

LambdaとDynamoDBで IoTバックエンド開発

今日のテーマ

LambdaとDynamoDBで IoTバックエンド開発

=>いろいろな「これどうしてるの？」
をご共有します！

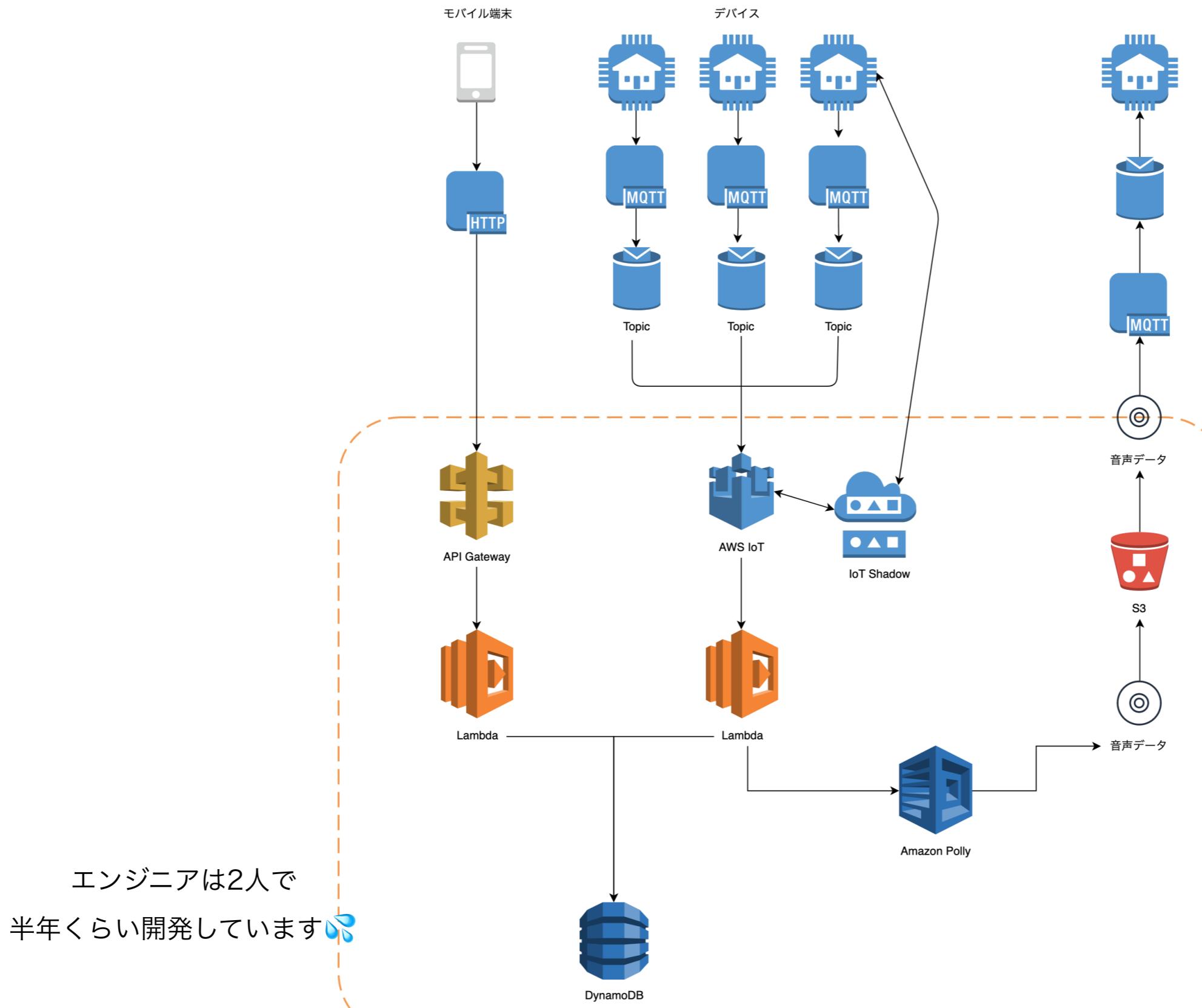
「これどうしてるの？」

- Serverlessアプリケーションの管理
- DynamoDBテーブル設計
- DynamoDBトランザクション
- MQTT Topic設計
- Lambdaの監視

機能要件

- ・ いわゆる見守りスピーカーのような機能
 - ・ 家に置くIoTスピーカー
 - ・ モバイルアプリ側からメッセージ登録
 - ・ デバイス側で人感センサー=>メッセージ再生
 - ・ 外出時のリマインダーや天気予報の設定なども
- ・ モバイルアプリでユーザー管理やビーコン管理もできる
- ・ モバイルアプリと通信するWeb API
- ・ AWS IoTからMQTTでつなぐバックエンド

設計概要



Serverlessアプリケーションの管理

Serverlessアプリケーションの管理

一般的なWebアプリケーションなどで行うこと
+ Serverlessでのみ行うこと

- ビルド
- パッケージ
- デプロイ
- テスト
- 開発・ステージング・プロダクションなどの環境管理
- Lambda以外の関連リソース管理
- 各リソースへの適切なロール管理
- 環境変数

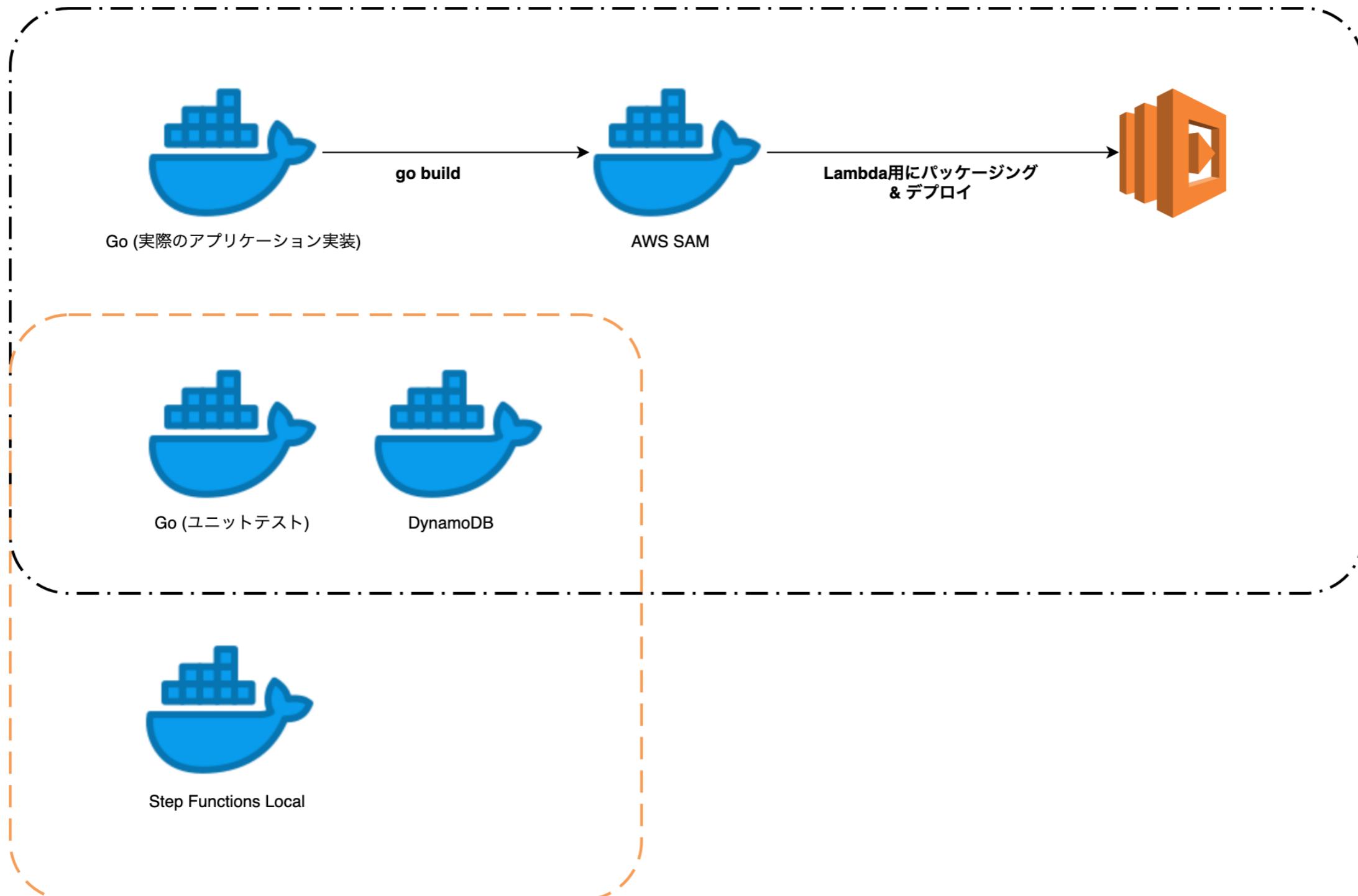
=>AWS SAM

AWS SAMについて

- Lambdaを使ったServerlessアプリケーションを管理できるモデル
 - ※実態はCloudFormation
 - いろいろいい感じにやってくれる
 - 大小関わらず他の案件での実績・知見がある
-
- AWS SAMで始めるサーバーレスアプリケーション開発 <https://www.slideshare.net/YoshidaShingo/getting-started-with-aws-sam>
 - サーバーレスなバックアップシステムを AWS SAM を用いてシュッと構築する - クックパッド開発者ブログ <https://techlife.cookpad.com/entry/2018/02/07/090645>

設計概要

CircleCI



ローカル開発環境

その他、様々なリソースをAWS SAMで管理



API Gateway



CloudWatch



S3



sqS



sns

AWS SAMでまかなえない部分は別リポジトリのCloudFormationで管理



各種ロール



KMS(環境変数の暗号化に使用)

現在、100以上のLambda関数がある

=>Lambda関数を作るたびにAWS SAM(CloudFormation)の
設定を書く必要があって大変

- 設定し忘れたり…
- タイポしたり…
- 全ての関数にひとつずつ共通のプレフィックスを付けたり…

template.ymlの自動生成

例. APIのRoutesを読み込む

```
func SetRoutes() *routes.Router {
    var router = &routes.Router{}

    router.Post("/v1/hoges", PostHoge)
}
```

template.ymlの自動生成

例. CloudFormationテンプレートに変換する

```
func (s *APIGatewayConfig) ToLambdaYAML(stackNumber int) string {
    t := map[string]interface{}{}
    s.HandlerName(): map[string]interface{}{
        "Type": "AWS::Serverless::Function",
        "Properties": map[string]interface{}{
            "FunctionName": s.LambdaFunctionNameWithRef(stackNumber),
            "CodeUri":      APIHandlerCodeURI,
            "Role":          RoleRefStr,
            "Handler":       "main",
        },
    },
}

d, err := yaml.Marshal(&t)
if err != nil {
    log.Panicf("%+v", err)
}

str := strings.Replace(string(d), "'", "", -1)

return str
}
```

template.ymlの自動生成

例. CloudFormationテンプレートに変換する

```
PostV1HogesHandler:  
  Properties:  
    CodeUri: ../handlers/api  
    FunctionName: !Sub api-1-PostV1Hoges${ExtraNameForTest}  
    Handler: main  
    Role: {"Fn::ImportValue": !Sub "${InfraStackName}-LambdaRoleArn"}  
  Type: AWS::Serverless::Function
```

ビジネスロジックに集中できる 😊

- Go言語でメタプログラミング <https://blog.mmmcorp.co.jp/blog/2019/07/01/metago/>

DynamoDBのテーブル設計

DynamoDBのベストプラクティス

https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/best-practices.html

AWS ドキュメント » Amazon DynamoDB » 開発者ガイド » DynamoDB のベストプラクティス

DynamoDB のベストプラクティス

このセクションでは、Amazon DynamoDB 使用時にパフォーマンスを最大にしてスループットコストを最小にするための推奨事項をすばやく確認することができます。

目次

- [DynamoDB に合わせた NoSQL 設計](#)
 - [リレーションナルデータ設計と NoSQL の相違点](#)
 - [NoSQL 設計の 2 つの重要な概念。](#)
 - [NoSQL 設計へのアプローチ](#)
- [パーティションキーを効率的に設計し、使用するためのベストプラクティス](#)
 - [バーストキャパシティーを効率的に使用する](#)
 - [DynamoDB アダプティブキャパシティーを理解する](#)
 - [パーティションキーを設計してワークロードを均等に分散する](#)
 - [書き込みシャーディングを使用してワークロードを均等に分散させる](#)
 - [ランダムなサフィックスを使用したシャーディング](#)
 - [計算されたサフィックスを使用したシャーディング](#)
 - [データアップロード時に書き込みアクティビティを効率的に分散する](#)
- [ソートキーを使用してデータを整理するためのベストプラクティス](#)
 - [バージョンコントロールにソートキーを使用する](#)
- [DynamoDB でセカンダリインデックスを使用するためのベストプラクティス](#)
 - [DynamoDB のセカンダリインデックスの一般的なガイドライン](#)
 - [インデックスを効率的に使用する](#)
 - [射影を慎重に選択する](#)

=>基本的にこれに従えばOK

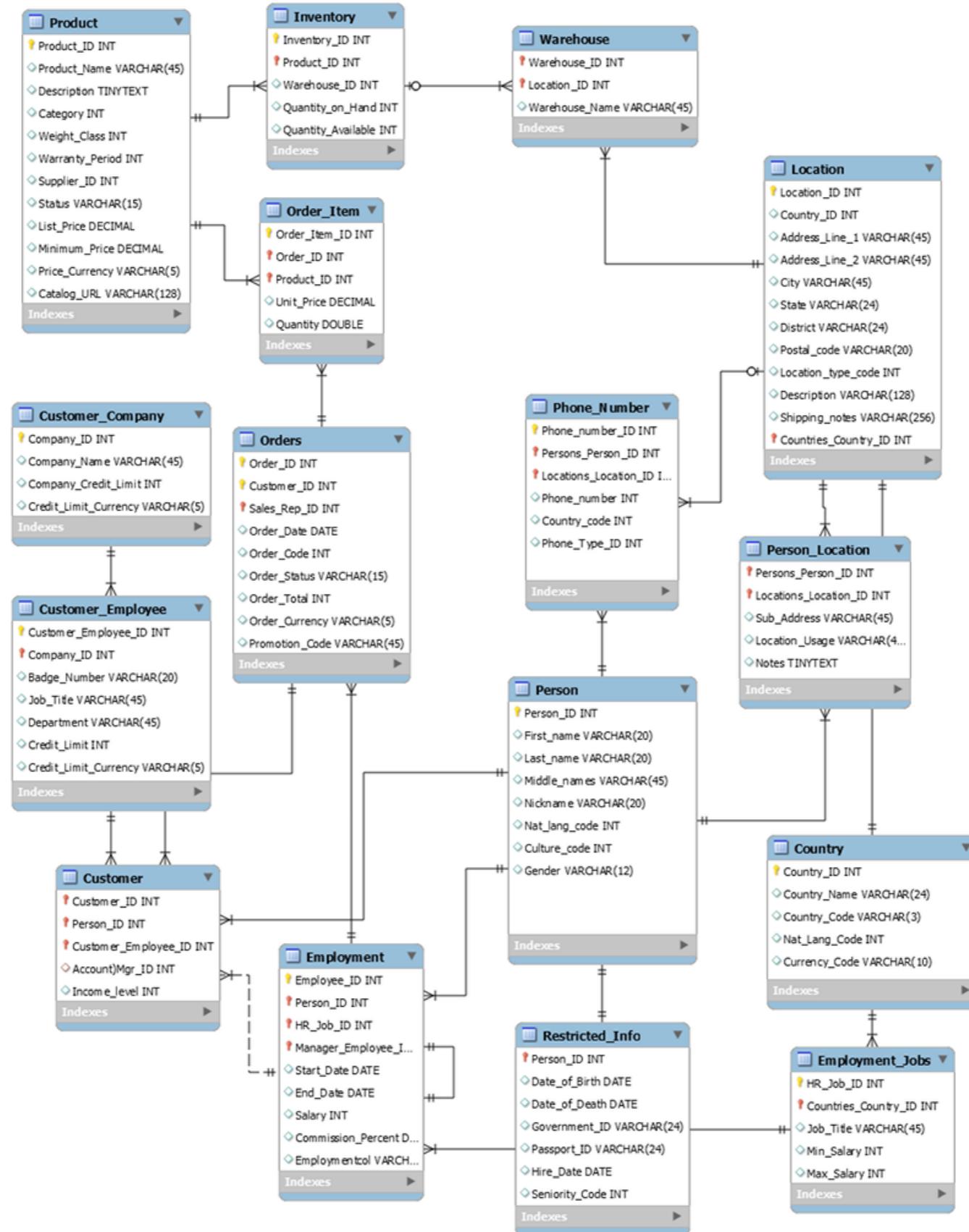
参考記事

- ・ サーバーレスアプリケーション向きの DB 設計ベストプラクティス <https://www.slideshare.net/AmazonWebServicesJapan/db-144577033>
- ・ Serverlessを極めるためにDynamoDBデータモデリングを極めよう
<https://speakerdeck.com/marcyterui/lets-become-the-master-of-dynamodb-data-modeling-to-become-the-master-of-serverless>
- ・ AppSyncを使いこなすためのDynamoDB設計パターン <https://speakerdeck.com/ktsukago/appsyncwoshi-ikonasutamefalsedynamodbsho-ji-patan>
- ・ AWS Black Belt Online Seminar 2017 Amazon DynamoDB <https://www.slideshare.net/AmazonWebServicesJapan/20170809-black-belt-dynamodb?ref=http://marcy.hatenablog.com/entry/2018/07/31/213705>

テーブル設計の大まかな手順

1. RDB風にリレーションナルなER図を書く
2. アクセスパターンを列挙
3. DynamoDBのスキーマに落とし込む

1. RDB風にリレーショナルなER図を書く



2. アクセスパターンを列挙

/RDBとはアプローチが全く異なる

- RDB
 - まずスキーマを考える
 - 正規化
 - それに対してどうアクセスするか(SQL)考える
- DynamoDB
 - スキーマレス
 - 非正規化
 - まずアクセスを考えて、それに合わせたデータを作る
 - アプリケーションとデータモデルは同時に設計する

<https://speakerdeck.com/marcyterui/lets-become-the-master-of-dynamodb-data-modeling-to-become-the-master-of-serverless>

2. アクセスパターンを列挙

Most Common/Import Access Patterns in Our Organization	
1	Look up employee details by employee ID
2	Query employee details by employee name
3	Find an employee's phone number(s)
4	Find a customer's phone number(s)
5	Get orders for a given customer within a given date range
6	Show all open orders within a given date range across all customers
7	See all employees hired recently
8	Find all employees working in a given warehouse
9	Get all items on order for a given product
10	Get current inventories for a given product at all warehouses
11	Get customers by account representative
12	Get orders by account representative and date
13	Get all items on order for a given product
14	Get all employees with a given job title
15	Get inventory by product and warehouse
16	Get total product inventory
17	Get account representatives ranked by order total and sales period

3. DynamoDBのスキーマに落とし込む

例えば検索

アクセスパターン(フィルタ条件)ごとにレコードをつくる

PK	SK	SortKey	Name	Email	CreatedAt	UpdatedAt
User-1	User-AllAttribute		Tanaka	example@example.com	2019-07-30T00:00:00Z	2019-07-30T00:00:00Z

PK	SK (GSI-PK)	SortKey (GSI-SK)	Email
User-1	SK-User-Name	Tanaka	example@example.com

PK	SK (GSI-PK)	SortKey (GSI-SK)	Name
User-1	SK-User-Email	example@example.com	Tanaka

- ・すべての項目の一覧表示
- ・ユーザーネームによる並び替え、メールアドレスによる検索
- ・メールアドレスによる並び替え、ユーザーネームによる検索

3. DynamoDBのスキーマに落とし込む

```
type EntityBase struct {
    ID      uint64   `dynamo:"ID"`
    Version int      `dynamo:"Version"`
    CreatedAt time.Time `dynamo:"CreatedAt"`
    UpdatedAt time.Time `dynamo:"UpdatedAt"`
}

type EntityBaseDynamo struct {
    PK      string `dynamo:"PK,hash"`
    SK      string `dynamo:"SK,range" index:"GSI-SortKey-1,hash"`
    SortKey string `dynamo:"SortKey" index:"GSI-SortKey-1,range"`
}

// User 構造体の定義
type User struct {
    EntityBase
    Name  string `dynamo:"Name"`
    Email string `dynamo:"Email"`
}

// UserBase すべての項目を含めたレコード
type UserBase struct {
    EntityBaseDynamo
    User
}

// UserFilter 並び替え、検索用のレコード
type UserFilter struct {
    EntityBaseDynamo
    Name  string `dynamo:"Name"`
    Email string `dynamo:"Email"`
}
```

3. DynamoDBのスキーマに落とし込む

```
// 並び替え、検索に必要なエンティティのみを含める
func (u *User) filter() *UserFilter {
    return &UserFilter{
        Name: u.Name,
        Email: u.Email,
    }
}

// GSI-SKの定義
func (u *User) sortKeys() map[string]string {
    return map[string]string{
        Name: u.Name,
        Email: u.Email,
    }
}

// 並び替え、検索用のレコード作成
func (u *User) createFilterRecord(keyName, sortKey string) interface{} {
    f := u.filter()
    f.PK = u.DynamoID()
    f.SK = keyName
    f.SortKey = sortKey
    return f
}
```

ORM風なメソッド&適切なモジュール分け

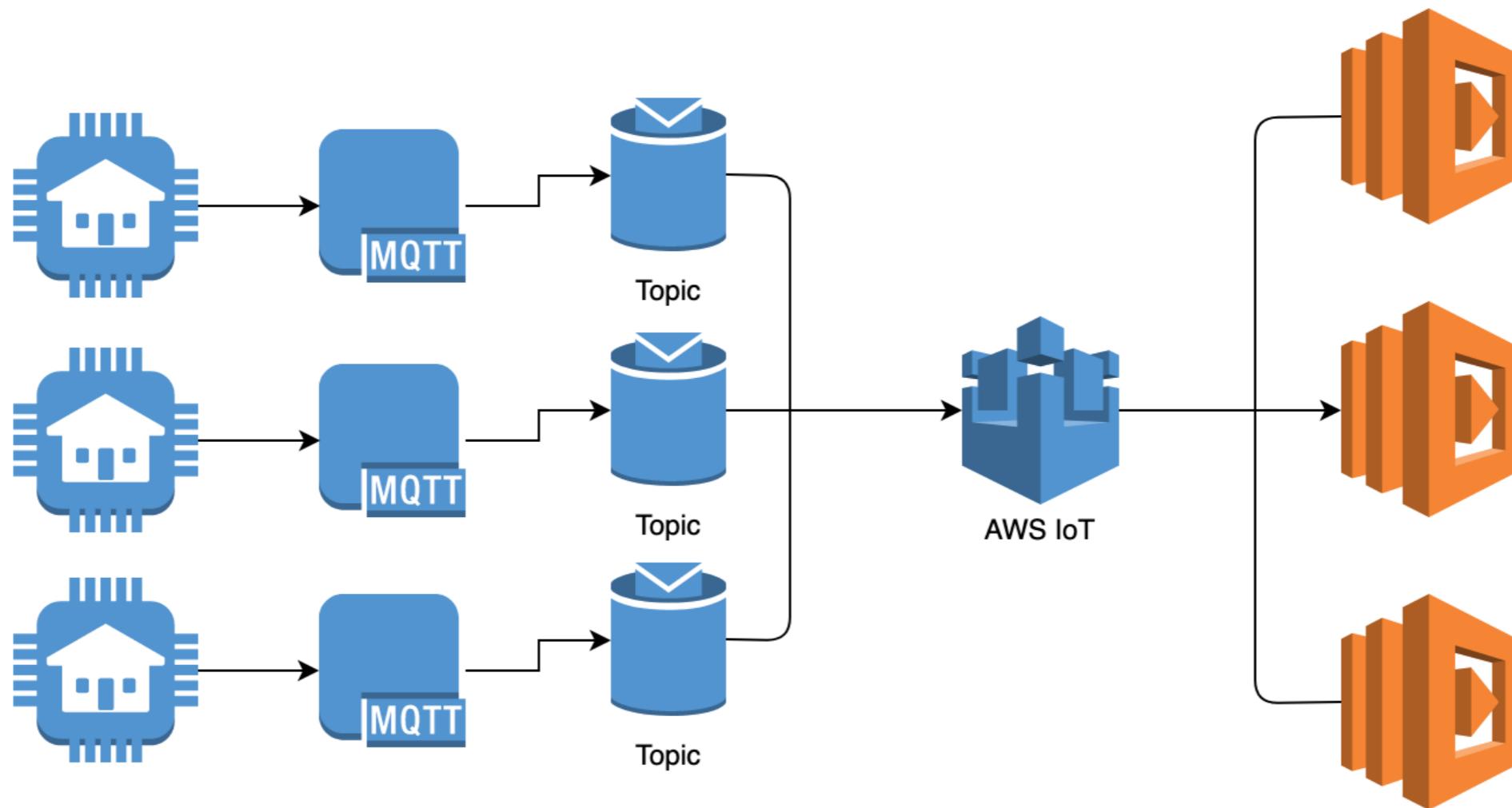
=>DynamoDBを意識することが減り、ロジック実装に集中できる👍

<https://mememememomo.booth.pm/items/1317078>

MQTT Topic設計

MQTTのTopicとは？

- ・イメージとしてはREST APIでいうところのエンドポイントのようなもの(例「/device/detection/{device-id}」)。
- ・AWS IoTでMQTTのTopicを元にLambdaへ振り分けを行う(デバイス=>クラウド)。
- ・また、Lambda側からIoTデバイスへMQTT通信を行うこともある(クラウド=>デバイス)。



MQTT設計に関するホワイトペーパーがある👉

[https://d1.awsstatic.com/whitepapers/
Designing MQTT Topics for AWS IoT Core.pdf](https://d1.awsstatic.com/whitepapers/Designing_MQTT_Topics_for_AWS_IoT_Core.pdf)

Designing MQTT Topics for AWS IoT Core

May 2019



設計概要

Commands

- クラウドからデバイスに向けて何らかの操作をする場合や、双方向にコミュニケーションが必要な場合。
- Broadcastパターンと呼ばれる、一対多(一つのクラウド、多くのデバイス)のコミュニケーション。
- 例: クラウドから音声データをデバイスへ送信、再生させる

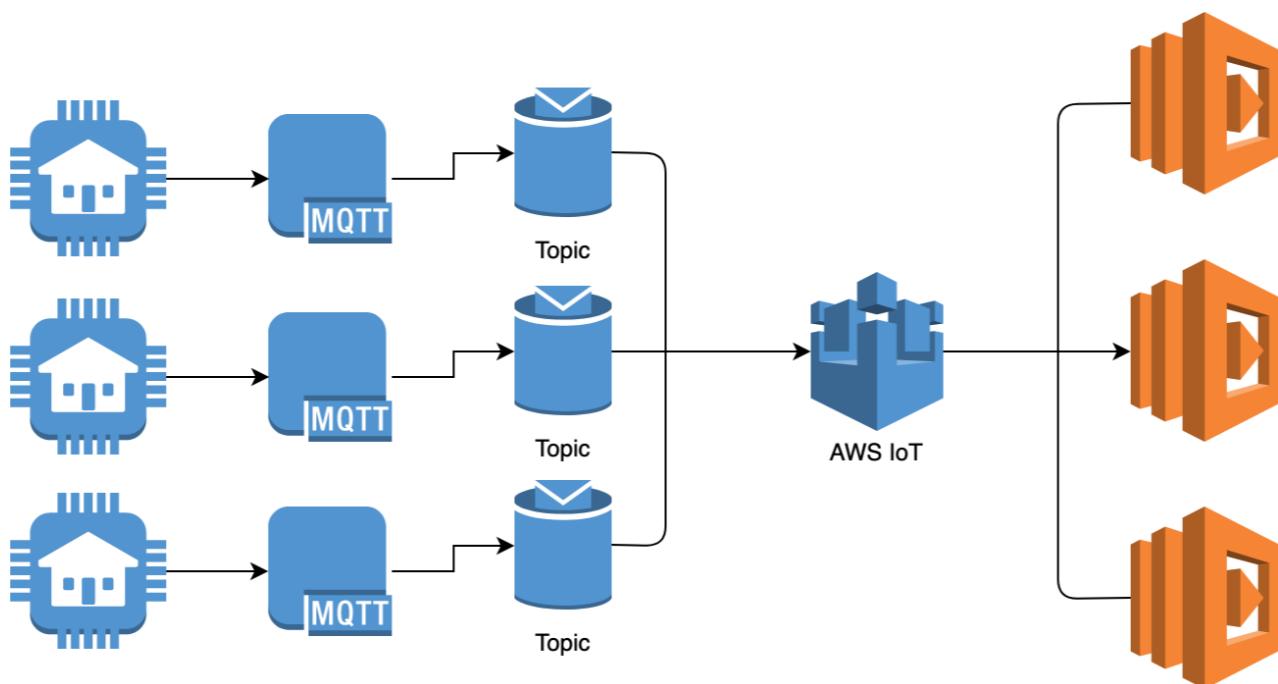
Telemetry

- デバイスからクラウドへデータを送信する場合。
- Fan-Inパターンと呼ばれる、多対一(多くのデバイス、一つのクラウド)のコミュニケーション。
- 例: 電球の色をデバイスで設定、クラウド側のシャドウに保持

Telemetryの例

Telemetryのシンタックスは
「dt/<application>/<context>/<thing-name>/<dt-type>」

対象	設定値
dt	dt (固定値)
application	speaker-app (固定値)
context	例えば detect-person , detect-fire など
thing-name	スピーカーのデバイスID
dt-type	例えば enter(帰宅時) , leave(外出時) など

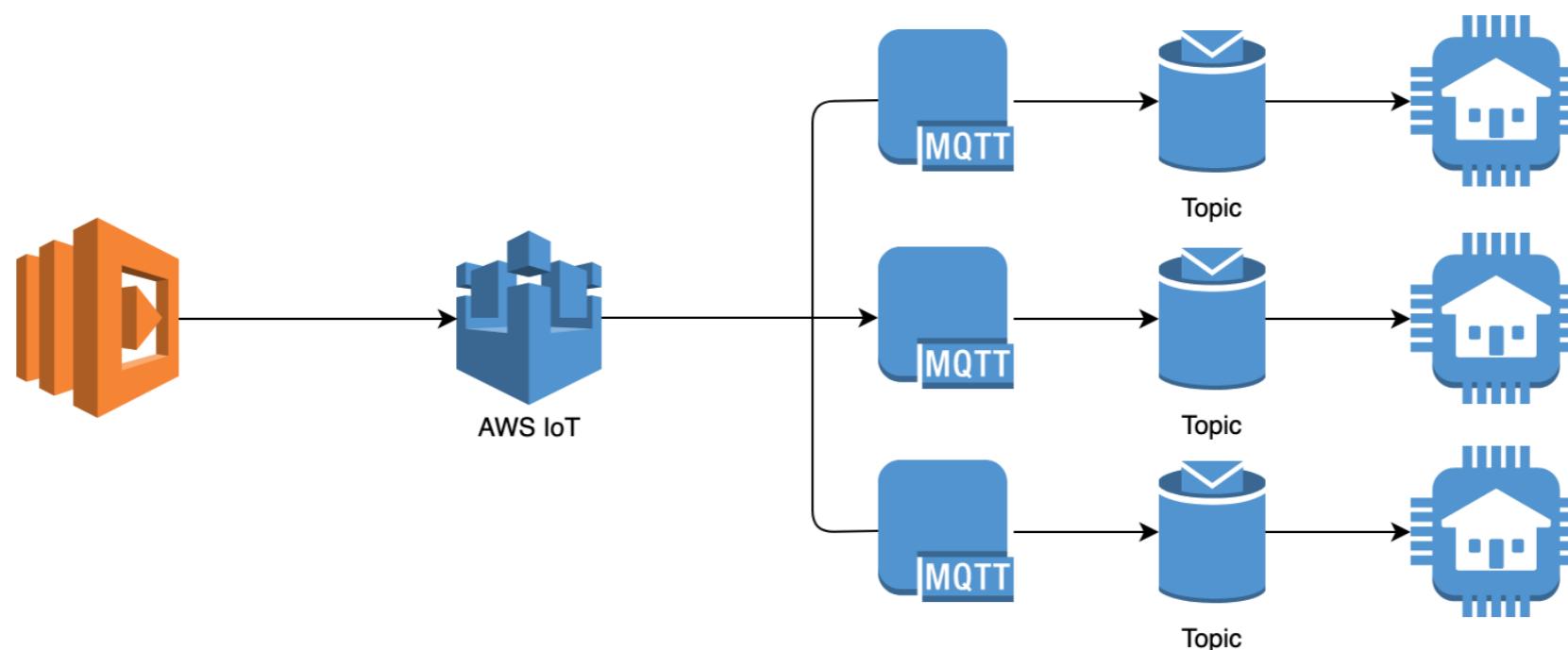


Commandsの例

Commandsのシンタックスは

「cmd/<application>/<context>/<destination-id>/<req-type>」

対象	設定値
cmd	cmd (固定値)
application	speaker-app (固定値)
context	例えば play-reminder , play-weather-forecast など
destination-id	スピーカーデバイスのID
req-type	req or res



DynamoDBトランザクション

DynamoDBでトランザクションのサポート

The screenshot shows the AWS blog header with the AWS logo and a search bar. Below the header, there are navigation links for 'Blog Home', 'Category', 'Edition', and 'Feed'.

Amazon Web Services ブログ

新機能 – DynamoDB Transactions

by AWS Japan Staff | on 28 NOV 2018 | in Amazon DynamoDB, Launch, News | Permalink | Share

Amazon DynamoDBがローンチされてから多くのユースケースで使られてきました。microservicesやゲームなどのモバイルバックエンドシステム、IoTソリューションなど様々です。例えばCapital Oneのユースケースではモバイルアプリケーションから使われていたメインフレームの処理をサーバレスアーキテクチャに移行しレイテンシの軽減にもなりました。TinderはDynamoDBにゼロダウンタイムで移行し、世界中のユーザーのために必要なスケーラビリティを獲得しました。

開発者の多くはビジネスロジックの実装に複数のItemを操作し、all-or-nothingな結果を一つか複数のtableにまたがって行いたいと思う場面があると思います。要件によっては実装に不必要的複雑さが加わることがあります。今日、我々はこのユースケースに対応するためにDynamoDBにtransaction機能をネイティブサポートしました！

Amazon DynamoDB Transactionsについて

DynamoDB transactionsは開発者に原子性、一貫性、分離性、永続性（ACID）を保証した操作を一つか複数のテーブルに対して提供します。一つのAWSアカウントの単一リージョンで行います。アプリケーションからinsert、delete、updateを複数のアイテムに対して実施出来る事でビジネスロジックの操作を一回のリクエストで実行出来ます。DynamoDBは、複数のパーティショニングとテーブル間でトランザクションをサポートする唯一の非リレーションナルDBです。

TransactionsはDynamoDBを利用してより広いワークロードに対してエンタープライズレベルでのbenefitとスケール、パフォーマンスをもたらします。多くのユースケースではすぐに簡単にtransactionsを利用することが可能です。例えば

- 金融取引
- 商品の受注から決済までの管理
- マルチプレイヤーでの操作を提供するゲーム

一般的なRDBのトランザクションとは異なる制限がある

DynamoDBのトランザクションについてFAQ形式で答えてみる

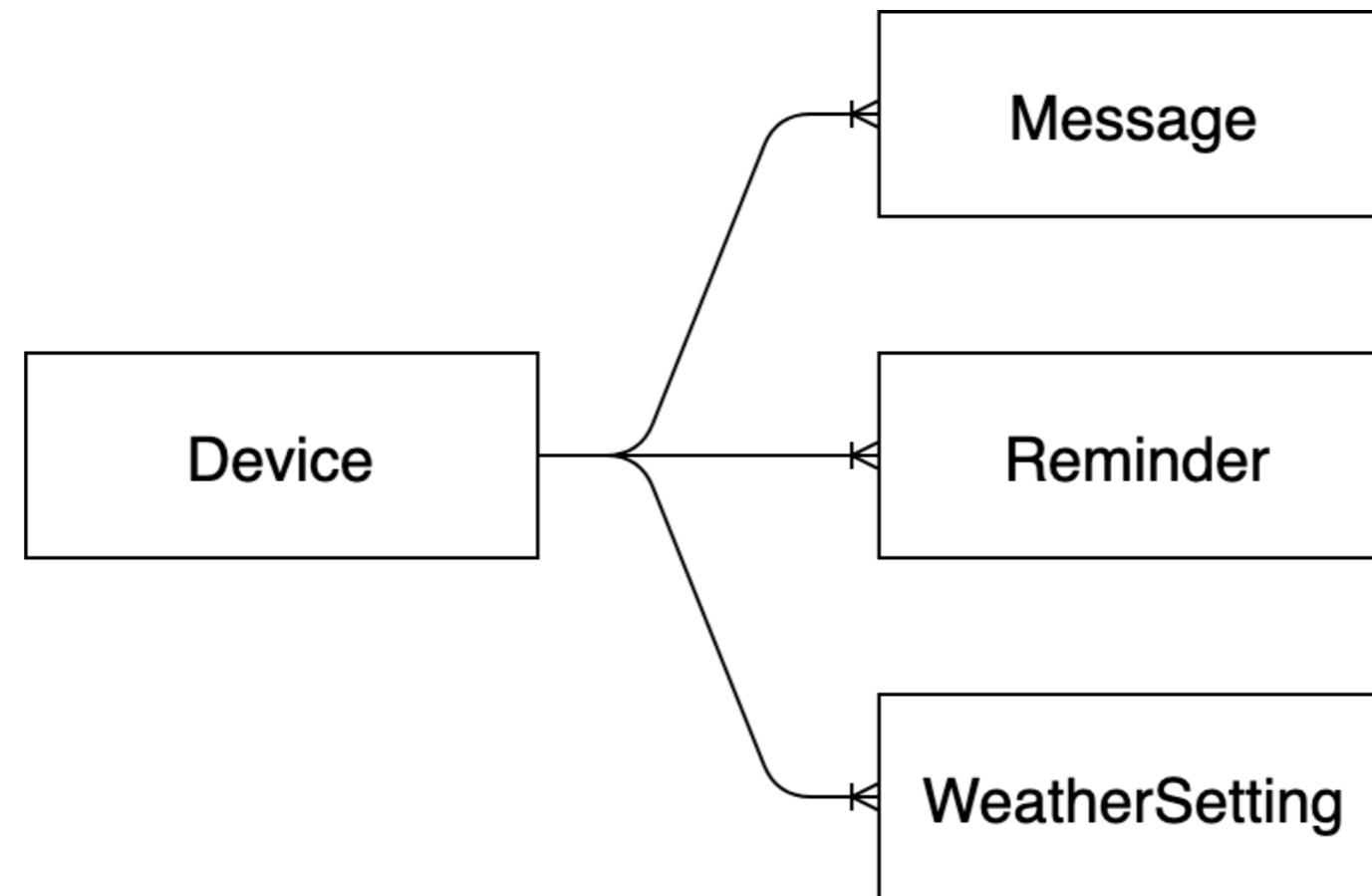
<https://dev.classmethod.jp/cloud/aws/dynamodb-transactions-faq/>

中でも

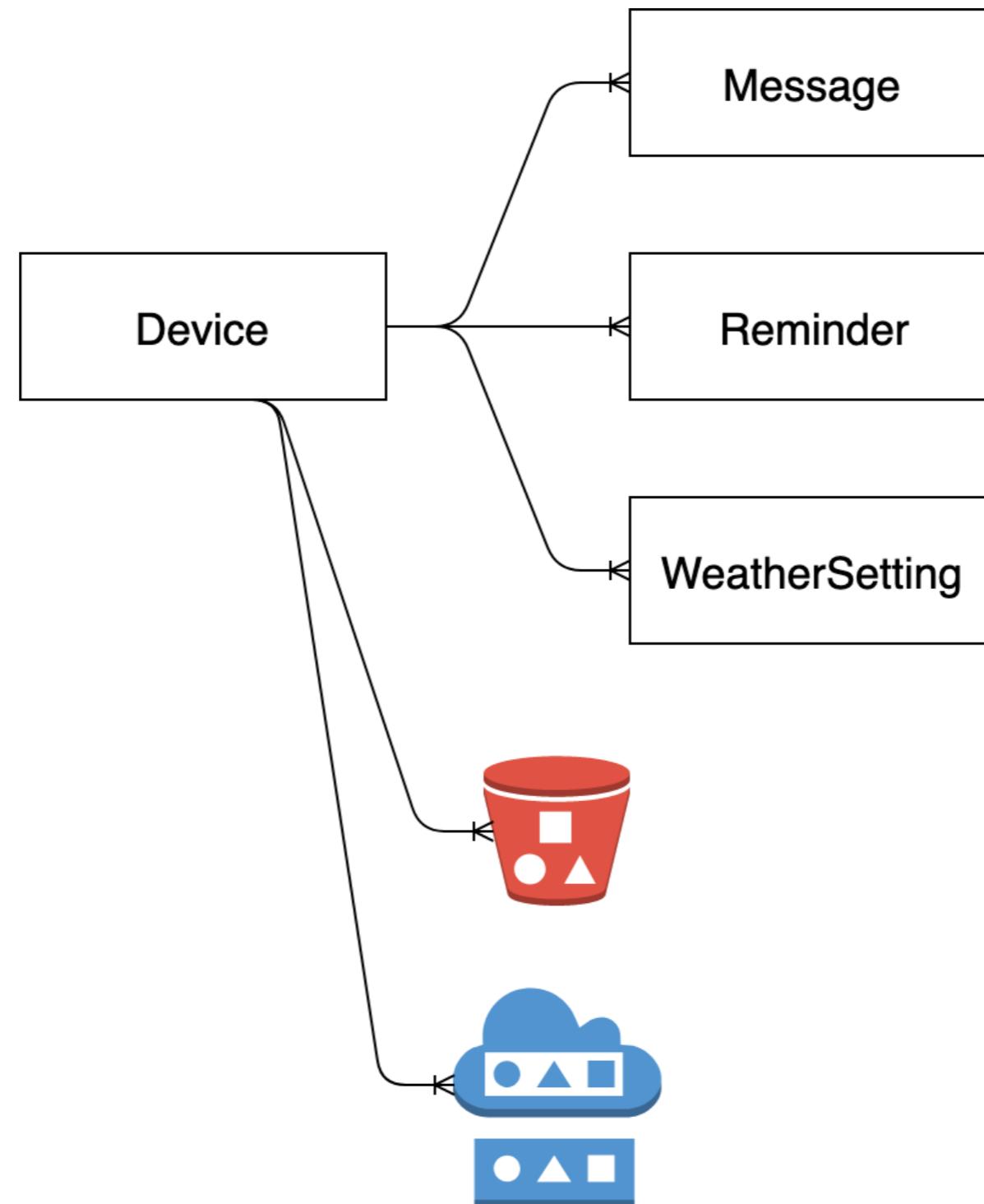
「同時に操作できるItemは10件まで」

=>この制限は意識しないと簡単に超過してしまう

関連するレコードを削除しておきたい



DynamoDB以外のサービスの更新も必要



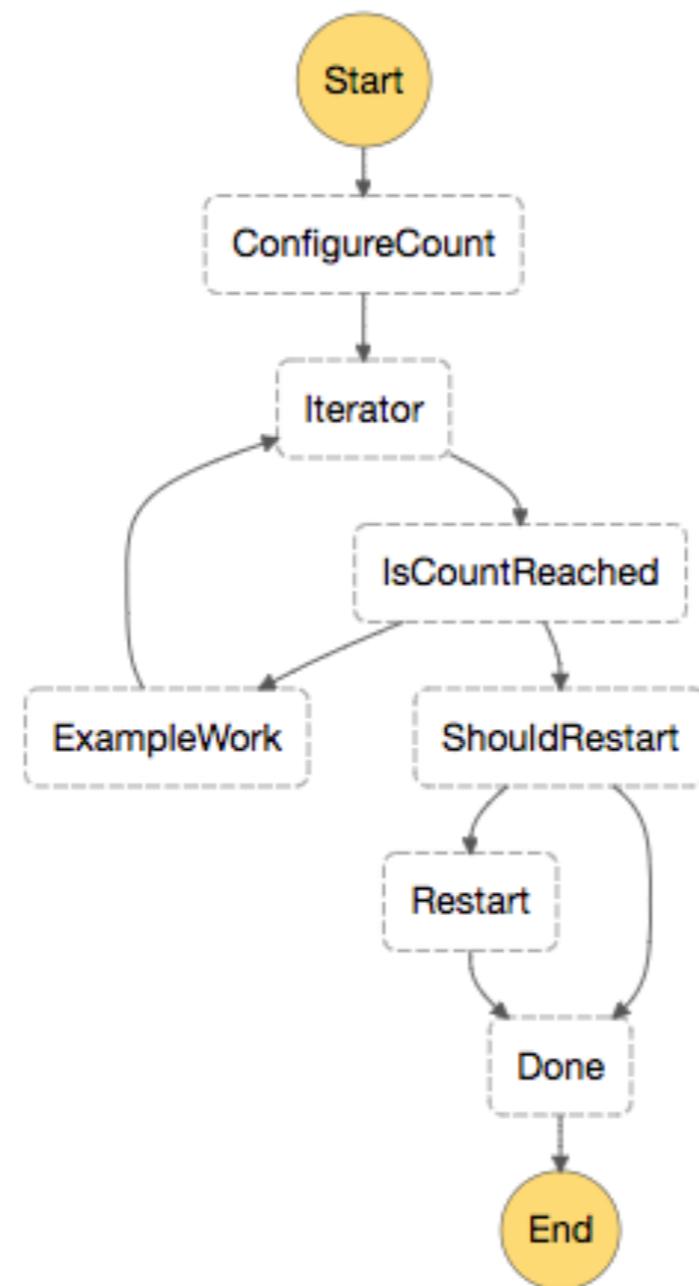
=>レコードはゆうに10件を超える
=>実行時間も、APIとして許容できなくなってしまう

そもそも、結果整合性の考え方従うと、
疎結合に更新するべではないか？

=>AWS Step Functions

Step Functionsとは？

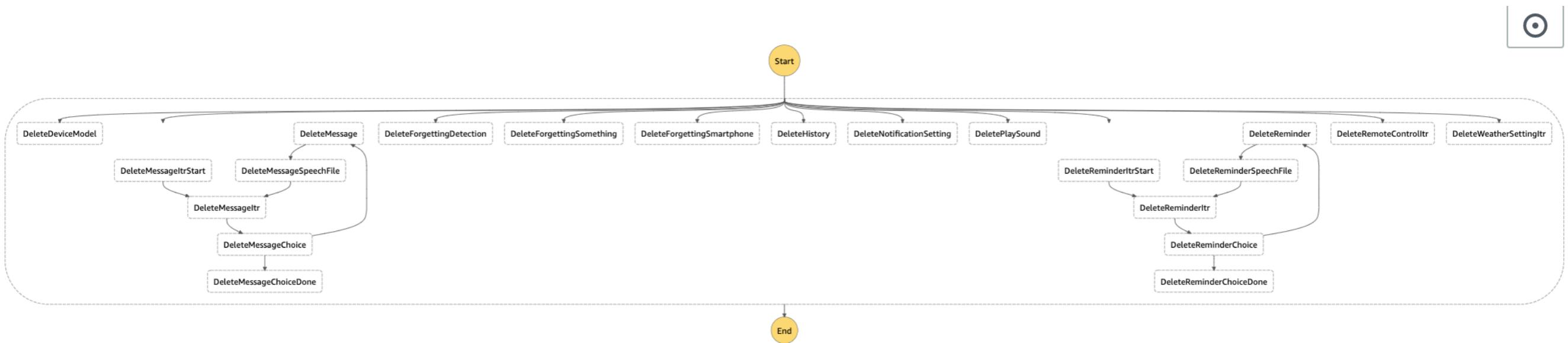
複数のLambdaを組み合わせて、連動的に実行することができる。



今回なにがよかったか？

- 処理を疎結合に分離できる
- 一度APIとしてのレスポンスは返したあと、Step Functionsで非同期に別の処理実行することができる
- 複数のAWSサービスも非同期に処理できる
- 豊富なリトライ設定(エラー種別によるリトライ是非判別、最大試行回数、バックオフなど)
- Lambda同士の関係を視覚化

肩幅の広いStep Functionsができあがりました…🎉😷



マネージドならではの制限。仕様の合意が重要

要件調整 => AWS利用サービス選定 => 検証 => 要件確定

仕様調整 => アーキテクチャ設計 => PoC開発 => 仕様確定

このステップがそもそも必要であることをステークホルダーに理解してもらう

このステップのための期間を設ける

開発者がビジネスを把握し、このステップを効率化・省力化する

<https://speakerdeck.com/wadayusuke/the-concept-of-serverless-in-application-development?slide=79>

Lambdaの監視

Lambdaの監視

例外監視についてはDatadog+CloudWatch



- ・ ひとつのLambda関数(handler)単位で監視
- ・ AWSマネージドの性質上、サーバーエラーというよりは、基本的にアプリケーション側のエラーとなる

実際にあった怖い話…

あるときミスってしまい、Lambda関数を無限呼び出しする
ような実装を書いてしまった。

=>約10日で28億回の実行

2桁万円で収まったが、付随するサービス(X-Ray/
CloudWatchLogなど)も加わり、
けっこうな額を請求されてしまった…。

複数の要因

- Goの関数名がわかりにくく、あるサービスを呼び出すべきところで、Lambdaの実行をするような処理をしてしまっていた。
- 開発環境だったので、監視をしていなかった。
- (開発環境でも)Lambdaが無限にスケールしてしまうので気づかない。

=>Serverlessならではの監視観点

実際の対処

- 請求アラートを設定する
 - => 上限いくらをAWS側で設定できる
- 開発環境でもDatadog監視を行う
 - => 同時実行数の制限をかける
- 関数名をいい感じにする

どうにかしたいこと

- Lambda関数の結合テスト

=>開発中なこと、DynamoDBのデータを整備する工数がかかることから、1ヶ月など大きな単位でしか結合テストができていない

まとめ

- Serverlessアプリケーションの管理
 - => AWS SAMと自動生成
- DynamoDBテーブル設計
 - => ベストプラクティスに従う + DynamoDBを意識しない仕組みづくり
- DynamoDBトランザクション
 - => Step Functionsでがんばる
- MQTT Topic設計
 - => Commands / Telemetry
- Lambdaの監視
 - => クラウド破産に注意