서버리스 슈퍼 컴퓨터를 최대한 활용하려면 최대한 병렬화해야 합니다. 예를 들어, 일부 세그먼트는 결합할 준비가 되어 있고 다른 세그먼트는 아직 처리 중이므로 준비된 것을 결합해야 합니다. 성능은 이름입니다. 여기 게임의. 따라서 이를 염두에 두고 비디오를 1비트 전송률에서

또 다른:

1. 사용자가 Lambda 함수를 호출하는 비디오 파일을 S3에 업로드합니다.
 2. Lambda 함수는 파일을 분석하고 소스를 잘라내는 방법을 알아냅니다. 파일을 변환하여 트랜스코딩을 위한 더 작은 비디오 파일(세그먼트)을 생성합니다.
 3. 작업을 조금 더 빠르게 진행하기 위해 비디오에서 오디오를 제거하고 저장합니다. 다른 파일로. 오디오에 대해 걱정할 필요 없이 단계를 실행합니다.
- 4-6 더 빠릅니다.

4. 이 단계는 작은 비디오 세그먼트를 생성하는 분할 프로세스를 수행합니다.
트랜스코딩.
5. 이제 세그먼트를 원하는 형식 또는
비트 레이트. 시스템은 이러한 세그먼트를 병렬로 트랜스코딩할 수 있습니다.
6. 지도 프로세스 다음에는 이러한 작은
비디오 파일을 함께.
7. 마지막 단계는 오디오와 비디오를 병합하고 파일을
사용자. 우리는 작업을 최종 결과물로 줄였습니다.
8. 아이들이 말했듯이 진정한 마지막 단계는 이입니다.

다음은 트랜스코더를 구축하는 데 사용할 주요 AWS 서비스 및 소프트웨어입니다.

FFmpeg - 이 책의 초판에서는 크로스 플랫폼인 FFmpeg를 간략하게 사용했습니다.
오디오 녹음, 변환 및 스트리밍을 위해 생성된 라이브러리/응용 프로그램 및
동영상. 이것은 모든 사람이 사용하는 응용 프로그램의 강국입니다.
취미 생활에서 상업 TV 채널에 이르기까지 다양합니다.

이 장에서 ffmpeg를 사용하여 비디오 파일의 트랜스코딩, 분할 및 병합을 수행합니다. 또한
ffprobe라는 유ти리티를 사용하여 비디오 파일을 분석합니다.
키프레임에서 잘라내는 방법을 알아보세요. ffmpeg 및 ffprobe 바이너리
Lambda 레이어 (<http://mng.bz/N4MN>)로 배송되며, 다른 허용
액세스하고 실행하기 위한 Lambda 함수. 반드시 사용할 필요는 없습니다.
Lambda 계층(각 함수와 함께 ffmpeg 바이너리를 업로드할 수 있습니다.
사용) 하지만 이는 중복되고 불편하며 배포하는 데 오랜 시간이 걸립니다.
따라서 Lambda 계층을 통해 ffmpeg를 사용할 수 있도록 하는 것이 좋습니다.
및 선호하는 접근 방식.

정의 키 프레임은 비디오 스트림에 전체 이미지를 저장하는 반면
일반 프레임은 이전 프레임의 충분 변경 사항을 저장합니다. 절단
on keyframes는 비디오에서 정보가 손실되는 것을 방지합니다.

AWS Lambda— 거의 모든 일에 Lambda를 사용할 것이라는 것은 두말할 나위 없는 일입니다.
Lambda는 ffmpeg를 실행하고 대부분의 로직을 실행합니다. 비디오를 분석하고, 오디오를 추출하고,
원본 파일을 분할하고, 변환하는 6가지 기능을 작성합니다.
세그먼트를 만들고 비디오 세그먼트를 병합한 다음 최종적으로 비디오와 오디오를 병합합니다.
단계.

Step Functions - 이 워크플로를 조정하는 데 도움이 되도록 Step Functions를 사용합니다. 이 서
비스는 실행 단계의 방법과 순서를 정의하는 데 도움이 됩니다.
전체 워크플로를 강력하게 만들고 추가 가시성을 제공합니다.
실행에.

S3 - 초기 및 최종 저장을 위해 Simple Storage Service(S3)를 사용합니다.
동영상. 임시 비디오 청크를 저장하는 데 사용할 수도 있지만
이를 위한 EFS. EFS를 선택한 이유는 장착이 쉽고
Lambda에서 파일 시스템으로 액세스합니다. 우리는 S3를 사용하는 대체 구현을 제공할 것이지만 제대
로 하기는 약간 더 어렵습니다.

EFS - 서비스 트랜스코더용 Elastic File System(EFS)을 사용하여 생성한 임시 파일을 저장합니다. 작은 비디오 파일이 많이 있을 것입니다. 다행히 EFS는 필요에 따라 확장 및 축소할 수 있습니다.

DynamoDB - Step Functions가 Lambda 함수의 전체 워크플로 및 실행을 관리하는 데 도움이 되지만 여전히 일부 상태를 유지해야 합니다. 특정 비디오 청크가 생성되었거나 병합될 수 있는지 여부를 알아야 합니다. DynamoDB를 사용하여 이 정보를 저장합니다. 저장된 모든 것은 임시이며 DynamoDB의 TTL(Time to Live) 기능을 사용하여 삭제됩니다.

그림 8.2는 우리가 구축하려는 시스템의 높은 수준의 개요를 보여줍니다. 다음 섹션에서 개별 구성 요소에 대해 살펴보겠습니다.

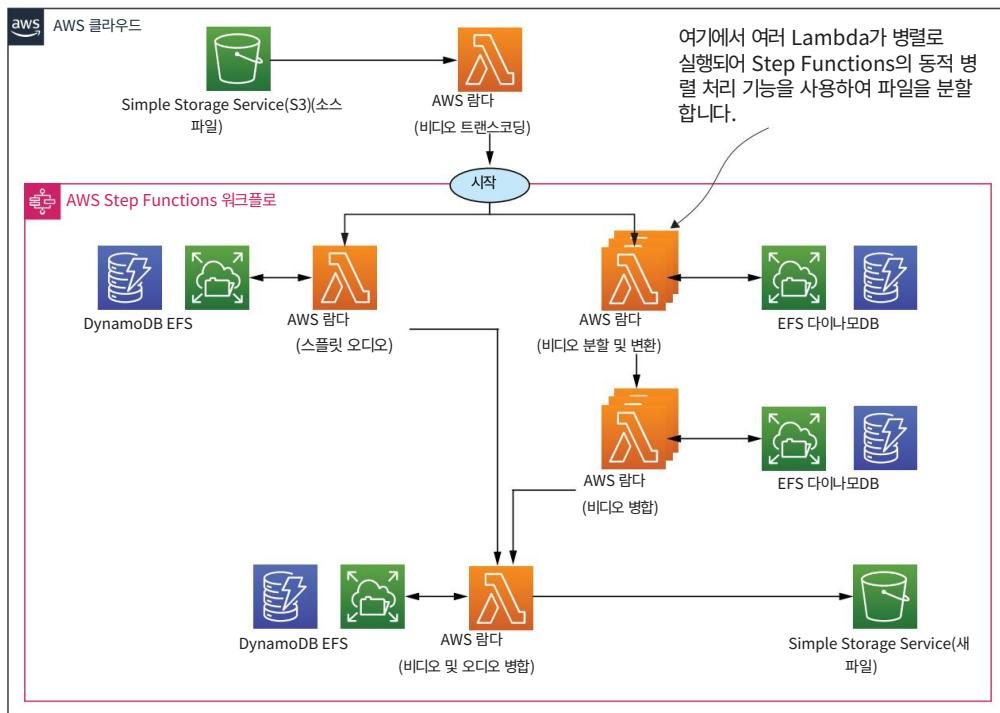


그림 8.2 이것은 서비스 트랜스코더의 단순화된 보기입니다. 여기에는 몇 가지 구성 요소가 더 있지만 아키텍처의 필수 요소에 초점을 맞추기 위해 이 그림에 포함하지 않았습니다.

8.2 아키텍처 심층 분석 아키텍처를 더 자

세히 살펴보겠습니다. 구현과 작동 방식에는 미묘한 차이가 있습니다. 나중에 문제가 발생하지 않도록 서비스 트랜스코더를 설계, 구축 및 배포하는 방법을 계획해 보겠습니다. 더 진행하기 전에 전체 아이디어는 거대한 비디오 파일을 여러 개의 작은 세그먼트로 분할하고 이러한 세그먼트를 병렬로 코드 변환한 다음 새 파일로 함께 병합하는 것임을 기억하십시오.

8.2.1 상태 유지

DynamoDB를 사용하여 서비스의 전체 작업에서 상태를 유지합니다.

관로. 어떤 비디오가 생성되었고 어떤 비디오가 생성되지 않았는지 추적합니다. 파이프라인을 단순화하고 특히 어떤 세그먼트를 모니터링하는 코드를 단순화하려면

생성되거나 트랜스코딩된 경우 트리를 사용할 것입니다. (더 나아가기 전에 이전에 있었던 모든 작은 비디오 세그먼트를 추적하는 방법에 대해 생각하십시오.

세그먼트가 생성 및 처리되는 경우 생성, 트랜스코딩 및 병합됩니다.

평행한.)

트리은 n^2 개의 더 작은 비디오 세그먼트를 만드는 것입니다. 우리는 하나의 큰 비디오 파일에서 2, 4, 8, 256, 512 또는 그 이상의 세그먼트를 생성해야 합니다. 다만 기억하십시오

세그먼트 수를 n^2 로 유지합니다. 왜 이런거야? 아이디어는

비디오 세그먼트를 트랜스코딩하면 어떤 순서로든 병합을 시작할 수 있습니다. 데

n^2 세그먼트를 사용하면 병합할 수 있는 두 개의 인접 세그먼트를 쉽게 식별할 수 있습니다.

그리고 데이터베이스에서 이 정보를 추적할 수 있습니다.

이 알고리즘에 대한 논리를 만드는 데 도움이 되는 기본 이진 트리를 만들겠습니다.

조금 더 쉽게 관리할 수 있습니다. 8개의 세그먼트가 있다고 가정해 보겠습니다. 프로세스가 어떻게 진행될 수 있는지는 다음과 같습니다.

1. 세그먼트 3과 4는 나머지보다 빠르게 코드 변환되고 병합될 수 있습니다.
함께(그들은 이웃) 3-4라는 새 세그먼트로 연결됩니다.
2. 그런 다음 세그먼트 7이 생성되지만 세그먼트 8은 아직 사용할 수 없습니다. 시스템 대기
세그먼트 8은 7과 8을 병합하기 전에 준비됩니다.
3. 세그먼트 8이 생성되고 세그먼트 7과 8은 새 세그먼트로 병합될 수 있습니다.
세그먼트 7-8.
4. 그런 다음 세그먼트 1과 2가 트랜스코딩을 완료하고 세그먼트 1-2로 병합됩니다.
5. 좋은 소식은 이웃 세그먼트 3-4가 이미 사용 가능하다는 것입니다. 그러므로,
세그먼트 3-4 및 1-2는 1-4라는 새 세그먼트로 함께 병합될 수 있습니다.
6. 세그먼트 5와 6은 코드 변환되어 세그먼트 5-6으로 병합됩니다.
7. 세그먼트 5-6에는 인접 세그먼트 7-8이 있습니다. 이 두 세그먼트가 병합됩니다.
함께 세그먼트 5-8을 만듭니다.
8. 마지막으로 세그먼트 1-4 및 5-8을 병합하여 최종 비디오를 만들 수 있습니다.
세그먼트 1에서 8로 구성됩니다.

n^2 개의 세그먼트 가 있으므로 인접 세그먼트를 추적하고

두 이웃(왼쪽 및 오른쪽)을 모두 사용할 수 있게 되면 즉시 병합합니다.

또 다른 흥미로운 측면은 세그먼트 자체가 병합할 이웃이 누구인지 파악할 수 있다는 것입니다. 2로 깔끔하게 나눌 수 있는 인덱스가 있는 세그먼트는 항상 켜져 있습니다.

오른쪽이 2로 깔끔하게 나누어지지 않는 부분이 왼쪽에 있습니다. 예를 들어 인덱스가 6인 세그먼트는 2로 나눌 수 있으므로 켜져 있음을 알 수 있습니다.

오른쪽 및 병합해야 하는 이웃(사용 가능하게 될 때)에는

index of 5. 그림 8.3은 블록을 병합하는 방법을 보여줍니다.

DynamoDB는 생성 및 병합된 세그먼트를 추적하기 위한 훌륭한 도구입니다. 사실, 우리는 가능한 모든 세그먼트를 미리 계산하고 추가할 것입니다.

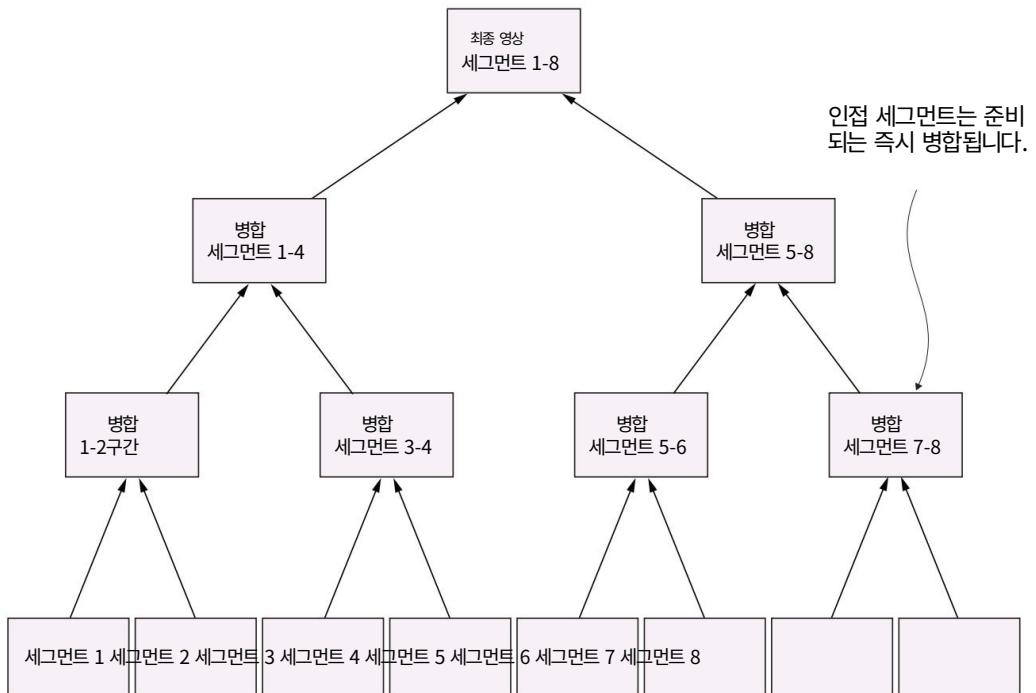


그림 8.3 세그먼트는 새 비디오로 병합됩니다. 우리 엔진의 장점은 인접 세그먼트가 준비되는 즉시 병합될 수 있다는 것입니다. 관련되지 않은 다른 세그먼트가 처리를 마칠 때까지 기다릴 필요가 없습니다.

다이나모DB. 그런 다음 DynamoDB에서 카운터를 원자 단위로 증가시켜 세그먼트가 생성 및 병합된 시기를 기록합니다. 이를 통해 처리 엔진은 아직 병합되지 않은 블록과 다음에 수행해야 하는 블록을 파악할 수 있습니다.

이것은 알고리즘의 중요한 부분이므로 다시 설명할 가치가 있습니다. DynamoDB의 각 레코드는 두 개의 인접 세그먼트(예: 세그먼트 1 및 세그먼트 2)를 나타냅니다. 분할 및 변환 작업은 세그먼트가 생성될 때마다 확인 카운터를 증가시킵니다. 확인 카운터가 2일 때 시스템은 두 개의 인접한 세그먼트가 존재하고 함께 병합될 수 있음을 알고 있습니다.

이 정보와 논리는 분할 및 변환 기능과 비디오 병합 기능에서 사용됩니다. 우리 알고리즘은 세그먼트를 계속 병합하고 병합할 항목이 없을 때까지 DynamoDB에서 확인 카운터를 증가시킵니다.

여러 가지 방법이 있습니다 . 이진 트리를 사용

하는 것은 이 문제를 해결하고 세그먼트를 추적하고 궁극적으로 함께 병합하는 한 가지 방법 일 뿐입니다. 이 작업을 수행할 수 있는 다른 방법이 무수히 많습니다. 다른 접근 방식을 생각 해 내야 한다면 어떻게 하시겠습니까?

트랜스코드 비디오

S3 버켓에 파일을 업로드하면 서비스 트랜스코더가 시작됩니다. S3 이벤트가 Transcode Video Lambda를 호출하고 프로세스가 시작됩니다. 그림 8.4는 이것이 어떻게 생겼는지 보여줍니다.

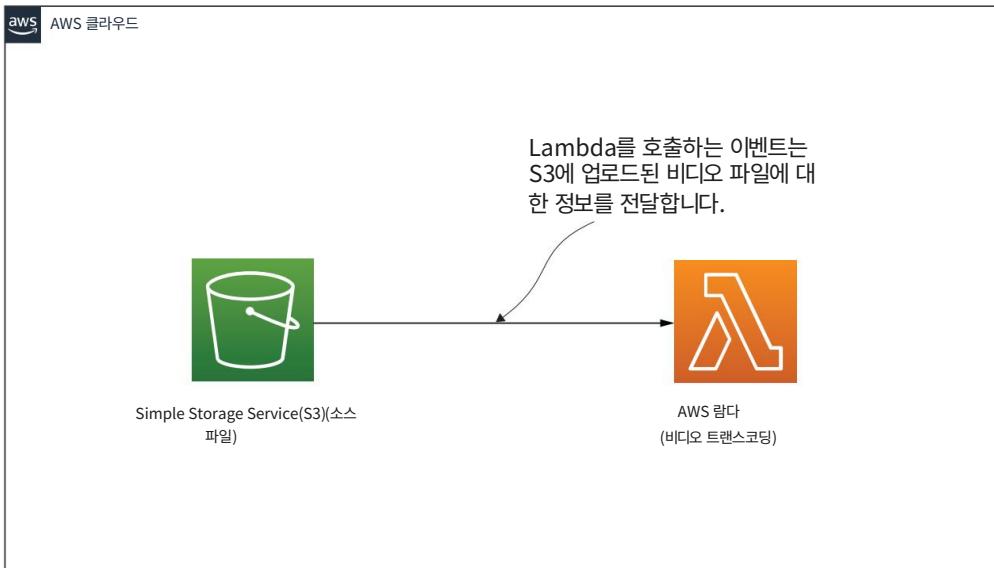


그림 8.4 이것은 기본적이고 일반적인 서비스 아키텍처입니다. S3에서 코드를 호출하는 것은 Lambda 함수의 핵심입니다.

- Transcode Video 기능은 다음 단계를 수행합니다.
1. S3에서 EFS의 로컬 디렉토리로 파일을 다운로드합니다.
 2. 다운로드한 비디오 파일을 분석하고 메타데이터를 추출합니다. 비디오 키 프레임은 이 메타데이터에서 제공됩니다.
 3. EFS에서 앞으로 나아갈 모든 세그먼트에 필요한 디렉토리를 생성합니다.
 4. 얼마나 많은 세그먼트를 생성해야 하는지 계산합니다.
키프레임.
- 우리는 항상 n^2 개의 세그먼트를 생성한다는 것을 기억하십시오. 즉, DynamoDB에서 일부 가짜 세그먼트를 생성해야 할 수 있습니다. 이것들은 정말로 아무것도 하지 않을 것입니다. 이미 생성된 세그먼트로 간주되므로 처리할 필요가 없습니다.
5. 모든 가짜 레코드를 포함하여 DynamoDB에 필요한 레코드를 생성합니다.
 - 필요.
 6. 두 개의 다른 입력으로 Step Functions 워크플로를 실행합니다. 첫 번째 파라미터는 오디오를 추출하고 EFS에 저장하기 위해 Lambda를 실행하도록 Step Functions에 지시합니다. 그만큼

두 번째 매개변수는 모든 시작 및 종료 시간을 지정하는 객체의 배열입니다.
생성해야 하는 세그먼트. Step Functions는 이 배열을 사용하여 적용합니다.
지도 절차. 각 객체에 대한 Lambda 함수를 팬아웃하고 생성합니다.
따라서 원본 파일이 많은 Lambda 함수에 의해 병렬로 분할되도록 합니다.

Transcode Video 기능은 다양한 일을 하기 때문에 모놀리식 또는 "뚱뚱한" Lambda 기능의 한 예입니다.
나눠서 하는 것도 나쁘지 않을 것 같지만,
이는 또한 자체적인 트레이드오프와 함께 제공됩니다. 결국 이 기능을 함께 유지해야 하는지 아닌지는 개인의
취향과 철학에 따라 달라질 수 있습니다. 우리는 이 기능이 우리가 해야 할 일에 대한 실용적인 솔루션이라고
생각하지만,
당신이 그것을 나누기로 결정했다면 당황하지 않을 것입니다.

8.2.2 단계 함수

Step Functions는 우리 시스템에서 중심적인 역할을 합니다. 이 서비스는 오케스트레이션 및 실행
비디오 파일을 세그먼트로 분할하고 트랜스코딩한 다음
그들을 병합합니다. Step Functions는 또한 오디오를 추출하는 기능을 실행합니다.
비디오 파일을 저장하고 보관을 위해 EFS에 저장합니다. 단계 함수가 호출하는 Lambda 함수는 다음과 같습니다.

오디오 분할 - 비디오에서 오디오를 추출하여 별도의 파일로 저장합니다.
EFS.

비디오 분할 및 변환 - 특정 시작 지점에서 비디오 파일을 분할합니다(
예: 5분 25초)에서 종료 지점(예: 6분 25초)
초) 그런 다음 새 세그먼트를 다른 형식이나 비트 전송률로 인코딩합니다.

비디오 병합 - 트랜스코딩된 세그먼트를 함께 병합합니다. 여러 개의 비디오 병합 기능은 하나의 최종
비디오 파일까지 세그먼트를 병합하기 위해 실행됩니다.

생산되었다.

비디오 및 오디오 병합 - 새로 생성된 비디오 파일과 오디오 파일을 병합합니다.
최종 출력을 만듭니다. 이 함수는 새 파일을 다시 S3에 업로드합니다.

팁 비디오에서 오디오를 추출한 다음 트랜스코딩할 필요가 없습니다.

동영상 파일만 따로. 우리는 테스트에서 비디오가 자체적으로 처리된 다음 오디오와 다시 결합될 때
시스템이 조금 더 빠르게 실행되기 때문에 그렇게 하기로 결정했습니다. 그러나 마일리지가 다를 수
있으므로 다음을 권장합니다.

먼저 오디오를 추출하거나 추출하지 않고 비디오 트랜스코딩을 테스트합니다.

Step Functions는 사용자 정의가 가능한 워크플로우 엔진입니다. 그것은 다른 지원
Task와 같은 상태(이는 Lambda 함수를 호출하거나 API에 매개변수를 전달합니다.
다른 서비스) 또는 Choice(분기 논리를 추가함).

우리가 사용할 한 가지 중요한 상태는 Map입니다. 이 상태는 배열을 취하고 exe는 해당 배열의 각 항목
에 대해 동일한 단계를 수행합니다. 따라서 배열을 전달하면

비디오를 세그먼트로 자르는 방법에 대한 정보를 제공하는 Map은 이러한 모든 인수를 병렬로 처리하기에 충분
한 Lambda 기능을 실행합니다. 이것이 바로 우리가 하려고 하는 것입니다
짓다. Map을 사용하여 Lambda 분할 및 변환 함수에 배열을 전달합니다.

유형. Step Functions는 원본을 자르는 데 필요한 만큼의 기능을 생성합니다.

비디오를 세그먼트로.

여기에도 더 흥미로운 부분이 있습니다. 세그먼트가 생성되자마자 Step

함수는 모든 세그먼트가 다음으로 병합될 때까지 Merge Video 함수를 호출하기 시작합니다.

새로운 영상. Step Functions 실행 워크플로에 몇 가지 로직을 추가하여

Merge Video를 호출해야 하는 경우 출력합니다. 모든 비디오 병합 작업이 호출되고 처리되면 Step Functions는 오디오 분할 및 비디오 병합의 결과를 가져옵니다.

최종 비디오 및 오디오 병합 기능을 호출합니다. 그림 8.5는 이 프로세스가 어떻게 생겼는지 보여줍니다.

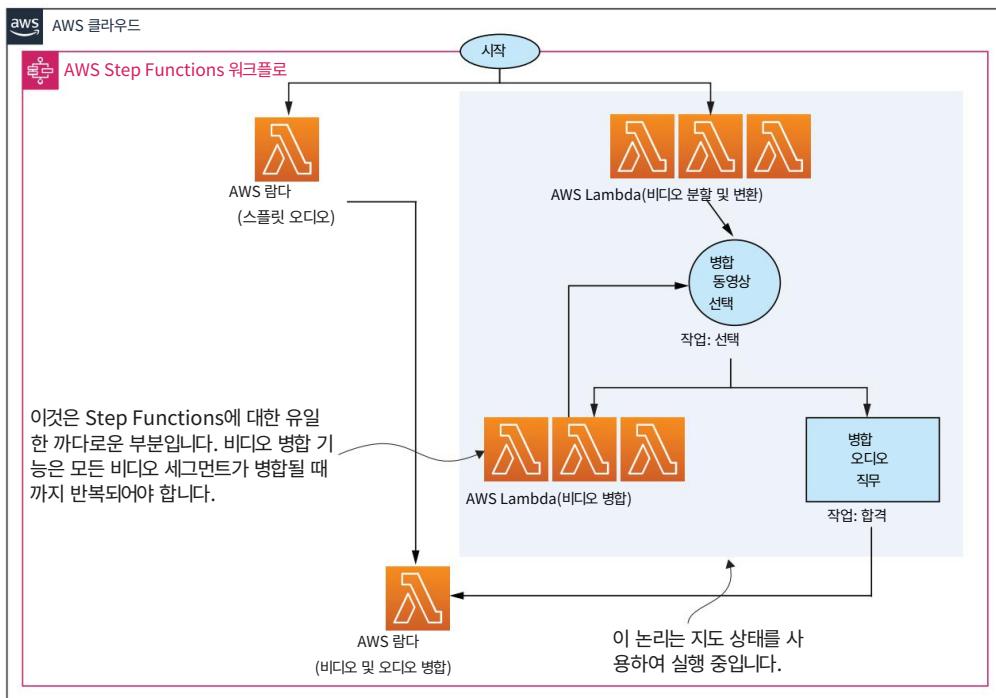


그림 8.5 Step Functions 실행 워크플로는 트랜스코더에서 모든 작업을 수행합니다. 비디오는 두 가지 주요 기능과 약간의 논리를 사용하여 분할, 변환 및 다시 병합됩니다.

이제 Step Functions 워크플로가 무엇을 하는지 알았으므로 각 람다 함수에 대해 자세히 알아보십시오.

오디오 분할

Step Functions는 오디오 분할 Lambda 함수를 실행하여 비디오에서 오디오를 추출합니다.

파일. 앞서 언급했듯이 이 단계는 전체 워크플로를 가속화하기 위해 수행됩니다.

거기에서 파일의 오디오 부분은 고려되지 않고 비디오 부분만 다른 비트 레이트로 트랜스코딩됩니다. 우리는 이것을 할 필요가 없습니다. 오디오를 남길 수 있습니다.

및 비디오를 함께 사용했지만 우리의 경우 테스트에서 이렇게 하면 전반적인 성능이 향상되는 것으로 나타났습니다. 오디오 분할 기능은 다음 단계를 실행합니다. 1. ffmpeg를 사용하여 오디오를 추출하고 EFS의 폴더에 저장합니다.

2. 관련 DynamoDB 레코드를 업데이트하여 이 작업이 완료되었음을 기록합니다.

3. 성공 메시지와 추가 매개변수(예:

오디오 파일)을 Step Functions 오케스트레이터로 전송합니다.

이후 단계에서 Step Functions는 오디오 분할 및 비디오 병합 기능에서 반환된 매개변수를 사용하여 비디오 오디오 병합 기능을 호출합니다.

비디오 분할 및 변환

비디오 분할 및 변환 기능은 원본 비디오 파일을 세그먼트로 분할하고 해당 세그먼트를 새 비트 전송률 또는 인코딩으로 변환합니다. 원본 비디오 파일은 이 과정에서 변경되지 않습니다. 대신 함수는 전달된 매개변수에 지정된 시작 시간과 종료 시간 사이의 세그먼트만 추출합니다. 이 매개변수는 Transcode Video 기능에 의해 해결됩니다.

수백 개의 비디오 분할 및 변환 기능을 병렬로 실행할 수 있습니다. 수행하는 주요 작업은 다음과 같습니다.

1. ffmpeg를 사용하여 이 기능은 원본 파일에서 새 비디오 파일을 만듭니다.

2. 적절한 DynamoDB 레코드의 확인 카운터를 증가시켜 세그먼트가 존재함을 지정합니다.

3. 확인 카운터가 2이면 병합 메시지와 함께 Step Functions 워크플로로 돌아갑니다. 그렇지 않으면 특정 Step Functions 병렬 실행의 실행을 중지하는 성공 메시지와 함께 반환됩니다.

이전 섹션에서 DynamoDB에서 각 레코드가 두 개의 인접 세그먼트를 나타낸다는 것을 기억할 수 있습니다. 레코드 카운터가 2로 증가하면 함수는 두 개의 인접 세그먼트가 있음을 알고 있습니다. 함수는 병합 메시지를 Step Functions에 반환하고 Step Functions는 이러한 세그먼트에 대해 병합 비디오 호출을 시작할 수 있음을 알고 있습니다.

비디오 병합

두 개의 인접 세그먼트를 새로운 단일 세그먼트로 병합할 준비가 되면 Step Functions가 비디오 병합 기능을 호출합니다. 병합 작업은 ffmpeg를 사용하여 발생하고 새 세그먼트는 EFS에 저장됩니다. 조금 더 자세히 설명하면 다음과 같습니다.

1. 비디오 병합 기능은 Step Functions에 의해 전달된 여러 매개변수와 함께 호출됩니다. 이러한 매개변수에는 왼쪽 및 오른쪽 세그먼트가 포함됩니다.

2. ffmpeg를 사용하여 왼쪽 및 오른쪽 세그먼트를 병합하여 새 세그먼트를 만듭니다.
이 새 세그먼트는 EFS에 저장됩니다.

3. DynamoDB 확인이 증가합니다. 두 가지 확인이 있는 경우 함수는 병합 메시지와 함께 워크플로로 돌아갑니다.

4. 그러나 두 개의 확인이 있고 마지막 두 개의 세그먼트가 병합된 경우 함수는 MergeAudio 메시지와 함께 워크플로로 반환됩니다.

보시다시피 비디오 병합 기능은 약간의 루프를 만듭니다. 계속 합쳐진다 세그먼트, 병합 메시지를 반환하고 Step Functions가 스스로를 호출하도록 합니다. 다시. 이것은 마지막 두 세그먼트가 병합될 때까지 발생하고 반환 유형은 다음과 같습니다. MergeAudio로 변경되었습니다. 이것은 Step Functions가 결합할 때임을 알게 된 때입니다. 오디오 및 비디오 및 비디오 및 오디오 병합 기능을 호출합니다.

비디오와 오디오 병합

마지막 기능은 비디오 및 오디오 병합입니다. 스플릿 오디오에서 입력을 받고 비디오 기능을 병합하고 다음을 사용하여 오디오와 새 비디오 파일을 병합합니다. ffmpeg. 새 파일은 EFS의 다른 위치(다른 디렉토리)에 저장됩니다. 그만큼 함수는 더 쉽게 액세스할 수 있도록 새 파일을 S3 버킷에 업로드할 수도 있습니다.

8.3 대체 아키텍처

EFS(또는 이를 위한 Step Functions)를 사용하지 않고 이 서비스 트랜스코더를 구축할 수 있습니다. 문제). 실제로 첫 번째 반복에서는 S3와 SNS만 사용하여 팬아웃을 수행했습니다. 우리 원하지 않는 경우 Step Functions 또는 EFS를 사용할 필요가 없음을 보여주는 대체 아키텍처를 보여주고 싶었습니다. 이 섹션에서는 SNS와 S3를 대신 사용하여 동일한 결과를 얻을 수 있음을 보여줍니다. 다행이다

AWS는 원하는 아키텍처를 구축할 수 있는 많은 빌딩 블록을 제공합니다.
다른 방법들.

팁 다른 아키텍처를 선택하는 한 가지 이유는 다음과 같을 수 있습니다.
Step Functions 및 EFS 비용을 지불하고 싶지 않습니다. 그것은 합리적인 우려입니다.
S3를 사용하는 것은 EFS를 사용하는 것보다 훨씬 저렴할 것이며 아마도 똑같이 수행하십시오. 코드가 작동하면 S3를 사용하는 것이 간단하고 권장하지 않을 이유가 없습니다. 당신이 해야 하는지 여부
Step Functions 대신 SNS를 사용하는 것은 더 어려운 제안입니다. SNS가 싸다.
그러나 당신은 당신이 얻을 수 있는 견고성과 관찰 가능성을 많이 잃을 것입니다.
단계 기능. 아마도 가장 좋은 솔루션은 S3에서 Step Functions를 사용하는 것입니까?
우리는 그것을 달성하기 위한 연습으로 당신에게 맡길 것입니다.

이 대체 구현은 앞서 언급했듯이 Step Functions를 SNS로, EFS를 S3로 대체한다는 점을 제외하고는 이전 섹션에서 생성한 것과 매우 유사합니다.

그림 8.6은 이 아키텍처가 어떻게 생겼는지 보여줍니다.

이 아키텍처는 잘 작동하지만 몇 가지 개선할 수 있는 사항이 있습니다.
그것에. 첫째, 다음과 같은 오류가 발생한 경우 구현을 개선해야 합니다.
분할 또는 병합 작업이 실패했습니다. 다행히도 Lambda에는 실패한 호출을 저장, 검토 및 재생할 수 있는 DLQ(Dead Letter Queue) 기능이 있습니다. 만약 너라면
도전을 원하신다면 이 아키텍처에 대한 DLQ를 구현하여
오류에 더 탄력적입니다.

두 번째 문제는 관찰 가능성과 시스템에서 무슨 일이 일어나고 있는지 아는 것입니다.
Step Functions는 어느 정도의 가시성을 제공하지만
SNS. 스스로를 듣는 데 사용할 수 있는 도구 중 하나는 AWS X-Ray입니다. 이 AWS 서비스가 도움이 될 수 있습니다.
시스템 내 다양한 서비스의 상호 작용을 이해합니다. CloudWatch도 필수적인 것은 당연합니다.

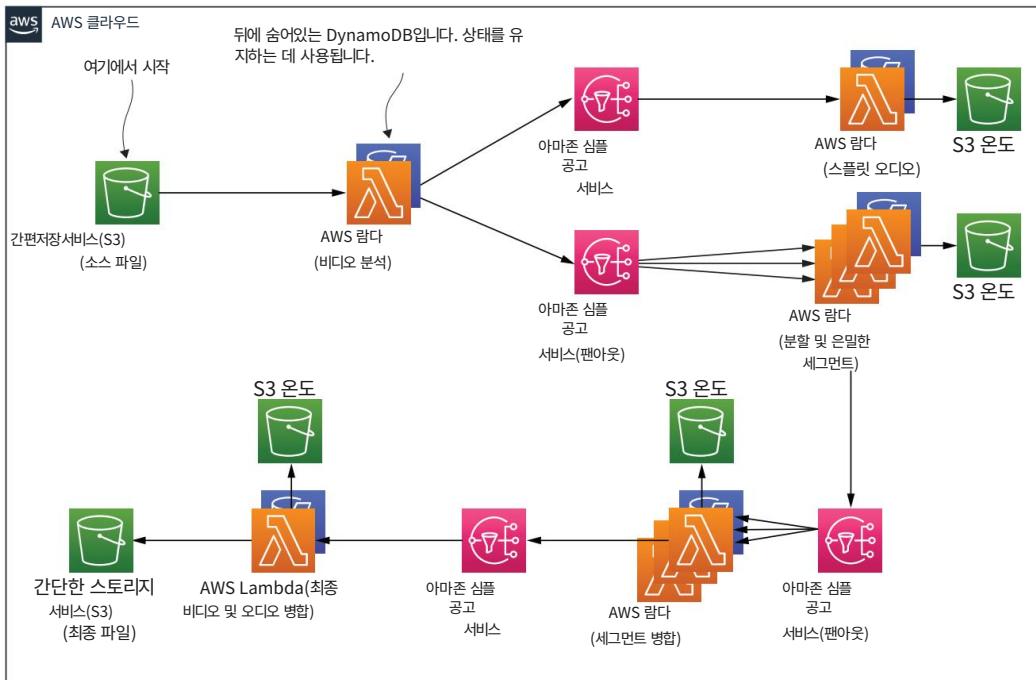


그림 8.6 서비스 트랜스코더를 위한 SNS 및 S3 아키텍처

요약

MapReduce는 서비스 접근 방식에서 정말 잘 작동합니다. 람다가 당신을 초대합니다
병렬화에 대해 처음부터 생각하려면 이를 활용하십시오.

Lambda에서 많은 문제를 해결하고 방대한 양의 데이터를 처리할 수 있습니다.

더 작은 청크로 분할하고 작업을 병렬화합니다.

Step Functions는 워크플로 정의를 위한 훌륭한 서비스입니다. 그것은 당신이 할 수 있습니다
팬아웃 및 팬인 작업.

Lambda용 EFS는 무한 로컬 디스크로, 필요한 만큼 확장됩니다. 당신은 할 수 있습니다
이전에는 실행할 수 없었던 EFS 및 Lambda로 애플리케이션을 실행합니다. S3가 여전히 더 저렴할
가능성이 높으므로 계산을 수행해야 합니다.

EFS를 선택하기 전에 분석합니다.

다양한 방법으로 문제를 해결할 수 있습니다. – SNS

를 사용할 수 있기 때문에 Step Functions를 사용할 필요가 없습니다(비록

Step Functions는 추가 수준의 견고성과 가시성을 추가합니다.

- S3를 사용할 수 있으므로 EFS를 사용할 필요가 없습니다.

시스템을 위한 아키텍처를 구상할 때 사용 가능한

다른 절충안을 가진 다른 대안이 있기 때문입니다.

코드 개발자 대학

이 장에서는 다음을 다룹니다.

AWS Glue 및 Amazon Athena

EventBridge를 사용하여 시스템 구성 요소 연결

대규모 데이터 처리를 위해 Kinesis Firehose 및 Lambda 사용

우리가 한동안 숙고해 왔던 아이디어 중 하나는 도움이 되도록 설계된 웹 앱이었습니다.

개발자는 게임화 및 유용한 분석을 통해 재미있는 방식으로 프로그래밍 기술을 배웁니다. 우리의 아이디어는 코드 개발자 대학(CDU)이라고 하며 흥미로운 프로그래밍 과제 모음이 있는 개념 증명 웹 사이트로 발전했습니다.

문제를 해결하고 기술을 구축할 신진 개발자.

각각의 도전은 문제를 제기할 것입니다. 학생은 입력할 공간이 있습니다.

솔루션을 제공한 다음 처리를 위해 시스템에 제출합니다. 시스템은

일련의 테스트를 통해 솔루션을 실행하고 솔루션이 통과했는지 여부를 결정합니다.

또는 실패했습니다. 솔루션이 실패한 경우 사용자는 코드를 업데이트할 수 있습니다.

다시 제출하십시오. 솔루션이 통과되면 사용자는 다음 챌린지로 이동하여 난이도에 따라 50~500 경험치(XP)를 받습니다.

문제의.

전체 경험을 더 흥미롭고 흥미롭게 만들기 위해 시스템 전체에 게임화 요소가 내장되어 있을 것입니다. 예를 들어, 경험

포인트는 다양한 리더보드를 만드는 데 사용됩니다. 이를 통해 사용자는

친선 경쟁은 상위 10위권을 놓고 경쟁할 수 있습니다. 더 많은 문제를 풀수록 점수가 높아집니다. 이 리더보드는 전체 상위 10위를 보여줍니다.

Python 또는 JavaScript와 같은 각 언어에 대한 최고의 성능을 제공합니다.

학생이 더 많은 데이터를 살펴보고 더 많이 보고, 검색하고, 필터링하려는 경우 고급 보고서도 지원됩니다. 예를 들어, 학생은 다음을 볼 수 있습니다.

다른 사용자가 저지르고 배우는 가장 흔한 실수(물론 익명 처리됨)
그것으로부터도.

원래 아이디어는 고상했지만 실행 가능했습니다. 이 프로젝트를 구축하는 핵심은 가능한 한 다양한 AWS 서비스에 의존하십시오. 그렇게 하면 우리는 시스템의 고유한 측면과 차별화되지 않은 무거운 작업은 남겨두고, 인증처럼 AWS에. 결국, 보시게 되겠지만 다음 서비스를 사용하여 모든 것을 통합했습니다.

EventBridge(메시징)

접착제(데이터 준비 및 변환)

Athena(데이터 쿼리)

DynamoDB(데이터베이스)

QuickSight(보고)

S3(스토리지)

람다 및 API 게이트웨이

이 장에서는 특히 CDU에 대한 데이터, 리더보드 및 보고에 중점을 둡니다.

AWS 생태계의 많은 부분을 사용하기 때문에 시스템의 매력적인 부분입니다.

그리고 비슷한 것을 스스로 빠르게 만들 수 있기 때문입니다. 의 다른 기능

CDU는 웹 앱의 표준입니다. 사용자 계정, HTML5 사용자 인터페이스, 예상할 수 있는 모든 기본 볼트 및 비트가 있습니다. 방법을 배우고 싶다면

그러한 시스템을 직접 구축하려면 이 책의 초판을 살펴보십시오.

비디오 중심의 웹 애플리케이션이지만 유사하지만 설명합니다.

9.1 솔루션 개요

CDU의 리더보드와 보고 측면은 서비스이기 때문에 흥미롭고,

확장 가능하고 솔직히 구현하기 재미있습니다. 많은 서비스 AWS 서비스가 있습니다.

과거의 전통적인 보고 및 데이터 워어하우징 제품에 의존하지 않고 데이터 수집, 집계 및 분석을 가능하게 합니다. 먼저 살펴보자

요구 사항 및 전체 솔루션.

9.1.1 나열된 요구 사항

CDU는 사용자 등록 및 계정 기능이 있는 웹사이트이며, 사용자가

코드 연습에 액세스하여 시도하고 코딩 문제를 구현하고 해결하는 데 성공하면 점수를 받습니다. 이를 위해 다음 섹션에서는 다음 목록을 제공합니다.

CDU에 대한 높은 수준의 요구 사항.

일반

사용자는 자신의 코드 솔루션을 실행할 수 있어야 하며 테스트 통과 또는 실패 여부를 결정할 수 있어야 합니다.

테스트를 통과하면 솔루션이 올바른 것으로 간주됩니다.

올바른 솔루션은 사용자에게 일정 수의 포인트를 부여하고 사용자 프로필에 저장됩니다.

사용자가 볼 수 있는 리더보드와 고급 보고서가 있어야 합니다.

전체 시스템은 서비스, 이벤트 중심, 최대한 자동화되어야 합니다(리더보드 및 보고서를 업데이트하기 위해 관리자의 개입이 필요하지 않음).

사용자 및 경험 포인트는 과

제를 해결하는 데 사용되는 프로그래밍 언어에 대해 수여됩니다. 예를 들어 사용자가 Python으로 코드를 작성하면 Python에 할당된 점수를 받습니다. JavaScript를 사용하는 경우 점수는 JavaScript 점수에 할당됩니다.

사용자 프로필에는 전체 점수(모든 프로그래밍 언어에 대한 이전 점수의 합계)와 각 프로그래밍 언어에 대한 점수가 개별적으로 표시되어야 합니다. 사용자의 프로필과 점수는 거의 실시간으로 업데이트되어야 합니다.

사용자는 동일한 챌린지에 대해 두 번 이상 포인트를 받을 수 없습니다.

리더보드

CDU에는 다양한 프로그래밍 언어(예: Python 및 JavaScript)에서 최고 득점을 표시하는 리더보드가 있어야 합니다.

전체 순위표에는 지난 달, 작년 및 모든 기간 동안 최고의 수행자(프로그래밍 언어에 관계없이)가 표시되어야 합니다.

순위표는 실시간으로 업데이트할 필요는 없지만 최소한 60분마다 새로 고쳐야 합니다. 관리자의 요청에 따라 새로 고칠 수 있는 방법도 있어야 합니다.

보고서

순위표 외에도 사용자는 검색 및 필터링할 수 있는 보다 심층적인 보고서에 액세스할 수 있어야 합니다.

보고서의 정확한 구현은 데이터 팀에 맡길 수 있습니다. 그러나 기본 보고서는 리더보드와 유사하게 최고의 성과를 보일 수 있습니다.

보고서는 최소한 60분마다 새로 고쳐야 하지만 관리자가 더 빨리(필요한 경우) 새로 고칠 수도 있습니다.

관리자뿐만 아니라 모든 사용자가 리더보드에 액세스할 수 있어야 합니다.

9.1.2 솔루션 아키텍처

이제 주요 요구 사항을 해결해야 하는 가능한 아키텍처를 살펴보겠습니다. 그림 9.1은 대부분의 주요 아키텍처 부분을 보여줍니다. 여기에는 다음 세 가지 주요 마이크로서비스가 포함됩니다.

코드 스코어링 서비스

학생 프로필 서비스

분석 서비스

다음 섹션에서 솔루션을 분해할 것이지만

그림 9.1에 표시된 고수준 아키텍처. Code Scoring Service는 Lambda를 실행합니다.

제출된 코드를 처리하는 기능. 테스트를 통하여 정보가 다음을 통해 전송됩니다.

두 개의 다른 마이크로 서비스를 호출하는 EventBridge에:

학생 프로필 서비스는 데이터베이스에서 학생의 프로필을 업데이트하고
학생의 전체 점수에 추가됩니다.

Analytics Service는 사용자의 테스트 데이터를 처리하고 S3에 저장합니다.

QuickSight 대시보드 생성을 활성화합니다.

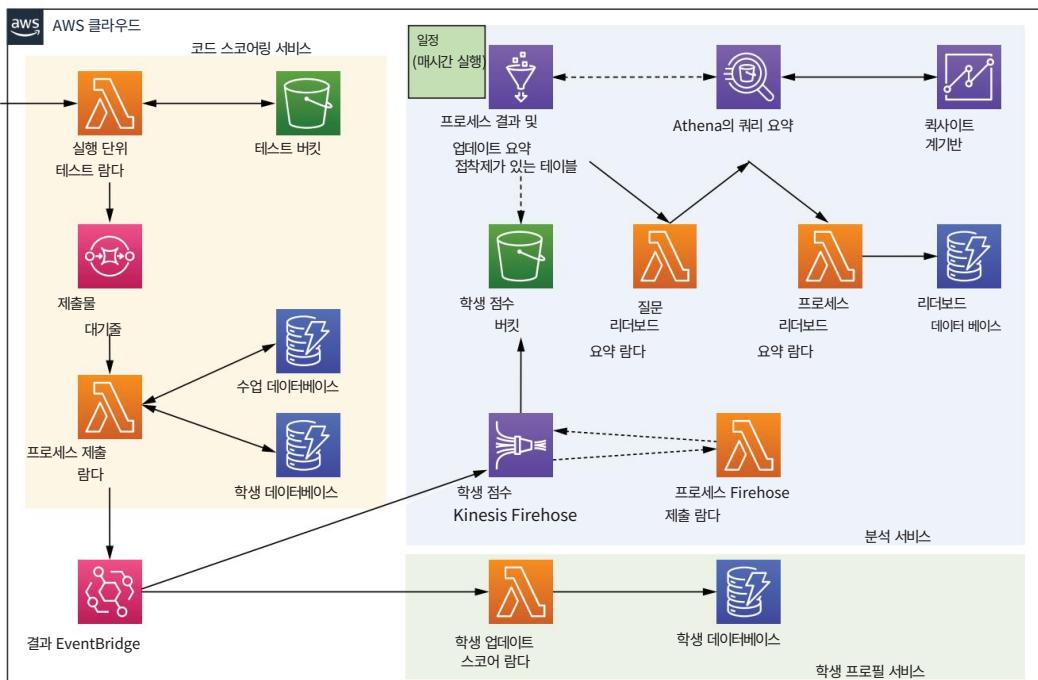


그림 9.1 점수 및 리더보드를 담당하는 CDU(Code Developer University)의 아키텍처

실제로 분석 서비스에서 꽤 많은 일이 발생합니다. 자세하게 다루고 있습니다

섹션 9.4에 있지만, 여기에서 실제로 일어나는 일에 대한 높은 수준의 개요가 있습니다.

마이크로서비스:

메시지(사용자 솔루션 포함)가 Kinesis Firehose로 푸시되고

Lambda 함수를 사용하여 메시지 형식을 수정할 수 있습니다.

나중에 다른 AWS 서비스에서 처리됩니다.

그런 다음 Kinesis는 새로 처리된 메시지(JSON 파일로)를 S3 버킷에 저장합니다.

AWS Glue는 60분마다 트리거되도록 설정된 일정에 따라 실행됩니다. 이 경우 Glue는 앞서 언급한 S3 버킷을 처리하고 S3에 저장된 데이터를 가리키는 Glue 데이터 카탈로그(메타데이터가 있는 테이블)를 업데이트합니다.

그런 다음 Glue는 Amazon Athena를 사용하여 Glue 데이터 카탈로그를 통해 S3에 저장된 데이터를 쿼리하는 Lambda 함수를 트리거합니다.

Athena가 완료되면 쿼리 결과를 가져오고 DynamoDB에 저장된 적절한 리더보드를 업데이트하는 또 다른 Lambda 함수를 트리거합니다.

마지막으로 사용자가 추가 정보를 보고 싶어할 때 Athena를 사용하여 S3 버킷의 데이터를 쿼리하는 Amazon QuickSight 보고서가 있습니다.

모든 서비스에 대해 조금 더 자세히 설명하고 다른 질문이 있을 수 있습니다. 이에 대해서는 다음 섹션에서 해결해야 합니다. 일어!

QuickSight 대 DynamoDB 스스로

에게 물어볼 수 있는 한 가지 질문은 보고에 Amazon QuickSight를 사용하고 DynamoDB에 리더보드도 저장하는 이유입니다. 중복 아닌가요? 그 이유는 QuickSight가 무겁고 강력하지만 느리기 때문입니다. iFrame에서 웹사이트에 통합할 수 있지만 로드하는 데 시간이 오래 걸립니다. QuickSight를 사용하여 데이터를 자세히 탐색하려는 경우 10초 또는 20초 동안 기다려야 합니다. 하지만 결과를 빨리 보고 싶다면 기다리기 힘들다(AWS, 성능 좀 봐주세요!).

이것이 우리가 중요한 리더보드 결과를 DynamoDB에 저장하는 이유입니다. DynamoDB는 사용자에게 빠르게 로드되고 표시될 수 있습니다. 그런 다음 특히 더 자세한 정보가 필요한 경우 이 데이터의 QuickSight 버전을 선택하는 것은 사용자의 몫입니다. DynamoDB 리더보드를 QuickSight의 비공식 캐시로 생각할 수 있습니다.

이 구현의 부정적인 측면은 DynamoDB 테이블과 QuickSight의 데이터가 동기화되어야 한다는 것입니다. 학생 프로필 서비스가 학생의 점수를 업데이트하지만 분석 서비스가 실패하면 DynamoDB에 QuickSight와 다른 내용이 표시될 수 있습니다. 이 문제를 해결할 수 있는 방법이 있습니다. 어떻게 하시겠습니까?

이에 대해서는 9.3절에서 논의할 것이다.

9.2 코드 채점 서비스 코드 채점 서비스의 목적

은 사용자로부터 제출된 코드를 받아 일련의 테스트에 대해 실행하는 것입니다. 테스트가 통과하면 실행 단위 테스트 Lambda가 제출을 생성하여 제출 대기열에 넣습니다. 제출은 Process Submission Lambda에 의해 대기열에서 선택되고 몇 가지 DynamoDB 테이블의 데이터로 보강됩니다. 마지막으로 프로세스 제출 Lambda는 시스템의 다른 서비스에서 사용할 수 있도록 새로 강화된 메시지를 Amazon EventBridge로 푸시합니다.

Code Scoring Service의 실제 디자인은 상당히 간단하지만 그 디자인을 좀 더 자세히 살펴보겠습니다. 그림 9.2는 Run Unit Test Lambda로 시작하는 아키텍처의 근접 촬영을 보여줍니다. 이 Run Unit Test Lambda 함수는 HTTPS를 통해(API 게이트웨이를 통해) 호출되고 zip 패이로드를

요청 본문. zip 페이로드에는 사용자의 코드 제출 및 메타데이터(예: 사용자가 시도하는 챌린지 및 사용 중인 프로그래밍 언어)가 포함됩니다. Lambda 함수는 테스트 버킷에서 적절한 테스트를 조회하고(수업 이름을 기반으로 어떤 테스트를 가져올지 알고 있음) S3에서 해당 테스트 파일을 다운로드합니다.

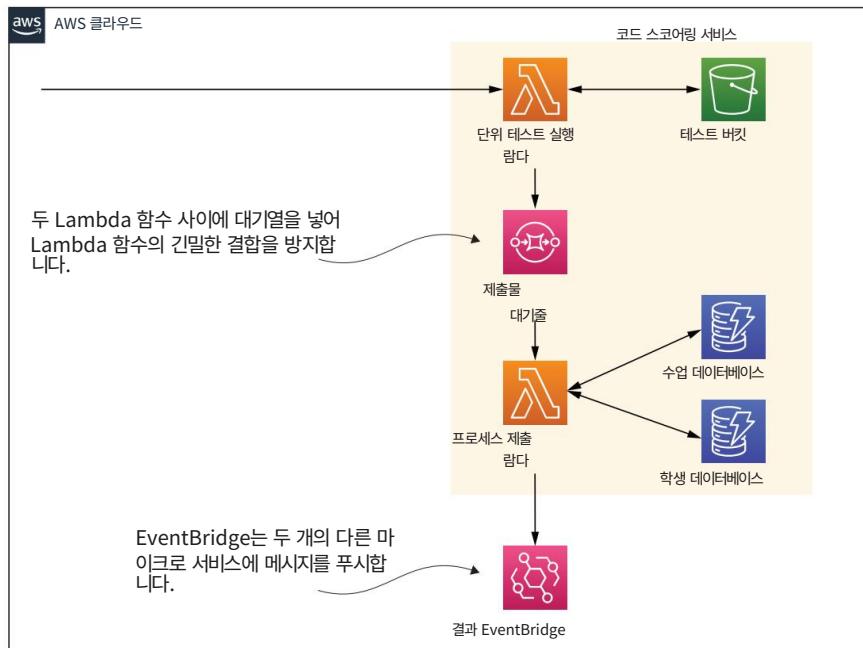


그림 9.2 Code Scoring Service는 사용자의 코드를 실행하고 성공하면 시스템의 나머지 이벤트 체인을 시작합니다.

이제 Lambda 함수는 적절한 인터프리터 또는 컴파일러를 실행하고, 단위 테스트를 실행하고, 사용자의 제출을 테스트할 수 있습니다.

Lambda 계층

Python, JavaScript, C++, C#, Java 등과 같은 여러 언어를 지원하려면 Lambda 계층을 사용하십시오. 계층은 추가 라이브러리 또는 사용자 정의 런타임을 포함할 수 있는 zip 파일입니다.

Python 인터프리터가 있는 Lambda 계층 또는 C 컴파일러가 있는 계층이 있을 수 있습니다.

또한 Lambda 함수와 별도로 계층을 배포하므로 실제 Lambda 함수를 쉽게 유지합니다. 런타임에 올바르게 구성된 한 Lambda 함수는 계층의 콘텐츠(함수 실행 환경의 /opt 디렉터리로 추출됨)에 액세스할 수 있습니다. 원하는 만큼 계층을 배포할 수 있지만 Lambda는 한 번에 최대 5개의 계층만 사용할 수 있습니다. 자세한 내용은 <http://mng.bz/doJO>에서 읽을 수 있습니다.

Run Unit Test Lambda 자체는 특별히 복잡하지 않습니다. 단위 테스트를 실행하고 성공적으로 통과했는지 여부를 파악하기 위해 결과를 구문 분석하는 방법을 알아야 합니다.

테스트가 실패하면 함수는 인터프리터 또는 컴파일러의 출력과 함께 HTTP 응답을 다시 보냅니다. 따라서 사용자는 오류 메시지를 보고 코드 문제를 수정한 다음 다시 제출할 수 있습니다. 그렇지 않고 통과하면 함수는 사용자에게 축하 메시지를 다시 보내고 추가 처리를 위해 사용자의 제출이 포함된 메시지를 제출 SQS 대기열에 배치합니다.

9.2.1 제출 대기열

제출 대기열은 이 서비스의 단 두 개의 Lambda 함수 사이에 있는 SQS 대기열입니다. 메시지가 대기열에 배치되면 메시지를 검색하고 EventBridge로 푸시하기 전에 더 많은 데이터로 보강하는 Process Submission Lambda가 호출됩니다. 다음을 포함하여 몇 가지 이유가 있습니다. 요구 사항 중 하나는 사용자가 동일한 챌린지에 대해 여러 번 포인트를 받는 것을 방지하는 것입니다. 프로세스 제출 Lambda는 사용자가 이미 이 챌린지를 완료했는지 여부를 파악하기 위해 학생 DynamoDB 테이블을 조회해야 합니다. 사용자가 이미 해당 챌린지를 완료한 경우 해당 사항이 표시되고 포인트가 적립되지 않습니다.

학생이 처음으로 문제를 해결하고 점수를 받는다고 가정하면 Process Submission Lambda는 Lesson 데이터베이스에서 몇 점을 받아야 하는지도 조회합니다.

대기열에서 온 메시지를 포함하여 이 모든 정보가 결합되어 Amazon EventBridge로 푸시됩니다.

지금쯤이면 "초기 Run Unit Test Lambda에서 모든 작업을 수행하지 않는 이유는 무엇입니까?"라고 생각할 수 있습니다. 그 이유는 책임을 분리하기 위함입니다. 단위 테스트 실행 기능은 코드를 실행하고 테스트를 통과하는지 확인하기 위한 것입니다. 두 번째 Process Submission Lambda 함수는 데이터베이스 조회를 수행하고 학생이 점수를 받아야 하는지 여부를 평가해야 합니다. 경험에 따르면 모든 것을 하나로 묶기보다 다양한 문제를 처리할 때 여러 Lambda 함수를 사용해야 합니다. 따라서 이것이 우리가 두 개의 함수를 만들고 그들 사이에 메시지 큐를 도입한 이유입니다.

또 다른 질문은 함수가 서로를 직접 호출하지 않고 SQS를 사용하는 이유입니다. Lambda 대상이라는 기능(어쨌든 두 함수 사이에 숨겨진 대기열을 추가함)을 사용하지 않는 한 함수가 서로를 직접 호출하지 않는 것이 좋습니다. 그러나 Lambda 대상은 비동기식 호출에 대해서만 작동하므로 우리의 경우에는 불가능했을 것입니다. Run Unit Test Lambda는 HTTP를 통해 동기적으로 호출되었습니다. 두 함수 사이에 대기열이 있는 이유는 결합을 줄이고(예: 두 함수가 서로에 대한 직접적인 지식이 없음) 오류 및 재시도를 처리하는 데 더 쉬운 시간을 갖기 위함입니다.

대신 Amazon EventBridge를 사용하도록 선택할 수도 있었지만 SQS는 이 시나리오에서 허용됩니다. 그리고 FIFO(선입선출)를 활성화하려면 이후 단계에서 큐를 사용하려면 EventBridge가 지원하지 않기 때문에 SQS를 사용해야 합니다. 이 기능은 우리의 결정에 무게를 더했습니다.

Process Submission Lambda가 수행하는 마지막 작업은 메시지를 Amazon EventBridge로 푸시하는 것입니다. 기억하시겠지만 이 메시지에는 원본이 포함되어 있습니다. 사용자에게 경험 포인트를 부여해야 하는지 여부 및 금액에 대한 정보로 구성된 추가 세부 정보와 함께 사용자가 제출 그 포인트의 (이 정보는에서 두 개의 테이블을 조회하여 얻은 것입니다. 프로세스 제출 Lambda 함수).

아마존 이벤트브리지

Amazon EventBridge는 서로 다른 AWS(및 비 AWS) 서비스. SQS, SNS, Kinesis와 같은 서비스에는 없는 몇 가지 훌륭한 기능이 있습니다. 그 중 가장 중요한 것은 90개 이상의 AWS 서비스를 이벤트 소스로, 17개 서비스를 대상으로 사용하는 기능, 자동화된 확장, 콘텐츠 기반 필터링, 스키마 검색 및 입력 변환입니다. 그러나 다른 기술과 마찬가지로 이벤트 순서나 버퍼링에 대한 보장이 없는 것과 같은 특정 결함이 있습니다.

항상 그렇듯이 결국 선택하는 것은 요구 사항과 사용 중인 제품의 기능.

9.2.2 코드 스코어링 서비스 요약

Code Scoring Service는

사용자가 제공한 코드. 그렇더라도 압축을 풀고 실행하는 것은 그리 어렵지 않습니다 Lambda 내의 인터프리터(또는 컴파일러). 한 가지 중요한 언급은 보안. 함수에서 다른 사람의 코드를 실행하는 경우 준비해야 합니다. 누군가가 그것을 전복시키려고 시도하고, 악용할 취약점을 찾아내고, 무언가를 할 것이라는 나쁜. 따라서 최소 권한의 원칙을 따라야 하며 어떤 것도 하용하지 않아야 합니다. 이는 기능 실행에 중요하지 않습니다. 이것은 모든 Lambda에 대한 규칙이어야 합니다. 기능을 수행하지만 이 경우 두 배로 주의하고 경계해야 합니다.

9.3 학생 프로필 서비스

학생 프로필 서비스는 소규모입니다. 그 목적은 DynamoDB 테이블의 학생 레코드에 있는 경험 포인트 수를 늘리는 것입니다. 그렇게 하면 학생은 즉시 집계에 추가된 누적 점수를 확인하고 성취. 이 서비스는 통신하는 단일 Lambda 함수로 구성됩니다. DynamoDB와 함께. 이 함수는 EventBridge에서 이벤트를 수신하고 이를 읽고, 사용자가 포인트를 받은 경우 사용자 프로필을 업데이트합니다. 그림 9.3은 이것이 무엇인지 보여줍니다. 기본 서비스가 보입니다.

이전에(섹션 9.1에서) 서로 다른 테이블을 동기화 상태로 유지하는 것에 대한 질문을 했다는 것을 기억할 것입니다. 학생 프로필 서비스와 분석이

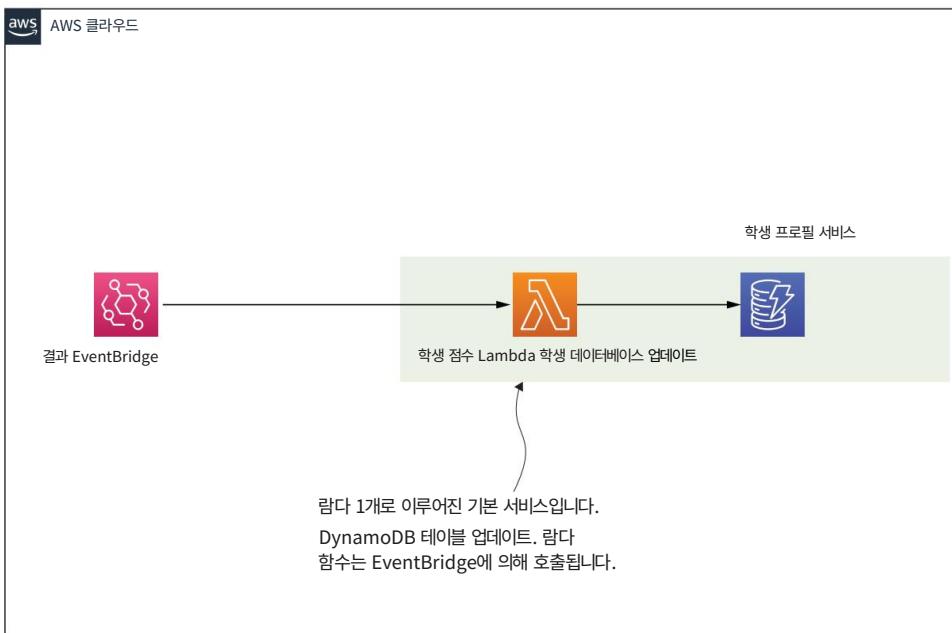


그림 9.3 학생 프로필 서비스는 이 전체 아키텍처에서 가장 간단한 서비스입니다. Dynamo 테이블에 쓰는 Lambda 함수입니다.

서비스는 유사한 데이터(즉, 사용자의 점수)를 저장합니다. 서비스 중 하나가 다른 서비스와 동기화되지 않으면 어떻게 됩니까? 다시 말해, 만약 데이터 불일치를 일으키는 서비스의 오류, 이에 대해 우리는 무엇을 할 수 있습니까? 있다 이 문제를 해결하기 위해 구현할 수 있는 솔루션의 수:

- 직렬 호출 - 한 가지 접근 방식은 분석 서비스와 학생 프로필 서비스를 병렬이 아닌 직렬로 실행하는 것입니다. 그렇게 하면 시스템이 먼저 학생 프로필 서비스를 업데이트한 다음 업데이트를 실행합니다.
- 분석 서비스의 프로시저(다른 EventBridge를 통해 호출). 만약 분석 서비스가 실패하면 시스템이 학생의 변경 사항을 롤백합니다.
- 프로필 서비스와 두 서비스는 계속 동기화되어 작동합니다.
- 하나의 진실 소스—또는 분석 서비스를 데이터를 학생 프로필 서비스에 복사하기만 하면 됩니다. 그렇게 하면 학생 프로필 서비스의 모든 데이터를 삭제할 수도 있고
- 분석 서비스에서 필요한 만큼 재생성합니다.
- 데이터베이스 공유 - 두 서비스 모두 동일한 데이터베이스를 읽고 쓸 수 있습니다. 저것 몇 가지 문제를 피할 수는 있지만 더 이상 각 서비스가 자신의 세계관을 책임지는 마이크로서비스 아키텍처가 없습니다. 우리
- 분산 모노리스로 끝날 것입니다. 많은 상황에서 분산된 단일체를 갖는 것이 훌륭하고 수용 가능한 솔루션이라는 점을 언급해야 합니다.

okestr레이터 - 또 다른 접근 방식은 okestr레이터를 두 서비스 위에 배치하는 것입니다. 그리고 무슨 일이 일어나고 있는지 모니터링하십시오. 오류가 있는 경우 조정자는 문제를 보완하기 위해 추가 작업을 실행할 수 있습니다(예: 재시도 또는 롤백).

솔직히 말해서 이것은 마이크로 서비스 기반 접근 방식의 일반적인 상황입니다. 어떻게 모든 마이크로서비스를 중앙 데이터베이스에 연결하지 않고도 서비스를 동기화 상태로 유지하고 계십니까? 이 문제에 대한 다양한 솔루션이 있지만 소프트웨어의 모든 것과 마찬가지로 엔지니어링, 그들은 모두 다른 트레이드 오프를 가지고 있습니다. CDU의 경우 업데이트하기로 결정했습니다. 두 서비스를 동시에. 문제가 발생하면 분석 서비스를 다음과 같이 사용합니다. 우리의 진실의 근원을 찾고 학생 프로필 서비스에 필요한 데이터를 재생성합니다.

9.3.1 학생 점수 업데이트 기능

학생 점수 업데이트 기능은 목록 9.1에 나와 있습니다. 세 가지 기본 작업을 수행합니다. 행위:

- 점수/데이터가 있는 EventBridge에서 수신한 이벤트를 구문 분석합니다.
- JavaScript 또는 Python과 같은 주제에 대해 획득한 XP의 양을 업데이트합니다.
- 사용자가 획득한 총 XP를 업데이트합니다.

목록 9.1 학생 점수 Lambda 업데이트

```
'엄격한 사용';

const AWS = 요구('aws-sdk');
const sns = 새로운 AWS.SNS();

const dynamoDB = 요구('aws-sdk/clients/dynamodb');
const 문서 = 새로운 dynamoDB.DocumentClient();

const updateTotalXP = (레코드, 수업) => {
    const 날짜 = new Date(Date.now()).toISOString();

    const xp = Lessons.filter(m => m.xp)
        .map(m => m.xp)
        .reduce((a, b) => a+b);

    모든 수업에 대한 XP를 합산하여 사용자의 총 XP를 계산합니다. 이것은 매번 수행하는 것이 비참할 정도로 비효율적이지만 예를 들면 괜찮습니다.
    더 나은 방법이 생각나나요?

    const params = { 테이블 이름:
        process.env.USER_DATABASE,
        열쇠: {
            사용자 ID: 기록.사용자 이름
        },
        UpdateExpression: `set \
            수정된 =:날짜, \
            xp.#총계 =:xp `,
        표현식 속성 이름: {
            '#총': '총'
        },
        표현식 속성값: {
            ':날짜': 날짜,
            ':xp': xp
        },
    };
}

이 params 객체에는 관련 DynamoDB 테이블을 업데이트하는 데 필요한 모든 속성이 있습니다. ExpressionAttributeNames의 '총계'를 확인하세요. 예약된 키워드이므로 ExpressionAttributeNames를 사용하여 지정해야 합니다.
```

```

반환 값: 'ALL_NEW'
};

반환 doc.update(params).promise();
}

const updateTopicXP = (레코드) => {
  const 날짜 = new Date(Date.now()).toISOString();

  const 수업 = {
    수업: 기록.수업,
    주제: 기록.주제,
    수정: 날짜,
    xp: 레코드.xp,
    isCompleted: 기록.isCompleted
  };

  상수 매개변수 = {
    테이블 이름: process.env.USER_DATABASE,
    열쇠: {
      사용자 ID: 기록.사용자 이름
    },
    UpdateExpression: `set \
      수정된 = :날짜, \
      수업 = list_append(if_not_exists(수업,
      :empty_list), :lesson), \
      xp.${레코드.주제} = \
      if_not_exists(xp.${record.topic}, :zero) + :xp` ,
    표현식 속성값: {
      ':레슨': [ 레슨 ], ':empty_list': [], ':zero':
      0,
      ':날짜': 날짜,
      ':xp': parseInt(record.xp, 10)
    },
    반환 값: 'ALL_NEW'
  };

  반환 doc.update(params).promise();
}

export.handler = 비동기(이벤트, 컨텍스트) => {
  노력하다 {
    const 레코드 = event.detail;

    if (record.isCompleted) {
      const 사용자 = updateTopicXP(레코드)를 기다립니다.

      if (user.Attributes.lessons.length > 0) {
        updateTotalXP(레코드, user.Attributes.lessons)를 기다립니다.
      }
    }
  } 잡기(오류) {
    console.log(오류);
  }
}

```

이 업데이트 표현식은 수업 목록에 수업을 추가합니다.

다이나모DB. 그렇지 않고 목록이 없으면 비어 있는 새 목록이 만들어집니다.

이 함수는 다음을 통해 호출됩니다.

이벤트브리지. event.detail 파라미터에는 이전 섹션의 Process Submission Lambda 함수에서 전송된 정보가 포함되어 있습니다.

학생 프로필 서비스는 단일 Lambda 함수가 있는 소규모 마이크로서비스입니다. 그 목적은 DynamoDB 테이블을 업데이트하는 것이며 이는 얻을 수 있는 가장 기본적인 것입니다.
그러나 다음 서비스는 간단하지 않습니다. 이제 살펴보겠습니다.

9.4 분석 서비스 이것은 큰 일

이 될 것이므로 뛰어들기 전에 차나 커피를 준비하십시오. 기억한다면 분석 서비스의 목적은 두 가지입니다.

QuickSight 대시보드 생성 및 표시를 활성화합니다.

빠르게 액세스하고 읽을 수 있는 DynamoDB의 리더보드를 유지 관리 합니다.

분석 서비스에서 수집 및 처리하는 데이터는 당사가 이 두 가지 목표를 달성할 수 있도록 해야 합니다. 그럼 9.4의 아키텍처를 살펴보겠습니다. 분석 서비스가 수행하는 단계는 다음과 같습니다.

1. EventBridge 서비스는 코드 채점 서비스의 메시지를 학생 점수 Kinesis Firehose로 푸시합니다.
2. Firehose는 수신되는 각 메시지를 처리하고 Amazon Glue가 나중에 작업하기에 적합한 형식으로 변환하는 Lambda를 실행합니다.
3. 메시지가 Lambda에 의해 변환된 후 Firehose는 이를 S3 버킷에 저장합니다.
4. 매시간(또는 온디マン드) AWS Glue는 S3 버킷에 저장된 메시지를 실행하고 크롤링합니다. 크롤링을 기반으로 하는 메타데이터로 AWS Glue 데이터 카탈로그 내의 테이블을 업데이트합니다.
5. Glue 처리가 완료되면 Query Leaderboard Summary 기능이 실행됩니다. Lambda 함수는 쿼리를 실행하는 Athena를 호출하여 리더보드를 계산합니다.
6. Athena는 Glue 및 S3에 액세스하고 쿼리에 대한 관련 데이터를 추출합니다. 쿼리가 완료되면 Process Leaderboard Summary Lambda가 호출됩니다.
7. 이 프로세스 리더보드 요약 Lambda 함수는 Athena에서 쿼리 결과를 수신하고 이를 읽고 Leaderboard DynamoDB 테이블을 업데이트합니다.
8. 마지막으로 QuickSight 대시보드 구성 요소는 Athena를 사용하여 쿼리를 실행합니다.

사용자가 QuickSight 보고서에서 보려는 내용을 기반으로 합니다.

한 번에 처리하기에는 이 작업이 상당히 많다는 데 동의할 수 있으므로 가장 흥미로운 구성 요소를 분해해 보겠습니다. 다음 섹션에서 이를 수행합니다.

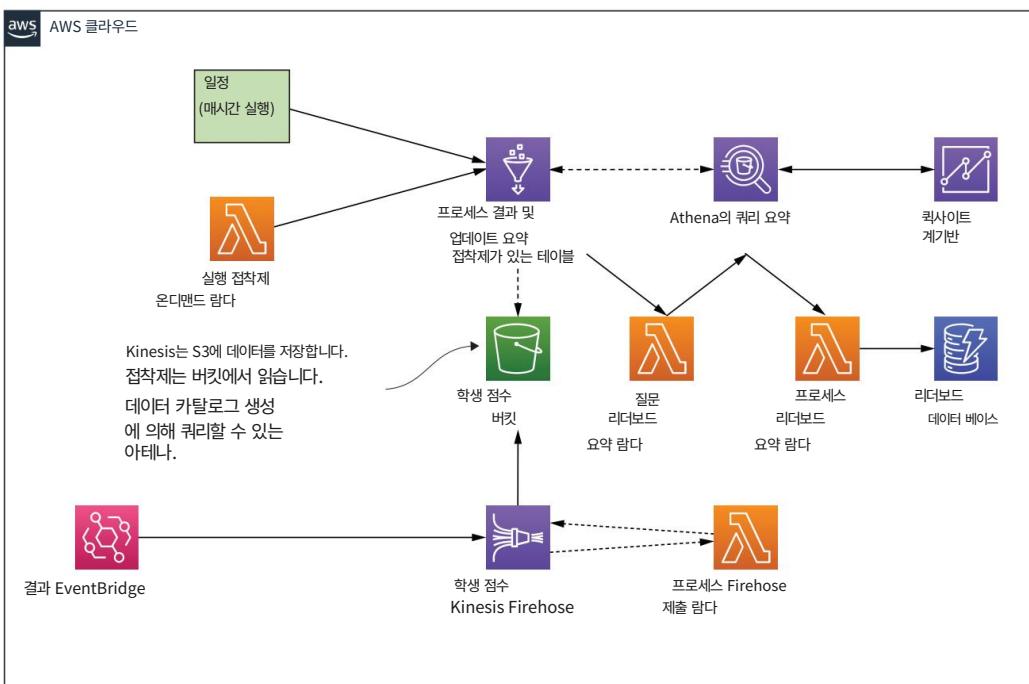


그림 9.4 분석 서비스 아키텍처에는 Glue, Athena, DynamoDB, Kinesis Firehose, QuickSight 및 Lambda가 포함됩니다.

9.4.1 Kinesis Firehose

Kinesis Firehose는 데이터를 캡처하여 Elasticsearch, Redshift 및 S3로 스트리밍하는 방법을 제공합니다.

AWS는 "... 스트리밍 데이터를 안정적으로 로드하는 가장 쉬운 방법

데이터 레이크, 데이터 스토어 및 분석 서비스" (<https://aws.amazon.com/kinesis/data-firehose/>)

, 우리의 사용 사례에 완벽하게 들립니다. Kinesis Firehose는 다른 제품과 달리

Kinesis 서비스는 서비스입니다. 즉, Kinesis Data Stream의 경우처럼 패티션이나 색인을 확장하는 것에 대해 걱정할 필요가 없습니다. 그것은 모두 당신을 위해 이루어집니다

자동으로. Firehose의 또 다른 좋은 기능은 메시지에 대해 Lambda를 실행할 수 있다는 것입니다.

그들이 섭취됨에 따라. Lambda는 원시 스트리밍 데이터를 다른 데이터로 변환하는 데 사용할 수 있습니다.

더 유용하고 형식이 지정되며 이것이 바로 우리가 할 일입니다. 우리의 사용 사례에서는 다음을 사용할 수 있습니다.

메시지를 나중에 읽을 수 있는 JSON 형식으로 변환하는 Lambda 함수

S3에 저장하기 전에 AWS Glue 서비스에 의해.

목록 9.2는 메시지를 받아 처리하고, 다른 필드 세트로 새 레코드를 생성하고, S3에 저장하기 위해 Firehose로 다시 푸시하는 Kinesis Firehose 처리 기능을 보여줍니다. 이 목록에서는 몇 가지 속성만 추출합니다.

모든 것을 유지하고 싶지 않기 때문에 원본 메시지. 예를 들어 사용자가 제출한 소스 코드를 폐기할 수 있습니다.

사용자가 테스트를 통과한 경우에만 관심을 갖기 때문입니다.

아니면. 이 목록에서 염두에 두어야 할 몇 가지 사항이 있습니다.

변환된 모든 레코드에는 recordId, 결과 및 데이터가 포함되어야 합니다. 그렇지 않으면 Kinesis Firehose가 전체 레코드를 거부하고 "변환 실패."

recordId라는 속성은 Firehose에서 Lambda로 전달됩니다. 변환된 레코드는 원본과 동일한 recordId를 포함해야 합니다. 불일치

변환 실패로 이어집니다(따라서 직접 만들거나 아무 것도 추가하지 마십시오. 그것에).

속성 결과는 Ok (레코드 변환) 또는 Dropped (레코드 의도적으로 삭제됨). 유일하게 하용되는 값은 ProcessingFailed입니다.

변환을 수행할 수 없다는 플래그를 지정하려는 경우.

속성 데이터는 base-64로 인코딩된 변환 레코드입니다.

목록 9.2 Kinesis Firehose 처리 기능

'엄격한 사용':

```
export.handler = (이벤트, 컨텍스트) => {
    레코드 = [];

    for (let i = 0; i < event.records.length; i++) {
        const 페이로드 = Buffer.from(
            event.records[i].data, 'base64')
            .toString('utf-8'); const 데이터 =
        JSON.parse(페이로드);

        상수 레코드 = {
            사용자 이름: data.detail.username,
            이름: data.detail.user.name,
            수업: data.detail.lesson,
            주제: data.detail.topic,
            xp: data.detail.xp,
            hasPassedTests: (data.detail.hasPassedTests || 거짓),
            runTests: (data.detail.runTests || 거짓),
            isCompleted: (data.detail.isCompleted || 거짓),
            시간: 데이터.시간,
        };
        기록.푸시({
            recordId: event.records[i].recordId,
            결과: '확인',
            데이터: Buffer.from(JSON.stringify(레코드)).toString('base64')
        });
    }

    console.log(`변환: ${ JSON.stringify({기록}) }`);

    반환 Promise.resolve({
        기록
    });
};
```

Firehose로 푸시된 원본 메시지입니다. 당신은 할 수 있습니다. 이제 S3에 저장하려는 관련 비트를 추출합니다.

여기서 생성하고 S3에 저장하는 레코드는 JSON입니다.

변환된 모든 레코드에는 recordId, result 및 data라는 속성이 포함되어야 합니다. 변환된 레코드는 Lambda의 궁극적인 목표입니다.

마지막으로 응답이 6MB를 초과하지 않도록 해야 합니다. 그렇지 않으면 Firehose 같이 노는 것을 거부할 것입니다.

9.4.2 AWS Glue 및 Amazon Athena

AWS Glue는 검색할 수 있는 서비스 ETL(추출, 변환 및 로드) 서비스입니다.

크롤러가 있는 S3 버킷 및 Glue라는 중앙 메타데이터 저장소 업데이트

데이터 카탈로그. 그러면 귀하와 다른 서비스가 이 메타데이터 저장소를 사용하여 신속하게

S3에 흘어져 있는 기록들 중에서 관련 정보를 검색합니다. Glue는 실제로 데이터를 이동하거나 복사하지 않습니다.

Glue Data Cat 로그에서 생성하는 메타데이터가 있는 테이블은 S3(또는 Amazon Redshift 또는 RDS와 같은 다른 소스)의 데이터를 가리킵니다. 이것

필요한 경우 데이터 카탈로그를 원래 데이터에서 다시 만들 수 있음을 의미합니다.

Amazon Athena는 표준 SQL을 사용하여 S3의 데이터를 분석할 수 있는 서비스 쿼리 서비스입니다.

Athena를 시도하지 않았다면 시도해야 합니다. 당신은 단순히 그것을 가리킵니다

S3, 스키마를 정의하고 SQL을 사용하여 쿼리를 시작합니다. 더 좋은 점은 Glue 및 해당 데이터 카탈로그(스키마를 처리함)와 긴밀하게 통합된다는 것입니다. 한번

AWS Glue를 구성하고 데이터 카탈로그를 생성했다면 쿼리를 시작할 수 있습니다.

바로 아테나.

목록 9.3은 쿼리를 수행하는 방법을 보여줍니다. 주목해야 할 중요한 점은

쿼리는 비동기식입니다. 일단 실행하면 응답이 없습니다. 시작해야 합니다.

쿼리를 실행한 다음 CloudWatch 이벤트를 사용하여

결과. 운 좋게도 두 개의 Lambda 함수로 모든 것을 수행할 수 있습니다. 목록 9.3

쿼리를 실행하는 방법을 보여주고 목록 9.4는 다음이 있는 경우 쿼리를 처리하는 방법을 보여줍니다.

응답하기 위해 CloudWatch 이벤트를 연결했습니다.

Listing 9.3 쿼리 리더보드 요약 Lambda

'엄격한 사용';

```
const AWS = 요구('aws-sdk');
const athena = 새로운 AWS.Athena();

const runQuery = (보기) => {
    상수 매개변수 = {
        QueryString: `SELECT * FROM "${view}"`, QueryExecutionContext: { 카탈로그: process.env.ATHENA_DATA_SOURCE, 데이터베이스: 보기는 Athena에서 지원되며 일반 SQL
            process.env.ATHENA_DATABASE
        },
        작업 그룹: process.env.ATHENA_WORKGROUP
    };
    반환 athena.startQueryExecution(params).promise();
}

export.handler = 비동기(이벤트) => {
    약속하자 = [];
}
```

만큼 유용합니다.

카탈로그, 데이터베이스 및 작업 그룹과 같은 매개변수는 Athena를 구성할 때 설정됩니다.

```

promise.push(runQuery(process.env
    .ATHENA_LEADERBOARD_VIEW_TOPICS));
promise.push(runQuery(process.env
    .ATHENA_LEADERBOARD_VIEW_OVERALL));

const 쿼리 = Promise.all(약속)을 기다립니다.

console.log('Athena 쿼리 ID', 쿼리);
}

```

보기는 Athena에서 지원되며 일반 SQL
만큼 유용합니다.

목록 9.4는 목록 9.3에서 수행된 쿼리에 대한 정보가 포함된 Cloud Watch 이벤트에 응답하는 Process Leaderboard Lambda 함수를 보여줍니다. 메모
실제 결과(데이터 자체를 의미)는 다음을 사용하여 Athena에서 검색해야 합니다.
GetQueryResults API 호출 프로세스 리더보드 요약 기능이 다음과 같을 때
호출되면 queryExecutionId 만 전달되지만 수행하기에 충분합니다.
GetQueryResults API를 호출하여 데이터를 가져옵니다. 다음 목록의 코드는 상당히
Athena에서 결과를 가져오는 방법 외에 DynamoDB 테이블을 업데이트하는 방법을 보여주기 때문입니다.

Listing 9.4 프로세스 리더보드 요약 Lambda

```

'엄격한 사용';

const AWS = 요구('aws-sdk');
const athena = 새로운 AWS.Athena();
const dynamodb = 새로운 AWS.DynamoDB.DocumentClient();

const getQueryResults = (queryExecutionId) => {
    상수 매개변수 = {
        QueryExecutionId: queryExecutionId
    };

    반환 athena.getQueryResults(params).promise();
}

const updateDynamoLeaderboard = (행, 인덱스) => {
    let transactItems = [];
    const 날짜 = new Date(Date.now()).toISOString();

    //

    // 레이블이기 때문에 첫 번째 행을 건너뜁니다.
    // 데이터: [
    //     { VarCharValue: '주제' },
    //     { VarCharValue: '사용자 이름' },
    //     { VarCharValue: '이름' },
    //     { VarCharValue: '점수' },
    //     { VarCharValue: 'm' }
    // ]

    for (let i = 0; i < rows.length; i++) {
        const 행 = 행[i].Data;

```

실행하려면
QueryExecutionId가 필요합니다.
GetQueryResults를 사용하면
쿼리 결과가 사용자의 것입니다.

```

상수 매개변수 = {
    테이블 이름: process.env.LEADERBOARD_DATABASE,
    열쇠: {
        uniqueld: row[1].VarCharValue, //사용자 이름
        유형: row[0].VarCharValue //주제
    },
    UpdateExpression: `set \
        #이름 = :이름, \
        수정된 = :날짜, \
        #순위 = :순위,
        점수 = :점수`,
    표현식 속성 이름: {
        '#이름': '이름',
        '#순위': '순위'
    },
    표현식 속성값: {
        ':날짜': 날짜,
        ':이름': 행[2].VarCharValue,
        ':점수': parseInt(row[3].VarCharValue, 10),
        ':순위': parseInt(row[4].VarCharValue, 10)
    },
    반환 값: 'ALL_NEW'
}

transactItems.push({업데이트: 매개변수});
}

반환 dynamodb.transactWrite({TransactItems:transactItems}).promise();
}

export.handler = 비동기(이벤트) => {

    노력하다 {
        만약 (event.detail
            .currentState === '성공') { const queryExecutionId =
            event.detail.queryExecutionId;

        상수 결과 =
            getQueryResults(queryExecutionId)를 기다립니다.

        result.ResultSet.Rows.shift();

        if (result.ResultSet.Rows.length > 0) {

            const maxItemsPerTransaction = 20;

            for (let i = 0; i <
            result.ResultSet.Rows.length/maxItemsPerTransaction; i++) {

                상수 인자 =
                    result.ResultSet.Rows.length/maxItemsPerTransaction;
                상수 나머지 =
                    result.ResultSet.Rows.length%maxItemsPerTransaction;
}

```

쿼리가 성공적으로 실행된 경우에
만 결과를 검색하고 저장하려고 합
니다.

운 좋게도 이 매개변수는 모든 것이 양호
한지 확인합니다.

배열의 첫 번째 행에는

열(예: 주제, 점수 등). 값에만
관심이 있기 때문에 해당 행을
이동하면 제거됩니다.

20개 항목 단위로 업데이트됩니다.
DynamoDB는 트랜잭션에서 25개
항목을 처리할 수 있지만 여기서는 20
개만 처리합니다.

다시보드를 다시 만드십시오. 가장 중요한 것은 올바른 형식의 데이터와 이 장에서 설명하는 도구와 함께 사용할 수 있습니다.

요약하면, Analytics Service를 구축하기 위해 Kinesis Firehose와 같은 AWS 서비스, Athena와 Glue가 필요할 수 있습니다. 이들은 서비스 서비스입니다. 즉, 필요한 것과 동일한 방식으로 확장하거나 관리하는 것에 대해 생각할 필요가 없습니다.

Amazon Redshift에 대해 생각합니다. 그럼에도 불구하고 서비스를 시작하기로 결정했다면 이러한 서비스를 사용하는 여행은 먼저 평가를 수행해야 합니다.

모든 요구 사항을 충족할 수 있습니까?

귀하의 경우 Amazon Redshift가 더 나은 상황이 있습니까?

Athena의 요금은 각 쿼리에서 스캔한 데이터의 양을 기준으로 합니다. 레드시프트는 인스턴스 크기에 따라 가격이 책정됩니다. 아테나가 있는 상황이 있을 수 있습니다. 더 저렴하지만 Redshift가 더 빠르므로 Excel 예측 비용에 약간의 시간을 할애해야 합니다. 그럼에도 불구하고 많은 경우, 특히 소규모 데이터 세트의 경우 Athena와 Glue의 조합으로 대부분의 요구 사항을 충족하기에 충분합니다.

요약

AWS에는 캡처, 변환, 분석 및 보고하는 다양한 서비스와 방법이 있습니다.

귀하의 응용 프로그램과 관련된 데이터.

Event Bridge, DynamoDB, Amazon Glue, Amazon Athena 및 Amazon QuickSight와 같은 서비스를 사용하여 데이터 캡처, 처리 및 보고

3개의 마이크로서비스로 웹 애플리케이션을 구축하기 위해 다음을 포함하여 몇 가지 요점을 남겨야 합니다. – Amazon QuickSight는 느리고 비용이 많이 들 수 있습니다. 표시해야 하는 경우

리더보드의 경우 빠른 검색을 위해 DynamoDB와 같은 곳에 캐시합니다.

- Glue와 Athena는 환상적인 도구입니다. Glue는 S3에 저장된 데이터를 색인화할 수 있으며,

Athena는 표준 SQL을 사용하여 검색할 수 있습니다. 그 결과 "리프팅"과 코딩이 줄어듭니다.

- Kinesis Firehose에는 레코드를 수정할 수 있는 환상적인 기능이 있습니다.

그들이 가고자 하는 목적지에 도착하기 전에. 이것은 환상적인 입장료의 가치가 있는 기능.

- 사용하지 않는 한 Lambda 함수가 각각을 직접 호출하지 마십시오.

람다 목적지. Lambda의 경우 항상 SQS 또는 EventBridge와 같은 대기열을 사용하십시오. 목적지를 사용할 수 없습니다.

- EventBridge는 AWS 내에서 사용하기에 탁월한 메시지 버스입니다. 말고는

FIFO 기능이 있는 경우(이 내용을 읽을 때 변경될 수 있음)

뛰어난 기능이 많이 있으며 적극 권장합니다.

4부

미래

이 책의 마지막 두 장은 정말 재미있습니다. 다음 챕터는

AWS Lambda는 내부에 있으며 Lambda가 어떻게 작동하는지 알고 싶어하는 모든 사람에게 매력적입니다. 이 책의 마지막 장은 새로운 관행에 관한 것입니다. 여러 AWS 계정 사용, 임시 CloudFormation 스택, 민감한 데이터 관리, 이벤트 기반 아키텍처에서 EventBridge 사용을 다룹니다. 이 두 장은 우리가 가장 좋아하는 것 중 일부입니다. 저희가 작업하는 것을 좋아했던 만큼 여러분도 책을 좋아하시기를 바랍니다.

블랙벨트 람다

이 장에서는 다음을 다룹니다.

서비스 애플리케이션의 대기 시간, 초당 요청 및 동시성 모니터링

지연 시간 최적화 기법

성능(응용 프로그램의 응답 속도) 및 가용성(여부

애플리케이션이 유효한 응답을 제공함)은 최종 사용자 경험의 중요한 측면입니다. 서비스 아키텍처를 사용할 때 성능도 직접적인 영향을 받습니다.

비용에 미치는 영향 예를 들어, AWS Lambda는 기능이 실행되는 기간 동안 사용자가 할당한 메모리에 따라 가중치를 부여하여 비용을 청구합니다. 서비스 아키텍처는 확장과 같은 성능 최적화를 위해 많은 공통 표면 영역을 제거합니다.

사용 가능한 서버 또는 서버 구성 조정으로 인해 어려울 수 있습니다.

새로운 사용자가 이러한 최적화를 수행하는 방법을 이해할 수 있습니다.

이 장에서는 다음 작업에 사용할 수 있는 주요 도구와 접근 방식을 소개합니다.

서비스 애플리케이션을 구성하는 다양한 서비스에서 성능을 향상시킵니다. 관련 예제를 사용하여 이러한 기술이 작동하는 방식을 보여줍니다.

10.1 어디에서 최적화할 것인가?

서비스 아키텍처를 최적화하는 방법을 알아보기 전에 간단히 요약해 보겠습니다.

그들에 대해 생각하는 방법. 서비스 아키텍처에는 여러 개념 계층이 있습니다.

그림 10.1과 같이, 끝점은 최종 사용자 및 장치와의 보안 상호 작용과 최종 사용자의 요청 또는 이벤트 수신을 담당합니다. AWS 서비스 아키텍처에서 사용할 수 있는 엔드포인트의 예로는 API 게이트웨이, AWS IoT, Amazon Alexa(Alexa 기술을 구축하는 경우) 또는 AWS SDK만 포함됩니다.

워크로드의 컴퓨팅 계층은 외부 시스템의 요청(엔드포인트를 통해 수신)을 관리하는 동시에 액세스를 제어하고 요청이 적절하게 승인되었는지 확인합니다. 여기에는 Lambda 함수로 구현된 비즈니스 로직을 배포 및 실행하는 런타임 환경이 포함되어 있습니다(이에 대해서는 곧 자세히 설명하겠습니다).

워크로드의 데이터 계층은 시스템 내에서 영구 저장소를 관리합니다. 비즈니스 로직에 필요한 상태를 저장하는 보안 메커니즘을 제공합니다. 또한 데이터 변경에 대한 응답으로 이벤트를 트리거하는 메커니즘을 제공하여 비즈니스 로직의 다른 부분으로 전달될 수 있습니다. 상상할 수 있듯이 이것은 최적화에 대해 논의할 수 있는 광범위한 표면 영역이므로 그림 10.1에서 강조 표시된 다음 사항에 중점을 둘 것입니다.

기능

이러한 기능의 호출(엔드포인트의 요청 또는 백엔드 시스템의 이벤트를 통해)

함수와 다운스트림 리소스의 상호 작용

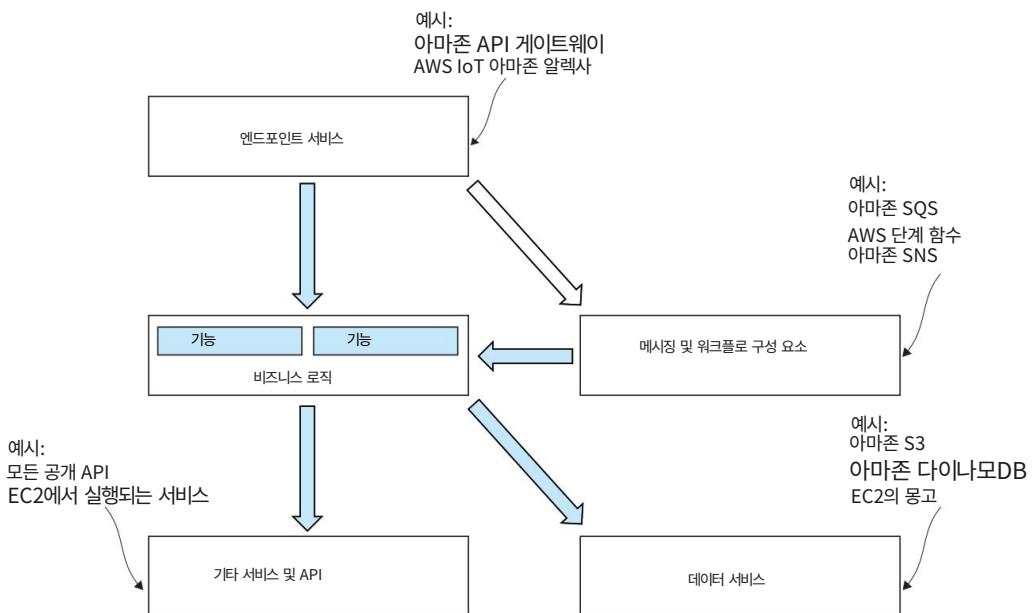


그림 10.1 서비스 애플리케이션의 개념적 아키텍처

이제 다양한 최적화 지점에 대한 개념을 이해했으므로 이를 수행하는 데 사용할 수 있는 도구를 살펴보겠습니다. 이에 대해서는 다음 섹션에서 논의할 것입니다.

10.2 시작하기 전에

응용 프로그램을 효과적으로 최적화하려면 특정 도구와 개념이 있어야 합니다.

익숙한. 이 섹션에서는 Lambda 함수가

실행 및 지연 시간에 미치는 영향, 지연 시간 및 기여자를 관찰하는 방법,

충분한 샘플 데이터를 얻기 위해 함수에 로드를 생성하는 방법.

10.2.1 Lambda 함수가 요청을 처리하는 방법

기능을 최적화하는 방법을 이해하려면

Lambda가 우리의 기능을 실행하는 방법. 무엇을 설명하기 위해 예를 사용합시다.

함수가 배포될 때 발생합니다. image-resizer-service 응용 프로그램을 사용할 것입니다.

서비스 애플리케이션 저장소 (<http://mng.bz/WBy4>)에서 참고로. 이것

서비스 애플리케이션은 이미지를 읽는 미국 동부(버지니아 북부)east-1 리전의 AWS 계정에 Lambda 함수 (Node.js로 작성) 및 API Gate 방식을 배포합니다.

S3 버킷(배포 시 이름이 정의됨)에서 이를 통해 제공

API 게이트웨이. 이 함수는 ImageMagick 라이브러리를 사용하여 이미지를 처리합니다.

참고 사용할 애플리케이션에 대해 새 버킷 이름을 지정해야 합니다. 사용

이 예에서는 "image-resizer-service-demo"라는 이름을 사용합니다.

배포가 완료되면 페이지에서 앱 테스트 버튼을 클릭하면 다음 페이지로 이동합니다.

새로 배포된 것을 볼 수 있는 Lambda 콘솔의 애플리케이션 목록 보기

애플리케이션. 그림 10.2에서 이들은 각각 (1)과 (2)로 표시되어 있습니다.

| Name | Description |
|--|--|
| serverlessrepo-image-resizer-service | Serverless REST API for image resizing |
| serverlessrepo-aws-lambda-power-tuning | |
| sls-transcoder-pete-dev | The AWS CloudFormation template for this application |

그림 10.2 애플리케이션 보기에는 배포된 모든 서비스와 애플리케이션이 표시됩니다.

응용 프로그램을 테스트하려면 기본 기능으로 이동해야 합니다. 응용 프로그램을 클릭하고 자세히 보기(그림

10.3)를 클릭합니다. 이미지의 ResizeFunction(1)을 클릭하여

기능에 액세스합니다.

serverlessrepo-image-resizer-service

Overview | Deployments | Monitoring

▶ Getting started Dismiss

API endpoint

Endpoint
<https://vrtd72fh6i.execute-api.us-west-1.amazonaws.com/production>

Resources (6) G

Filter by tags and attributes or search by keyword

| Logical ID | Physical ID | Type | Last modified |
|----------------|---|--------------------|---------------|
| Api | vrtd72fh6i | ApiGateway RestApi | 5 days ago |
| ResizeFunction | serverlessrepo-image-resizer-servic-ResizeFunction-JMzXqwYaaSV6 | Lambda Function | 5 days ago |

그림 10.3 이미지 리사이저 서비스의 애플리케이션 상세 보기

기능을 선택하면 기능 개요 페이지로 이동합니다(그림 10.4). 여기에서 테스트(1)를 선택하여 기능을 테스트 할 수 있지만 먼저 함수에 제공할 샘플 이벤트를 구성해야 합니다(2).

▶ Function overview Info

1 2

Code | **Test** | Monitor | Configuration | Aliases | Versions

Test event For

Invoke your function with a test event. Choose a template that matches the service that triggers your function, or enter your event document in JSON

New event 2

Saved event

Template

hello-world

Name

SampleImage

```

1  {
2   "Records": [
3     {
4       "eventVersion": "2.0",
5       "eventTime": "1970-01-01T00:00:00.000Z",
6       "requestParameters": {
7         "sourceIPAddress": "127.0.0.1"
8       },
9       "s3": {
10        "configurationId": "testConfigRule",
11        "object": {
12          "eTag": "0123456789abcdef0123456789abcdef",

```

Feedback English (US) ▾

© 2008 - 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.

그림 10.4 기능 개요 페이지에서 기능을 사용자 정의하고 실행할 수 있습니다.

목록 10.1의 테스트 이벤트를 사용하여 기능을 테스트할 수 있습니다. 그러나 먼저 https://commons.wikimedia.org/wiki/File:Happy_smiley_face.png에서 파일을 업로드하십시오. image-resizer-service-demo 버킷으로 이동합니다. 다른 이미지를 업로드하도록 선택한 경우 이 목록의 키 필드에서 개체 이름을 변경해야 합니다. 함수가 존재하지 않는 객체를 찾는 데 오류가 발생하지 않도록 이 작업을 수행해야 합니다!

목록 10.1 샘플 이벤트 추가하기

```
{
  "기록": [
    {
      "eventVersion": "2.0", "eventTime": "1970-01-01T00:00:00.000Z", "requestParameters": {

        "sourceIPAddress": "127.0.0.1"
      },
      "s3": {
        "configurationId": "testConfigRule", "개체": {

          "eTag": "0123456789abcdef0123456789abcdef", "sequencer": "0A1B2C3D4E5F678901", "key": "Happy_smiley_face.png", "크기": 1024
        }
      },
      "버킷": {
        "arn": "arn:aws:s3:::image-resizer-service-demo", "name": "image-resizer-service-demo",
        "ownerIdentity": { "principalId": "예"
        }
      },
      "s3SchemaVersion": "1.0"
    },
    {
      "응답 요소": {
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklmдаісауесоме/mnopqrstuvwxyz    wxyzABCDEFGH", "x-amz-request-id": "EXAMPLE123456789", "awsRegion", "us-event-Name": "us-event-id" : "ObjectCreated:Put", "userIdentity": { "principalId": "EXAMPLE" }, "eventSource": "aws:s3"
      }
    }
  ]
}
```

함수를 몇 번 호출하여 동작을 평가합니다. 우리는 이 함수를 사용하여 뒤따르는 다양한 최적화에 대해 논의할 것입니다. 이 함수를 호출하면 Lambda 컴퓨팅 기판, 실행 환경 및 함수 코드와 같은 다양한 계층이 작동합니다 (그림 10.5). 기질은 당신에게 보이지 않습니다. 처럼

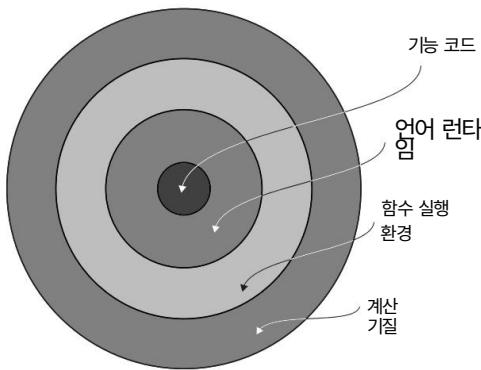
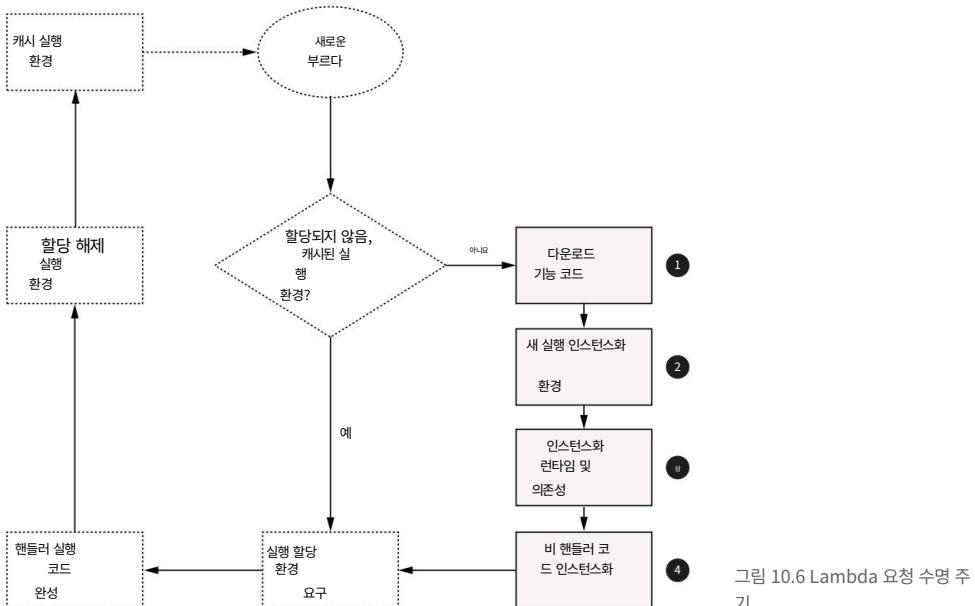


그림 10.5 기능 실행에 관련된 계층

환경은 규모 이벤트(예: 요청 버스트)에 대한 요청 시 인스턴스화됩니다. 그만큼 기능 코드는 모든 요청에 대해 인스턴스화됩니다.

함수에 대한 첫 번째 요청 또는 이벤트가 도착하면 AWS Lambda 서비스가 일련의 단계를 수행합니다. 환경이 존재하면 Lambda는 내부에서 코드를 실행합니다. 당신의 기능 핸들러. 그림 10.6은 다음과 같은 단계를 보여줍니다.

1. Lambda 함수 Node.js 코드를 컴퓨팅 부분에 다운로드합니다.
코드가 실행될 기판.
2. 새 실행 환경을 인스턴스화합니다(크기는 함수 allo를 기반으로 합니다.
양이온) Node.js 런타임으로.
3. 비기능 종속성을 인스턴스화합니다(이 경우 ImageMagick).
4. 핸들러 외부에 작성된 함수의 일부를 실행합니다.
이 예에서는).



`ResizeFunction` 예제에서는 함수 핸들러가 실행될 때 이미지를 생성하고 이미지 메타데이터를 반환합니다. Lambda는 핸들러 로직(및 내부에서 생성된 모든 스레드)이 요청 처리를 완료한 것으로 간주합니다.
함수 핸들러가 실행을 완료합니다. 그러나 요청이 완료되면 AWS Lambda는 실행 환경을 버리지 않습니다(런타임 및 코드가 초기화됨). 대신 실행 환경을 캐시합니다.

실행 환경이 일시 중지됩니다. AWS는 방법에 대한 공식 지침을 게시하지 않습니다.
오랫동안 환경이 이 상태로 유지되지만 다양한 게시된 실험 (<https://www.usenix.org/conference/atc18/presentation/wang-liang>) 5분에서 20분 사이를 보여주세요.

이 시간 동안 후속 요청이 도착하고 캐시된 실행 환경을 사용할 수 있는 경우 AWS Lambda는 해당 실행 환경을 재사용하여 서비스를 제공합니다.

요청. 반면에 캐시된 실행 환경을 사용할 수 없는 경우

AWS Lambda는 요청을 처리하기 위해 모든 단계를 반복합니다. 이것은 함수의 성능과 함수를 작성하는 방법 모두에 중요한 의미를 갖습니다. 잘

이 장의 뒷부분에서 이에 대해 자세히 설명합니다. 기억해야 할 한 가지 중요한 행동은

AWS Lambda는 항상 실행 환경당 하나의 요청만 실행합니다. 이것은 의미

모든 실행 환경이 요청을 처리하고 있고 새로운 환경이 들어오는 경우

AWS Lambda는 새로운 실행 환경을 인스턴스화합니다.

10.2.2 대기 시간: 콜드 vs. 월

이 예(그림 10.6)에서 1~4단계로 인해 발생하는 지역 시간을 참조하십시오.

일반적으로 콜드 스타트 패널티로 사용됩니다. 요청에 대한 요청 대기 시간을 참조합니다.

콜드 스타트를 콜드 레이턴시라고 하며 실제 함수 실행을 참조합니다.

대기 시간을 월 대기 시간으로. 참고로 콜드 스타트 패널티는 두 가지 경우에만 발생합니다.

상황. 먼저 함수가 이전에 호출된 적이 없는 경우 콜드 스타트가 표시됩니다.

또는 장기간(캐시된 모든 실행 환경이 제거됨) 후에 호출됩니다. 둘째, AWS Lambda가 새로운 실행 환경을 생성해야 할 정도로 수신 요청 비율이 증가하면 콜드 스타트가 나타납니다.

사용 가능한 모든 항목이 요청을 처리하고 있기 때문입니다.

대부분의 프로덕션 시나리오에서 콜드 스타트는 요청의 0.5% 미만에 영향을 미치지만

콜드 스타트는 드물게 호출되는 기능과 트래픽 버스트가 있는 기능에 불균형적으로 영향을 미칩니다(특히

트래픽 증가). 콜드 스타트를 경험하는 요청에도 시간 초과가 발생할 수 있습니다.

AWS Lambda 제한 시간 설정이 총 요청 지역 시간에 적용되기 때문입니다.

10.2.3 함수 및 애플리케이션에 대한 부하 생성

응용 프로그램을 최적화할 때 부하 담당자에서 수행하려고 합니다.

실생활 사용의. 보시다시피 대기 시간 특성은 다음과 같이 부하에 따라 다를 수 있습니다.

잘. Serverless-artillery는 Nordstrom 오픈 소스 프로젝트입니다. 그것은 artillery.io를 기반으로 하며 serverless.com은 AWS의 수평적 확장성과 사용한 만큼만 지불하는 특성을 사용하여

서비스에 임의의 부하를 즉각적이고 저렴하게 던지고 보고하는 Lambda

결과를 InfluxDB 시계열 데이터베이스에 표시합니다(다른 보고 플러그인 사용 가능). 이것

기능은 CI/CD 파이프라인 초기에 모든 커밋에 대한 성능 및 부하 테스트를 제공하므로 성능 버그를 즉시 포착하고 수정할 수 있습니다. <https://github.com/Nordstrom/serverless-artillery-workshop> 은 도구 사용 및 설정에 대한 자세한 [안내](#) 를 제공합니다.

10.2.4 성능 및 가용성 추적

측정할 수 없는 것은 최적화할 수 없습니다. 대기 시간을 줄이고 서비스 애플리케이션의 가용성을 개선하는 방법을 알아내기 전에 이 정보를 모니터링하기 위한 일관된 접근 방식이 있어야 합니다. AWS는 이 작업을 위해 다양한 기본 도구와 타사 도구를 모두 제공합니다. 사용 가능한 항목을 보려면 AWS Lambda 콘솔에서 함수를 선택하고 AWS Lambda 콘솔의 함수 목록에서 함수를 클릭한 다음 모니터 탭으로 이동합니다(그림 10.7). 선택한 페이지의 CloudWatch 지표(1), CloudWatch 로그(2) 및 AWS X-Ray(3)의 세 가지 도구를 즉시 사용할 수 있습니다.

모니터 탭에서는 필요한 모든 메트릭과 통찰력에 액세스할 수 있습니다. 여기에서 관련 CloudWatch 로그 및 X-Ray 추적에 액세스할 수도 있습니다.

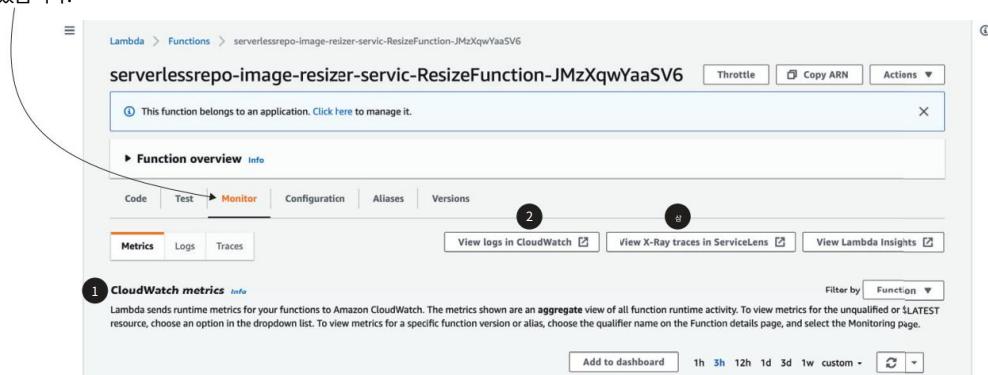


그림 10.7 모니터링을 위한 세 가지 도구를 보여주는 AWS Lambda 함수의 모니터링 탭

이 장에서는 CloudWatch 지표와 X-Ray를 두 가지 기본 도구로 사용하여 애플리케이션의 자연 시간 특성을 관찰합니다.

CloudWatch 측정 항목

각 서비스 서비스(예: AWS Lambda 및 API Gateway)는 성능 및 가용성 특성을 이해하는 데 도움이 되는 표준 지표를 내보냅니다. Lambda의 경우 AWS는 무엇보다도 다음과 같은 지표를 제공합니다.

호출 - 주어진 함수에서 수신한 총 요청 수입니다. 여기에는 성공적으로 완전히 처리되었는지, 조절되었는지 또는 오류가 발생했는지 여부와 관계없이 모든 요청이 포함됩니다. 여기에는 Lambda의 기본 제공 재시도 정책으로 인해 재시도된 모든 요청도 포함됩니다(나중에 자세히 설명).

지속 시간 - 함수 코드가 호출로 인해 실행을 시작할 때부터 실행이 중지될 때까지 경과된 벽시계 시간을 측정합니다. 이는 AWS Lambda가 청구 기간을 가장 가까운 1밀리초로 반올림하기 때문에 정확하지는 않지만 함수에 청구될 항목에 대한 합리적인 프록시입니다.

오류 - 오류로 인해 실패한 호출 수를 측정합니다.

기능. 이것은 AWS Lambda 서비스의 문제 또는 조절로 인한 오류를 측정하지 않습니다.

스로틀 - 함수가 동시성 한도에 도달했거나 계정이 동시성 한도에 도달했기 때문에 함수 코드가 실행되지 않은 호출 수를 측정합니다(동시 실행 1,000개가 기본 한도이지만 AWS에 문의하여 늘릴 수 있습니다.).

AWS 엑스레이

AWS X-Ray는 AWS Lambda 함수의 성능 문제를 감지, 분석 및 최적화하고 서비스 아키텍처 내의 여러 서비스에서 요청을 추적할 수 있는 서비스입니다. X-Ray는 각 기능이 수신하는 요청 샘플에 대한 추적을 생성합니다. 여기서 추적은 요청이 통과하는 각 서비스에 대한 세그먼트로 구성됩니다. 세그먼트에는 요청 대기 시간에 추가된 서비스의 특정 측면을 자세히 설명하는 하위 세그먼트가 포함될 수 있습니다. X-Ray를 켜려면 기능의 구성 탭에 있는 모니터링 및 작업 도구에서 활성 추적을 활성화해야 합니다.

예를 들어, 그림 10.8은 간단한 샘플 애플리케이션에 대한 추적을 보여줍니다. Lambda에서 소요된 총 시간(1), 함수를 실행하는 데 걸린 시간(2), 콜드 스타트에 소요된 시간(3)을 볼 수 있습니다. X-Ray는 최고 기여자가 콜드 스타트인지 여부를 포함하여 함수 실행의 병목 지점을 파악하는 데 유용한 도구가 될 수 있습니다.

AWS X-Ray는 성능 문제를 발견하고 애플리케이션 성능을 개선하는 데 적합합니다.

서비스 애플리케이션을 최적화하는 데 도움이 되는 유용한 정보를 많이 볼 수 있습니다.

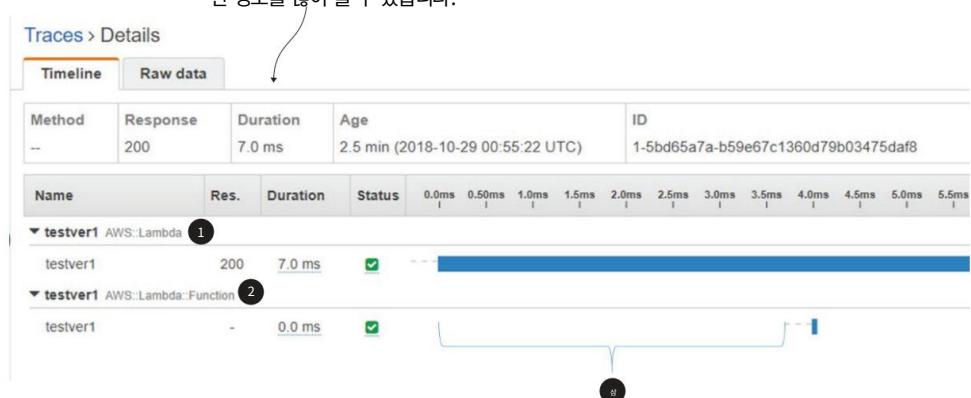


그림 10.8 샘플 애플리케이션에 대한 X-Ray 추적

타사 도구

NewRelic(<https://newrelic.com/>)과 같은 잘 확립된 회사와 Epsagon (<https://epsagon.com>) 과 같은 서비스 우선 회사의 성능 및 가용성 모니터링에도 사용할 수 있는 비 AWS 도구 에코시스템이 성장하고 있습니다. . /).

이 장에서는 이러한 도구에 대해 자세히 설명하지 않겠지만 <https://aws.amazon.com/lambda/partners/> (AWS Lambda 파트너 페이지)에서 모든 옵션을 살펴보고 가장 적합한 것을 선택하는 것이 좋습니다.. .

CloudWatch 로그에 대한 참고 사

항 이전 장에서 설명한 것처럼 CloudWatch 로그는 Lambda 함수 내에 지정된 모든 로그 활동을 캡처합니다. CloudWatch 로그는 다음 두 가지 추가 방법으로 사용할 수도 있습니다.

사용자 지정 지표의 데이터 원본으로 사용 - 예를 들어 Lambda 함수의 특정 메서드 내에서 소요된 시간에 대한 데이터 포인트를 내보내고 해당 정보를 CloudWatch 지표의 사용자 지정 지표로 시각화 및 경보할 수 있습니다.

타사 도구에 데이터를 표시하기 위한 다리 역할 - CloudWatch 로그를 사용하면 NewRelic과 같은 타사 도구에 데이터를 쉽게 보낼 수 있으며, 이를 통해 추가 시각화 및 추적을 제공할 수 있습니다. Lambda는 AWS Lambda Extensions를 통해 직접 타사 애이전트를 포함하는 것을 지원하지만 CloudWatch 로그는 여전히 다른 서비스에 운영 정보를 표시하는 쉬운 방법입니다.

10.3 대기 시간 최적화 이제 애플리케이션에 부하를 생성하는 방법, 대기 시간을 관찰하기 위한 도구를 이해했습니다. 이 섹션에서는 이를 개선하는 방법에 대해 설명합니다.

대기 시간을 최적화하기 위한 최선의 노력은 개별 기능 내에서 이루어집니다. 애플리케이션의 핵심 접착제 및 논리 구성 요소로서 기능에 대한 변경 사항은 고객이 경험하는 대기 시간과 전체 애플리케이션 비용에 직접적이고 즉각적인 영향을 미칠 수 있습니다. 예를 들어, 기능 실행 시간을 10% 줄이면 기능 비용이 10% 감소하며, 이는 대규모에서 중요할 수 있습니다. 최적화할 백분위수와 숫자에 대한 결정은 고객에 따라 귀하의 선택입니다. 예를 들어 웹 사이트를 구축하는 경우 응답 시간이 99번째 백분위수에서 2초 미만이 되기를 원합니다. 백엔드 API를 실행하는 경우 99번째 백분위수에서 10초의 응답 시간을 허용할 수 있습니다.

10.3.1 배포 아티팩트 크기 최소화

배포 패키지의 크기는 두 가지 방식으로 콜드 스타트 패널티에 직접적인 영향을 미칩니다. 참고로 AWS Lambda가 "콜드" 호출에서 수행하는 단계 중 하나는 코드를 다운로드하는 것입니다(그림 10.9의 1단계).

함수가 클수록 이 단계는 더 오래 걸립니다. 간단합니다! AWS Lambda는 함수의 배포 패키지에 대해 250MB의 제한을 적용하므로 자연스러운

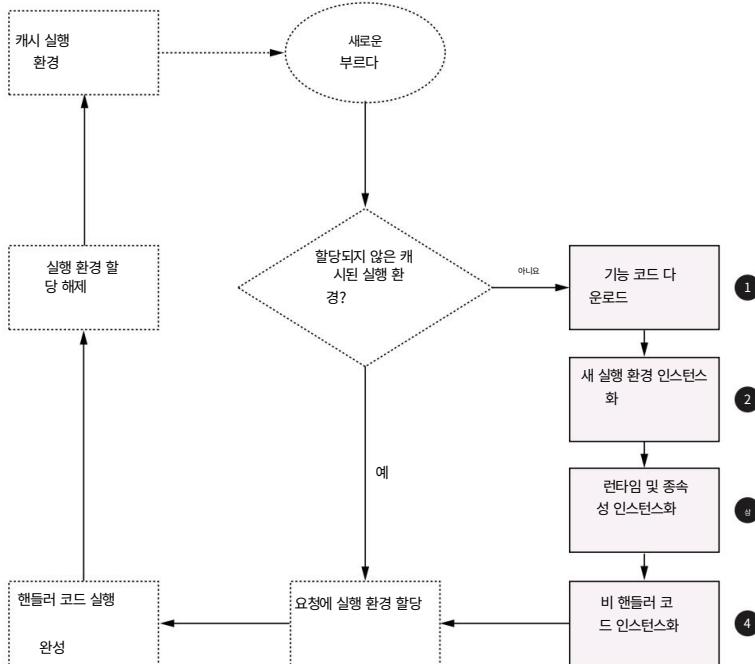


그림 10.9 Lambda 실행 요청 수명 주기

배포 패키지에 대한 "최악의 경우" 영향. 둘째, Java 및 C#과 같은 컴파일된 언어로 작성된 함수의 경우 종속성이 많은 대규모 배포 패키지는 CLASSPATH에 로드할 클래스가 많을 때 인스턴스화하는 데 시간이 더 오래 걸립니다. 예를 들어, Java의 간단한 "hello world"는 약 0.1초 만에 JVM에서 429개의 클래스만 로드하는 반면 Clojure를 사용하여 동일한 "hello world"를 수행하면 1,988개의 클래스를 로드합니다. 3배 더 많고 약 1초가 걸립니다.

따라야 할 모범 사례는 모든 기능 종속성을 감사하는 것입니다. 제거할 수 있는 중량 라이브러리 종속성이나 사용할 수 있는 경량 버전이 있습니까? 특히 HTTP 서버 또는 에이전트 역할을 하는 라이브러리를 찾으십시오. Lambda는 서버 역할을 하기 때문에 Lambda 함수 내에서는 아무 소용이 없습니다. 예를 들어 기본 Java Spring 라이브러리를 사용하는 대신 간소화 된 <https://github.com/awslabs/aws-serverless-java-container> 라이브러리를 사용할 수 있습니다. 이는 실험에서 약 30% 더 빠릅니다. 이 예에서는 전체 AWS SDK를 패키징하는 대신 S3에 액세스하는 데 필요한 SDK만 포함할 수 있습니다. <https://npm.anvaka.com/> 과 같은 도구를 사용하여 Node.js에 대한 종속성을 감사할 수 있습니다. <https://pypi.org/project/modulegraph/>를 사용하는 Python의 경우, 또는 Maven 종속성 트리를 사용하는 Java의 경우.

또한 언어는 배포 패키지 크기를 줄이기 위한 특정 도구를 제공합니다. 예를 들어 Node.js에 minify를 사용할 수 있습니다 (<https://www.npmjs.com/package/node-minify>). Node.js 함수 패키지의 전체 크기를 줄이려면 당신은 또한 사용할 수 있습니다.

프로가드 (<https://www.guardsquare.com/en/products/proguard>) 줄이기 위해 Java 배포 패키지(JAR 파일)의 크기입니다.

10.3.2 실행 환경에 충분한 리소스 할당

코드를 실행하려면 컴퓨팅 리소스(CPU, 메모리)가 필요합니다. AWS Lambda 제공 기능에 필요한 리소스를 설정하는 단일 단계: 메모리 설정. 너 AWS 콘솔에서 Lambda 함수를 열고 다음을 선택하여 이 설정을 변경할 수 있습니다. 구성 탭을 클릭한 다음 일반 구성 옆에 있는 편집을 선택합니다. 너 그런 다음 다른 메모리 할당을 실험할 수 있습니다(그림 10.10의 1). 당신은 할 수 있습니다 또한 API 및 CLI를 통해 동일한 값을 설정합니다.

2020년 12월경에 AWS Lambda는 새로운 기준 Lambda 함수에 대해 10,240MB의 메모리(및 6개의 vCPU)를 지원하기 시작했습니다.

AWS에 따르면 10GB의 메모리와 6개의 vCPU가 있는 Lambda 함수는 머신 러닝, 모델링, 유전체학, 더 전통적인 ETL 및 미디어 처리 애플리케이션에 특히 유용할 수 있습니다.

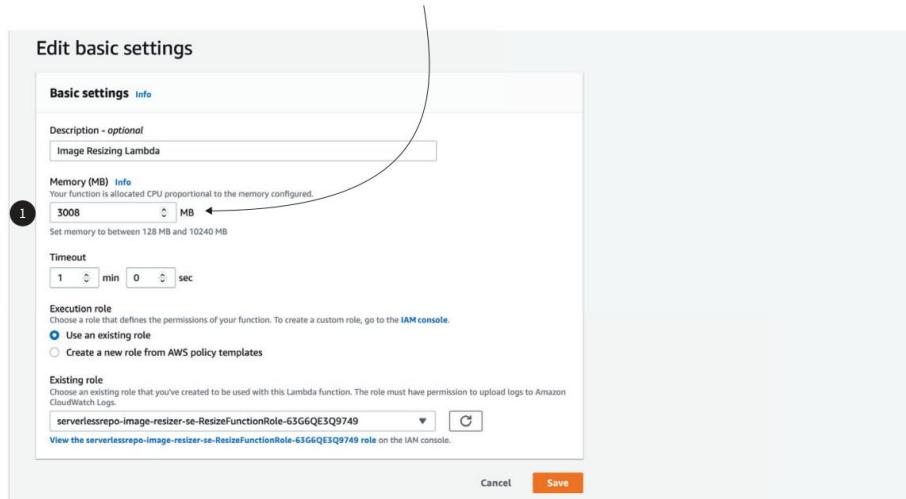


그림 10.10 기본 설정 편집에서 기능에 할당된 메모리 양을 조정할 수 있습니다.

AWS Lambda는 동일한 메모리를 사용하여 메모리에 비례하여 CPU 전력을 할당합니다.

M3 유형과 같은 범용 Amazon EC2 인스턴스 유형으로 비율. 예를 들어 256MB 메모리를 할당하면 Lambda 함수는 CPU를 두 배 받습니다.

128MB만 할당하는 경우보다 공유합니다. 구성을 업데이트하고 요청할 수 있습니다.

추가 메모리는 128MB에서 10240MB로 64MB씩 증가합니다. 이 변경 사항은

무료 아님: AWS Lambda 요금은 함수에 대해 청구된 기간에 가중치를 부여합니다.

메모리 설정: 1024MB에서 1초의 기능 실행 시간은 8과 동일

128MB에서 실행 시간(초).

image-resizer-service 함수에 대한 메모리 설정을 실험해 보겠습니다.

영향을 볼 수 있도록 생성했습니다(아직 생성하지 않은 경우 이 섹션 앞부분의 섹션 10.2.1 참조).
장). 메모리를 128MB, 256MB, 512MB 및 1024MB로 설정하고 몇 가지를 실행합니다.

테스트는 콘솔을 사용하여 호출합니다(최소 10개 권장). 이제 CloudWatch 지표에서 해당 호출에 대한 평균 실행 시간을 확인합니다. 결과를 봐야 합니다

표 10.1과 유사합니다. 예상 비용은 AWS Lambda 공개 기준입니다.

함수에 대한 1,000개 요청에 대한 가격입니다.

표 10.1 1,000개 요청에 대한 예상 비용

| 메모리 | 지속 | 1,000개 요청에 대한 예상 비용 |
|--------|------------|---------------------|
| 128MB | 11.722965초 | \$0.024628 |
| 256MB | 6.678945초 | \$0.028035 |
| 512MB | 3.194954s | \$0.026830 |
| 1024MB | 1.465984초 | \$0.024638 |

이 경우 메모리를 늘리면 비용이 비교적 일정하게 유지되는 반면

성능을 ~10배 증가시킵니다. 일반적으로 이미지 처리와 같은 CPU 바운드 기능에 대해 이러한 종류의 이득을 볼 수 있습니다.
더 많은 리소스가 기능 실행에 도움이 될 수 있습니다.

비용을 변경하지 않고 더 빠르게. I/O 바운드 작업(예: 대기 작업)의 경우

다운스트림 서비스가 응답하는 경우) 리소스를 늘려도 아무런 이점이 없습니다.

배당. Node.js 및 Go와 같은 가벼운 실행 시간의 경우 다음을 줄일 수 있습니다.

가장 낮은 설정(128MB); Java 및 C#과 같은 런타임의 경우

256MB는 런타임이 함수 코드를 로드하는 방법에 해로운 영향을 미칠 수 있습니다.

기능에 적합한 리소스 할당을 찾으려면 약간의 실험이 필요합니다. 가장 쉬운 방법은 높은 설정으로 시작하여 변경 사항이 나타날 때까지 낮추는 것입니다.

성능 특성에서. <https://>에서 인기 있는 튜닝 도구를 사용할 수 있습니다 .

github.com/alexcasalboni/aws-lambda-power-tuning 함수를 조정하는 데 도움이 됩니다.

자원 사용.

콜드 스타트 중 리소스 할당

AWS Lambda는 함수를 실행하는 동안 리소스 할당을 존중하지만
함수를 로드하고 초기화하는 동안 사용 가능한 CPU를 "부스트"하려고 시도합니다.
의존성. 이는 리소스 할당을 늘리는 것이 실제로
당신의 콜드 스타트에 차이를 만드십시오.

10.3.3 기능 로직 최적화

AWS Lambda는 함수 실행이 시작되는 시간을 기준으로 사용량을 청구합니다.

소비된 CPU 주기나 다른 시간 기반 메트릭이 아닌 실행이 중지된 시간입니다. 이것
그 시간 동안 당신의 기능이 하는 일이 중요하다는 것을 의미합니다. 고려하다

이미지 크기 조정기 서비스 기능. S3 객체를 다운로드할 때 코드는 단순히 S3 서비스가 응답하기를 기다리고 있으며 해당 대기 시간에 대해 비용을 지불하고 있습니다. 이 함수의 경우 소요된 시간은 무시할 수 있지만 응답 시간이 긴 서비스(예: 프로비저닝되는 EC2 인스턴스 대기) 또는 대기 시간(매우 큰 파일 다운로드 등)의 경우 이 대기 시간이 과도해질 수 있습니다. . 이 유휴 시간을 최소화하기 위한 두 가지 옵션이 있습니다. 코드에서 오케스트레이션 최소화 - 함수 내에서 작업을 기다리는 대신 AWS Step 함수를 사용하여 "이전" 및 "이후" 로직을 두 개의 개별 함수로 분리합니다. 예를 들어, API 호출 전후에 실행해야 하는 로직이 있는 경우 두 개의 개별 함수로 시퀀스를 지정하고 AWS Step 함수를 사용하여 이들 사이를 오케스트레이션합니다.

I/O 집약적 작업에 스레드 사용 - 모든 컴퓨팅 환경에서 실행되는 코드와 마찬가지로 Lambda 함수 내에서 여러 스레드를 사용할 수 있습니다(프로그래밍 언어가 지원하는 경우). 그러나 기존 프로그램과 달리 다중 스레딩의 가장 좋은 용도는 계산을 병렬화하는 것이 아닙니다. 이는 Lambda가 1.8GB 미만의 메모리로 실행되는 Lambda 함수에 여러 코어를 할당하지 않기 때문입니다. 따라서 병렬화 이점을 얻으려면 더 많은 리소스를 할당해야 합니다. 대신 I/O 작업을 병렬화하는 방법으로 스레드를 사용할 수 있습니다. 예를 들어, `image_resizer` 함수의 Python 버전은 챕터일에 대한 별도의 스레드에서 S3 다운로드를 실행하여 여러 함수에서 작동할 수 있습니다.

이러한 모범 사례를 따르면 서비스 애플리케이션의 대기 시간(및 비용!)을 크게 줄일 수 있습니다. 마지막으로 동시성을 살펴보고 다음 섹션에서 이에 대해 설명합니다.

10.4 동시성

AWS Lambda 함수에 대해 이해해야 할 또 다른 중요한 개념은 동시성입니다.

동시성은 Lambda 함수의 규모 단위입니다. 덮개 아래에는 요청에 할당된 실행 환경의 수에 맵핑됩니다. 다음 공식을 사용하여 언제든지 함수의 동시성을 추정할 수 있습니다.

동시성 = 초당 요청 수(TPS) * 함수 기간

최대값을 사용하면 최대 동시성을 얻을 수 있습니다. 평균 값을 사용하면 평균 동시성을 얻을 수 있습니다. `ConcurrentExecutions` CloudWatch 지표를 사용하여 주어진 함수(및 전체 계정)에 대한 동시성을 모니터링할 수 있습니다. AWS Lambda는 함수의 동시성에 대해 두 가지 제한을 적용합니다.

계정 내 모든 기능의 총 동시 실행에 대한 계정 전체 소프트 제한이 있습니다. 작성 시 기본적으로 1,000으로 설정되어 있으며, 지원 케이스를 통해 원하는 값으로 올릴 수 있습니다. `GetAccountSettings` API를 사용하고 `AccountLimit` 개체를 보면 계정 수준 설정을 볼 수 있습니다.

동시 실행을 확장할 수 있는 비율에 대한 계정 전체 제한도 있습니다. 더 큰 AWS 리전에서는 3,000개의 동시 실행으로 즉시 확장한 다음 이후 1분마다 500개의 동시 실행을 추가할 수 있습니다. 이 제한은 더 작은 지역에서 더 낮습니다. 이러한 제한은 변경될 수 있으므로 <http://mng.bz/80PZ> 에 나열된 최신 값을 참조하십시오.

따라서 최대 및 평균 동시성 요구 사항이 무엇인지, 얼마나 빨리 증가해야 하는지, 필요에 따라 한도 증가 요청을 제출해야 하는지를 항상 추정하는 것이 중요합니다.

10.4.1 요청, 대기 시간 및 동시성 간의 상관 관계

대부분의 함수에서 동시성은 함수 및 계정에 대한 동시성 제한에 따라 요청 및 함수 기간의 함수로 증가합니다. 그러나 스트림 데이터를 처리하는 데 사용되는 함수(Kinesis 및 DynamoDB 스트림)의 경우 동시성은 처리 중인 스트림의 색드 수에 따라 결정됩니다. 지연 시간이 함수 자체에 의해 결정된다는 점을 감안할 때 스트림 처리 함수의 경우 요청 속도 또는 처리량이 가변적일 수 있습니다. 다른 말로 하자면,

$$\text{유효 처리 속도} = \text{유효 동시성} / \text{평균 지속 시간(초당 이벤트 수)}$$

5개의 색드가 있고 배치 크기가 100인 스트림을 처리하는 데 1초가 걸리는 함수를 고려하십시오. 이는 함수가 처리할 수 있는 최대 요청 수(각각 100개의 레코드가 있음)가 5이고 최대 레코드 수를 의미합니다. 주어진 시간에 처리되는 데이터는 $5 * 100 = 500$ 입니다. 반면에 동일한 스트림에 10개의 색드가 있는 경우 처리량도 두 배가 됩니다.

10.4.2 동시성 관리

AWS는 동시성 관리를 위한 두 가지 설정을 제공합니다. 첫 번째는 계정 내 모든 기능의 총 동시성에 적용되는 계정 수준 동시성 제한입니다. 이 한도는 기본적으로 1,000으로 설정되어 있으며 서비스 한도 증가 티켓을 통해 올릴 수 있습니다. 이 증가는 작성 시점에서 "셀프 서비스"할 수 없습니다. 두 번째는 기능별 동시성 제어로, 개별 기능의 동시성을 제어하는 데 사용할 수 있습니다. 동시성을 "예약"하려는 함수 또는 동시성을 제한해야 하는 함수(다운스트림 리소스로 인해)가 있는 경우에만 함수별 동시성 제어를 사용합니다.

예를 들어 특정 로드만 처리할 수 있는 API를 호출하기 때문에 Lambda 함수가 확장되는 높이를 제한할 수 있습니다. 이 옵션을 선택하지 않으면 함수로 인해 다운스트림 API가 오버로드되어 전체 애플리케이션에 대한 가용성이 발생할 수 있습니다. 따라서 동시성을 모니터링하고 관리하는 것이 중요한 단계입니다. 제한 및 제어에 대한 자세한 내용은 <http://mng.bz/v4mq>에서 확인할 수 있습니다.

요약 서버리

스 애플리케이션에는 기존 애플리케이션 성능 모니터링 단계가 필요하지 않습니다. 대신, 함수 코드의 성능을 최적화하면 가장 큰 이득을 얻을 수 있습니다.

도구 세트(예: X-Ray) 및 구성(예: 메모리 설정)을 사용하여 성능을 쉽게 찾고 최적화하십시오.

Lambda 함수의 동시성은 함수 지연 시간에 영향을 미칠 수 있으므로(반대의 경우도 마찬가지) 중요한 함수에 대해 이를 모니터링하고 관리해야 합니다.

새로운 관행

이 장에서는 다음을 다룹니다.

여러 AWS 계정 사용

임시 스택 사용

환경 변수의 일반 텍스트로 민감한 데이터 유지 방지

이벤트 기반 아키텍처에서 EventBridge 사용

서비스라는 용어는 AWS가 2014년에 Lambda 서비스를 출시한 후 생겨났습니다.

그런 의미에서 서비스 패러다임(모든 컴퓨팅 요구 사항을 포함하여 관리 서비스를 사용하여 애플리케이션 구축)은 블록체인의 새로운 개념입니다.

새로운 패러다임은 문제를 보고 다르게, 아마도 더 효율적으로 해결할 수 있는 새로운 방법을 제공합니다. 이것은 우리가 이 책에서 여러 서비스 아키텍처에 대해 논의했기 때문에 지금쯤 분명할 것이며, 그것들이 동등한 서버풀 아키텍처와 매우 다르게 보인다는 것을 인정해야 합니다. 그들은 더 이벤트 중심적이며 종종 함께 작동하는 많은 다른 서비스를 포함합니다.

새로운 패러다임은 또한 우리가 다르게 생각하고 일할 것을 요구합니다. 예를 들어 비용을 가상 머신 집합의 크기와 필요한 기간의 함수로 생각하는 대신 요청 측면에서 비용을 생각해야 합니다.

카운트 및 실행 시간. 우리가 작성하는 코드와 애플리케이션을 배포하고 모니터링하는 방식도 이 새로운 패러다임을 최대한 활용하고 일부 제한 사항을 완화하기 위해 변경해야 합니다.

조직에서 서비스 기술을 성공적으로 채택한 팀은 다음과 같은 새로운 사례를 사용합니다. 여러 AWS 계정을 사용하고 이벤트 기반 아키텍처에서 EventBridge를 사용하는 것과 같이 많은 것이 서비스 컨텍스트 외부에서 유용합니다. 그들 중 어느 것도 은색 총알은 아니지만(무엇이 있습니까?), 그들은 적절한 맥락에서 유용하고 고려할 가치가 있는 아이디어입니다.

11.1 여러 AWS 계정 사용

요즘 대화하는 모든 AWS 직원은 여러 AWS 계정이 있어야 하며 AWS Organizations (<https://aws.amazon.com/organizations>)에서 관리해야 한다고 말합니다. 최소한 환경당 하나 이상의 AWS 계정이 있어야 합니다. 대규모 조직의 경우 더 나아가 환경당 팀당 하나 이상의 AWS 계정이 있어야 합니다. 다음 섹션에서 설명하는 것을 포함하여 서비스 기술을 사용하는지 여부에 관계없이 이것이 모범 사례로 간주되는 데는 여러 가지 이유가 있습니다.

11.1.1 보안 침해 경리

공격자가 AWS 환경에 대한 액세스 권한을 얻은 후 사용자 데이터에 액세스하여 훔칠 수 있는 악동 같은 시나리오를 상상해 보십시오. 이 악동 같은 시나리오는 여러 가지 방법으로 발생할 수 있으며 다음 세 가지가 바로 떠오릅니다.

EC2 인스턴스는 공개적으로 노출되며 공격자는 무차별 대입을 사용하여 인스턴스에 SSH로 연결할 수 있습니다. 내부에 들어가면 인스턴스의 IAM 역할을 사용하여 다른 AWS 리소스에 액세스 할 수 있습니다.

잘못 구성된 WAF(웹 애플리케이션 방화벽)를 통해 공격자는 SSRF(서버 측 요청 위조) 공격을 실행하고 WAF를 속여 EC2 메타데이터 서비스에 요청을 릴레이할 수 있습니다. 이를 통해 공격자는 WAF 서버에서 사용하는 임시 AWS 자격 증명을 찾을 수 있습니다. 여기에서 공격자는 계정의 다른 AWS 리소스에 액세스할 수 있습니다. 이것이 2019년 캐피털원 데이터 유출 사건에서 일어난 일입니다.

직원이 실수로 공개 GitHub 리포지토리의 Git 커밋에 AWS 자격 증명을 포함했습니다. 공격자는 AWS 자격 증명에 대한 공개 GitHub 저장소를 스캔하고 이 커밋을 찾습니다. 그런 다음 공격자는 모든 AWS 리소스에 액세스할 수 있습니다.

직원이 액세스할 수 있었습니다. AWS는 또한 활성 AWS 자격 증명에 대해 공개 GitHub 리포지토리를 검색하고 발견하면 고객에게 경고합니다. 그러나 고객이 인지할 때 이미 피해가 발생한 경우가 많습니다.

여러 계정을 사용하는 것이 이러한 공격 벡터를 막지는 못하지만 보안 침해의 폭발 반경을 단일 계정으로 제한합니다.

11.1.2 공유 서비스 제한에 대한 경합 제거

이 책 전체에서 AWS 서비스 제한에 대해 이미 여러 번 이야기했습니다.

조직과 시스템이 성장함에 따라 더 많은 엔지니어가 시스템에서 작업해야 하며 특정

더 큰 시스템 내의 도메인(マイクロサービス를 생각하십시오). 이렇게 되면 당신은 공유 서비스 제한에 대한 더 많은 경합이 있기 때문에 성가신 서비스 제한에 더 자주 부딪힐 가능성이 높습니다.

여기서부터 더 나빠집니다. 서비스 제한은 지역 수준에서 적용되고 영향을 미치기 때문에 지역의 모든 자원, 즉 한 팀 또는 한 서비스가 모든 자원을 소진할 수 있음을 의미합니다.

리전에서 사용 가능한 처리량(예: Lambda 동시 실행)

그리고 다른 모든 것을 조절하십시오.

또한 모든 환경이 동일한 AWS 계정에서 실행되는 경우

비프로덕션 환경에서 발생하는 일이 프로덕션 사용자에게도 영향을 미칠 수 있습니다. 예를 들어, 스테이징의 로드 테스트는 Lambda 동시 실행을 너무 많이 소비하여 사용자가 프로덕션에서 API에 액세스할 수 없도록 할 수 있습니다.

이러한 API 기능은 제한됩니다.

각 팀과 각 환경에 대해 별도의 계정이 있으면 경합이 완전히 사라집니다. 팀이 실수를 하거나 트래픽이 갑자기 급증하는 경우

그들이 소비하는 추가 처리량은 다른 서비스에 영향을 미치지 않습니다. 어느 서비스 제한 관련 제한은 해당 계정에 포함되어 폭발을 제한합니다.

이러한 사건의 반경. 마찬가지로 프로덕션 환경의 사용자에게 영향을 미치지 않는다는 사실을 알고 비프로덕션 환경에서 로드 테스트를 안전하게 실행할 수 있습니다.

팀 내에서 서로 다른 서비스 간에 동일한 경합이 존재한다면 어떻게 될까요?

팀의 서비스 중 하나가 나머지 서비스보다 훨씬 많은 트래픽을 처리하고 때때로 다른 서비스가 제한될 수 있습니다. 글쎄, 당신은 그 서비스를 이동하고 싶습니다

자체 개발, 테스트, 스테이징 및 프로덕션 계정 세트에 이 기술의 AWS 계정을 격벽으로 사용하여 폭발 반경을 격리하고 억제할 수 있습니다.

당신이 필요로. 환경당 팀당 하나의 계정으로 멈출 필요가 없습니다. 만들다 기술은 당신을 위해 작동하지만 그 반대는 아닙니다.

11.1.3 더 나은 비용 모니터링

모든 것이 동일한 AWS 계정에서 실행되는 경우 AWS 비용을 다른 환경이나 팀 또는 서비스에 할당하는 데 어려움을 겪을 것입니다. 여러

계정을 사용하면 해당 계정의 비용을 쉽게 확인할 수 있습니다.

11.1.4 팀의 자율성 향상

보안 및 액세스 제어 관점에서 각 팀에 자체 AWS 세트가 있는 경우

계정에 대한 더 많은 자율성과 제어 권한을 부여할 수 있습니다.

AWS 계정. 모든 사람이 동일한 AWS 계정을 공유하고 해당 계정이 다음 용도로 사용되는 경우 비프로덕션 환경과 프로덕션 환경 모두에서 위험이 높습니다.

실수는 큰 폭발 반경을 가지며 팀은 실수로 다른 사람을 삭제하거나 업데이트할 수 있습니다.

팀의 리소스를 삭제하거나 프로덕션에서 사용자 데이터를 삭제할 수도 있습니다. 그렇기 때문에 당신은 있어야합니다

액세스 관리 측면에서 주의하십시오. 액세스를 관리해야 하는 사람(일반적으로 보안 팀 또는 클라우드 플랫폼 팀)에게 많은 복잡성과 스트레스가 발생합니다.

내 경험상 높은 이해관계와 복잡성으로 인해 게이트 키핑이 필요하고 다양한 분야 간에 마찰이 발생합니다. 기능 팀은 종종 지연을 겪어야 합니다.

과로한 플랫폼 팀이 필요한 액세스 권한을 부여할 때까지 기다리세요. 원한 구축과 조화가 약화되고 곧 "우리 대 그들" 상황이 됩니다.

모든 팀에 자체 AWS 계정을 제공하면 문제의 폭발 반경이 제한되고 판돈을 낮춥니다. 그러면 팀에 더 많은 자율성을 부여할 수 있습니다. 자신의 계정. 플랫폼 팀/보안 팀은 대신 설정에 집중할 수 있습니다. 가드레일 및 거버넌스 인프라를 통해 신속하게 문제를 식별할 수 있습니다. 그리고 그들은 기능 팀과 협력하여 조직에서 가장 잘 따르도록 해야 합니다.

보안 요구 사항을 충족합니다.

11.1.5 AWS Organizations를 위한 코드로서의 인프라

여러 AWS 계정이 있다는 것은 이를 관리할 방법이 필요하다는 것을 의미합니다.

특히 조직을 확장할 때. AWS 계정의 수는 증가할 수 있으며

더 많은 엔지니어가 조직에 합류할수록 강력한 AWS 환경의 거버넌스 및 감독.

AWS Organizations의 단점 중 하나는 IaC(Infrastructure as Code)를 사용하여 조직의 구성을 업데이트할 수 없다는 것입니다. 예를 들어 Cloud Formation은 지역 서비스이며 단일 내에서 리소스를 프로비저닝하는 것으로 제한됩니다.

계정 및 지역. 글을 쓰는 시점에서 IaC를 적용할 수 있는 유일한 도구는

AWS Organizations는 org-formation (<https://github.com/org-formation/org>)입니다.
-형성-cli). 구성은 캡처할 수 있는 오픈 소스 도구입니다.

IaC를 사용하는 AWS 계정 및 전체 AWS 조직. 나는 여러 가지와 함께 그것을 사용 프로젝트이며 충분히 높게 추천할 수 없습니다!

여러 AWS 계정 사용과 관련된 주제는 기능 분기 또는

종단 간(e2e) 테스트를 수행합니다. 다음으로 임시 스택에 대해 설명합니다.

11.2 임시 스택 사용

서비스 기술의 이점 중 하나는 사람들이 귀하의 애플리케이션을 사용할 때만 비용을 지불한다는 것입니다. 코드가 실행되고 있지 않으면 비용이 청구되지 않습니다. 다음과 같은 도구를 사용하여 서비스 애플리케이션을 배포하는 것이 쉽다는 사실과 이것을 결합하십시오.

서비스 프레임워크로. 새로운 환경을 만드는 것이 너무 쉽기 때문에

이러한 환경을 유지하기 위한 가동 시간 비용은 없으며 많은 팀에서 임시로 기능 분기에서 작업하거나 e2e 테스트를 실행하기 위한 환경입니다.

11.2.1 일반적인 AWS 계정 구조

팀에는 환경마다 하나씩 여러 AWS 계정이 있는 것이 일반적입니다.

이러한 환경을 사용하는 방법에 대한 합의가 없는 것 같지만 우리는 다음 규칙을 따르는 경향이 있습니다.

개발 환경은 팀에서 공유합니다. 여기에서 최신 개발 변경 사항이 배포되고 종단 간 테스트가 이루어집니다. 이 환경은 본질적으로 불안정하므로 다른 팀에서 사용해서는 안 됩니다.

테스트 환경은 다른 팀이 팀의 작업과 통합할 수 있는 곳입니다. 이 환경은 다른 팀의 속도를 늦추지 않도록 안정적이어야 합니다.

스테이징 환경은 프로덕션 환경과 매우 유사해야 하며 종종 프로덕션 데이터 덤프를 포함할 수 있습니다. 여기에서 프로덕션과 같은 환경에서 릴리스 후보를 스트레스 테스트할 수 있습니다.

그리고 생산 환경이 있습니다.

이 장의 앞부분에서 논의한 것처럼 여러 AWS 계정(환경당 팀당 최소 하나의 계정)을 갖는 것이 모범 사례입니다. dev 계정에는 각 개발자 또는 각 기능 분기에 대해 하나씩, 둘 이상의 환경이 있을 수도 있습니다.

11.2.2 기능 분기에 임시 스택 사용

우리가 새로운 기능에 대한 작업을 시작할 때, 우리는 여전히 문제에 대한 최상의 솔루션을 향한 길을 느낍니다. 코드베이스가 불안정하고 많은 버그가 아직 해결되지 않았습니다.

반쯤 구운 변경 사항을 개발 환경에 배포하면 상당히 혼란스러울 수 있습니다.

팀의 공유 환경이 불안정해질 위험이 있습니다.

팀에서 작업 중인 다른 기능을 덮어씁니다.

팀 구성원은 기능 분기를 공유 환경에 배포할 대상을 놓고 싸울 수 있습니다.

대신 기능 분기를 임시 환경에 배포할 수 있습니다. Serverless Framework를 사용하는 것은 `sls deploy -s my-feature` 명령을 실행하는 것만큼 쉽습니다. 여기서 `my-feature`는 환경 이름이자 Cloud Formation 스택 이름입니다. 이렇게 하면 모든 Lambda 함수, API 게이트웨이 및 DynamoDB 테이블과 같은 기타 관련 리소스가 자체 CloudFormation 스택에 배포됩니다. 다른 팀원의 작업에 영향을 주지 않고 AWS 계정에서 진행 중인 작업 기능을 테스트할 수 있습니다.

각 기능 분기에 대해 이러한 임시 CloudFormation 스택을 사용하면 비용 오버헤드가 무시할 수 있습니다. 개발자가 기능을 완료하면 `sls remove -s my-feature` 명령을 실행하여 임시 스택을 쉽게 제거할 수 있습니다. 그러나 이러한 임시 스택은 기능 분기의 확장이기 때문에 수명이 긴 기능 분기가 있는 경우 동일한 문제가 나타납니다. 즉, 통합해야 하는 다른 시스템과 동기화되지 않습니다. 이는 Lambda 함수를 트리거하는 수신 이벤트(예: SQS/SNS/Kinesis의 페이로드)와 함수가 의존하는 데이터(예: DynamoDB 테이블의 데이터 스키마)에 적용됩니다. 우리는 서비스 기술을 사용하는 팀이 더 빨리 움직이는 경향이 있다는 것을 알게 되었고, 이로 인해 수명이 긴 기능 분기의 문제가 더 두드러지고 눈에 띄게 되었습니다.

일반적으로 기능 분기를 일주일 이상 방치하지 마십시오. 작업이 크고 구현하는 데 시간이 오래 걸리는 경우 더 작게 분할하십시오.

특징. 가능 브랜치에서 작업할 때 기본 개발 브랜치에서도 하루에 한 번 이상 정기적으로 통합해야 합니다.

11.2.3 e2e 테스트에 임시 스택 사용

임시 CloudFormation 스택의 또 다른 일반적인 용도는 e2e 테스트를 실행하는 것입니다.

이러한 테스트의 일반적인 문제 중 하나는 테스트 데이터를 공유 AWS 환경에 삽입해야 한다는 것입니다. 시간이 지남에 따라 이러한 환경에 많은 정크 데이터가 추가되고 다른 팀 구성원을 어렵게 만들 수 있습니다. 예를 들어 테스터는 종종 모바일 또는 웹 앱에서 수동 테스트를 수행해야 하며 자동화된 테스트에서 남겨진 모든 테스트 데이터는 혼란을 일으키고 필요 이상으로 작업을 어렵게 만들 수 있습니다. 일반적으로 우리는 항상 다음을 수행합니다.

테스트 전에 테스트 케이스에 필요한 데이터를 삽입하십시오.

테스트가 완료된 후 데이터를 삭제하십시오.

Jest 사용하기 (<https://jestjs.io>) JavaScript 프레임워크를 사용하면 테스트 스위트의 일부로 전후 단계를 캡처할 수 있습니다. 데이터 존재에 암시적으로 의존하지 않기 때문에 테스트를 강력하고 독립적으로 유지하는 데 도움이 됩니다. 또한 공유 개발 환경에서 정크 데이터의 양을 줄이는 데 도움이 됩니다.

그러나 우리의 최선의 의도에도 불구하고 실수가 발생하고 때로는 단기적으로 민첩성을 얻기 위해 의도적으로 모퉁이를 잘라냅니다. 시간이 지남에 따라 이러한 공유 환경은 여전히 수많은 테스트 데이터로 끝납니다. 이에 대한 대책으로 많은 팀에서 cron 작업을 사용하여 때때로 이러한 환경을 삭제합니다.

이러한 문제를 해결하기 위한 새로운 방법은 CI/CD 파이프라인 동안 임시 Cloud Formation 스택을 만드는 것입니다. 임시 스택은 e2 테스트를 실행하는 데 사용되며 나중에 삭제됩니다. 이렇게 하면 테스트 픽스처의 일부로 또는 cron 작업으로 테스트 데이터를 정리할 필요가 없습니다. 단점은 다음과 같습니다.

CI/CD 파이프라인을 실행하는 데 시간이 더 오래 걸립니다.

여전히 테스트 데이터를 외부 시스템에 남겨두므로 완전한 솔루션이 아닙니다.

CI/CD 파이프라인에 추가되는 지연과 이 접근 방식의 이점을 비교해야 합니다. 개인적으로 우리는 이것이 훌륭한 접근 방식이라고 생각하고 더 많은 팀이 이를 채택하기 시작하는 것을 봅니다. CI/CD 파이프라인의 속도를 높이기 위해 일부 팀에서는 이러한 임시 스택을 여러 개 유지하고 라운드 로бин 방식으로 재사용합니다. 이렇게 하면 임시 환경에 대해 e2e 테스트를 실행할 수 있다는 이점이 있지만 임시 환경을 배포하는 데 걸리는 시간을 단축할 수 있습니다(기존 CloudFormation 스택을 업데이트하는 것이 새 스택을 생성하는 것보다 훨씬 빠름).

11.3 환경 변수에서 일반 텍스트의 민감한 데이터 피하기

서버풀 및 서비스 애플리케이션 모두에서 볼 수 있는 일반적인 실수 중 하나는 민감한 데이터(예: API 키 및 자격 증명)가 환경 변수의 일반 텍스트로 남아 있다는 것입니다. 보안과 관련하여 AWS가 애플리케이션의 운영 환경 보안을 관리하기 때문에 서비스 애플리케이션이 더 안전합니다. 이것

여기에는 우리의 코드가 실행되는 가상 머신과 네트워크 구성의 보안이 포함되며 운영 체제 자체의 보안도 포함됩니다.

Lambda 함수는 AWS가 관리하는 베어메탈 EC2 인스턴스에서 실행되고 EC2 인스턴스는 AWS 관리 VPC에 상주합니다. 공격자가 가상 머신 자체에 대한 정보를 쉽게 찾을 수 있는 방법은 없으며 공격자가 이러한 가상 머신에 SSH로 접속할 수 있는 방법도 없습니다.

운영 체제는 최신 보안 패치로 지속적으로 업데이트되고 패치되며, 때로는 일반 대중이 패치를 사용할 수 있기 전에도 있습니다. Meltdown 및 Spectre 사태 동안 Lambda 및 Fargate 뒤에 있는 모든 EC2 인스턴스가 나머지 사람들이 컨테이너 및 EC2 이미지를 패치할 수 있기 훨씬 전에 취약점에 대해 신속하게 패치된 경우가 그러했습니다. AWS가 우리 코드의 운영 환경을 관리하게 하면 우리 플레이트에서 엄청난 종류의 공격 벡터가 제거되지만 우리는 여전히 애플리케이션과 해당 데이터의 보안을 책임지고 있습니다.

11.3.1 공격자는 여전히 들어갈 수 있습니다

우리 코드의 운영 환경은 AWS에 의해 보호되지만, 공격자는 여전히 다음을 포함한 다른 수단을 통해 우리 함수의 실행 환경에 들어갈 수 있습니다.

공격자는 코드 주입 공격을 성공적으로 실행합니다. 예를 들어, 애플리케이션 또는 해당 종속성 이 사용자 입력에 대해 JavaScript의 eval() 함수를 사용하는 경우 이러한 공격에 취약합니다.

공격자는 종속성 중 하나를 손상시키고 런타임에 애플리케이션에서 정보를 훔치는 악성 버전의 종속성을 게시합니다. 보안 연구원이 NPM 패키지 (<http://mng.bz/N4PN>) 의 14%에 대한 게시 액세스 권한을 얻었던 때를 기억 하십니까? 아니면 그 때 공격자가 EsLint의 유지 관리자 중 한 명의 NPM 계정을 손상시키고 eslint-scope 및 eslint-config-eslint (<http://mng.bz/DKPn>)의 악성 버전을 게시했습니까?

공격자는 인기 있는 NPM 패키지와 유사한 이름을 가진 악성 NPM 패키지를 게시하고 초기화 시 애플리케이션에서 정보를 훔칩니다. 예를 들어 공격자가 인기 있는 NPM 패키지 cross-env 를 미끼로 사용하여 crossenv 라는 악성 패키지를 게시한 경우가 있습니다(<http://mng.bz/l9d6>).

일단 침입하면 공격자는 환경 변수와 같이 일반적이고 쉽게 액세스할 수 있는 위치에서 정보를 훔치는 경우가 많습니다. 이것이 환경 변수의 일반 텍스트에 민감한 데이터를 넣지 않는 것이 중요한 이유입니다.

11.3.2 민감한 데이터를 안전하게 처리

민감한 데이터는 전송 및 저장 모두에서 암호화되어야 합니다. 이는 암호화된 형식으로 저장해야 함을 의미합니다. AWS 내에서 SSM Parameter Store와 Secrets Manager를 모두 사용하여 저장할 수 있습니다. 두 서비스 모두 저장 데이터 암호화를 지원하고 AWS Key Management Service(KMS)와 직접 통합되며 고객 관리 키(CMK)를 사용할 수 있습니다. 동일한 암호화된 미사용 원칙을 적용해야 합니다.

민감한 데이터가 애플리케이션에 저장되는 정도. 이를 달성하는 방법에는 여러 가지가 있습니다. 예를 들어:

민감한 데이터를 암호화된 형태로 환경 변수에 저장하고 콜드 스타트 중에 KMS를 사용하여 암호를 해독합니다.

민감한 데이터는 SSM Parameter Store 또는 Secrets Manager에 보관하고 Lambda 함수 콜드 스타트 중에 SSM Parameter Store/Secrets Manager에서 가져옵니다.

암호가 해독되면 코드에서 쉽게 액세스할 수 있는 애플리케이션 변수 또는 클로저에 데이터를 보관할 수 있습니다. 중요한 것은 민감한 데이터가 암호화되지 않은 형태로 환경 변수에 다시 저장되어서는 안 된다는 것입니다. 우리의 개인적인 선호는 콜드 스타트 동안 SSM Parameter Store/Secrets Manager에서 민감한 데이터를 가져오는 것입니다. 우리는 middy의 SSM 미들웨어 (<https://github.com/middyjs/middy/tree/main/packages/ssm>)를 사용하여 복호화된 데이터를 컨텍스트 변수에 주입하고 한동안 캐시합니다.

이렇게 하면 애플리케이션을 다시 배포할 필요 없이 소스에서 이러한 비밀을 순환할 수 있습니다. 캐시가 만료되면 미들웨어는 다음 Lambda 호출 시 새 값을 가져옵니다. 또한 여러 서비스가 동일한 암호에 액세스해야 하는 경우 공유 암호를 더 쉽게 관리할 수 있습니다. 마지막으로, 이 접근 방식을 사용하면 Lambda 함수가 SSM Parameter Store/Secrets Manager의 비밀에 액세스할 수 있는 권한이 필요하기 때문에 권한을 보다 세부적으로 제어할 수 있습니다.

이 두 가지 접근 방식의 다른 변형이 있습니다. 예를 들어 암호화된 비밀을 환경 변수에 저장하는 대신 애플리케이션의 일부로 배포되는 암호화된 파일에 저장할 수 있습니다. Lambda 콜드 스타트 동안 이 파일은 KMS로 암호 해독되고 여기에 포함된 비밀은 추출되어 환경 변수에서 떨어져 저장됩니다.

11.4 이벤트 기반 아키텍처에서 EventBridge 사용

Amazon SNS 및 SQS는 서비스 통합과 관련하여 AWS 개발자가 오랫동안 선택하는 옵션이었습니다. 그러나 브랜드 변경 이후 Amazon EventBridge(이전의 Amazon CloudWatch Events)가 인기 있는 대안이 되었으며 실제로 이벤트 기반 아키텍처의 이벤트 버스로 훨씬 더 나은 옵션이라고 주장하고 싶습니다.

11.4.1 콘텐츠 기반 필터링

SNS를 사용하면 필터링 정책을 통해 메시지를 필터링할 수 있습니다. 그러나 내용으로 메시지를 필터링할 수 없고 메시지 속성으로만 필터링할 수 있으며 메시지당 최대 10개의 속성만 가질 수 있습니다. 콘텐츠 기반 필터링이 필요한 경우 코드에서 수행해야 합니다. 반면 EventBridge는 콘텐츠 기반 필터링을 지원하며 이벤트 콘텐츠에 대한 패턴 일치를 허용합니다. 또한 다음과 같은 고급 필터링 규칙을 지원합니다.

- 수치 비교
- 접두사 일치
- IP 주소 일치
- 존재 매칭
- 일치 이외의 모든 것

[참고 블로그 게시물\(http://mng.bz/B1w0 \)](http://mng.bz/B1w0)을 확인하세요. 이러한 고급 규칙에 대한 자세한 내용은 EventBridge의 콘텐츠 기반 필터링을 참조하십시오.

이벤트 중심 아키텍처에서는 중앙 집중식 이벤트 버스를 사용하는 것이 바람직한 경우가 많습니다. 이를 통해 하위 시스템이 다른 하위 시스템에 의해 트리거된 이벤트를 구독하고 전체 애플리케이션에서 발생하는 모든 것을 캡처하는 아카이브를 생성할 수 있습니다.(감사 및 재생 목적 모두).

콘텐츠 기반 필터링을 사용하면 Event Bridge에서 중앙 집중식 이벤트 버스를 가질 수 있습니다. 구독자는 포함할 속성에 대해 이벤트 게시자와 협상하지 않고도 원하는 정확한 이벤트를 자유롭게 구독할 수 있습니다. 이는 일반적으로 SNS에서는 불가능하며 여러 SNS 주제를 사용해야 합니다.

11.4.2 스키마 발견

이벤트 기반 아키텍처의 일반적인 문제는 이벤트 스키마를 식별하고 버전을 지정하는 것입니다. EventBridge는 스키마 레지스트리를 사용하여 이 문제를 처리하고 스키마 검색을 위한 기본 제공 메커니즘을 제공합니다.

EventBridge는 기본 이벤트 버스에서 AWS 서비스의 광범위한 이벤트(예: EC2 인스턴스의 상태가 변경된 경우)를 캡처합니다. 기본 스키마 레지스트리에서 이러한 AWS 이벤트에 대한 스키마를 제공합니다. 또한 모든 이벤트 버스에서 스키마 검색을 활성화할 수 있으며 EventBridge는 수집된 이벤트를 샘플링하고 이러한 이벤트에 대한 스키마 정의를 생성 및 버전화합니다.

이미 애플리케이션 이벤트에 대한 스키마 정의를 프로그래밍 방식으로 생성하고 있다면 사용자 지정 스키마 레지스트리를 생성하고 CI/CD 파이프라인의 일부로 스키마 정의를 게시할 수도 있습니다. 그렇게 하면 개발자는 항상 최신 순환 이벤트 목록과 이러한 이벤트에서 찾을 수 있는 정보를 얻을 수 있습니다.

eVB-CLI (<https://www.npmjs.com/package/@mhlabs/eVB-CLI>) 와 같은 오픈 소스 도구를 사용하면 스키마 레지스트리의 스키마 정의를 사용하여 EventBridge 패턴을 생성할 수도 있습니다. 이것은 특히 EventBridge의 패턴 언어를 처음 접하는 경우에 유용합니다!

11.4.3 이벤트 보관 및 재생

이벤트 중심 아키텍처의 또 다른 일반적인 요구 사항은 수집된 이벤트를 보관하고 나중에 재생할 수 있어야 한다는 것입니다. 아카이브 요구 사항은 종종 더 큰 감사 또는 규정 준수 요구 사항 집합의 일부이므로 많은 시스템에서 필수 항목입니다. 운 좋게도 EventBridge는 즉시 보관 및 재생 기능을 제공합니다.

아카이브를 생성할 때 보존 기간을 구성할 수 있습니다.

무기한으로. 선택적으로 일치하는 이벤트만 아카이브에 포함되도록 필터를 구성할 수 있습니다.

아카이브에서 이벤트를 재생해야 하는 경우 지정된 시간 범위에 캡처된 이벤트만 재생되도록 시작 및 종료 시간을 선택할 수 있습니다. 이벤트 재생에 대해 염두에 두어야 할 가지는 EventBridge가 수신된 이벤트의 원래 순서를 유지하지 않는다는 것입니다. 대신 EventBridge는 이러한 이벤트를 최대한 빨리 재생하려고 합니다. 즉, 많은 동시성을 기대할 수 있고 대부분의 이벤트가 순서 없이 재생됩니다.

이벤트를 재생할 때 순서가 중요하다면 앞서 언급한 evb-cli 프로젝트를 확인해야 합니다. evb replay 명령은 이벤트 순서를 유지하고 이벤트가 재생되는 속도를 제어할 수 있는 페이싱 재생을 지원합니다. 예를 들어, 실시간으로 100회 재생 이벤트의 재생 속도를 사용하면 1시간 분량의 이벤트를 재생하는 데 1시간이 걸립니다.

11.4.4 더 많은 목표

SNS는 소수의 대상(예: HTTP, 이메일, SQS, Lambda 및 SMS), EventBridge는 15개 이상의 AWS 서비스(SNS, SQS, Kinesis 및 Lambda 포함)를 지원하며 이벤트를 다른 EventBridge 버스에 전달할 수 있습니다.
계정.

이 광범위한 범위는 불필요한 글루 코드를 많이 제거하는 데 도움이 됩니다. 예를 들어, Step Functions 상태 머신을 시작하려면 SNS와 Step Functions 사이에 Lambda 함수가 필요했을 것입니다. EventBridge를 사용하면 규칙을 상태 시스템에 직접 연결할 수 있습니다.

11.4.5 토플로지

EventBridge에서 이벤트 버스를 정렬하는 방법에는 여러 가지가 있습니다. 예를 들어 중앙 집중식 이벤트 버스가 있거나 모든 서비스가 자체 이벤트 버스에 이벤트를 게시하거나 관련 서비스에서 공유하는 몇 가지 도메인별 이벤트 버스가 있을 수 있습니다. 사람마다 상황이 다르고 접근 방식마다 장단점이 있기 때문에 어떤 접근 방식이 최선인지에 대한 명확한 합의가 없습니다. 그러나 다음과 같은 몇 가지 큰 이점이 있기 때문에 우리는 개인적으로 중앙 집중식 이벤트 버스 접근 방식을 선호합니다.

아카이브와 스키마 레지스트리를 한 곳에서 구현할 수 있습니다.

액세스 및 권한을 한 곳에서 관리할 수 있습니다.

필요한 모든 이벤트를 하나의 이벤트 버스에서 사용할 수 있습니다.

관리할 리소스가 적습니다.

그러나 고려해야 할 몇 가지 단점도 있습니다. 단일 실패 지점이 있습니다. 하지

만 EventBridge는 이미 고가용성이며 이벤트를 수집, 필터링 및 구성된 대상으로 전달하는 인프라는 여러 가용성 영역에 분산되어 있습니다.

서비스 팀은 모두 중앙 집중식 이벤트 버스에 의존하기 때문에 자율성이 떨어집니다.

AWS 계정 토플로지에 대한 질문도 있습니다. 즉, 주어진 환경이 여러 AWS 계정으로 구성된 경우(예: 팀당 하나의 계정이 있는 경우) 이벤트 버스를 어떤 계정에 배포합니까? 중앙 집중식 이벤트 버스를 자체 계정에 배포해야 합니까 아니면 가장 합리적인 계정에 배포해야 합니까?

이는 이 장의 범위를 벗어나는 더 넓은 주제이지만 Stephen Liedig의 re:Invent 2020 세션(<https://www.youtube.com/watch?v=Wk0FoXTUEjo>)을 확인하는 것이 좋습니다. 다양한 구성과 각각의 장단점에 대해 자세히 설명합니다.

요약 여기까지

가 프로젝트에서 채택하는 것을 진지하게 고려해야 하는 새로운 사례 목록입니다. 우리는 이러한 새로운 방식을 유비쿼터스 방식으로 채택되지 않았지만 AWS 커뮤니티에서 주목을 받고 있기 때문에 이러한 방식이라고 부릅니다. AWS 에코시스템과 서비스 기술이 발전하고 성숙해짐에 따라 더 많은 관행이 나타나고 뿌리를 내리게 됩니다. 어떤 관행도 그 자체로 최선의 것으로 간주되어서는 안 되며 항상 해당 관행이 적용되는 맥락과 환경을 고려해야 한다는 점을 기억할 가치가 있습니다.

기술과 조직이 변하면 상황도 변합니다. 한 때 모범 사례라고 생각할 수 있는 많은 것들이 쉽게 악티 패턴이 될 수 있습니다.

예를 들어, monorepos는 소규모 팀일 때 훌륭하게 작동하지만 수백 또는 아마도 수천 명의 엔지니어로 성장할 때 monorepos는 해결하기 위해 복잡한 솔루션이 필요한 많은 문제를 제시합니다.

소프트웨어를 구축, 테스트, 배포 및 운영하는 방법도 마찬가지입니다. 사실 데이터 센터와 서버 팜에서 잘 작동하던 것이 클라우드로 제대로 변환되지 않을 수 있습니다.

그리고 코드가 실행되는 인프라와 코드 자체를 모두 관리해야 할 때 우리에게 도움이 되는 관행은 서비스 기술로 애플리케이션을 구축할 때 우리에게 불리할 수 있습니다.

모범 사례와 디자인 패턴은 대화의 끝이 아니라 시작이어야 합니다. 결국, 이러한 소위 모범 사례와 디자인 패턴은 다른 사람들이 한 일에 대해 어느 정도 효과가 있었던 집합적인 문서입니다. 그들이 오늘 당신을 위해 일할 것이라는 보장은 없습니다. 그리고 다른 산업의 유사점을 쉽게 볼 수 있습니다. 예를 들어, 1930년대부터 1950년대까지 뇌염 절제술이 주류 정신 건강 관리의 일부였다는 것을 알고 계셨습니까?

부록 서버리스 아키텍 처를 위한 서비스

AWS는 서비스 애플리케이션을 구축하는 데 사용할 수 있는 다양한 서비스와 제품이 있는 거대한 놀이터입니다. Lambda는 이 책에서 논의한 핵심 서비스이지만 다른 서비스와 제품도 특정 문제를 해결하는 데 중요하지는 않더라도 유용할 수 있습니다. AWS가 아닌 우수한 제품도 많이 있으므로 Amazon이 제공하는 제품만 사용해야 한다고 생각하지 마십시오. Microsoft와 Google의 제품도 살펴보십시오. 다음 섹션에서는 유용하다고 판단한 서비스 샘플을 제공합니다. 이 부록을 책 전체에서 논의한 다양한 서비스 및 제품에 대한 지침으로 사용할 수 있습니다.

A.1 API 게이트웨이

Amazon API Gateway는 프런트엔드와 백엔드 서비스 간에 API 계층을 생성하는 데 사용할 수 있는 서비스입니다. API Gateway의 수명 주기 관리를 통해 여러 버전의 API를 동시에 실행할 수 있으며 개발, 스테이징 및 프로덕션과 같은 여러 릴리스 단계를 지원합니다. API Gateway에는 요청 캐싱 및 조절과 같은 유용한 기능도 있습니다.

API는 리소스 및 메서드를 중심으로 정의됩니다. 리소스는 사용자 또는 제품과 같은 논리적 엔티티입니다. 메서드는 HTTP 동사(예: GET, POST, PUT 또는 DELETE)와 리소스 경로의 조합입니다. API Gateway는 Lambda 및 기타 AWS 서비스와 통합됩니다. 프록시 서비스로 사용하고 일반 HTTP 끝점에 요청을 전달할 수 있습니다.

A.2 단순 알림 서비스(SNS)

Amazon Simple Notification Service(SNS)는 메시지를 전달하도록 설계된 확장 가능한 게시/구독 서비스입니다. 생산자 또는 게시자는 주제에 대한 메시지를 작성하고 보냅니다. 구독자 또는 소비자는 주제를 구독하고 다음 중 하나를 통해 메시지를 받습니다.

지원되는 프로토콜. SNS는 중복성을 위해 여러 서버와 데이터 센터에 메시지를 저장하고 최소 1회 전달을 보장합니다. 최소 1회 전달은 구독자에게 메시지가 1회 이상 전달되도록 규정하지만 드물게 SNS의 분산 특성으로 인해 여러 번 전달될 수 있습니다.

SNS에서 HTTP 엔드포인트로 메시지를 전달할 수 없는 경우 나중에 전달을 다시 시도하도록 구성할 수 있습니다. SNS는 조절이 적용될 때 Lambda에 실패한 전송을 재시도할 수도 있습니다. SNS는 최대 256KB의 메시지 페이로드를 지원합니다.

A.3 단순 저장 서비스(S3)

Simple Storage Service(S3)는 Amazon의 확장 가능한 스토리지 솔루션입니다. S3의 데이터는 여러 시설과 서버에 중복 저장됩니다. 이벤트 알림 시스템을 사용하면 객체가 생성되거나 삭제될 때 S3가 SNS, SQS 또는 Lambda에 이벤트를 보낼 수 있습니다.

S3는 기본적으로 소유자만 자신이 생성한 리소스에 액세스할 수 있는 안전하지만 액세스 제어 목록 및 버킷 정책을 사용하여 보다 세분화되고 유연한 액세스 권한을 설정할 수 있습니다.

S3는 버킷과 객체의 개념을 사용합니다. 버킷은 객체에 대한 상위 수준 디렉터리 또는 컨테이너입니다. 객체는 데이터, 메타데이터 및 키의 조합입니다. 키는 버킷에 있는 객체의 고유 식별자입니다.

S3는 또한 S3 콘솔에서 객체를 그룹화하는 수단으로 폴더 개념을 지원합니다. 폴더는 키 이름 접두사를 사용하여 작동합니다. 키 이름의 슬래시 문자(/)는 폴더를 나타냅니다. 예를 들어, 키 이름이 문서/개인/myfile.txt인 객체는 S3 콘솔에 myfile.txt 파일이 포함된 개인이라는 폴더를 포함하는 문서라는 폴더로 표시됩니다.

A.4 단순 대기열 서비스(SQS)

SQS(Simple Queue Service)는 Amazon의 분산 및 내결합성 대기열 서비스입니다. SNS와 유사한 메시지 최소 1회 전달을 보장하고 최대 256KB의 메시지 페이로드를 지원합니다. SQS를 사용하면 여러 게시자와 소비자가 동일한 대기열과 상호 작용할 수 있으며 미리 설정된 보존 기간 후에 메시지가 자동으로 만료되고 삭제되는 기본 제공 메시지 수명 주기가 있습니다. 대부분의 AWS 제품과 마찬가지로 대기열에 대한 액세스를 제어하는 데 도움이 되는 액세스 제어가 있습니다. SQS는 SNS와 통합되어 메시지를 자동으로 수신하고 대기열에 넣습니다.

A.5 단순 이메일 서비스(SES)

SES(Simple Email Service)는 이메일을 주고받도록 설계된 서비스입니다. SES는 스팸 및 바이러스 검사 및 신뢰할 수 없는 출처의 이메일 거부와 같은 이메일 수신 작업을 처리합니다. 수신 이메일은 S3 버킷으로 전달되거나 Lambda 알림을 호출하거나 SNS 알림을 생성하는 데 사용할 수 있습니다. 이러한 작업은 수신 규칙의 일부로 구성할 수 있으며, 이는 SES에게 이메일로 수행할 작업을 알려줍니다.

일단 도착합니다.

SES를 사용하여 이메일을 보내는 것은 간단하지만 전송되는 메시지의 수와 비율을 규제하는 데 제한이 있습니다. SES는 스팸이 아닌 고품질 이메일이 전송되는 한 자동으로 할당량을 늘립니다.

A.6 관계형 데이터베이스 서비스(RDS)

Amazon Relational Database Service(RDS)는 AWS 인프라에서 관계형 데이터베이스의 설정 및 운영을 지원하는 웹 서비스입니다. RDS는 Amazon Aurora, MySQL, MariaDB, Oracle, MS-SQL 및 PostgreSQL 데이터베이스 엔진을 지원합니다. 프로비저닝, 백업, 패치, 복구, 복구 및 오류 감지와 같은 일상적인 작업을 처리합니다. 모니터링 및 지표, 데이터베이스 스냅샷, 다중 가용 영역(AZ) 지원이 즉시 제공됩니다. RDS는 이벤트 발생 시 SNS를 통해 알림을 전달합니다. 이를 통해 생성, 삭제, 장애 조치, 복구 및 복원과 같은 데이터베이스 이벤트가 발생했을 때 쉽게 대응할 수 있습니다.

A.7 다이나모DB

DynamoDB는 Amazon의 NoSQL 데이터베이스입니다. 테이블, 항목 및 속성은 Dynamo의 주요 개념입니다. 테이블은 항목 모음을 저장합니다. 항목은 속성 모음으로 구성됩니다. 각 속성은 사람의 이름이나 전화번호와 같은 단순한 데이터 조각입니다. 모든 항목은 고유하게 식별할 수 있습니다. Lambda는 DynamoDB 테이블과 통합되며 테이블 업데이트로 트리거될 수 있습니다. 글로벌 테이블은 다양한 AWS 리전에서 테이블을 원활하게 복제하고 모든 데이터 충돌을 해결하는 Dynamo의 주목할만한 기능입니다(동시 업데이트를 처리하기 위해 "최종 작성자 우선" 조정 사용). DynamoDB는 확장 가능한 글로벌 애플리케이션에 적합한 데이터베이스입니다. 마지막으로 DynamoDB에 인메모리 캐시(DAX)를 사용할 수 있습니다. 응답 시간이 단축되지만 대가가 따릅니다.

A.8 Algolia

(비 AWS) 관리 검색 엔진 API입니다. 반구조화된 데이터를 통해 검색할 수 있으며 개발자가 검색을 웹사이트 및 모바일 애플리케이션에 직접 통합할 수 있도록 하는 API가 있습니다. Algolia의 뛰어난 기능 중 하나는 속도입니다.

Algolia는 전 세계 15개 지역에 데이터를 배포 및 동기화하고 가장 가까운 데이터 센터에 쿼리를 보낼 수 있습니다.

Algolia는 인덱스 (".. 검색하려는 데이터를 가져오는 엔티티 ... 데이터베이스 내의 테이블과 유사 ..."), 레코드 (".. JSON 스키마가 없는 객체 검색할 수 있기를 원합니다...") 및 작업 (업데이트 또는 삭제와 같은 본질적으로 원자적 작업)입니다. 이러한 개념은 간단하며 Algolia를 사용하기 쉬운 검색 플랫폼 중 하나로 만듭니다. 유료 플랜은 한 달에 약 \$35부터 시작하지만 애플리케이션과 사용자가 수행하는 레코드 및 작업 수에 따라 비용이 빠르게 증가할 수 있습니다.

A.9 미디어 서비스

AWS 미디어 서비스는 개발자가 비디오 워크플로를 구축할 수 있도록 설계된 신제품입니다. 미디어 서비스는 다음 제품으로 구성됩니다.

MediaConvert는 규모에 따라 다양한 비디오 형식 간에 트랜스코딩하도록 설계되었습니다.

MediaLive는 라이브 비디오 처리 서비스입니다. 라이브 비디오 소스를 가져 와서 배포를 위해 더 작은 버전으로 압축합니다.

MediaPackage를 통해 개발자는 일시 중지 및 되감기와 같은 비디오 기능을 구현할 수 있습니다. DRM(디지털 권한 관리)을 추가하는 데에도 사용할 수 있습니다.

콘텐츠.

MediaStore는 미디어에 최적화된 스토리지 서비스입니다. 목표는 라이브 및 주문형 비디오 콘텐츠를 위한 저지연 스토리지 시스템을 제공하는 것입니다.

MediaTailor를 사용하면 개발자가 개별적으로 타겟팅된 광고를 비디오 스트림에 삽입할 수 있습니다.

Media Services는 Elastic Transcoder보다 우수한 고급 서비스 제품군을 제공합니다. 그럼에도 불구하고 Elastic Transcoder에는 Media Services에 없는 몇 가지 기능(예: WebM 파일 및 애니메이션 GIF 생성 기능)이 있습니다.

A.10 Kinesis 스트림

Kinesis Streams는 스트리밍 빅 데이터의 실시간 처리를 위한 서비스입니다. 일반적으로 빠른 로그 및 데이터 수집, 메트릭, 분석 및 보고에 사용됩니다. Amazon에서는 Kinesis Streams를 빅 데이터 스트리밍에 주로 사용하도록 권장하는 반면 SQS는 특히 가시성 시간 초과 또는 개별 지연과 같은 메시지에 대한 보다 세분화된 제어가 필요한 경우 안정적인 호스팅 대기열로 사용된다는 점에서 SQS와 다릅니다.

Kinesis Streams에서 색인은 스트림의 처리 용량을 지정합니다. 스트림 생성 시 색인 수를 규정해야 하지만 처리량을 늘리거나 줄여야 하는 경우 리사이징이 가능합니다. 이에 비해 SQS는 확장을 훨씬 더 투명하게 만듭니다. Lambda는 Kinesis와 통합하여 감지되는 즉시 스트림에서 레코드 배치를 읽을 수 있습니다.

A.11 아테나

AWS는 Athena를 서비스 대화형 쿼리 서비스로 청구합니다. 기본적으로 이 서비스를 사용하면 표준 SQL을 사용하여 S3에 배치된 데이터를 쿼리할 수 있습니다. 많은 경우 쿼리가 발생하기 전에 데이터를 변환하기 위해 ETL(추출, 변환 및 로드) 작업을 실행할 필요가 없습니다(데이터를 특정 방식으로 변환해야 하는 경우 Athena와 AWS Glue를 결합할 수 있음). 사용자는 S3에 데이터를 업로드하고 스키마를 준비하며 거의 즉시 쿼리를 시작합니다.

A.12 AppSync AppSync

는 개발자가 ... 실시간 및 오프라인 기능을 갖춘 데이터 기반 앱." 실제로 AppSync는 AWS에서 제공하는 관리형 GraphQL 엔드포인트입니다. DynamoDB, Lambda 및 Amazon Elasticsearch와 통합됩니다. GraphQL 및 GraphQL 스키마에 익숙하다면 AppSync를 바로 시작할 수 있습니다. GraphQL에 익숙하지 않다면 사전에 약간의 읽기를 권장합니다 (<http://graphql.org/learn/>). GraphQL은 지난 몇 년 동안 특히 서비스 기술 채택자들 사이에서 호평을 받아왔습니다.

A.13 Cognito

Amazon Cognito는 자격 증명 관리 서비스입니다. Google, Facebook, Twitter 및 Amazon과 같은 공개 자격 증명 공급자 또는 자체 시스템과 통합됩니다.

Cognito는 사용자 풀을 지원하므로 고유한 사용자 디렉터리를 생성할 수 있습니다. 이를 통해 별도의 사용자 데이터베이스 및 인증 서비스를 실행하지 않고도 사용자를 등록하고 인증할 수 있습니다. Cognito는 다양한 장치에서 사용자 응용 프로그램 데이터의 동기화를 지원하고 인터넷에 액세스할 수 없는 경우에도 모바일 장치가 작동할 수 있도록 하는 오프라인 지원을 제공합니다.

A.14 인증0

Auth0(최근 Okta에서 인수)은 Cognito에 없는 몇 가지 기능이 있는 비 AWS ID 관리 제품입니다. Auth0는 Google, Facebook, Twitter, Amazon, LinkedIn 및 Windows Live를 포함한 30개 이상의 자격 증명 공급자와 통합됩니다. ID 공급자와 통합할 필요 없이 자체 사용자 데이터베이스를 사용하여 새 사용자를 등록하는 방법을 제공합니다. 또한 다른 데이터베이스에서 사용자를 가져올 수 있는 기능이 있습니다. 예상대로 Auth0은 SAML, OpenID Connect, OAuth 2.0, OAuth 1.0 및 JSON 웹 토큰(JWT)을 포함한 표준 산업 프로토콜을 지원합니다. AWS Identity, Access Management 및 Cognito와 간편하게 통합할 수 있습니다.

A.15 기타 서비스

이 섹션에서 제공하는 서비스 목록은 애플리케이션을 구축하는 데 사용할 수 있는 다양한 제품의 간단한 샘플입니다. Google 및 Microsoft와 같은 대규모 클라우드 중심 회사와 Auth0와 같은 소규모 독립 회사에서 제공하는 서비스를 포함하여 더 많은 서비스가 있습니다. 또한 알고 있어야 하는 보조 서비스가 있습니다. 이를 통해 보다 효율적이고 소프트웨어를 더 빠르게 구축하고 성능을 개선하거나 다른 목표를 달성할 수 있습니다. 소프트웨어를 구축할 때 다음 제품 및 서비스를 고려하십시오.

콘텐츠 전송 네트워크(CloudFront, CloudFlare)

DNS 관리(Route 53)

캐싱(ElastiCache)

소스 제어(GitHub, GitLab)

지속적인 통합 및 배포(GitHub 작업)

모든 서비스 제안에 대해 상황에 따라 동일하거나 더 나은 대안을 찾을 수 있습니다. 더 많은 연구를 수행하고 현재 사용 가능한 다양한 서비스를 탐색할 것을 촉구합니다.

부록 B

클라우드 설정

이 책에서 설명하는 대부분의 아키텍처는 AWS를 기반으로 구축되었습니다. 즉, 보안, 알림 및 비용의 관점에서 AWS에 대한 명확한 이해가 필요합니다. Lambda만 사용하는 여러 서비스를 혼합하여 사용하는 상관없습니다. 보안을 구성할 수 있고, 경고를 설정하는 방법을 알고, 비용을 관리하는 것이 중요합니다. 이 부록은 이러한 우려 사항을 이해하고 AWS에서 중요한 정보를 찾을 수 있는 위치를 배울 수 있도록 설계되었습니다.

AWS 보안은 복잡한 주제이지만 이 부록에서는 사용자와 역할 간의 차이점에 대한 개요를 제공하고 정책을 생성하는 방법을 보여줍니다. 이 정보는 서비스가 효과적이고 안전하게 통신할 수 있는 시스템을 구성하는 데 필요합니다. 때로는 정책을 직접 만들거나 구성할 필요가 없습니다. 서비스 프레임워크와 같은 도구가 당신을 위해 그것을 할 것입니다. 그러나 조각이 어떻게 서로 맞물리는지 이해하고 문제가 발생할 경우 어디에 도움을 청해야 하는지 이해하는 것이 여전히 중요합니다.

비용은 AWS와 같은 플랫폼을 사용하고 서비스 아키텍처를 구현할 때 중요한 고려 사항입니다. 사용하려는 서비스의 비용 계산을 이해하는 것이 중요합니다. 이는 청구서 쇼크를 방지할 뿐만 아니라 다음 달 청구서 및 그 이후를 예측하는 데에도 유용합니다. 우리는 서비스 비용 추정을 살펴보고 비용을 추적하고 비용을 통제하기 위한 전략에 대해 논의합니다. 이 부록은 AWS에 대한 완전한 안내서가 아닙니다. 이 부록을 읽은 후 추가 질문이 있는 경우 AWS 설명서 (<https://aws.amazon.com/documentation>)를 참조하십시오.

B.1 보안 모델 및 ID 관리

2장에서는 Lambda, S3 및 MediaConvert

를 사용하기 위해 IAM(Identity and Access Management) 사용자와 여러 역할을 생성했습니다. 이 섹션에서는 새로 발견한 지식을 사용하여 사용자, 그룹, 역할 및 정책에 대해 더 자세히 학습하여 지식을 더욱 발전시킬 것입니다.

B.1.1 IAM 사용자 생성 및 관리

기억하시겠지만, IAM 사용자는 인간 사용자, 애플리케이션 또는 서비스를 식별하는 AWS의 엔터티입니다. 사용자는 일반적으로 다음을 수행할 수 있는 자격 증명 및 권한 집합을 가지고 있습니다.
AWS에서 리소스 및 서비스에 액세스하는 데 사용됩니다.

IAM 사용자는 일반적으로 사용자를 식별하는 데 도움이 되는 친숙한 이름을 가지고 있습니다.
AWS에서 고유하게 식별하는 Amazon 리소스 이름(ARN)입니다. 그림 B.1
Alfred라는 가상의 사용자에 대한 요약 페이지와 ARN을 보여줍니다. 당신은 얻을 수 있습니다
IAM을 클릭하고 탐색에서 사용자를 클릭하여 AWS 콘솔에서 이 요약
창을 클릭한 다음 보려는 사용자의 이름을 클릭합니다.

그림 B.1 IAM 콘솔은 계정의 모든 IAM 사용자에 대한 ARN, 그룹 및 생성 시간과 같은 메타데이터를 보여줍니다.

- 인간 사용자, 애플리케이션 또는 서비스를 나타내는 IAM 사용자를 생성할 수 있습니다. 그래요
응용 프로그램이나 서비스를 대신하여 작업하도록 생성된 사용자가 참조되는 경우가 있습니다.
- 서비스 계정으로 . 이러한 유형의 IAM 사용자는 다음을 사용하여 AWS 서비스 API에 액세스할 수 있습니다.
액세스 키. IAM 사용자에 대한 액세스 키는 사용자가 처음에 있을 때 생성될 수 있습니다.
생성하거나 IAM 콘솔에서 사용자를 클릭하고 나중에 생성할 수 있습니다.
필수 사용자 이름, 보안 자격 증명을 선택한 다음 만들기를 클릭합니다.
액세스 키 버튼.
액세스 키의 두 가지 구성 요소는 액세스 키 ID와 비밀 액세스입니다.
열쇠. Access Key ID는 공개적으로 공유할 수 있지만 Secret Access Key는 유지해야 합니다.
숨겨진. Secret Access Key가 노출되면 전체 키를 즉시 무효화하고 다시 생성해야 합니다. IAM 사용자는 최
대 2개의 활성 액세스 키를 가질 수 있습니다.
- IAM 사용자가 실제 사람에 대해 생성된 경우 해당 사용자에게 할당되어야 합니다.
비밀번호. 이 암호를 사용하면 사람이 AWS 콘솔에 로그인하여 서비스와 API를 직접 사용할 수 있습니다.
IAM 사용자의 암호를 생성하려면 다음 단계를 따르십시오.

 1. IAM 콘솔의 탐색 창에서 사용자를 클릭합니다.
 2. 필요한 사용자 이름을 클릭하여 사용자 설정을 엽니다.
 3. 보안 자격 증명 탭을 클릭한 다음 콘솔 암호 옆에 있는 관리를 클릭합니다(그림 B.2).

Summary

User ARN arn:aws:iam::571060592278:user/Alfred

Path /

Creation time 2021-09-27 11:41 UTC+1000

| | | | | |
|-------------|--------|------|-----------------------------|----------------|
| Permissions | Groups | Tags | Security credentials | Access Advisor |
|-------------|--------|------|-----------------------------|----------------|

Sign-in credentials

| | |
|-----------------------------|--|
| Summary | <ul style="list-style-type: none"> User does not have console management access |
| Console password | Disabled Manage |
| Assigned MFA device | Not assigned Manage |
| Signing certificates | None |

관리 옵션은 모든 IAM 사용자가 사용할 수 있습니다.
암호가 있는 사용자는 AWS 콘솔에 로그인할 수 있습니다.

그림 B.2 IAM 사용자는 암호 설정, 액세스 키 변경, 다단계 인증 활성화를 비롯한 다양한 옵션을 사용할 수 있습니다.

4. 팝업에서 콘솔 액세스를 활성화할지 비활성화할지 선택하고 새

사용자 지정 암호를 사용하거나 시스템에서 자동 생성하도록 합니다. 당신은 또한 강제할 수 있습니다
사용자는 다음 로그인 시 새 암호를 생성합니다(그림 B.3).

Manage console access

Manage Alfred's AWS console access and password.

Console access Enable
 Disable
Disabling will remove pre-existing password.

Set password* Autogenerated password
 Custom password

Require password reset User must create a new password at next sign-in

Cancel **Apply**

올바른 암호 정책이 설정되어 있는 한 사용자에게 새 암호를 설정
하도록 요청하는 것이 좋습니다.

그림 B.3 사용자가 AWS 콘솔에 로그인할 수 있도록 허용하는 경우 복잡성이 높은 우수한 암호 정책을 생성해야 합니다. 암호 정책은 IAM 콘솔의 계정 설정에서 설정할 수 있습니다.

사용자에게 암호가 할당되면 <https://<Account-ID>.signin.aws.amazon.com/console>로 이동하여 AWS 콘솔에 로그인할 수 있습니다. 계정 ID를 얻으려면 오른쪽 상단 탐색 모음에서 지원을 클릭한 다음 지원 센터를 클릭합니다. 계정 ID(또는 계정 번호)는 콘솔 상단에 표시됩니다. 사용자가 기억하지 않아도 되도록 계정 ID에 대한 별칭을 설정할 수도 있습니다(별칭에 대한 자세한 내용은 <http://amzn.to/1MgvWvf> 참조).

다단계 인증

다단계 인증(MFA)은 사용자가 콘솔에 로그인을 시도할 때 MFA 장치에서 인증 코드를 입력하라는 메시지를 표시하여 보안의 또 다른 계층을 추가합니다(일반적인 사용자 이름 및 암호에 추가됨). 공격자가 계정을 손상시키는 것을 더 어렵게 만듭니다. 모든 최신 스마트폰은 Google Authenticator 또는 AWS Virtual MFA와 같은 애플리케이션을 사용하여 가상 MFA 어플라이언스로 작동할 수 있습니다. AWS 콘솔을 사용할 수 있는 모든 사용자에 대해 MFA를 활성화하는 것이 좋습니다. 콘솔에서 IAM 사용자를 클릭하면 보안 자격 증명 탭에서 MFA 디바이스 할당 옵션을 찾을 수 있습니다.

임시 보안 자격 증명 현재 AWS 계정당 사용자 수

는 5,000명으로 제한되어 있지만 필요한 경우 한도를 늘릴 수 있습니다. 사용자 수를 늘리는 대신 임시 보안 자격 증명을 사용하는 것이 좋습니다. 임시 보안 자격 증명은 잠시 후 만료되도록 설정할 수 있으며 동적으로 생성할 수 있습니다. <http://mng.bz/drnN> 에서 Amazon의 온라인 설명서를 참조하십시오. 임시 보안 자격 증명에 대한 자세한 내용은 IAM 사용자에 대한 자세한 정보는 <http://mng.bz/r6zB>에서 확인할 수 있습니다.

B.1.2 그룹 그룹은

IAM 사용자 모음을 나타냅니다. 한 번에 여러 사용자에 대해 임무별로 쉽게 지정할 수 있는 방법을 제공합니다. 예를 들어 조직의 개발자 또는 테스터를 위한 그룹을 생성하거나 Lambda라는 그룹을 만들어 해당 그룹의 모든 구성원이 Lambda 기능을 실행할 수 있도록 할 수 있습니다. Amazon은 권한을 개별적으로 정의하는 것보다 그룹을 사용하여 IAM 사용자에게 권한을 할당할 것을 권장합니다. 그룹에 가입한 모든 사용자는 그룹에 할당된 권한을 상속합니다. 마찬가지로 사용자가 그룹을 떠나면 그룹의 권한이 사용자에게서 제거됩니다. 또한 그룹은 역할과 같은 다른 그룹이나 엔터티가 아닌 사용자만 포함할 수 있습니다.

B.1.3 역할

역할은 사용자, 애플리케이션 또는 서비스가 일정 기간 동안 맡을 수 있는 권한 집합입니다. 역할은 특정 사용자에게 고유하게 연결되지 않으며 암호나 액세스 키와 같은 연결된 자격 증명도 없습니다. 일반적으로 필요한 리소스에 대한 액세스 권한이 없는 사용자 또는 서비스에 권한을 부여하도록 설계되었습니다.

위임은 역할과 관련된 중요한 개념입니다. 위임 이란 쉽게 말해서 특정 리소스에 대한 액세스를 허용하기 위해 제3자에게 권한을 부여하는 것과 관련됩니다. 그것은 신뢰하는 계정 사이의 신뢰 관계를 설정하는 것을 포함합니다. 리소스를 소유하고 사용자 또는 애플리케이션을 포함하는 신뢰할 수 있는 계정 리소스에 액세스해야 합니다. 그림 B.4는 신뢰 관계가 있는 역할을 보여줍니다. CloudCheckr라는 서비스를 위해 설정되었습니다.

The screenshot shows the AWS IAM Roles page with the CloudCheckrRole selected. The top navigation bar shows 'Roles > CloudCheckrRole'. The main section is titled 'Summary' with a 'Delete role' button. Below are several details:

- Role ARN:** arn:aws:iam::038221756127:role/CloudCheckrRole
- Role description:** Edit
- Instance Profile ARNs:** [Edit]
- Path:** /
- Creation time:** 2016-01-30 15:39 UTC+1000
- Last activity:** Not accessed in the tracking period
- Maximum session duration:** 1 hour Edit
- Give this link to users who can switch roles in the console:** <https://signin.aws.amazon.com/switchrole?roleName=CloudCheckrRole>

Below the summary are tabs: Permissions, Trust relationships, Tags, Access Advisor, and Revoke sessions. The 'Trust relationships' tab is selected. It contains the following information:

You can view the trusted entities that can assume the role and the access conditions for the role. Show policy document

Edit trust relationship

Trusted entities: The following trusted entities can assume this role.

| Trusted entities |
|------------------|
| The account 352 |

Conditions: The following conditions define how and when trusted entities can assume the role.

| Condition | Key | Value |
|--------------|----------------|----------|
| StringEquals | sts:ExternalId | CC-3985C |

Annotations with arrows point from the text '신뢰할 수 있는 엔티티는 역할을 맡을 수 있는 엔티티를 정의합니다.' to the 'Trusted entities' table, and from the text '외부 ID는 권한 상승의 한 형태인 혼란스러운 대리인 문제를 방지합니다. 제3자가 AWS 계정에 액세스할 수 있도록 액세스를 구성한 경우 필요합니다.' to the 'Conditions' table.

그림 B.4 이 역할은 CloudCheckr에게 AWS 계정에 대한 액세스 권한을 부여하여 비용 분석을 수행하고 개선 사항을 권장합니다.

연합은 역할의 맥락에서 자주 논의되는 또 다른 개념입니다. 연합 외부 ID 제공자 간의 신뢰 관계를 생성하는 프로세스입니다. Facebook, Google 또는 보안을 지원하는 엔터프라이즈 ID 시스템과 같은 SAML(Assertion Markup Language) 2.0 및 AWS. 사용자가 다음 중 하나를 통해 로그인할 수 있습니다. 이러한 외부 자격 증명 공급자와 임시 자격 증명으로 IAM 역할을 맡습니다.

B.1.4 자원

AWS의 권한은 자격 증명 기반 또는 리소스 기반입니다. ID 기반 권한 IAM 사용자 또는 역할이 수행할 수 있는 작업을 지정합니다. 리소스 기반 권한은 다음을 지정합니다. S3 버킷 또는 SNS 주제와 같은 AWS 리소스는 수행할 수 있거나 가질 수 있는 사람

그것에 대한 액세스. 리소스 기반 정책은 종종 누가 주어진 정보에 액세스할 수 있는지 지정합니다. 자원. 이를 통해 신뢰할 수 있는 사용자는 다음을 가정할 필요 없이 리소스에 액세스할 수 있습니다. 역할. <http://mng.bz/VBJP> 의 AWS 사용 설명서 상태:

리소스 기반 정책을 통한 교차 계정 액세스는 역할보다 이점이 있습니다. 와 리소스 기반 정책을 통해 액세스되는 리소스에 대해 사용자는 여전히 신뢰할 수 있는 역할 대신 사용자 권한을 포기할 필요가 없습니다. 권한. 즉, 사용자는 신뢰할 수 있는 리소스에 계속 액세스할 수 있습니다. 동시에 신뢰하는 계정의 리소스에 액세스할 수 있습니다.

모든 AWS 서비스가 리소스 기반 정책을 지원하는 것은 아닙니다(사용 설명서 <http://mng.bz/xX8W> 는 수행하는 모든 서비스를 나열합니다).

B.1.5 권한 및 정책

IAM 사용자를 처음 생성할 때 IAM 사용자는 액세스하거나 수행할 수 없습니다.

계정. 설명하는 정책을 생성하여 사용자 권한을 부여해야 합니다.

사용자가 할 수 있는 것. 새로운 그룹이나 역할도 마찬가지입니다. 새로운 그룹이나 영향을 미치려면 역할에 정책을 할당해야 합니다.

모든 정책의 범위는 다를 수 있습니다. 사용자 또는 역할에 관리자 액세스 권한을 부여할 수 있습니다. 전체 계정에 적용하거나 개별 작업을 지정합니다. 세분화하고 지정하는 것이 좋습니다.

작업을 완료하는 데 필요한 권한만 있습니다(최소 권한 액세스). 시작

최소한의 권한 집합을 만들고 필요한 경우에만 추가 권한을 추가합니다.

관리 형 정책 과 인라인 정책의 두 가지 유형이 있습니다. 관리형 정책 적용 대상

리소스가 아닌 사용자, 그룹 및 역할. 관리형 정책은 독립 실행형입니다. 일부

관리형 정책은 AWS에서 생성 및 유지 관리합니다. 당신은 또한 만들 수 있습니다

고객 관리 정책을 유지합니다. 관리형 정책은 재사용성과

변경 관리. 고객 관리형 정책을 사용하고 수정하기로 결정한 경우,

모든 변경 사항은 정책이 연결된 모든 IAM 사용자, 역할 및 그룹에 자동으로 적용됩니다. 관리형 정책을 사용하면 버전 관리 및 룰백이 더 쉬워집니다.

인라인 정책이 생성되어 특정 사용자, 그룹 또는 역할에 직접 연결됩니다.

엔터티가 삭제되면 엔터티에 포함된 인라인 정책도 삭제됩니다.

리소스 기반 정책은 항상 인라인입니다. 인라인 또는 관리형 정책을 추가하려면

필요한 사용자, 그룹 또는 역할을 선택한 다음 권한 탭을 클릭합니다. 불일 수 있고, 관리형 정책을 보거나 분리하고 유사하게 인라인 정책을 생성, 보거나 제거합니다.

정책은 JSON 표기법을 사용하여 지정됩니다. 다음 목록은 관리되는 AWSLambdaExecute 정책.

목록 B.1 AWSLambdaExecute 정책

버전은 정책 언어 버전을 지정합니다. 현재 버전은 2012-10-17입니다. 사용자 지정 정책을 생성하는 경우 버전을 포함하고 2012-10-17로 설정해야 합니다.

```
{
    "버전": "2012-10-17",
    "설명": {
        "효과": "허용",
        "정책": "정책을 구성하는 실제 권한을 지정하는 하나 이상의 명령문을 포함합니다."
    }
}
```

```

    "작업": "로그:*",
    "리소스": "arn:aws:logs:*:*"
},
{
    "효과": "허용",
    "작업": [
        "s3:GetObject",
        "s3:PutObject"
    ],
    "리소스": "arn:aws:s3:::/*"
}
]
}

```

Effect 요소는 필수 요소이며 명령문이 리소스에 대한 액세스를 허용할지 또는 거부할지 여부를 지정합니다. 사용 가능한 옵션은 허용 및 거부뿐입니다.

허용하거나 거부해야 하는 리소스에 대한 특정 작업을 지정합니다. 와일드카드(*) 문자 사용 허용됩니다(예: "Action": "s3:*").

Resource 요소는 명령문이 적용되는 개체를 식별합니다. 구체적이거나 여러 엔터티를 참조하는 와일드카드를 포함합니다.

많은 IAM 정책에는 Principal, Sid 및 Condition 과 같은 추가 요소가 포함되어 있습니다 . Principal 요소는 IAM 사용자, 계정 또는 다음과 같은 서비스를 지정합니다 . 리소스에 대한 액세스를 허용하거나 거부합니다. Principal 요소는 정책에서 사용되지 않습니다 . IAM 사용자 또는 그룹에 연결된 대신 누가 할 수 있는지 지정하기 위해 역할에서 사용됩니다. 역할을 맙습니다. 리소스 기본 정책에도 일반적입니다. 명세서 ID (Sid) : SNS와 같은 특정 AWS 서비스에 대한 정책에서 필요합니다. 조건을 통해 다음을 수행할 수 있습니다. 정책을 적용해야 하는 시기를 지시하는 규칙을 지정합니다. 조건의 예는 다음 목록에 나와 있습니다.

목록 B.2 정책 조건

DateEquals, DateLessThan, DateMoreThan, StringEquals, StringLike, StringNotEquals 및 ArnEquals를 비롯한 여러 조건부 요소를 사용할 수 있습니다.

```

    "상태": {
        "DateLessThan": {
            "aws:CurrentTime": "2020-09-12T12:00:00Z"
        },
        "IP 주소": {
            "aws:SourceIp": "127.0.0.1"
        }
    }
}

```

조건 키는 사용자가 발행한 요청에서 가져온 값을 나타냅니다. 가능한 키는 SourceIp, CurrentTime, Referer, SourceArn, 사용자 ID 및 사용자 이름입니다. 값은 "127.0.0.1"과 같은 특정 리터럴 값 또는 정책 변수.

여러 조건

<http://amzn.to/21UofNi> 의 AWS 설명서 조건 연산자가 여러 개 있거나 단일 조건 연산자에 여러 키가 연결된 경우

조건은 논리적 AND를 사용하여 평가됩니다. 단일 조건 연산자인 경우

하나의 키에 대해 여러 값을 포함하는 경우 해당 조건 연산자는 논리 OR를 사용하여 평가됩니다." <http://amzn.to/21UofNi> 참조 당신이 따를 수 있는 훌륭한 예와 전체 유용한 문서 더미.

Amazon은 보안에 실용적인 범위에서 조건을 사용할 것을 권장합니다. 그만큼 예를 들어 다음 목록은 콘텐츠를 강제로 제공하는 S3 버킷 정책을 보여줍니다. HTTPS/SSL을 통해서만 가능합니다. 이 정책은 암호화되지 않은 HTTP를 통한 연결을 거부합니다.

Listing B.3 HTTPS/SSL 시행을 위한 정책

```
{
    "버전": "2012-10-17",
    "아이디": "123",
    "성명": [
        {
            "효과": "거부",
            "주요한": "*",
            "동작": "s3:*",
            "리소스": "arn:aws:s3:::my-bucket/*",
            "상태": {
                "불": {
                    "aws:SecureTransport": 거짓
                }
            }
        }
    ]
}
```

조건이 충족되면 명시적으로 s3에 대한 액세스를 거부합니다.



SSL을 사용하여 요청을 보내지 않으면 조건이 충족됩니다. 이렇게 하면 사용자가 암호화되지 않은 일반 HTTP를 통해 버킷에 액세스하려고 하면 정책이 버킷에 대한 액세스를 차단합니다.

B.2 비용

월말에 큰 청구서 형태로 불쾌한 놀라움을 받는 것은

실망스럽고 스트레스. Amazon CloudWatch는 다음을 전송하는 결제 경보를 생성할 수 있습니다.

해당 월의 총 요금이 사전 정의된 임계값을 초과하는 경우 알림. 이것은 예기치 않게 큰 청구서를 방지할 뿐만 아니라 시스템의 잠재적인 구성 오류를 포착하는 데 유용합니다.

예를 들어, Lambda 함수를 잘못 구성하여 실수로 3.0GB의 RAM을 할당하기 쉽습니다. 함수는 다음을 기다리는 것 외에는 유용한 작업을 수행하지 않을 수 있습니다.

데이터베이스로부터 응답을 받는 데 15초. 과중한 환경에서 시스템

\$1,462가 조금 넘는 비용으로 한 달에 2백만 회의 함수 호출을 수행할 수 있습니다.

128MB RAM이 있는 동일한 기능의 비용은 한 달에 약 \$56입니다. 비용 계산을 미리 수행하고 합리적인 청구 알람이 있으면 빨리 깨닫게 될 것입니다.

결제 알림이 오기 시작할 때 무언가가 일어나고 있다는 것입니다.

B.2.1 결제 알림 생성

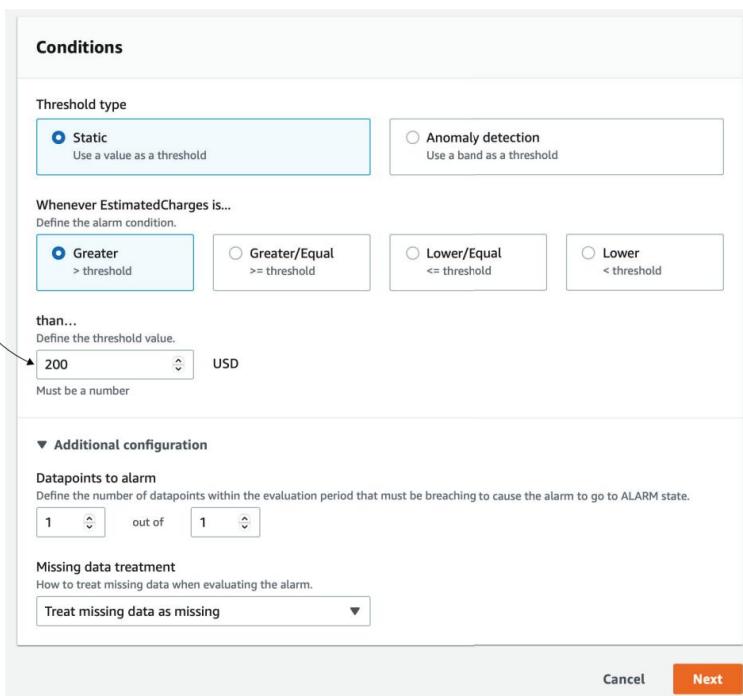
결제 알림을 생성하려면 다음 단계를 따르세요.

1. 기본 AWS 콘솔에서 자신의 이름(또는 해당 IAM 사용자의 이름)을 클릭합니다.
귀하를 대표함)을 클릭한 다음 내 결제 대시보드를 클릭합니다.
2. 탐색 창에서 결제 기본 설정을 클릭한 다음 확인란을 활성화합니다.
결제 알림 수신 옆에 있습니다.
3. 기본 설정 저장을 클릭한 다음 기본 AWS 콘솔로 돌아가서
클라우드워치 서비스.

4. CloudWatch 서비스를 열고 경보를 클릭한 다음 탐색 창에서 모든 경보를 선택합니다. 경보 생성 버튼을 클릭한 다음 메트릭 선택 버튼을 클릭합니다.
5. 메트릭 제목 아래에서 청구를 선택하고 총 예상 요금을 클릭합니다. (만약에 청구가 표시되지 않으면 청구 수신을 활성화하지 않았을 수 있습니다. 단계 2)의 경고 옵션.)
6. EstimatedCharges 확인란을 선택하고 Select metric to continue를 클릭합니다.
7. 임계값 유형이 정적으로 설정되어 있고 예상 요금이 있을 때마다 더 크게 설정되어 있는지 확인합니다.

8. 임계값 정의에서 트리거하려는 금액을 입력합니다.
알람(예: 그림 B.5에서 볼 수 있는 200).
9. 다음을 클릭하여 다음 페이지로 계속 진행합니다.
여기에서 알람이 울리면 알려주는 새 SNS 주제를 설정하거나 생성할 수 있습니다.
발동. 이것은 중요하다! 알림 이메일을 수신하려면 SNS 주제가 필요합니다.
무슨 일이 일어나고 있는지.
10. 알림 추가 버튼을 클릭합니다.
11. 새 주제 생성을 선택하고 이름을 입력한 다음 이메일을 입력합니다.
주소. 주제 생성 버튼을 클릭하여 SNS 주제 설정을 저장합니다. 때를
계속할 준비가 되었습니다. 다음을 클릭하십시오.
12. 알람 이름을 입력하고 다음을 다시 클릭합니다.
13. 마지막으로 하단의 알람 생성 버튼을 클릭하여 완료합니다.

이 핵심은 예산을 초과했을 때 알려주는 금액을 설정하는 것입니다.
항상 예산을 유지하고 비용이 폭발하지 않도록 하고 싶습니다.



Conditions

Threshold type

Static
Use a value as a threshold

Anomaly detection
Use a band as a threshold

Whenever EstimatedCharges is...
Define the alarm condition.

Greater
> threshold

Greater/Equal
>= threshold

Lower/Equal
<= threshold

Lower
< threshold

than...
Define the threshold value.

200 USD
Must be a number

▼ Additional configuration

Datapoints to alarm
Define the number of datapoints within the evaluation period that must be breaching to cause the alarm to go to ALARM state.

1 out of 1

Missing data treatment
How to treat missing data when evaluating the alarm.

Treat missing data as missing

Cancel Next

그림 B.5 지속적인 비용에 대한 정보를 제공하기 위해 여러 청구 경보를 생성하는 것이 좋습니다.

B.2.2 비용 모니터링 및 최적화

CloudCheckr (<http://cloudcheckr.com>) 와 같은 서비스 비용을 추적하는 데 도움이 될 수 있습니다.

사용 중인 서비스 및 리소스를 분석하여 경고를 제공하고 비용 절감을 제안할 수도 있습니다. 클라우드 채커 S3, CloudSearch, SES, SNS 및

다이나모DB. 일부 표준 AWS 기능보다 기능이 더 풍부하고 사용하기 쉽습니다. 권장 사항 및 일일 알림을 고려 할 가치가 있습니다.

AWS에는 성능, 내결함성, 보안 및 비용 최적화 개선을 제안하는 Trusted Advisor라는 서비스도 있습니다. 불행히도 Trusted Advisor의 무료 버전은 제한되어 있으므로 모든 기능과

제공해야 하는 권장 사항을 보려면 유료 월간 요금제로 업그레이드하거나 액세스해야 합니다.

AWS 엔터프라이즈 계정을 통해

비용 탐색기(그림 B.6)는 비록 높은 수준의 보고 및 분석 도구이기는 하지만 유용합니다.

AWS에 내장되어 있습니다. 먼저 이름(또는 IAM 사용자 이름)을 클릭하여 활성화해야 합니다.

AWS 콘솔의 오른쪽 상단 모서리에서 내 결제 대시보드를 선택한 다음

탐색 창에서 비용 탐색기를 클릭하고 활성화합니다. 비용 탐색기는 이번 달과 지난 4개월 동안의 비용을 분석합니다. 그런 다음 다음 3개월 동안 포캐스트를 생성합니다. 처음에는 정보가 표시되지 않을 수 있습니다.

AWS가 이번 달의 데이터를 처리하는 데 24시간이 걸립니다. 이전 달의 데이터를 처리하는 데 시간이 더 오래 걸립니다. 비용 탐색기에 대한 자세한 내용은 <http://amzn.to/1KvN0g2>에서 확인할 수 있습니다.

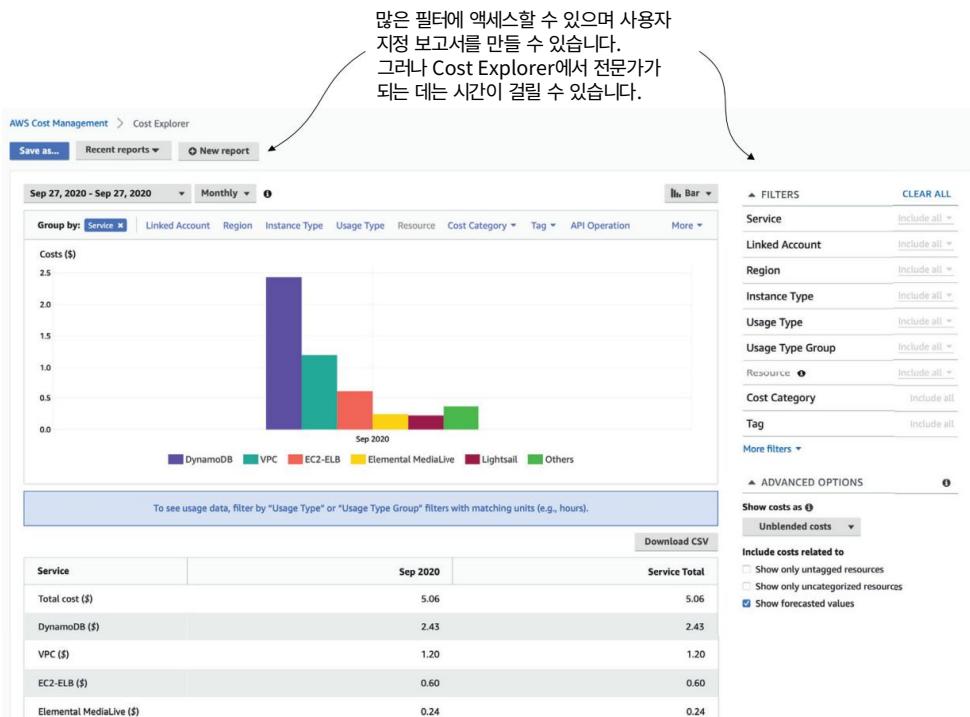


그림 B.6 비용 탐색기를 사용하면 과거 비용을 검토하고 향후 비용을 추정할 수 있습니다.

B.2.3 간단한 월별 계산기 사용하기

AWS 요금 계산기 (<https://calculator.aws>) 개발된 웹 애플리케이션입니다.

Amazon에서 많은 서비스에 대한 비용 모델링을 돋습니다. 이 도구를 사용하면 특정 자원의 소비와 관련된 정보를 입력하고, 표시 비용을 얻습니다.

B.2.4 Lambda 및 API Gateway 비용 계산

서비스 아키텍처를 실행하는 비용은 종종 기존 인프라를 실행하는 것보다 훨씬 적습니다. 당연히 사용하는 서비스마다 비용이 다르겠지만, 하지만 Lambda와 API를 사용하여 서비스 시스템을 실행하는 데 필요한 것을 볼 수 있습니다. 게이트웨이.

Amazon의 Lambda 요금 (<https://aws.amazon.com/lambda/pricing/>) ~이다

요청 수, 실행 시간 및 메모리 양에 따라

기능에 할당됩니다. 처음 100만 건은 무료이며 이후 100만 건은 0.20 USD의 요금이 부과됩니다. 시간은 함수를 실행하는 데 걸리는 시간을 기반으로 합니다.

밀리초(ms)로 측정됩니다. Amazon은 1ms 단위로 요금을 청구하는 동시에 함수에 예약된 메모리 양을 고려합니다. 1로 만든 함수

GB 메모리 비용은 실행 시간 100ms당 \$0.000001667인 반면, 128MB 메모리로 생성된 함수는 100ms당 \$0.000000208입니다.

참고 아마존 가격은 지역에 따라 다를 수 있습니다.

언제든지 변경될 수 있습니다.

Amazon은 1백만 개의 무료 요청과 400,000GB-초의 영구 프리 티어를 제공합니다.

월별 컴퓨팅 시간. 이는 사용자가 백만 개의 요청을 수행할 수 있음을 의미합니다.

1GB로 생성된 함수를 실행하는 데 400,000초에 해당하는 시간을

지불해야 하기 전에 메모리. 예를 들어 다음과 같은 시나리오를 고려하십시오.

256MB 기능을 한 달에 5백만 번 실행합니다. 함수는 매번 2초 동안 실행됩니다. 비용 계산은 다음과 같습니다.

월별 요청 요금: – 프리 티어는 1백

만 개의 요청을 제공합니다. 즉, 4개의 요청만 있습니다.

백만 개의 청구 가능 요청(5M 요청 – 1M 무료 요청 = 4M 요청).

– 각 백만 달러의 가격은 \$0.20이므로 요청 요금은 \$0.80(4M

요청 × \$0.2/M = \$0.80).

월별 컴퓨팅 요금:

– GB-초당 함수의 컴퓨팅 가격은 \$0.00001667입니다. 무료

계층은 400,000GB-초를 무료로 제공합니다.

– 가격 계산 시나리오에서 함수는 10ms(5M × 2s) 동안 실행됩니다.

– 256MB 메모리에서 10M초는 2,500,000GB-초에 해당

($10,000,000 \times 256\text{MB} / 1024 = 2,500,000$).

– 해당 월의 총 청구 가능한 GB-초는 2,100,000입니다.

($2,500,000\text{GB-초} - 400,000\text{프리 티어 GB-초} = 2,100,000$). 그만큼

따라서 컴퓨팅 요금은 $\$35.007(2,100,000\text{GB-초} \times \$0.00001667 = \$35.007)$.

- 이 예에서 Lambda를 실행하는 데 드는 총 비용은 \$35.807입니다.

API Gateway 가격은 수신된 API 호출 수와

AWS에서 전송된 데이터의 양. 미국 동부에서는 Amazon이 청구합니다.

수신된 100만 API 호출당 \$3.50 및 처음 10TB 전송에 대해 \$0.09/GB

밖으로. 이전 예에서 월별 아웃바운드 데이터 전송이 다음과 같다고 가정합니다.

월 100GB, API Gateway 가격은 다음과 같습니다.

월간 API 요금:

- 프리 티어는 매월 1M API 호출을 포함하지만 12회까지만 유효합니다.

개월. 영구 프리 티어가 아니므로 여기에 포함되지 않습니다.

계산.

- 총 API 비용은 \$17.50입니다($5\text{M 요청} \times \$3.50/\text{M} = \17.50).

월 데이터 요금은 $\$9.00(100\text{GB} \times \$0.09/\text{GB} = \$9)$ 입니다.

이 예에서 API 게이트웨이 비용은 \$26.50입니다.

Lambda 및 API Gateway의 총 비용은 월 \$62.307입니다.

얼마나 많은 요청과 작업이 있는지 모델링해 볼 가치가 있습니다.

지속적으로 처리합니다. Lambda 함수의 2M 호출이 예상되는 경우

128MB의 메모리만 사용하고 1초 동안 실행되는 경우 약

\$0.20 월. 512MB RAM이 있는 함수의 2M 호출이 예상되는 경우

5초 동안 실행하면 \$75.00보다 약간 더 많은 비용을 지불하게 됩니다. Lambda를 사용하면

비용을 평가하고 미리 계획하고 실제로 사용한 만큼만 비용을 지불할 수 있는 기회. 드디어,

S3나 SNS와 같은 다른 서비스는 아무리 사소한 것이라도 고려하는 것을 잊지 마십시오.

그들의 비용은 그렇게 보일 수 있습니다.

부록 C

배포 프레임워크

AWS와 같은 클라우드 플랫폼에서 무엇이든 구축하는 경우 자동화 및 지속적 전달이 중요합니다. 서비스 접근 방식을 취하는 경우 더 많은 서비스, 더 많은 기능 및 구성할 항목이 더 많아지기 때문에 서비스 접근 방식이 훨씬 더 중요해집니다. 전체 애플리케이션을 스크립팅하고 테스트를 실행하고 자동으로 배포할 수 있어야 합니다. 수동으로 Lambda 함수를 배포하거나 API Gateway를 자체 구성해야 하는 유일한 시간은 학습하는 동안입니다. 실제 서비스 애플리케이션 작업을 시작하면 반복 가능하고 자동화된 강력한 시스템 프로비저닝 방법이 필요합니다. Terraform을 제외하고 이 부록에서 논의된 다른 프레임 작업은 자체적으로 리소스를 프로비저닝하지 않습니다. 대신 AWS CloudFormation (<https://aws.amazon.com/cloudformation/>)에 의존합니다. 리소스를 프로비저닝하므로 CloudFormation의 제한 사항이 적용됩니다.

여기에는 다음이 포함됩니다.

CloudFormation 템플릿에는 500개 이하의 리소스가 포함될 수 있습니다. 이 제한을 초과하려면 중첩된 CloudFormation 스택을 사용할 수 있습니다.

CloudFormation 템플릿에는 200개 이하의 매개변수 또는 출력이 포함될 수 있습니다.

CloudFormation 스택에 기존 리소스를 추가하는 것은 번거롭습니다.

Terraform은 이러한 제한을 완화하지만 나름대로의 단점이 있습니다. 그 중 가장 눈에 띄는 것은 룰렛에 대한 지원이 부족하다는 것입니다. 배포 중에 문제가 발생한 경우 일부 리소스는 업데이트되지만 다른 리소스는 업데이트되지 않으면 애플리케이션이 손상된 상태가 될 수 있습니다.

이 부록에서 논의된 일부 프레임워크는 Lambda 함수를 로컬로 호출하거나 API Gateway를 로컬로 시뮬레이션하는 기능과 같은 추가 유ти리티도 제공합니다. 이제 서비스 애플리케이션을 위한 가장 인기 있는 배포 프레임워크 중 일부를 살펴보겠습니다.

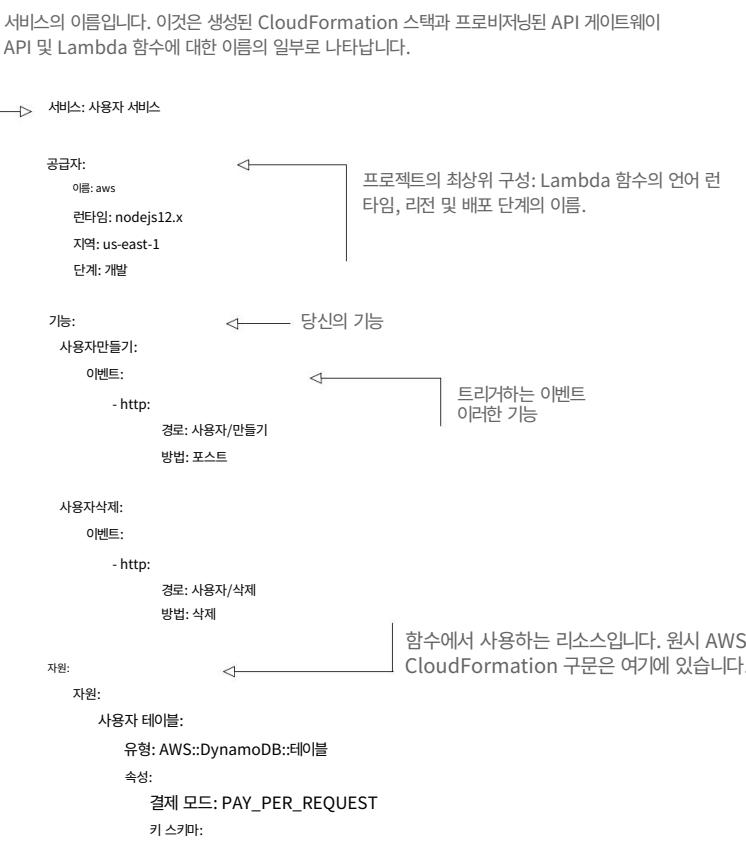
C.1 서비스 프레임워크

서비스 프레임워크 (<https://serverless.com>) 오픈 소스 프레임워크이며 가장 인기 있고 성숙한 배포 프레임워크 중 하나입니다. 그것의 본질적으로 사용자는 전체 서비스 애플리케이션(Lambda 포함)을 정의할 수 있습니다. 함수, API Gateway API, SNS 주제 및 기타 CloudFormation 리소스) 그런 다음 CLI(명령줄 인터페이스)를 사용하여 배포합니다. 정리하는 데 도움이 되고 서비스 애플리케이션을 구조화하여 더 큰 규모의 구축을 시작할 때 큰 이점을 얻을 수 있습니다. 플러그인 시스템을 통해 완전히 확장할 수 있습니다.

C.1.1 시작하기

Serverless Framework는 JSON과 YAML을 모두 지원합니다. 또한 설명할 수 있습니다. 다음 목록에 표시된 것과 같은 매니페스트 파일의 애플리케이션.

목록 C.1 serverless.yml의 서비스 설명



애플리케이션을 배포하려면 단일 명령만 실행하면 됩니다.

서비스 배포

Serverless Framework는 코드를 패키징하여 S3에 업로드하고 CloudFormation을 통해 serverless.yml에 지정된 리소스를 프로비저닝합니다. 다음과 같이 CLI 옵션을 사용하여 기본 지역 및 단계 이름을 덮어쓸 수도 있습니다 (<http://mng.bz/AOJz>).

```
서비스 배포 -s prod -r eu-west-1
```

C.1.2 언어 지원

Serverless Framework는 Node.js, Python, Java, Golang, C# 및 Scala와 같은 여러 언어 런타임을 지원합니다. Serverless Framework가 기능을 패키징하는 방법을 많이 제어할 수 있습니다. 기본적으로 serverless.yml에서 구성한 모든 기능에 대해 동일한 패키지 아티팩트를 사용합니다. 그러나 선택적으로 각 기능을 개별적으로 패키지하고 특정 폴더 또는 파일을 포함하거나 제외할 수 있습니다.

serverless-webpack 플러그인 (<https://bit.ly/sls-webpack>)을 통해, 또한 웹팩을 패키징 프로세스에 통합하여 Java Script 기능을 트리 쉐이크 및 번들로 만들 수 있습니다. 그렇게 하면 훨씬 작은 아티팩트가 생성될 수 있으므로 배포 시간과 콜드 스타트 성능 모두에 도움이 됩니다.

Python 함수의 경우 배포 아티팩트에 타사 라이브러리를 포함하는 것이 어려울 수 있습니다. serverless-python-requirements 플러그인 (<http://bit.ly/sls-python-reqs>)은 이를 투명하게 처리하고 기존 requirements.txt 파일을 사용할 수 있도록 합니다.

C.1.3 로컬에서 기능 호출 서비스 응용 프로

그램을 패키징하고 배포하는 것 외에도 Serverless Framework에는 여러 가지 유용한 유틸리티가 있습니다. 가장 주목할만한 것은 invoke local 명령을 사용하여 로컬에서 기능을 호출하는 기능입니다.

```
서비스 호출 로컬 -f functionName -d "{}"
```

invoke local 명령은 기능을 로컬에서 빠르게 테스트하는 데 유용합니다. 함수를 AWS에 먼저 배포할 필요 없이 빠른 피드백을 제공합니다. 디버거를 연결하고 코드를 한 줄씩 단계별로 실행할 수도 있습니다(자세한 내용은 <http://bit.ly/sls-debug-vscode> 게시물 참조). VS Code로 수행하는 방법).

하지만 API 게이트웨이를 로컬에서 에뮬레이트하려면 어떻게 해야 할까요? 서버 리스 오프라인 플러그인 (<http://bit.ly/sls-offline>) 정확히 그렇게 할 수 있고 localhost 게시물에서 API 게이트웨이를 에뮬레이트합니다. Lambda로 서버 측 렌더링을 수행할 때 이것이 유용하다는 것을 알았습니다.

invoke local을 사용하여 함수를 로컬에서 테스트하고 출력을 검사할 수 있지만 머리에서 HTML을 렌더링할 수는 없습니다! 로컬 엔드포인트가 있으면 브라우저에서 해당 엔드포인트를 가리키고 모든 CSS 영광에서 서버 측 렌더링된 HTML을 검사할 수 있습니다.

C.1.4 플러그인

Serverless Framework에는 프레임워크가 즉시 사용할 수 있는 것 이상으로 기능을 확장하는 풍부한 플러그인 애플리케이션이 있습니다. 일부 플러그인은 Serverless Framework가 생성하는 CloudFormation 템플릿을 수정합니다. 예를 들어 서비스 프레임워크는 프로젝트의 모든 기능에 대해 공유 ID 및 액세스 관리(IAM) 역할을 생성하는 반면 인기 있는 serverless-iam-roles-per-function 플러그인을 사용하면 각 기능에 대한 IAM 역할을 구성할 수 있습니다.

일부 플러그인은 Serverless Framework가 기본적으로 지원하지 않는 서비스에 대한 지원을 추가합니다. 예를 들어 Serverless Framework는 기본적으로 AppSync를 지원하지 않습니다. 원시 CloudFormation 구문(serverless.yml의 리소스 섹션)을 사용하여 serverless.yml에서 AppSync API를 구성할 수 있지만 이는 지루하고 힘든 작업입니다. serverless-appsync-plugin 플러그인은 Serverless Framework를 확장하여 AppSync를 지원하고 훨씬 더 간결한 구문으로 AppSync API를 구성할 수 있도록 합니다. 마찬가지로 serverless-step-functions 플러그인은 Step Functions에 대한 지원을 추가합니다.

일부 플러그인은 Serverless Framework의 CLI에 추가 명령을 추가할 수 있습니다. 예를 들어 serverless-offline 플러그인은 API Gateway의 로컬 인스턴스를 시작하는 오프라인 명령을 추가합니다. 마찬가지로 serverless-export-env 플러그인은 Lambda 함수에서 참조하는 환경 변수를 캡처하고 .env 파일로 내보내는 export-env 명령을 추가합니다.

Serverless Framework에는 유연한 플러그인 아키텍처가 있으며 프레임워크가 수행하는 거의 모든 것을 사용자 정의할 수 있습니다. 이러한 유연성을 통해 프레임워크 기본값에 동의하지 않고 필요에 맞게 동작을 조정할 수 있습니다. 사용 가능한 플러그인의 풍부한 애플리케이션은 AWS SAM과 차별화되는 요소이기도 합니다.

C.2 서비스 애플리케이션 모델(SAM)

서비스 애플리케이션 모델 (<https://aws.amazon.com/serverless/sam>) (SAM)은 서비스 프레임워크에 대한 AWS의 답변이며 서비스 프레임워크와 많은 유사점을 공유합니다. Serverless Framework와 마찬가지로 SAM은 CloudFormation을 사용하여 리소스를 프로비저닝하고 더 간단한(CloudFormation과 비교하여) 구문을 사용하여 Lambda 함수, API 게이트웨이 등의 측면에서 서비스 애플리케이션을 정의할 수 있습니다. 또한 로컬에서 Lambda 함수를 호출하거나 API Gateway의 로컬 인스턴스도 시작할 수 있는 여러 CLI 명령이 있습니다. SAM과 Serverless Framework의 가장 큰 차이점은 SAM의 구문이 원시 CloudFormation 구문에 훨씬 가깝고 플러그인 시스템이 없다는 것입니다.

전자는 종종 서비스 프레임워크보다 SAM을 선호해야 하는 이유로 간주되지만 개인 취향의 문제입니다. 궁극적으로 CloudFormation 구문은 장황합니다. 이것이 더 간단한 구문과 보다 생산적인 추상화 수준을 제공하는 이러한 프레임워크를 사용하여 작업하는 것을 선호하는 이유 중 하나입니다. 그렇다면 구문이 멀리하려고 하는 것에 더 가깝기 때문에 프레임워크를 선호해야 하는 이유는 무엇입니까? 말도 안 돼

반면에 플러그인 시스템의 부족은 종종 거래 차단기입니다. 이는 프레임워크가 지원하는 것에 제한을 받고 프레임워크 기본값을 재정의할 수 있는 쉬운 방법이 없음을 의미합니다(물론 프레임워크에서 구성 가능한 옵션이 아닌 경우).

예를 들어 SAM은 2020년 5월에 Step Functions에 대한 지원을 추가했지만([serverless-step-functions](#) 플러그인이 Serverless Framework에 대해 동일한 작업을 수행한 지 3년 이상 경과됨) 작성 당시에는 여전히 AppSync에 대한 지원이 없습니다([2021년 4월](#)).

그리고 Serverless Framework의 플러그인 시스템은 프레임워크의 기본값에 동의하지 않을 때를 위한 탈출구를 제공하지만 플러그인 시스템이 없기 때문에 프레임워크에서 SAM으로 구성할 수 있는 항목으로 제한됩니다. SAM의 선택에 동의하지 않으려면 CloudFormation 매크로로 문제를 해결하고 해당 매크로를 사용하여 배포 시 SAM에서 생성한 Cloud Formation 템플릿을 수정해야 합니다. 이것이 지루한 해결책처럼 들린다면 그것은 우리가 2년 전에 어렵게 배웠기 때문입니다 (<http://mng.bz/ZxJP>).

SAM은 특정 작업을 매우 잘 수행합니다. 예를 들어 개별 기능에 대한 IAM 역할을 즉시 정의할 수 있습니다. 그리고 API Gate 방식으로 리소스를 프로비저닝하는 방식도 CloudFormation 리소스 수면에서 (Serverless Framework와 비교하여) 더 효율적입니다. 서비스 프레임

작업은 API 리소스와 메서드를 개별 리소스로 프로비저닝하고 SAM은 AWS::ApiGateway::RestApi 리소스의 Body 속성에서 모든 것을 인코딩합니다.

이 접근 방식은 CloudFormation 스택의 리소스 수를 최소화하고 CloudFormation 스택의 리소스 제한 500개에 도달할 위험을 줄이는 데 도움이 됩니다.

이는 대규모 API 프로젝트에서 유용합니다. Serverless Framework를 사용하면 이러한 대규모 프로젝트는 종종 500개 리소스 제한을 해결하기 위해 serverless-plugin-split-stacks 플러그인과 같은 플러그인에 의존해야 합니다.

C.3 테라폼

테라폼 (<https://www.terraform.io>) HashiCorp의 인기 있는 코드로서의 인프라(IaC) 도구입니다. 이것은 이 부록에서 가장 독단적인 프레임워크입니다. "코드로 인프라를 작성, 계획 및 생성"이라는 모토에 따라 Terraform은 오랫동안 인프라 엔지니어에게 사랑받아 왔으며 Lambda를 중심으로 설계되지 않았습니다.

대신 Lambda 함수를 AWS 리소스로 취급합니다. 그 이상도 그 이하도 아닙니다. 따라서 사용자가 원하는 대로 Lambda, API Gateway 및 기타 리소스를 최대한 제어하고 구성할 수 있습니다. 그러나 이렇게 하면 해당 리소스의 모든 근본적인 복잡성에 노출됩니다. 다른 도구가 관리하기 위해 열심히 노력하는 복잡성.

예를 들어 API Gateway 리소스가 구성되는 방식을 이해해야 하며, 이는 Terraform for Lambda를 사용할 때 가장 힘든 측면 중 하나입니다. Serverless Framework 또는 SAM에서 사람이 읽을 수 있는 URL 한 줄은 Terraform 코드 50줄로 쉽게 변환할 수 있습니다(그림 C.1).

Terraform은 인프라 구조를 설명하고 생성하는 방법을 제공하도록 설계되었기 때문에 서비스 애플리케이션을 위한 부가 가치 서비스를 제공하지 않습니다. 없다

서비스 프레임워크

조회: 차리기:

functions/lookup.handler 설명: 조회/국가/우편번호 endpoint
이벤트를 처리합니다.

- http: 경로:

lookup/{country}/{zipcode} 메소드: get

테라폼

```

1  리소스 "aws_api_gateway_rest_api" "rest_api" { name = "${var.stage}-$"
2    ${var.feature_name}" description = "우편번호 조회를 위한 REST API"
3
4  }
5
6  리소스 "aws_api_gateway_resource" "조회" {
7    rest_api_id = "${aws_api_gateway_rest_api.rest_api.id}" parent_id = "$
8    ${aws_api_gateway_rest_api.rest_api.root_resource_id}" path_part = "조회"
9
10 }
11
12 리소스 "aws_api_gateway_resource" "국가" {
13   rest_api_id = "${aws_api_gateway_rest_api.rest_api.id}" parent_id = "$
14   ${aws_api_gateway_resource.lookup.id}" path_part = "{국가}"
15
16 }
17
18 리소스 "aws_api_gateway_resource" "우편번호" {
19   rest_api_id = "${aws_api_gateway_rest_api.rest_api.id}" parent_id = "$
20   ${aws_api_gateway_resource.country.id}" path_part = "{zipcode}"
21
22 }
23
24 리소스 "aws_api_gateway_method" "get_zipcode" {
25   rest_api_id = "${aws_api_gateway_rest_api.rest_api.id}" resource_id = "$
26   ${aws_api_gateway_resource.zipcode.id}" http_method = "GET" 인증 = "NONE"
27
28 }
29
30
31 리소스 "aws_api_gateway_integration" "zipcode_lookup" { rest_api_id = "$
32   ${aws_api_gateway_rest_api.rest_api.id}" resource_id = "${aws_api_gateway_resource.zipcode.id}"
33
34   유형 = "AWS_PROXY"
35
36   http_method = "${aws_api_gateway_method.get_zipcode.http_method}"
37
38   통합_http_method = "POST" uri = "$
39   ${aws_lambda_function.lookup.invoke_arn}"
40
41
42 리소스 "aws_api_gateway_deployment" "zipcode_api" {depends_on =
43   ["aws_api_gateway_integration.zipcode_lookup"
44
45   ]
46
47   rest_api_id = "${aws_api_gateway_rest_api.rest_api.id}" stage_name = "${var.stage}"
48
49 }

```

그림 C.1 Serverless Framework와 Terraform으로 API 게이트웨이 기능 구성하기

배포 아티팩트 패키징을 위한 기본 제공 지원이 없으며 기능을 로컬로 실행하기 위한 기본 제공 지원도 없습니다.

이 목록의 다른 모든 도구는 CloudFormation을 기반으로 구축되지만 Terra 양식은 자체 작업을 수행하고 AWS API에 의존하여 리소스를 생성합니다. 이는 Terra 양식이 앞서 언급한 500과 같은 CloudFormation 제한 사항에 구속되지 않음을 의미합니다.

스택당 리소스는 없지만 CloudFormation이 제공하는 기능도 부족합니다.

예를 들어 Terraform은 배포가 중간에 실패할 때 변경 사항을 자동으로 롤백하지 않습니다. 많은 HashiCorp 팬은 이것이 기능이 아니라 기능이라고 말할 것입니다.

버그가 있지만 그들이 당신을 속이게 두지 마십시오. 배포가 실패할 때 애플리케이션이 중단된 상태로 중단되는 것을 원하지 않습니다.

서비스에서 Terraform을 사용할 때 고려해야 할 다른 문제도 있습니다.

응용 프로그램 예를 들어 Terraform은 AWS API를 사용하여 리소스를 생성하기 때문에 CloudFormation에서 발생하지 않는 제한 제한에 부딪히는 경우가 많습니다. 일반적인 예는 동시 업데이트 수로 인한 ResourceConflictException입니다.

람다 함수. 이는 Lambda에 특정 변경 사항을 적용할 때 발생할 수 있습니다.

여러 API 호출이 필요한 기능입니다. 이것은 오랫동안

문제(<http://mng.bz/RqJK>에서 2018년 이 문제 참조), 그리고 유일하게 실행 가능한 해결 방법은 extends_on 절을 사용하여 매일 체인 변경하는 것입니다.

Terraform은 리소스 상태를 추적하고 다음과 같은 데이터 저장소에 유지할 수 있습니다.

예스3. 그러나 이러한 상태 파일은 암호화하지 않으므로 자격 증명 및 API 키와 같은 민감한 정보는 일반 텍스트로 저장됩니다. 확인하는 것은 귀하에게 달려 있습니다.

S3 버킷이 SSE(서버 측 암호화)를 활성화합니다. 더욱 안전하게 사용하려면

고객 관리 키를 사용하여 사용자만 데이터를 해독할 수 있습니다.

전반적으로 심각한 생산성 부족만으로도 Terraform은 다음과 같은 경우에 나쁜 선택이 됩니다.

서비스 애플리케이션을 구축하는 것입니다. 우리는 그것에 대해 강력히 주장합니다. 그러나 DevOps 문화와 많은 인프라 팀에서 강력한 영향력을 행사하고 있습니다.

조직 내에서 Terraform 사용을 의무화합니다. 서비스에서 Terraform이 아닌 다른 것을 사용할 수 있도록 관리자를 설득하는 데 어려움을 겪고 있다면

프로젝트를 수행한 후 다음을 수행하는 것이 좋습니다.

다음과 같은 간단한 작업을 위해 작성해야 하는 코드 행의 차이를 보여줍니다.

단일 API 엔드포인트. 이것을 개발 시간과 비용으로 변환하십시오. 예를 들어, "Terraform을 사용하는 데 일주일이 소요되는 반면 Serverless Framework 또는 SAM을 사용하는 경우 몇 시간이 걸릴 것입니다"는 설득력 있는 주장입니다.

인프라 팀에 설명합니다(개발자의 DevOps 팀 레이블이 잘못되었을 수 있습니다).

귀하의 조직) Terraform과 Serverless 사이에 통합 경로가 있음

프레임워크 또는 SAM. 그들은 여전히 Terraform을 사용하여 VPC와 같은 공유 인프라 리소스를 프로비저닝할 수 있습니다. 이들의 ARN 또는 이름을 공유하기만 하면 됩니다.

리소스를 SSM 매개변수로 사용합니다. Serverless Framework와 SAM은 모두 이러한 매개변수를 참조할 수 있습니다. 이렇게 하면 인프라 팀과 기능 팀 모두

작업에 적합한 도구를 사용할 수 있고 모두가 행복합니다.

C.4 클라우드 개발 키트 AWS 클라우드 개

발 키트(CDK)는 <https://aws.amazon.com/cdk>에서 사용할 수 있습니다.

는 비교적 새로운 아동이지만 커뮤니티에서 많은 관심을 받았습니다. CDK는 사용하지 않는다는 점에서 앞서 언급한 프레임워크와 다릅니다.

마크업 언어. 대신 CDK를 사용하여 프로비저닝하려는 리소스를 설명할 수 있습니다.

TypeScript 또는 Python과 같은 범용 프로그래밍 언어를 사용합니다.

범용 프로그래밍 언어를 사용하는 것의 매력을 쉽게 알 수 있습니다.

IAC 도구. 개발자는 선호하는 프로그래밍 언어를 사용하여 응용 프로그램을 작성하고 배포 방법을 작성할 수 있습니다. YAML 또는 HCL(Terraform에서 제공하는 JSON 기반 구성 언어)과 같은 다른 언어를 배울 필요가 없습니다.

사용). 이것이 CDK가 더 나은 IaC 도구라는 것을 의미하지는 않습니다.

개발자가 원하는 것. 결국 우리가 아무리 케이스를 먹고 좋아해도

사탕, 이 달콤한 즐거움이 우리 건강에 나쁘다는 사실은 변하지 않습니다.

YAML 또는 HCL이 코드가 아니라고 선언하는 사람은 그렇지 않다는 것을 기억하십시오.

오래 전에 Java와 .Net 개발자는 JavaScript와 Python에 대해 같은 말을 했습니다.

이런 종류의 문지기와 자신의 자위를 높이기 위해 남을 낚추는 일이 일어난다.

우리 지역 사회에는 장소가 많고 자리가 없습니다. 구성 파일은 코드입니다. Cloud Formation 템플릿은 CloudFormation에 사용할 리소스를 알려주는 일련의 지침입니다.

규정하고 그것이 코드의 사전 정의입니다. 이제 우리는 공통점을 얻었습니다.

오해를 없애고 CDK가 실제로 빛나는 곳과 직면한 문제에 대해 이야기해 보겠습니다.

C.4.1 CDK가 빛나는 곳

범용 프로그래밍 언어는 YAML과 같은 구성 파일에 비해 훨씬 더 많은 표현력을 제공합니다. 이것은 CDK를 환상적인 선택으로 만듭니다.

일부 복잡한 AWS 환경을 템플릿화합니다. CloudFormation은

고유한 기능과 조건이 있는 템플릿 옵션의 범위는 다음과 같습니다.

기본 분기 논리 또는

사전에 대한 입력 값 매핑. CDK는 이 아이들의 놀이를 만들고 쉽게 할 수 있습니다

TypeScript 또는 Python에서 몇 줄의 코드로 표현하십시오.

TypeScript와 같은 범용 프로그래밍 언어를 사용할 수 있어야 합니다.

Python은 또한 해당 언어의 패키지 관리자에 액세스할 수 있음을 의미합니다. 이것 일반적인 아키텍처 패턴을 취하고 재사용 가능한 구성을 만들 수 있음을 의미합니다.

패키지로 공유합니다. CDK 패턴 (<https://cdkpatterns.com>) 프로젝트는

이것의 좋은 예. 모두가 같은 레시피를 가지고 구현하는 대신

이러한 일반적인 패턴을 처음부터 시작하여 관련 패키지를 다운로드할 수 있습니다.

NPM을 만들고 간단히 사용자 정의하십시오. 이는 대규모 조직 내에서 모범 사례를 영속화하고 전파하는 좋은 방법입니다. 이를 통해 팀은 모범 사례와 조직 규범이 적용된 구성을 쉽게 발견하고 공유할 수 있습니다.

C.4.2 CDK 문제

React 및 Vue.js와 같은 단일 페이지 애플리케이션(SPA) 프레임워크는 HTML, CSS 및 애플리케이션 코드를 응집력 있고 생산적인 JavaScript 프레임워크로 통합하려는 성공적인 시도를 했습니다. CDK는 인프라 코드에 대해 유사한 작업을 수행하고 있습니다.

그러나 JavaScript가 프론트엔드 세계에서 어디에나 있는 반면 백엔드 애플리케이션을 위한 프로그래밍 언어에 대한 선택과 선호도는 사용 사례를 중심으로 세분화되고 맥락화됩니다. 마이크로서비스의 이점 중 하나는 팀이

작업에 가장 적합한 언어를 선택합니다. 예를 들어 Node.js는 REST API를 빌드하는 데 적합할 수 있지만 Python은 대부분의 라이브러리가 Python으로 작성되기 때문에 기계 학습(ML) 워크로드에 더 적합합니다. 조직의 다른 팀이 다른 언어를 선호한다는 사실은 CDK에 문제가 될 수 있습니다.

모든 사람이 하나의 프로그래밍 언어를 사용하는 데 동의하면 CDK를 사용하면 재사용 가능한 구성을 쉽게 공유할 수 있습니다. 그러나 팀이 다른 언어를 사용하려면 이러한 구성의 다른 버전을 유지해야 합니다. <https://cdkpatterns.com>의 패턴에서 이 문제 매니페스트를 볼 수도 있습니다. 여기서 일부 패턴은 TypeScript, Python, Java 및 C#을 지원하지만 대부분은 네 가지 언어를 모두 지원하지 않습니다.

CDK에 대한 다른 우려 사항은 모든 사람이 동일한 구성으로 리소스를 프로비저닝하려는 경우 동일한 YAML을 작성해야 하지만 범용 프로그래밍 언어의 경우에는 그렇지 않다는 것입니다. 개인 취향과 관용구가 작용할 수 있으며 갑자기 인프라 코드를 이해하는 데 더 많은 인지 에너지가 필요합니다. 더 이상 구성이 아닙니다. 이제 인프라 코드에 비즈니스 로직이 포함됩니다.

이는 조직의 AWS 환경을 감독하고 기능 팀에 지침과 감독을 제공해야 하는 인프라 팀에게 특히 문제가 됩니다. 갑자기 그들은 익숙하지 않을 수 있는 여러 언어로 작성된 인프라 코드로 작업해야 합니다. 그리고 이 인프라 코드는 충분한 비즈니스 로직을 포함할 수 있기 때문에 인프라 팀이 작업을 수행하는 것을 두 배로 어렵게 만듭니다. 이것이 우리가 여전히 YAML의 선언적 접근 방식을 선호하고 인프라 코드에 복잡한 로직을 추가하는 것이 어렵다는 사실이 사실 축복이라고 생각하는 이유입니다.

C.5 Amplify AWS

Amplify (<https://aws.amazon.com/amplify>) 프론트엔드 웹 및 모바일 애플리케이션을 빠르게 구축하기 위해 함께 사용할 수 있는 도구 및 서비스 세트입니다. 다음으로 구성됩니다. Amplify CLI - AWS 리소스를 구성 할 수 있는 CLI 도구입니다.

라이브러리 증폭 - Cognito 및 AppSync와 같은 AWS 리소스를 사용하는 데 도움이 되는 오픈 소스 라이브러리 세트입니다.

UI 구성 요소 증폭 - AWS 리소스와 함께 작동하여 인증, 저장 및 상호 작용을 제공하는 드롭인 UI 구성 요소 모음입니다.

Amplify 콘솔 - 단일 페이지 애플리케이션을 구축 및 호스팅하는 AWS 서비스입니다(AWS 버전의 Netlify: <https://www.netlify.com>).

Amplify Admin UI - AWS 리소스를 프로비저닝 및 구성하고 애플리케이션의 데이터를 관리할 수 있는 시각적 UI입니다.

이러한 Amplify 도구는 각각 독립적으로 사용할 수 있습니다. 예를 들어 많은 팀은 Amplify CLI 또는 Admin UI를 사용하지 않고 Amplify 라이브러리와 UI 구성 요소를 사용하여 AWS 리소스를 관리합니다. 이 비교에서는 Amplify CLI만 고려합니다.

지금까지 논의한 다른 프레임워크는 리소스 중심

서비스 애플리케이션의 관점에서 Amplify CLI는 유ти리티 중심 접근 방식을 취합니다.

Cognito 사용자 풀을 리소스로 구성하는 대신 amplify add auth 명령을 실행합니다. 그러면 Amplify CLI가 수행하려는 작업에 대한 몇 가지 질문을 표시합니다. 이렇게 하면 CLI에서 질문에 답하는 방법에 따라 CloudFormation 템플릿을 부트스트랩하고 Cognito 사용자 풀과 Cognito 자격 증명 풀도 구성할 수 있습니다.

마찬가지로 단일 명령을 사용하여 새로운 AppSync API를 부트스트랩할 수 있습니다. amplify add api. 그런 다음 API 모델 정의에 집중할 수 있으며 Amplify CLI는 관련 AppSync 해석기 및 DynamoDB 테이블을 포함하여 많은 기본 AWS 리소스를 생성할 수 있습니다.

보시다시피 Amplify CLI는 많은 결정을 내리고 신속하게 정리할 수 있습니다. 따라서 개발자의 약간 다른 인구 통계를 대상으로 합니다. 이 목록의 다른 배포 프레임워크는 일반적으로 매일 AWS를 사용하는 백엔드 팀에서 사용합니다. 반면 Amplify는 AWS에 익숙하지 않고 작동하는 것을 원하는 프런트엔드 중심 팀을 대상으로 합니다.

이는 강력한 도구이며 이러한 프런트엔드 중심 팀이 사용 및 구성해야 하는 각 AWS 서비스에 대해 많은 시간을 배우지 않고도 신속하게 무언가를 구축할 수 있도록 많은 권한을 제공합니다. 그러나 팀이 Amplify CLI가 내리는 결정을 인지하지 못하고 이해하지 못하며 문제가 발생했을 때 디버깅할 수 없다는 함정도 있습니다. 예를 들어 Amplify CLI는 기본적으로 GraphQL 스키마의 목록 작업에 대해 DynamoDB 스캔을 사용합니다.

이 방법은 효과가 있지만 최적의 솔루션이 아니며 DynamoDB 스캔이 비싸고 드물게 사용해야 하기 때문에 시스템이 확장됨에 따라 문제가 될 수 있습니다.

Amplify CLI는 많은 것을 자동화하며 이것이 바로 이것이 생산적인 도구가 되는 이유입니다. 그러나 이러한 AWS 리소스를 구성하는 방법을 사용자 지정하는 기능도 제한합니다.

Amplify CLI로 달성할 수 있는 한계에 도달하면 현재로서는 이를 벗어날 탈출구가 없습니다. 많은 팀이 이 지점에 도달했을 때 전체 애플리케이션을 처음부터 다시 작성해야 했습니다. 많은 팀이 Amplify CLI가 자신을 위해 수행한 작업을 이해하지 못하고 설정을 복제하는 데 어려움을 겪을 수 있기 때문에 어려움을 겪을 수 있습니다. Amplify로 작업하세요.

Amplify CLI와 같은 도구에 대한 이상적인 사용자는 AWS를 잘 이해하고 기본 도구로 작업하고 구성한 경험이 있는 개발자입니다.

자원. 이해하지 못하는 것을 자동화해서는 안 됩니다.

도구에 지나치게 의존하고 꼬리가 개를 흔드는 위험한 위치.

Amplify 팀은 우리가 가져온 문제를 해결하기 위해 열심히 노력하고 있습니다.

팀이 다른 곳으로 이동할 수 있도록 탈출구를 만드는 방법을 찾고 있습니다.

그들이 필요할 때 그것으로부터. 그리고 우리는 그것이 어디로 가는지 보게되어 기쁩니다. 그러나 당분간은 Amplify CLI는 개념 증명을 구축하거나 매우

간단한 응용 프로그램. 시간이 지남에 따라 유지 관리하고 반복해야 하는 프로덕션 애플리케이션의 경우 차단기가 발생하고 쉽게 전환할 수 없는 위험이 있습니다.

그것과는 거리가 너무 큽니다. 그러나 다른 Amplify 사용을 중단해서는 안 됩니다.

프런트엔드 프로젝트의 Amplify 라이브러리 또는 Amplify 콘솔과 같은 구성 요소를 사용하여 SPA를 구축하고 호스팅할 수 있습니다.

이 부록에서는 사람들이 서비스 애플리케이션을 프로비저닝하고 배포하는 가장 인기 있는 5가지 방법을 살펴보았습니다. CloudFormation을 통해 추상화 계층을 제공하는 서비스 프레임 워크와 SAM을 살펴보았습니다.

서비스 애플리케이션을 더 쉽게 구축할 수 있습니다. 둘 다 부가 가치 CLI 명령 세트를 지원합니다.

기능을 로컬로 호출하거나 API Gateway의 로컬 인스턴스를 실행할 수 있는 것과 같이,

개발 워크플로에 도움이 됩니다.

이 둘의 주요 차이점은 Serverless Framework에 flexi가 있다는 것입니다.

ble 플러그인 시스템과 프레임워크의 기능을 확장할 수 있는 기존 플러그인의 풍부한 생태계. 반면 SAM을 사용하면 지원하는 항목에 따라 제한을 받습니다.

또한 사용 가능한 구성의 범위를 넘어 프레임워크의 기본 동작을 변경하는 쉬운 방법도 없습니다.

또한 많은 인프라 팀에서 널리 사용되는 IaC 도구인 Terraform도 살펴보았습니다. 서비스 애플리케이션에서 Terraform을 사용할 때의 문제점과 이를 강력히 권장하는 이유를 설명했습니다. 그럴 때 비생산적인 도구입니다.

서비스 애플리케이션을 구축하는 데 도움이 됩니다.

CDK와 Amplify CLI는 모두 이 분야에서 상대적으로 새로운 기업이며 둘 다 많은 주목을 받으며 많은 사랑을 받고 있습니다. 반면 서비스

Framework, SAM 및 Terraform은 모두 마크업 언어를 사용하여 리소스를 설명합니다.

서비스 애플리케이션의 경우 CDK와 Amplify CLI는 서로 다른 접근 방식을 취합니다.

CDK를 사용하면 범용 프로그래밍 언어를 사용하여 프로비저닝하려는 리소스. 범용 프로그래밍 언어가 제공하는 완전한 유연성과 패키지를 제공하는 양날의 검입니다.

해당 언어와 함께 제공되는 관리 시스템입니다. 이를 통해 개발자는 애플리케이션 코드와 인프라 코드 모두에 대해 선호하는 프로그래밍 언어를 사용할 수 있으며 재사용 가능한 패턴을 패키지로 쉽게 공유할 수 있습니다. 그러나 다음과 같이 할 수도 있습니다.

팀이 서로 다른 프로그래밍 언어를 사용하는 조직에서 문제

그들의 애플리케이션 코드. 이러한 구성의 작성자는 여러 언어를 지원해야 하기 때문에 CDK 구성의 공유하는 기능이 제한됩니다. 개발자가 추가할 수 있도록 허용

인프라 코드에 대한 비즈니스 로직은 광범위한 사용자 정의를 위한 문을 엽니다.

복잡한 AWS 환경에 적합하지만 인프라 팀이 인프라 코드를 이해하고 관리하기 어렵게 만듭니다.

마지막으로 Amplify CLI를 사용하면 AWS 리소스를 구성하는 것이 아니라 애플리케이션에 필요한 기능을 말하는 것입니다. Amplify CLI는 마법 같은 일이 일어나도록 하고 입력을 기반으로 합리적인 기본값으로 필요한 AWS 리소스를 구성합니다. 이것은 매우 생산적인 도구이며 즉시 완벽하게 작동하는 응용 프로그램을 빌드하는 데 도움이 될 수 있습니다. 하지만 그것은 블랙박스이기도 하고 할 수 있는 일의 한계에 도달했을 때 탈출할 수 있는 탈출구가 없습니다. 이는 Amplify CLI에 문제가 발생한 경우 애플리케이션을 처음부터 다시 빌드해야 하는 실제 가능성에 직면하게 되는 위태로운 위치에 놓이게 합니다.

이러한 도구는 각각 장단점이 있지만 완벽한 것은 없습니다. 어떤 도구를 사용하기로 결정했는지에 관계없이 좋은 원칙은 자동화를 시도하기 전에 작업해야 하는 AWS 서비스가 작동하는 방식과 배포 메커니즘이 작동하는 방식을 이해하는 것입니다. 이해하지 못하는 것을 맹목적으로 자동화하는 것은 위험합니다. 그러나 기본 기계를 이해하고 나면 추상화 수준을 높이고 생산성을 높일 수 있는 도구를 찾아야 합니다.

AWS/클라우드 컴퓨팅

AWS의 서비스 아키텍처

두번째 버전

스바르스키 • 쿠이 • 나이르

서버 하드웨어에 줄 프로그램에 대처하는 대처하는 그림을 알 수 있습니다.
스 아키텍처 o! 다음과 같은 핵심 작업

사전 구축된 클라우드 서비스에 대한 데이터 스토리지 및 하드웨어 관리. 또한 자체 사용자 지정 AWS Lambda 함수를 다른 서비스 서비스와 결합하여 온디맨드 방식으로 자동 시작 및 확장하고 호스팅 비용을 절감하고 유지 관리를 간소화하는 기능을 생성할 수 있습니다.

Serverless Architectures with AWS, Second Edition 에서는 AWS 플랫폼에서 Lambda 및 기타 서비스를 사용하여 서비스 시스템을 설계하는 방법을 배웁니다. 이벤트 기반 컴퓨팅을 탐색하고 다른 사람들이 서비스 디자인을 성공적으로 사용한 방법을 알아봅니다. "는 여러 대규모 서비스 시스템의 실제 사용 사례와 실용적인 통찰력을 제공하는 새 버전입니다.

혁신적인 서비스 디자인 패턴 및 아키텍처에 대한 챕터는 완전한 클라우드 전문가가 되는 데 도움이 될 것입니다.

안에 뭐가 들어있어

- 서비스 컴퓨팅의 첫 단계
- " 서비스 설계의 원칙
- 중요한 패턴 및 아키텍처
- 실제 아키텍처 및 해당 거리#

서버 측 및 전체 스택 소프트웨어 개발자용.

Peter Sbarski 는 A Cloud Guru의 교육 및 연구 부사장입니다. Yan Cui 는 독립적인 AWS 컨설턴트이자 교육자입니다. Ajay Nair 는 AWS Lambda 팀의 창립 멤버 중 한 명입니다.

이 인쇄판을 등록하면 모든 전자책 형식에 무료로 액세스할 수 있습니다.

방문 <https://www.manning.com/freebook>

" AWS에 대한 포괄적이고 실용적인 검토 끝없는 풍경. "

—유진 세르디오크
Primex 제품군

" AWS 서비스 아키텍처를 도입하는 데 사용할 수 있는 필수 조언으로 가득 차 있습니다." 다음 단계로 넘어갑니다.

—살 디스테파노
여행자 보험

"일반적인 서비스 아키텍처에 대한 개요와 설명을 제공하는 훌륭한 책입니다."

모든 클라우드 개발자가 반드시 읽어야 할 책입니다."
—미코와이 그라프
Cloudsail 디지털 솔루션

" AWS에서 제공하는 다양한 서비스를 탐색할 수 있는 명확한 경로입니다."

—지암피에로 그라나텔
많은 디자인



ISBN: 978-1-61729-542-3

