모바일 애플리케이션: **MyTodoList**

- MyTodoList라는 모바일 애플리케이션을 만들 경우
- 요구 사항
  - HTTPS 엔드 포인트가 있는 REST API가 노출돼야 함
  - 서버리스 아키텍처
  - 사용자가 원한다면 스스로 데이터를 관리하도록 S3에 있는 폴더와 직접 상호작용이 가능해야 함
  - 사용자가 관리형 서버리스 서비스로 인증할 수 있어야 함
  - 마지막으로 사용자가 할 일을 읽고 쓸 수 있지만 대부분 읽기를 많이 하니까 관련 성능이 필요
  - 데이터베이스 계층은 확장할 수 있도록 구축해 읽기 처리량을 높여야 함
- 모바일 클라이언트가 있어야 함
- REST HTTPS 얘기했으니까 Amazon API Gateway를 사용
- 서버리스 API 방식으로 API Gateway가 람다 함수를 호출해서 확장을 허용하고 서버리스 인프라를 쓰도록 함
- Amazon Lambda는 데이터베이스에서 할 일을 저장하거나 읽어 내야 하는데
- 서버리스이면서 확장이 잘 되는 건 DynamoDB
- DynamoDB를 백엔드로
- 인증 계층이 있어야 하는데 Amazon Cognito 같은 서버리스 기술을 쓸 수 있음
- 모바일 클라이언트가 Cognito에 연결하고 인증하면 API Gateway는 Cognito와 함께 인증을 확인해 줌 ->서버리스 API
- Amazon S3 버킷에 사용자 액세스 권한을 주려면 어떻게 해야 하나?
- Amazon Cognito에 인증을 요청하는 모바일 클라이언트가 있다고 할 경우 Cognito는 AWS STS를 통해 임시 자격 증명을 제공할 수 있음
- 자격 증명을 모바일 클라이언트에게 반환 함
- 이 자격 증명은 클라이언트가 Amazon S3에서 파일을 저장하고 회수하며 S3에서 전용 공간에 액세스하도록 허용
- 이를 어떻게 처리할지 묻는다면 오답은 'AWS 사용자 자격 증명을 모바일 클라이언트에 저장한다' 임
- Amazon Cognito와 STS Amazon S3를 임시 자격 증명에 사용해야 함
- 애플리케이션이 확장됨
- 사용자가 늘어나기 시작하고, 읽기 처리량이 아주 높다는 걸 알 수 있는데 RCU는 많고 할 일이 많이 바뀌지는 않으니까 자주 편집할 필요는 없음
- 읽기 처리량을 늘리면서 전체 비용을 줄이려면 아키텍처를 어떻게 바꿔야 할까?
- 캐싱 계층으로 DAX를 사용하면 됨
- DynamoDB 이전에 DynamoDB DAX를 사용해 캐싱 계층을 갖추는 것
- 읽기를 많이 하는데 읽어 낸 내용이 DAX 캐시에 저장돼 DynamoDB 읽기 용량 단위가 많이 필요하지 않게 되고, 확장이 용이하며 비용도 줄어듦
- 많은 내용을 읽어 캐시에 저장하니까 서버리스 방식으로 꾸준히 아키텍처를 개선하기에 적합
- 캐싱에는 다른 방법도 있음

- DAX도 사용 가능하지만 캐시에 응답을 저장할 때 Amazon API Gateway 레벨에서 진행할 수도 있는데 답이 거의 바뀌지 않으면서 API 라우트에 대한 응답을 캐시에 저장하는 경우 유용한 방법
- 임시 자격 증명을 생성하는 데 제한된 정책에서 추가 버킷에 액세스를 제공하는 STS와 함께 Cognito를 사용했으며 Cognito를 이용하면 똑같은 애플리케이션 패턴으로 애플리케이션이 DynamoDB나 Lambda에 직접 액세스할 수도 있음
- DAX로 DynamoDB상에서 읽은 내용을 캐시에 저장하면 쉽게 활성화하면서 성능을 개선하고 비용을 절감할 수 있음
- REST 요청 캐싱은 응답이 고정된 경우 API Gateway 레벨에서 가능
- 보안은 Cognito로 가능한데 Cognito는 API Gateway와 직접 통합되어 있음

서버리스 웹사이트: **MyBlog.com**

- 요구사항
  - 이 웹사이트는 글로벌 스케일 아웃이 가능해야 합니다, 보통은 블로그를 작성하는 빈도보다 읽는 빈도가 높음
  - 이 웹사이트는 대부분 정적 파일로 구성되고 일부는 동적인 REST API로 구성됨
  - 캐시를 적용해서 비용을 절감하고 응답 속도도 높이고 좋은 UX를 제공하고 싶음
  - 블로그에 처음으로 방문하는 사람에겐 따뜻한 환영 이메일을 발송하고 싶고, 서버리스로 구현하고 싶고, 블로그에 업로드 되는 사진은 섬네일이 생성되면 좋음
  - 서버리스로 구현하고 싶음
- 콘텐츠를 서빙해야 하는데요, 콘텐츠는 정적이고 글로벌을 대상으로 서빙해야 함
- 클라이언트가 있고 우리의 콘텐츠는 Amazon S3에 저장돼 있음
- 콘텐츠를 어떻게 제공하면 될까? -> 버킷으로
- Amazon S3는 특정 리전에 있어 글로벌을 대상으로 노출시킬 수 있을까? -> 전 세계를 대상으로 서비스되는 CDN 서비스인 Amazon CloudFront를 이용하면 됨
- 클라이언트는 Amazon CloudFront의 엣지 로케이션과 통신
- Amazon S3에게 받은 데이터를 캐시로 저장
- CloudFront는 전형적인 구조로 설계돼 있음
- 모든 작업을 안전하게 진행하려면 어떻게 할 수 있을까?
- 클라이언트는 글로벌 배포를 위한 CloudFront와 인터랙트
- 오리진 액세스를 제한해서 CloudFront로만 Amazon S3 버킷에 접근할 수 있도록 함
- CloudFront만 인가할 수 있도록 버킷 정책을 추가
- Amazon S3에 바로 접근하는 사용자는 인가를 받지 못하고 Amazon S3 버킷은 보호됨
- 퍼블릭 서버리스 REST API는 어떻게 추가할까?
- REST HTTPS로 Amazon API 게이트웨이와 통신하고 게이트웨이는 람다 함수를 호출하고 람다 함수는 Dynamo DB에 쿼리를 날리고, 읽기가 많이 발생하니 DAX를 캐시 레이어로 쓰면 좋음
- 글로벌로 서빙할 때 지역에 따라 발생하는 지연을 줄이기 위해 Dynamo DB의 글로벌 데이터베이스를 사용하는 것도 좋음
- 전체 인프라와 이 구성의 속도를 향상시켜 주는 좋은 방법
- 신규 방문자에게 보낼 환영 이메일 보내고 싶으면?
- 방문자가 구독을 하면 '반갑습니다, 안녕하세요'라는 인사를 담은 메일을 보내고 싶음

- Dynamo DB에서 변경사항을 전송하기 위해 Dynamo DB 스트림을 이용하고 스트림은 람다 함수를 호출
- IAM 역할을 부여 받아 Amazon SES를 사용할 수 있게 해줌
- Amazon SES를 아직 다루진 않았지만 어렵지 않음
- Amazon Simple Email Service가 전체 이름 줄여서 SES인 거죠, 이메일을 발송해 주는 서비스
- Amazon 람다 함수가 AWS SDK를 이용해 Amazon SES가 이메일을 발송시킴
- 서버리스로 환영 이메일 발송 기능을 이렇게 구현할 수 있습니다, 간단하죠, 관리해야 할 인프라도 없고, 작동도 잘하고 확장 가능
- 이미지를 업로드하면 섬네일을 생성은?
- 클라이언트에서 S3 버킷으로 바로 업로드를 할 수도 있고 아니면 CloudFront OAI를 이용 가능
- 이 방식이라면 클라이언트는 사진을 CloudFront에 업로드하고 CloudFront는 사진을 Amazon S3 버킷으로 전달 -> 이런 방식을 S3 Transfer Acceleration
- 사진은 바로 S3로 올리거나 S3 Transfer Acceleration을 이용하거나 S3에 파일이 추가되면 람다 함수를 호출
- S3가 람다를 호출할 수 있고, 람다는 섬네일을 생성하고 섬네일을 S3 버킷에 넣는데, 다른 버킷에 넣을 수도 있 dma
- Amazon S3는 SQS와 SNS를 호출할 수도 있음
- SQS나 SNS로 원하는 작업을 하시면 됩니다, 작업을 원하는 대로 구성 가능
- Amazon S3는 람다, SQS, SNS를 호출할 수 있고 솔루션을 자유롭게 구성하실 수도 있고, 또 서버리스로 구성하실 수도 있고, 편한 방법으로 구성할 수 있음
- 정적 콘텐츠를 CloudFront와 S3로 배포하는 방법을 살펴봤고 서버리스인 REST API를 붙이는 방법을 살펴봄
- API가 퍼블릭이어서 Cognito가 필요 없었습니다, 데이터를 글로벌로 서빙하기 위해 Dynamo DB를 붙임
- 오로라 글로벌 데이터베이스를 사용할 수도 있었지만 이 사례가 서버리스가 아닌 프로비저닝 오로라를 사용해야 함
- Dynamo DB 스트림을 활용할 수 있음
- 사용자 테이블에 발생하는 변경사항을 알려주는 용도이며 람다 함수 호출 용도도 있음
- 호출되는 람다 함수엔 IAM 역할을 부여해서 SES, 즉 simple email service를 이용하도록 함
- 서버리스 방식으로 이메일을 발송하기 위한 방법

마이크로 서비스 아키텍처

- 마이크로서비스 아키텍처로 전환하고자 함
- 많은 서비스가 REST API를 통해 상호작용 함
- REST API를 사용
- 마이크로서비스의 아키텍를 쓰려는 이유는 서비스 개발 수명 주기를 줄이고자 함
- 각 서비스가 독립적으로 확장하며 코드 리포지토리를 갖추길 원한다면 마이크로서비스 아키텍처를 사용하면 됨
- 사용자는 첫 마이크로서비스와 HTTPS를 통해 통신
- 일래스틱 로드 밸런서가 ECS와 통신하고 난 뒤에 DynamoDB와 통신하도록 함
- ECS는 AWS에 도커 컨테이너를 쓸 때 사용하는 서비스
- 마이크로서비스는 보통 DNS 이름이나 URL이 있음

- service1.example.com을 예시로
- 정보를 얻으려면 DNS 쿼리를 Route 53에 보내 별칭 레코드를 받고 나서 서비스와 상호작용이 가능 -> 이게 첫 번째 서비스
- 두 번째 서비스를 마련하려면 전형적인 서버리스용 아키텍처를 사용
- 하지만 DynamoDB 대신 일래스티 캐시가 있는데 Lambda의 백엔드로 일래스티 캐시를 써도 괜찮음
- 두 번째 마이크로서비스는 첫 번째 서비스와 상호작용 할 수도 있음
- 일래스틱 로드 밸런서에 람다 함수가 호출을 보내 첫 마이크로서비스에서 정보를 얻고, 응답을 생성
- 역시 ELB를 사용하는 세 번째 마이크로서비스를 마련할 수도 있는데 이건 서버리스가 아니라 Amazon EC2 오토 스케일링과 Amazon RDS 데이터베이스를 사용하는 거라 전형적인 아키텍처에 해당
- EC2 인스턴스가 결정을 내리기 전에 두 번째 마이크로서비스를 호출해야 함
- URL은 service3.example.com임
- 마이크로서비스에는 두 가지 패턴
- 동기식 패턴은 다른 마이크로서비스를 명시적으로 호출
  - API Gateway와 로드 밸런서는 다른 마이크로서비스에 HTTPS 호출을 보내기에 적합
- 비동기식 패턴도
  - S3처럼 SQS, Kinesis, SNS Lambda에 트리거로 작용하는 경우
  - SQS에 메시지를 남기지만 응답을 언제 받을지 내용이 뭔지 또 무슨 일이 일어날지 개의치 않음
- 마이크로서비스의 문제는 새로운 마이크로서비스를 생성할 때마다 **오버헤드가 발생**
- 서버 밀도나 사용률을 최적화하는 데 어려움
- 여러 버전의 마이크로서비스를 동시에 가동하려면 복잡
- 여러 서비스와 통합하려다 클라이언트 코드 요구사항이 급증하기도 함
- 서버리스 패턴으로 해결
- API Gateway나 Lambda는 자동으로 확장
- 사용량만큼 돈만 내면 되니까 서버 사용률을 걱정할 필요가 없음
- API Gateway에서 API를 쉽게 복제하고 재생산하며 API Gateway를 위한 Swagger 통합으로 클라이언트 SDK 생성이 가능
- 마이크로서비스는 AWS를 사용해 원하는 대로 설계할 수 있는데 문제를 해결해 주기도 하지만 말썽을 일으킬 때도 있음

소프트웨어 업데이트 배포

- 소프트웨어 업데이트 오프로딩
- EC2에서 작동하는 애플리케이션이 있고, 종종 소프트웨어 업데이트를 배포함
- 컴퓨터에서 패치를 다운로드해 EC2에 설치하는 상황일 때
- 소프트웨어가 새로 업데이트되면 요청을 많이 받게 됨
- 콘텐츠는 네트워킹을 통해 다수에게 배포되는데 비용이 많이 듦
- 애플리케이션을 변경하거나 아키텍팅을 다시 하는 일 없이 비용과 CPU를 최적화하고 싶을 때는?
- 애플리케이션의 현재 상태
  - 전형적인 ELB와 ASG 개발 애플리케이션이 있는데 다중 AZ에 걸쳐 작동합니다 여기 보이는 M5 인스턴스는 소프트웨어 업데이트를 배포

    - 소프트웨어 업데이트 모두 Amazon EFS에 위치한다고 가정
- 비용을 절감하는 방법 -> CloudFront를 상위에 두면 됨
- 아키텍처에는 변화가 없음
- 엣지에서 소프트웨어 업데이트 파일은 캐시에 저장
- 업데이트 파일은 동적이 아니라 정적이라서 변하지 않음
- EC2 인스턴스는 서버리스가 아니지만 CloudFront는 서버리스라서 확장 가능
- ASG가 많이 확장하지 않아 EC2와 네트워크, EFS 비용을 크게 절감
- 가용성도 확보
- **CloudFront가 기존 애플리케이션의 확장성을 높이고, 비용을 절감하는 데 용이**
- 엣지에서 캐싱을 활용하는 정적 콘텐츠가 대부분인 경우에