서버리스

- 서버리스는 새로운 서비스
- 서버리스 서비스를 사용하는 개발자는 서버를 관리할 필요가 없음
- 서버가 없다는 것은 아니고 관리할 필요가 없다는 뜻
- 그냥 코드를 배치하면 됩니다 함수를 배치하는 것
- 서버리스는 FaaS, 즉 Function as a Service를 뜻했지만 지금의 서버리스는 더 많은 것을 의미
- AWS Lambda에서 처음 사용 됐는데 현재는 원격 관리되는 것을 모두 포함
  - 데이터베이스, 메시징, 스토리지 등
  - 서버를 프로비저닝 하지 않는 모든 것들을 포함
- 서버리스란 서버가 없는 게 아니라 서버가 보이지 않거나 서버를 프로비저닝 하지 않는 것
- 사용자가 S3 버킷에서 정적 콘텐츠를 얻는데 웹 사이트 혹은 CloudFront 와 S3로 전달
- Cognito에 로그인하는데 사용자 신원 정보를 보관하는 곳
- API Gateway를 통해 사용자는 REST API를 호출하고, API Gateway는 람다 함수를 호출하고, 람다 함수는 DynamoDB에서 데이터를 저장하고 회수
- AWS에는 Lambda, DynamoDB Cognito, API Gateway, Amazon S3, SNS, SQS 등
- 사례
  - SQS와 SNS에서도 서버를 관리하지 않고, 자동으로 스케일링
  - Kinesis Data Firehose도 처리량에 맞춰 스케일링하고, 사용한 만큼 지불하며 서버를 프로비저닝 하지 않음
  - Aurora 서버리스는 관리 서버 없이도 데이터베이스가 온디맨드로 스케일링
  - 단계 함수와 Fargate의 경우 Fargate는 ECS의 서버리스 기능이며 여기서는 도커 컨테이너를 실행할 인프라를 프로비저닝하지 않음

Lambda 개요

- Amazon EC2로 시작할 때 클라우드의 가상 서버라서 프로비저닝이 필요합니다 따라서 프로비저닝 필요 -> 메모리와 CPU 크기가 제한됨
- 지속적으로 실행되어야 합니다 그래서 최적화를 하려면 효율적으로 시작하고 중단해야 함
- 그렇지 않으면 인스턴스에 어떤 일이 생기든 관계없이 EC2는 지속적으로 실행됨
- 오토 스케일링 그룹으로 스케일링할 수 있는데 자동으로 서버를 추가하고 제거하는 작업이 필요
- AWS Lambda를 사용해도 됨
- 관리할 서버 없이 코드를 프로비저닝하면 함수가 실행됨
- 제한 시간이 있어서 실행 시간이 짧지만 충분 -> 15분
- 온디맨드로 실행됨
- Lambda를 사용하지 않으면 람다 함수가 실행되지 않고, 비용 역시 함수가 실행되는 동안만 청구되며 호출을 받으면 온디맨드로 실행 -> Amazon EC2과는 차이
- 스케일링이 자동화

- 많은 람다 함수를 동시에 필요로 하는 경우 AWS가 자동으로 프로비저닝해서 람다 함수를 늘려줌
- 가격 책정이 아주 쉬워짐
- Lambda가 수신하는 요청의 횟수에 따라 과금되는데 호출 횟수와 컴퓨팅 시간, 즉 Lambda가 실행된 시간만큼 청구됨
- 프리 티어에서도 Lambda를 넉넉하게 제공, Lambda 요청 1백만 건과 40만 GB-초의 컴퓨팅 시간이 포함
- AWS 서비스와 통합 가능
- Lambda에 여러 가지 프로그래밍 언어를 사용할 수 있어서 상당히 자유로운 편
- CloudWatch와의 모니터링 통합도 쉬움
- 함수당 더 많은 리소스를 프로비저닝 하려면 함수당 최대 10GB의 램을 프로비저닝 할 수 있음
- 함수의 RAM을 증가시키면 CPU 및 네트워크의 품질과 성능도 함께 향상됨
- Lambda를 지원하는 언어
  - 먼저 JavaScript의 Node.js, Python, Java 즉 Java 8 호환이나 Java 11, C# .NET Core Golang, C# /Powershell, Ruby, 또 웬만한 언어는 모두 Lambda에서 사용 가능
  - Rust 언어 함수를 Lambda에서 사용하는 것도 오픈 소스 프로젝트가 있어서 가능
- Lambda 컨테이너 이미지를 지원
  - 컨테이너 이미지 자체가 Lambda의 런타임 API를 구현해야 함
  - 아무 컨테이너 이미지나 Lambda에서 실행되지는 않고, 컨테이너 이미지를 만들 때 전제 조건이 필요
  - ECS와 Fargate는 계속 임의의 도커 이미지를 실행할 때 더 많이 사용됨
  - 시험에서 Lambda에 컨테이너를 실행해야 할 경우 해당 컨테이너가 Lambda 런타임 API를 구현하지 않으면 ECS나 Fargate에서 컨테이너를 실행
- Lambda가 다양한 AWS 서비스와 통합 예시
  - API Gateway는 REST API를 생성
  - 람다 함수를 호출
  - Kinesis는 Lambda를 이용해 바로 데이터를 변환
  - DynamoDB는 트리거를 생성할 때 사용되는데 데이터베이스에 어떤 일이 생기면 람다 함수가 작동
  - Amazon S3은 언제든 람다 함수를 작동시킬 수 있음
    - S3에 파일이 생성되거나 할 때
  - CloudFront용 람다는 Lambda@Edge
  - CloudWatch 이벤트와 EventBridge에서는 AWS의 인프라에 어떤 일이 생기고, 상황에 대응하고자 할 때 사용
    - 파이프라인이 끊기거나, 상태가 바뀌는 경우 등
  - **상황에 따라 자동화를 실행하려고 람다 함수를 사용**
  - CloudWatch 로그는 어디든 해당 로그를 스트리밍
  - SNS로 알림과 SNS 토픽에 대처할 수 있으며 SQS로는 SQS 대기열 메시지를 처리할 수 있음
  - Cognito는 사용자가 데이터베이스에 로그인할 때마다 응답
- 서버리스 섬네일 생성 예시
  - S3 버킷이 있음
  - 바로 섬네일을 생성하고 싶음
  - Amazon S3에 새 이미지가 업로드되는 이벤트가 생기고, S3 이벤트 알림을 통해 람다 함수가 작동
  - 람다 함수에는 섬네일을 생성하는 코드가 있음

    - 해당 섬네일은 다른 S3 버킷이나 같은 S3 버킷으로 푸시 및 업로드
    - 람다 함수는 몇몇 데이터를 DynamoDB에 삽입할 수 있음
    - Lambda 덕분에 기능이 자동화되고, 또 S3에 새 이미지와 앱이 생성되는 이벤트에 대한 반응형 아키텍처를 생성할 수 있음
- 서버리스 CRON 작업
    - CRON이란 EC2 인스턴스에서 작업을 생성하는 방법인데 5분마다, 월요일 10시마다 등등을 지정
    - CRON은 가상 서버에 실행해야 함 -> EC2 인스턴스 등에
    - 인스턴스가 실행되지 않거나 CRON이 아무 일도 안 하면 인스턴스 시간이 낭비됨
    - CloudWatch 이벤트 규칙 또는 EventBridge 규칙을 만들고, 1시간마다 작동되게 설정해서 1시간마다 람다 함수와 통합되면 태스크를 수행할 수 있음
    - CloudWatch 이벤트는 서버리스이고, 람다 함수 역시 서버리스
- Lambda 가격 책정
    - 호출당 청구
        - 처음 1백만 건의 요청은 무료이고, 이후 1백만 건 요청마다 20센트가 과금
        - 1ms 단위로 요금이 부과
        - 한 달간 첫 40만 GB-초 동안의 컴퓨팅 시간은 무료로 사용
        - GB-초라는 것은 함수가 1GB RAM을 가질 때 실행 시간이 40만 초이며 실행 시간이 8배 늘어나려면 함수의 RAM은 8배 작은 128MB RAM이 되어야 함
        - 60만 GB-초당 1달러가 과금

Lamda 한도

- 한도는 리전당 존재
- 한도에는 실행 한도와 배포 한도가 있음
- 실행 한도
    - 실행 시 메모리 할당량은 128MB에서 10GB이고, 메모리는 1MB씩 증가
    - 메모리가 증가하면 더 많은 vCPU가 필요
    - 최대 실행 시간은 900초 즉 15분
    - 이를 초과하면 Lambda 실행에 바람직 하지 않음
    - 환경변수는 4KB까지 가질 수 있는데 제한적인 공간이나 Lambda 함수를 생성하는 동안 큰 파일을 가져올 때 사용할 수 있는 임시 공간이 있음
    - /tmp 폴더에 용량이 있으며 크기는 최대 10GB
    - Lambda 함수는 최대 1,000개까지 동시 실행이 가능하며 요청 시 증가할 수 있지만 동시성은 미리 예약해 두는 것이 좋음
- 배포 한도
    - 압축 시 최대 크기는 50MB, 압축하지 않았을 때는 250MB
    - 용량을 넘는 파일의 경우 /tmp 공간을 사용
    - 시작할 때 크기가 큰 파일이 있으면 /tmp 디렉터리를 사용
    - 배포 시에도 환경변수의 한도는 4KB

    - 예를 들어 30GB의 RAM과 30분의 실행 시간이 필요하고, 3GB의 큰 파일을 있을 때는 워크로드 처리에 Lambda 함수가 적합하지 않다는 걸 판단할 수 있음

Lambda@엣지 & CloudFront Functions

- 엣지에서의 사용자 지정
- 보통은 함수와 애플리케이션을 특정 리전에서 배포하지만 CloudFront를 사용할 때는 엣지 로케이션을 통해 콘텐츠를 배포
- 모던 애플리케이션에서는 애플리케이션에 도달하기 전에 엣지에서 로직을 실행하도록 요구하기도 함
- 이를 엣지 함수라고 하며 CloudFront 배포에 연결하는 코드
- 함수는 사용자 근처에서 실행하여 지연 시간을 최소화하는 것이 목적
- CloudFront에는 두 종류의 함수가 있는데 CloudFront 함수와 Lambda@Edge
- 엣지 함수
    - 서버를 관리할 필요가 없음 -> 전역으로 배포
    - 사용 사례로는 CloudFront의 CDN 콘텐츠를 사용자 지정하는 경우
    - 사용한 만큼만 비용을 지불하며 완전 서버리스
    - 첫 번째는 웹사이트 보안과 프라이버시일 경우
    - 엣지에서의 동적 웹 애플리케이션에도 쓰이고, 검색 엔진 최적화(SEO)에도 활용 가능
    - 오리진 및 데이터 센터 간 지능형 경로에도 활용
    - 엣지에서의 봇 완화 엣지에서의 실시간 이미지 변환 A/B 테스트 사용자 인증 및 권한 부여 사용자 우선순위 지정 사용자 추적 및 분석 등 다양한 사용자 지정에 CloudFront 함수와 Lambda@Edge가 활용
- CloudFront 함수의 활용과 원리
    - 일반적인 CloudFront의 요청 처리 과정은 화면의 도식과 같음
    - CloudFront에 클라이언트가 요청을 보내는 것을 뷰어 요청
    - CloudFront가 오리진 요청을 오리진 서버에 전송
    - 서버가 CloudFront에 오리진 응답을 보내고, CloudFront가 클라이언트에게 뷰어 응답을 전송
    - CloudFront 함수는 JavaScript로 작성된 경량 함수로 뷰어 요청과 응답을 수정
    - 확장성이 높고 지연 시간에 민감한 CDN 사용자 지정에 사용
    - 시작 시간은 1밀리초 미만이며 초당 백만 개의 요청을 처리
    - 뷰어 요청과 응답을 수정할 때만 사용
    - CloudFront가 뷰어로부터 요청을 받은 다음에 뷰어 요청을 수정할 수 있고, CloudFront가 뷰어에게 응답을 보내기 전에 뷰어 응답을 수정할 수 있음
    - CloudFront의 네이티브 기능으로 모든 코드가 CloudFront에서 직접 관리
    - CloudFront 함수는 고성능, 고확장성이 필요할 때 뷰어 요청과 뷰어 응답에만 사용됨
- Lambda@Edge의 기능
    - 함수는 Node.js나 Python으로 작성하며 초당 수천 개의 요청을 처리할 수 있음
    - 모든 CloudFront 요청 및 응답을 변경할 수 있음
    - 뷰어 요청은 위와 같고, 오리진 요청은 CloudFront가 오리진에 요청을 전송하기 전에 수정 가능
    - 오리진 응답은 CloudFront가 오리진에서 응답을 받은 후에 수정됨
    - 뷰어 응답은 CloudFront가 뷰어에게 응답을 보내기 전에 수정됨

    - 함수는 us-east-1 리전에만 작성할 수 있음
    - CloudFront 배포를 관리하는 리전과 같은 리전
    - 함수를 작성하면 CloudFront가 모든 로케이션에 해당 함수를 복제
    - CloudFront 함수와 Lambda@Edge의 비교표가 있음
    - **차이점은 런타임 지원**
    - CloudFront는 JavaScript만 Lambda@Edge는 Node.js와 Python을 지원
    - CloudFront 함수의 확장성은 매우 높음
    - 수백만 개의 요청을 처리하죠 Lambda@Edge는 수천 개 수준인 반면, 트리거가 발생 위치도 크게 다름
    - Lambda@Edge는 뷰어와 오리진 모두에게 영향을 미치는 반면 CloudFront 함수는 뷰어에만 영향력이 있음
    - CloudFront 함수의 최대 실행 시간은 1밀리초 미만 -> 빠르고 간단한 함수임
    - Lambda@Edge는 실행에 5~10초가 소요
- 사용 사례
    - CloudFront 함수는 캐시 키를 정규화
    - 요청 속성을 변환하여 최적의 캐시 키를 생성
    - 요청이나 응답에 HTTP 헤더를 삽입, 수정, 삭제하도록 헤더를 조작하고, URL을 다시 쓰거나 리디렉션
    - 요청을 허용 또는 거부하기 위해 JWT를 생성하거나 검증하는 요청 인증 및 권한 부여에도 사용
    - 모든 작업이 1밀리초 이내에 이뤄짐
    - 반면에 Lambda@Edge의 실행 시간은 10초가 걸릴 수도 있음
    - CPU와 메모리가 증가하므로 여러 라이브러리를 로드할 수 있고, 타사 라이브러리에 코드를 의존시킬 수 있음
    - 가령 SDK에서 다른 AWS 서비스에 액세스할 수 있도록
    - 네트워크 액세스를 통해 외부 서비스에서 데이터를 처리할 수 있어 대규모 데이터 통합도 수행 가능
    - 파일 시스템이나 HTTP 요청 본문에도 바로 액세스할 수 있음

Lambda in VPC

- 기본적으로 Lambda 함수를 시작하면 여러분의 VPC 외부에서 시작됨
- VPC는 AWS가 제공하는 서비스
- VPC 내에서 리소스에 액세스할 권한이 없음
- RDS 데이터베이스, ElastiCache 캐시 내부 로드 밸런서를 시작하면 Lambda 함수가 해당 서비스에 액세스할 수 없음
- Lambda 배포의 기본 설정
- 인터넷의 퍼블릭 API에 액세스하는 것은 가능
- DynamoDB에 액세스할 수 있는 건 DynamoDB가 AWS 클라우드의 퍼블릭 리소스이기 때문
- 하지만 프라이빗 RDS 데이터베이스에는 연결할 수 없음
- 이를 해결하려면 VPC에서 Lambda 함수를 시작하면 됨
- VPC ID Lambda 함수를 시작하려는 서브넷을 지정하고 Lambda 함수에 보안 그룹을 추가해야 함
- Lambda가 서브넷에 엘라스틱 네트워크 인터페이스를 생성해 VPC에서 실행되는Amazon RDS에 액세스 가능

- Lambda 함수를 프라이빗 서브넷에 시작하는 방법임
- VPC 내 모든 항목에 비공개로 연결할 수 있음
- VPC에서 Lambda를 사용하는 대표적인 사용 사례는 **RDS 프록시**
- 따라서 RDS 데이터베이스가 프라이빗 서브넷에 있어도 Lambda 함수로 직접 해당 DB에 액세스할 수 있음
- 이런 방법으로 RDS 데이터베이스에 직접 액세스하면 큰 문제가 발생
- Lambda 함수의 수가 너무 많이 생성되었다 사라지길 반복하면 개방된 연결이 너무 많아서 RDS 데이터베이스의 로드가 상승해 시간 초과 등의 문제로 이어짐
- 해결하는 방법은 RDS 프록시를 시작하는 것
- 프록시가 연결을 한곳으로 모으고, RDS 데이터베이스 인스턴스 연결의 수를 줄임
- Lambda 함수가 RDS 프록시에 연결되고, RDS 프록시가 RDS DB 인스턴스로 연결되므로 아키텍처상의 문제가 해결됨
- RDS 프록시에는 세 가지 장점
  - 데이터베이스 연결의 풀링과 공유를 통해 확장성을 향상
  - 장애가 발생할 경우 장애 조치 시간을 66%까지 줄여 가용성을 향상시키고 연결을 보존
  - RDS와 Aurora 모두에 적용
  - RDS 프록시 수준에서 IAM 인증을 강제하여 보안을 높일 수 있고, 자격 증명은 Secrets Manager에 저장
  - Lambda 함수가 RDS 프록시에 연결할 수 있으려면 VPC에서 Lambda 함수를 시작해야 함 -> RDS 프록시는 퍼블릭 액세스가 불가능하므로
  - Lambda 함수를 퍼블릭으로 시작하면 RDS 프록시에 네트워크 연결을 할 수가 없음

Amazon DynamoDB 개요

- 완전 관리형 데이터베이스로 데이터가 다중 AZ 간에 복제되므로 가용성이 높음
- DynamoDB는 클라우드 네이티브이며 AWS의 독점 서비스입니다 NoSQL 데이터베이스
- RDS나 Aurora 같은 관계형 데이터베이스는 아니지만 트랜잭션 지원 기능이 있음
- DynamoDB를 이용하면 방대한 워크로드로 확장이 가능 -> 데이터베이스가 내부에서 분산되기 때문
- **초당 수백만 개의 요청을 처리하고, 수조 개의 행, 수백 TB의 스토리지를 갖게 됨**
- 성능은 한 자릿수 밀리초를 자랑하고 일관성 또한 높음
- 보안과 관련된 모든 기능은 IAM과 통합되어 있음
- 보안, 권한 부여, 관리 기능이 포함
- 비용이 적게 들고 오토 스케일링 기능이 탑재
- 유지 관리나 패치 없이도 항상 사용할 수 있음
- 데이터베이스를 프로비저닝할 필요가 없음
- 항상 사용할 수 있으므로 테이블을 생성해 해당 테이블의 용량만 설정하면 됨
- 테이블 클래스는 두 종류
  - 액세스가 빈번한 데이터에는 Standard 클래스
  - 액세스가 빈번하지 않는 데이터는 IA 테이블 클래스에 저장
- DynamoDB는 테이블로 구성되며 데이터베이스를 생성할 필요가 없음
- Aurora나 RDS와 달리 DynamoDB는 데이터베이스가 이미 존재하는 서비스

- DynamoDB에 테이블을 생성하면 각 테이블에 기본 키가 부여되는데 기본 키는 생성 시 결정
- 각 테이블에 데이터를 추가합니다 항목, 즉 행을 무한히 추가할 수 있음
- 각 항목은 속성을 가지며 속성은 열에 표시
- 속성은 나중에 추가할 수도 있고 null이 될 수도 있음
- RDS나 Aurora 데이터베이스에서는 열을 나중에 추가할 수 있으나 과정이 복잡하고, 스키마 전개가 어려울 수 있지만 DynamoDB에서는 사전 요구 사항 없이 나중에 쉽게 속성을 추가할 수 있음
- DynamoDB 항목의 최대 크기는 400KB이므로 큰 객체를 저장할 때는 적합하지 않음
- DynamoDB는 다양한 데이터 유형을 지원
- 문자열, 숫자, 바이너리, 불리언 null과 같은 스칼라 유형 목록, 지도와 같은 문서 유형과 세트 유형을 지원
- 시험에서 스키마를 빠르게 전개해야 할 때 DynamoDB를 선택
- 데이터의 유형과 구성 면에서 스키마를 빠르게 전개해야 할 때 Aurora나 RDS보다는 DynamoDB가 더 나음
- DynamoDB 테이블의 예시
  - 기본 키는 파티션 키와 선택 사항인 정렬 키로 구성
  - 속성 테이블이 있음
  - 데이터베이스 형태
  - 속성은 null로 설정하거나 나중에 추가할 수 있음
  - DynamoDB를 사용하려면 읽기/쓰기 용량 모드도 설정
- 테이블 용량 관리 방식을 제어하는 데는 두 가지 모드
  - 기본 설정은 프로비저닝된 모드로 미리 용량을 프로비저닝
    - 초당 읽기/쓰기 요청 수를 예측해서 미리 지정하면 그것이 테이블의 용량이 됨
    - 미리 용량을 계획하고, 프로비저닝된 RCU와 WCU만큼의 비용을 지불하는 방식
    - RCU는 읽기 용량 단위 WCU는 쓰기 용량 단위를 뜻함
    - 미리 용량을 계획한 경우에도 오토 스케일링 기능이 있으므로 테이블의 로드에 따라 자동으로 RCU와 WCU를 늘리거나 줄일 수 있음
    - 프로비저닝된 모드는 로드를 예측할 수 있고 서서히 전개되며 비용 절감을 원할 때 적합
  - 온디맨드 모드
    - 온디맨드 모드에서는 읽기/쓰기 용량이 워크로드에 따라 자동으로 확장
    - 미리 용량 계획을 하지 않으므로 RCU나 WCU 개념 자체가 없음
    - 온디맨드 모드에서는 정확히 사용한 만큼의 비용을 지불
    - 모든 읽기와 쓰기에 비용을 지불
    - 비싼 솔루션으로 볼 수도 있지만 워크로드를 예측할 수 없거나 급격히 증가하는 경우에 유용
    - 수천 개의 트랜잭션을 수백만 개의 트랜잭션으로 1분 내로 확장해야 하는 애플리케이션에서는 빠르게 확장되지 않는 프로비저닝된 모드는 적합하지 않음
    - 트랜잭션이 없거나 하루에 많아야 4~5회밖에 되지 않는 워크로드라면 트랜잭션 횟수만큼만 비용을 지불하는 온디맨드 모드가 적절

Amazon DynamoDB 심화 기능

- DynamoDB Accelerato(DAX)메모리 캐시로 DynamoDB 테이블에 읽기 작업이 많을 때 DAX 클러스터를 생성하고 데이터를 캐싱하여 읽기 혼잡을 해결

- **DAX는 캐시 데이터에 마이크로초 수준의 지연 시간을 제공**
- DAX 클러스터는 기존 DynamoDB API와 호환되므로 애플리케이션 로직을 변경할 필요가 없음
- DynamoDB 테이블과 애플리케이션이 있을 때 몇몇 캐시 노드가 연결된 DAX 클러스터를 생성하면 백그라운드에서 DAX 클러스터가 Amazon DynamoDB 테이블에 연결됨
- 캐시의 기본 TTL은 5분으로 설정되어 있으나 변경할 수 있음
- ElastiCache가 아니라 DAX를 사용하는 이유는?
- DAX는 DynamoDB 앞에 있고 개별 객체 캐시와 쿼리와 스캔 캐시를 처리하는 데 유용
- 예를 들어 집계 결과를 저장할 때는 Amazon ElastiCache가 좋고, Amazon DynamoDB는 대용량의 연산을 저장할 때 좋음
- 두 서비스는 상호 보완적인 성격
- Amazon DynamoDB에 캐싱 솔루션을 추가할 때는 보통 DynamoDB Accelerator 즉 DAX를 사용
- DynamoDB에서는 스트림 처리도 가능
- 테이블의 모든 수정 사항 스트림을 생성할 수 있음
- 사용자 테이블에 새로운 사용자가 등록됐을 때 환영 이메일을 보내는 등 DynamoDB 테이블의 변경 사항에 실시간으로 반응하는 데 활용할 수 있음
- 실시간으로 사용 분석을 하거나 파생 테이블을 삽입할 수도 있음
- 리전 간 복제를 실행하거나 DynamoDB 테이블 변경 사항에 대해 Lambda 함수를 실행할 수도 있음
- DynamoDB에는 두 가지 스트림 처리
  - DynamoDB 스트림은 보존 기간이 24시간이고 소비자 수가 제한
  - Lambda 트리거와 함께 사용하면 좋음
  - 자체적으로 읽기를 실행하려면 DynamoDB Stream Kinesis 어댑터를 사용
  - Kinesis Data Streams에 변경 사항을 바로 보내는 방법도 있음
  - 이 스트림은 보존 기간이 1년이고, 더 많은 수의 소비자 수를 갖고, 데이터를 처리하는 방법이 훨씬 많음
  - AWS Lambda 함수 Kinesis Data Analytics, Kinesis Data Firehose Glue 스트리밍 ETL 등
- DynamoDB 스트림
  - 애플리케이션이 DynamoDB 테이블에서 작업을 생성, 업데이트, 삭제하면 이는 DynamoDB 스트림이나 Kinesis Data Streams로 전송
  - Kinesis Data Streams을 선택하면 Kinesis Data Firehose를 사용할 수 있음
  - 분석 목적으로 데이터를 Amazon Redshift로 전송하고, 데이터를 아카이빙하려면 Amazon S3로 전송
  - Amazon OpenSearch로 보내면 인덱싱이나 검색을 할 수 있음
  - DynamoDB 스트림을 사용하면 처리 계층을 둘 수 있음
  - EC2 인스턴스에서 애플리케이션을 실행하려면 KCL Adapter나 Lambda 함수를 사용
  - 처리 계층에서 SNS로 알림을 보내거나 DynamoDB 테이블을 필터링하거나 변환할 수 있음
  - Amazon OpenSearch로 처리 계층의 데이터를 전송할 수도 있음
- Kinesis Data Streams에서 EC2 인스턴스로 읽거나 Kinesis Data Streams에서 Kinesis Data Analytics를 사용할 수도 있음
- DynamoDB에는 글로벌 테이블이란 개념
  - 글로벌 테이블은 여러 리전 간에 복제가 가능한 테이블
  - 테이블을 US-EAST-1과 AP-SOUTHEAST-2에 둘 수 있음
  - 두 테이블 간에는 양방향 복제가 가능

    - US-EAST-1이나 AP-SOUTHEAST-2 테이블 둘 중 하나에 쓰기를 하면 된다는 뜻
    - DynamoDB 글로벌 테이블은 복수의 리전에서 짧은 지연 시간으로 액세스할 수 있음
    - 다중 활성 복제가 가능하므로 애플리케이션이 모든 리전에서 테이블을 읽고 쓸 수 있다는 뜻
    - 글로벌 테이블을 활성화하려면 DynamoDB 스트림을 활성화해야 리전 간 테이블을 복제할 수 있는 기본 인프라가 구축
    - DynamoDB의 기능 중에 타임 투 리브(TTL)라는 기능은 만료 타임스탬프가 지나면 자동으로 항목을 삭제하는 기능
    - 가령 SessionData라는 테이블에서 ExpTime(TTL)이라는 만료 기간 속성이 있음,이 안에 타임스탬프가 들어감
    - TTL을 정의한 다음 에포크 타임스탬프에서의 현재 시간이 ExpTime 열을 넘어설 경우 해당 항목을 만료 처리하고, 삭제 처리를 진행하는 개념
    - 데이터 테이블의 항목이 일정 시간 후에 삭제되도록 함
- 사용 사례
    - 최근 항목만 저장하도록 하거나 2년 후 데이터를 삭제해야 한다는 규정을 따라야 할 때 사용
    - 시험에서 자주 등장하는 사용 사례는 **웹 세션 핸들링**
    - 사용자가 웹사이트에 로그인했을 때 해당 세션을 중앙 저장소인 DynamoDB에 두 시간 동안 저장
    - 여기에 세션 데이터를 저장하면 모든 애플리케이션이 액세스할 수 있고, 두 시간 후에 세션이 갱신되지 않으면 만료되어 해당 테이블에서 삭제됨
    - **재해 복구**에도 DynamoDB를 활용합니다 백업 옵션
        - 지정 시간 복구(PITR)를 사용하여 지속적인 백업을 할 수 있음
        - 활성화를 선택할 수 있고 35일 동안 지속
        - 활성화하면 백업 기간 내에는 언제든 지정 시간 복구를 실행할 수 있음
        - 복구를 진행할 경우 새로운 테이블을 생성
        - 이보다 더 긴 백업 옵션으로는 온디맨드 백업이 있고, 백업은 직접 삭제할 때까지 보존
        - 온디맨드 백업은 DynamoDB의 성능이나 지연 시간에 영향을 주지 않음
        - 백업을 좀 더 제대로 관리할 수 있는 방법 중 하나로 AWS Backup 서비스가 있음
        - 백업에 수명 주기 정책을 활성화할 수 있고, 재해 복구 목적으로 리전 간 백업을 복사할 수 있음
        - 이 옵션 또한 백업으로 복구를 진행하면 새로운 테이블이 생성
- DynamoDB와 Amazon S3 간 통합
    - S3에 테이블을 내보낼 수 있는데요 이를 위해서는 지정 시간 복구 기능을 활성화해야 함
    - DynamoDB 테이블을 S3로 내보내고 가령 쿼리를 수행하려면 Amazon Athena 엔진을 사용
    - 지속적인 백업을 활성화한 상태이므로 최근 35일 이내 어떤 시점으로든 테이블을 내보낼 수 있음
    - 테이블을 내보내도 테이블의 읽기 용량이나 성능에 영향을 주지 않음
    - DynamoDB를 Amazon S3로 먼저 내보내기 하여 데이터 분석을 수행할 수 있음
    - 감사 목적으로 스냅샷을 확보할 수도 있고, 데이터를 DynamoDB로 다시 가져오기 전에 데이터 ETL 등 대규모 변경을 실행할 수도 있음
    - 내보낼 때는 DynamoDB JSON이나 ION 형식을 이용하여 Amazon S3에서 테이블을 가져올 수도 있음
    - S3에서 CSV, JSON 그리고 ION 형식으로 내보낸 다음 새로운 DynamoDB 테이블을 생성하는 식
    - 쓰기 용량을 소비하지 않고 새로운 테이블이 생성

- 가져올 때 발생한 오류는 모두 CloudWatch Logs에 기록

API Gateway 개요

- Lambda 함수에서 API의 데이터베이스로 DynamoDB를 사용할 수 있으며 테이블을 생성, 읽기, 업데이트 삭제할 수 있음
- 클라이언트도 이 Lambda 함수를 지연 호출(Invoke)하게 하고 싶을 때 클라이언트가 직접 Lambda 함수를 지연 호출할 수 있게 하려면 클라이언트에게 IAM 권한이 있어야 함
- 클라이언트와 Lambda 함수 사이에 애플리케이션 로드 밸런서를 배치하면 Lambda 함수가 HTTP 엔드포인트에 노출
- API Gateway를 사용하는 방법도 있음
- AWS의 서버리스 서비스로 REST API를 생성할 수 있으므로 클라이언트가 퍼블릭 액세스할 수 있음
- API Gateway에 클라이언트가 직접 소통하며 다양한 작업을 할 수 있고, Lambda 함수에 요청을 프록시
- API Gateway를 사용하는 이유는 HTTP 엔드포인트뿐만 아니라 본 섹션에서 배울 다양한 기능 예를 들어 인증부터 사용량 계획, 개발 단계 등의 기능을 제공하기 때문
- API Gateway의 다양한 기능
- API Gateway는 Lambda와 통합하면 완전 서버리스 애플리케이션이 구축되므로 인프라 관리가 필요 없음
- WebSocket 프로토콜을 지원하므로 API Gateway로 두 가지 방법의 실시간 스트리밍이 가능
- API 버저닝을 핸들링하므로 버전 1, 2, 3이 생겨도 클라이언트 연결이 끊기지 않음
- dev, test, prod 등 여러 환경을 핸들링함
- 보안에도 활용할 수 있음
- 인증, 권한 부여 등 수많은 보안 기능을 API Gateway에 활성화할 수 있음
- API 키를 생성할 수 있고, API Gateway에 클라이언트 요청이 과도할 때 요청을 스로틀링할 수 있음
- Swagger나 Open API 3.0과 같은 공통 표준을 사용하여 신속히 API를 정의하여 가져올 수 있음
- Swagger나 Open API로 내보낼 수도 있음
- API Gateway 수준에서 요청과 응답을 변형하거나 유효성을 검사해 올바른 호출이 실행되게 할 수 있음
- SDK나 API 스펙을 생성하거나 API 응답을 캐시 가능
- 애플리케이션 로드 밸런서 같은 간단한 서비스와 달리 API Gateway에는 다양한 기능이 있음
- API Gateway는 어떤 통합을 지원?
- Lambda 함수
  - Lambda 함수를 지연 호출, Lambda 함수를 사용하는 REST API를 완전 서버리스 애플리케이션에 노출시키는 가장 일반적이고 간단한 방법
- HTTP와도 통합
  - 백엔드의 HTTP의 엔드포인트를 노출시킬 수 있음
  - 온프레미스에 HTTP API가 있거나 클라우드 환경에 애플리케이션 로드 밸런서가 있을 때 API Gateway를 활용하면 속도 제한 기능 캐싱, 사용자 인증, API 키 등의 기능을 추가할 수 있음
  - HTTP 엔드포인트에서 API Gateway 계층을 활용
- AWS 서비스와도 통합
  - 어떤 AWS API라도 노출시킬 수 있음
  - 단계 함수 워크플로우를 시작할 수 있고, API Gateway에서 직접 SQS에 메시지를 게시할 수도 있음

    - 인증을 추가하거나 API를 퍼블릭으로 배포하거나 특정 AWS 서비스에 속도 제한을 추가하기 위해 통합
    - AWS 서비스에 API Gateway를 사용하는 예시
    - Kinesis Data Streams
    - Kinesis Data Streams에 사용자가 데이터를 전송할 수는 있지만 AWS 자격 증명은 가질 수 없도록 보안 설정을 할 수 있음
    - Kinesis Data Streams과 클라이언트 사이에 API Gateway를 둠
    - 클라이언트가 API Gateway로 HTTP 요청을 보내면 Kinesis Data Streams에 전송하는 메시지로 구성해 전송 ->따로 서버를 관리할 필요가 없음
    - Kinesis Data Streams에서 Kinesis Data Firehose로 레코드를 전송하고, 최종적으로 JSON 형식으로 Amazon S3 버킷에 저장
- API Gateway를 통해 AWS 서비스를 외부에 노출
- API Gateway 배포 방법은 세 가지이며 이를 엔드포인트 유형
    - 기본 유형인 엣지 최적화
        - 글로벌 클라이언트용
        - 전 세계 누구나 API Gateway에 액세스할 수 있음
        - 모든 요청이 CloudFront 엣지 로케이션을 통해 라우팅되므로 지연 시간이 개선됨
        - API Gateway는 여전히 생성된 리전에 위치하지만 모든 CloudFront 엣지 로케이션에서 액세스될 수 있음
        - CloudFront 엣지 로케이션을 원하지 않을 때는 리전 배포를 사용
        - 모든 사용자는 API Gateway를 생성한 리전과 같은 리전에 있어야 함
        - 자체 CloudFront 배포를 생성할 수도 있음
        - 엣지 최적화 배포와 동일한 결과를 내며 캐싱 전략과 CloudFront 설정에 더 많은 권한을 가질 수 있음
    - API Gateway 배포 유형은 프라이빗으로 퍼블릭 배포가 아님
    - 프라이빗 API Gateway는 VPC 내에서만 액세스할 수 있음
    - ENI 같은 인터페이스 VPC 엔드포인트를 사용
    - API Gateway에 액세스를 정의할 때는 리소스 정책을 사용
- API Gateway의 보안
- 사용자를 식별하는 방법에는 여러 가지가 있음
- 첫 번째 방법은 IAM 역할을 사용
- 가령 EC2 인스턴스에서 실행되는 내부 애플리케이션에서 유용
- API Gateway의 API에 액세스할 때 IAM 역할을 사용하도록
- 모바일 애플리케이션이나 웹 애플리케이션의 외부 사용자에 대한 보안 조치로 Amazon Cognito를 사용할 수 있음
- 자체 로직을 실행하려면 사용자 지정 권한 부여자를 사용 -> Lambda 함수
- HTTPS 보안
- 사용자 지정 도메인 이름을 AWS Certificate Manager 즉 ACM과 통합할 수 있음
- 엣지 최적화 엔드포인트를 사용할 경우 인증서는 us-east-1에 있어야 하고, 리전 엔드포인트를 사용한다면 인증서는 API Gateway 단계와 동일한 리전에 있어야 함
- Route 53에 CNAME이나 A-별칭 레코드를 설정해 도메인 및 API Gateway를 가리키도록 해야 함

Step Functions

- Step functions는 서버리스 워크플로를 시각적으로 구성할 수 있는 기능으로 주로 람다 함수를 오케스트레이션 하는 데 활용
- 그래프를 만드는데 각 그래프 단계별로 해당 단계의 결과에 따라 다음으로 수행하는 작업이 뭔지 정의
- 복잡한 워크플로를 만들어 AWS에서 실행시킬 수 있는 편리한 도구
- Step functions가 제공하는 기능으로는 시퀀싱, 병행 실행, 조건 설정, 타임아웃, 에러 처리하기 등
- 람다 함수만 처리하는 게 아니라 EC2랑도 연동할 수 있고, ECS, 온프레미스 서버, API 게이트웨이랑 SQS 큐 등 다양한 AWS 서비스를 워크플로에 넣을 수 있음
- 워크플로에 사람이 개입해서 승인을 해야만 진행되는 단계를 설정할 수 있음
- 예를 들어, 어떤 지점에 이르러서는 사람이 결과를 확인하고, 승인을 하면 다음 단계로 넘어 가고, 아니면 워크플로가 멈춰 실패
- Step functions가 제공하는 유용한 기능임
- 다양한 사용처가 있는데, 예를 들어 주문 이행이나 데이터 처리, 웹 애플리케이션 등 구성하기 복잡한 워크플로를 시각적으로 구성하려고 할 때 사용

Cognito

- Cognito는 사용자에게 웹 및 모바일 앱과 상호 작용할 수 있는 자격 증명을 부여
- 일반적으로 이 사용자들은 AWS 계정 외부에 있음
- 모르는 사용자들에게 자격 증명을 부여해 사용자를 인식(Cognito) 함
- Cognito에는 두 종류의 하위 서비스
  - 앱 사용자에게 가입 기능을 제공하는 Cognito 사용자 풀
    - API Gateway 및 애플리케이션 로드 밸런서와 원활히 통합
  - 다음은 페더레이션 자격 증명이라고 불리던 Cognito 자격 증명 풀
    - 앱에 등록된 사용자에게 임시 AWS 자격 증명을 제공해서 일부 AWS 리소스에 직접 액세스할 수 있도록 해 주고, Cognito 사용자 풀과 원활히 통합됨
    - IAM에 이미 사용자가 있다고 생각할 수 있지만 Cognito는 AWS 외부의 웹과 모바일 앱 사용자를 대상
- '수백 명의 사용자' '모바일 사용자' 'SAML을 통한 인증' 같은 키워드가 나오면 Cognito를 설명
- Cognito 사용자 풀(CUP)
  - 웹 및 모바일 앱을 대상으로 하는 서버리스 사용자 데이터베이스
  - 사용자 이름 또는 이메일, 비밀번호의 조합으로 간단한 로그인 절차를 정의할 수 있음
  - 비밀번호 재설정 기능이 있고 이메일 및 전화번호 검증 및 사용자 멀티팩터 인증이 가능
  - Facebook이나 Google과 통합할 수 있어 소셜 로그인도 가능
  - Cognito 사용자 풀은 API Gateway나 애플리케이션 로드 밸런서와 통합 가능
  - API Gateway를 예로 들면 사용자는 Cognito 사용자 풀에 접속해서 토큰을 받고, 검증을 위해 토큰을 API Gateway에 전달
  - 확인이 끝나면 사용자 자격 증명으로 변환되어 백엔드의 Lambda 함수로 전달
  - Lambda 함수는 처리할 사용자가 인증된 구체적인 사용자라는 사실을 인식
  - Cognito 사용자 풀을 애플리케이션 로드 밸런서 위에 배치하도록 할 수도 있음

    - 애플리케이션이 Cognito 사용자 풀에 연결한 다음 애플리케이션 로드 밸런서에 전달해서 유효한 로그인인지 확인
    - 유효하다면 요청을 백엔드로 리다이렉트하고, 사용자의 자격 증명과 함께 추가 헤더를 전송
    - API Gateway나 ALB를 통해 사용자의 로그인을 한곳에서 확실히 검증할 수 있고, 검증 책임을 백엔드로부터 백엔드의 로드를 밸런싱하는 실제 위치, 즉 API gateway 또는 ALB로 옮긴 것
- Cognito의 또 다른 하위 서비스는 Cognito 자격 증명 풀 -> 페더레이션 자격 증명
    - 사용자에게 자격 증명을 제공하지만 API Gateway나 애플리케이션 로드 밸런서를 통해서 애플리케이션에 액세스하지 않고, 임시 AWS 자격 증명을 사용해 AWS 계정에 직접 액세스
    - 사용자는 Cognito 사용자 풀 내의 사용자가 될 수도 있고, 타사 로그인이 될 수도 있음
    - 직접 또는 API Gateway를 통해 서비스에 액세스할 수도 있음
    - 자격 증명에 적용되는 IAM 정책은 Cognito 자격 증명 풀 서비스에 사전 정의되어 있음
    - user_id를 기반으로 사용자 정의하여 세분화된 제어를 할 수도 있음
    - 원한다면 기본 IAM 역할을 정의할 수도 있음
    - 게스트 사용자나 특정 역할이 정의되지 않은 인증된 사용자는 기본 IAM 역할을 상속
    - 웹이나 모바일 애플리케이션에서 S3 버킷 또는 DynamoDB 테이블에 직접 액세스하고 싶을 때
    - Cognito 자격 증명 풀을 사용하면 웹, 모바일 앱이 로그인해서 토큰을 받음
    - Cognito 사용자 풀에 대한 로그인일 수도 있고, 소셜 자격 증명 제공자나 SAML OpenID Connect일 수도 있음
    - 이 토큰을 Cognito 자격 증명 풀 서비스에 전달해서 토큰을 임시 AWS 자격 증명과 교환
    - Cognito 자격 증명 풀은 전달 받은 토큰이 올바른지, 즉 유효한 로그인인지 평가
    - 해당 사용자에게 적용되는 IAM 정책을 생성
    - IAM 정책이 적용된 임시 자격 증명 덕분에 AWS의 S3 버킷이나 DynamoDB 테이블에 직접 액세스할 수 있는 것 -> API Gateway나 애플리케이션 로드 밸런서를 거치지 않고도 가능
- Cognito 자격 증명 풀을 사용하면 DynamoDB에 행 수준 보안을 설정할 수 있음
- Cognito 자격 증명 풀에 화면과 같은 정책이 있을 때 이 안에 조건을 설정
- DynamoDB의 LeadingKeys가 Cognito 자격 증명 사용자 ID와 같아야 한다는 조건
- 이 정책이 적용된 사용자는 DynamoDB 테이블의 모든 항목을 읽고 쓸 수 있는 것이 아니라 이 조건을 통해 액세스를 얻은 항목에만 액세스할 수 있음
- 웹이나 모바일 애플리케이션의 사용자 기반을 생성할 수 있고, 세분화된 액세스 제어를 위해 DynamoDB에서 행 수준 보안을 활성화할 수 있음
- Cognito 사용자 풀과 API Gateway 또는 애플리케이션 로드 밸런서를 통합할 수 있음