



---

# Run Code in The Cloud

## AWS Lambda 概要

Amazon Data Service Japan K.K.  
Solutions Architect  
Keisuke Nishitani(@Keisuke69)  
2015.08.11 Update

# 自己紹介

- 西谷圭介

- @Keisuke69
- [www.facebook.com/keisuke69](https://www.facebook.com/keisuke69)

- □ール

- ソリューションアーキテクト
- Webサービス / EC / スタートアップを担当
- モバイルなどアプリ寄りなプロダクトを担当



クラウドは**新しい常識**と  
なりつつあります

この新しい常識の  
パターン  
とはどういったものか



# クラウドファーストから クラウドネイティブへ

# クラウドネイティブとは

- クラウドで提供されるサービス利用を前提に構築するシステムおよびアプリケーション
- 仮想サーバ上で1から全てを作り込むのではなく効率的にアプリケーションを実装
- ビジネスの差別化ポイントへの集中
  - 究極的にはビジネスに直結するアプリケーションの開発、管理のみ



# クラウドネイティブ時代の コンピュータサービス



# AWS Lambda



# AWS Lambda

- インフラを一切気にすることなくアプリケーションコードを実行できるコンピュータサービス
  - 実行基盤は全てAWSが管理
  - AWSサービスと連携させることで簡単にイベントドリブンのアプリケーションを実装可能
  - コード実行時間に対しての課金でありコスト効率が非常に高い

# AWS Lambda

- インフラを一切気にすることなくアプリケーションコードを実行できるコンピュータサービス
- やりたいこと  
だけに集中できる
- 実行基盤は全てAWSが管理
  - AWSサービスと連携させて簡単にコードドリブンのアプリケーションを実装可能
  - コード実行時間に対しての課金でありコスト効率が非常に高い

# AWS Lambda

- インフラを一切気にすることなくアプリケーションコードを実行できるコンピュータサービス

ビジネスロジック

だけに集中できる

- 実行基盤は全てAWSが管理
- AWSサービスと連携させて簡単にインフラを駆使したドリブンなアプリケーションを実装可能
- コード実行時間に対しての課金でありコスト効率が非常に高い

例えば、

S3のバケットに画像が保存  
されたらサムネイルイメー  
ジを用意したい



例えば、

DynamoDBに保存されるアドレス  
が全て正しい形式かチェックしたい





# AWS Lambda以前

# 解決方法は複雑



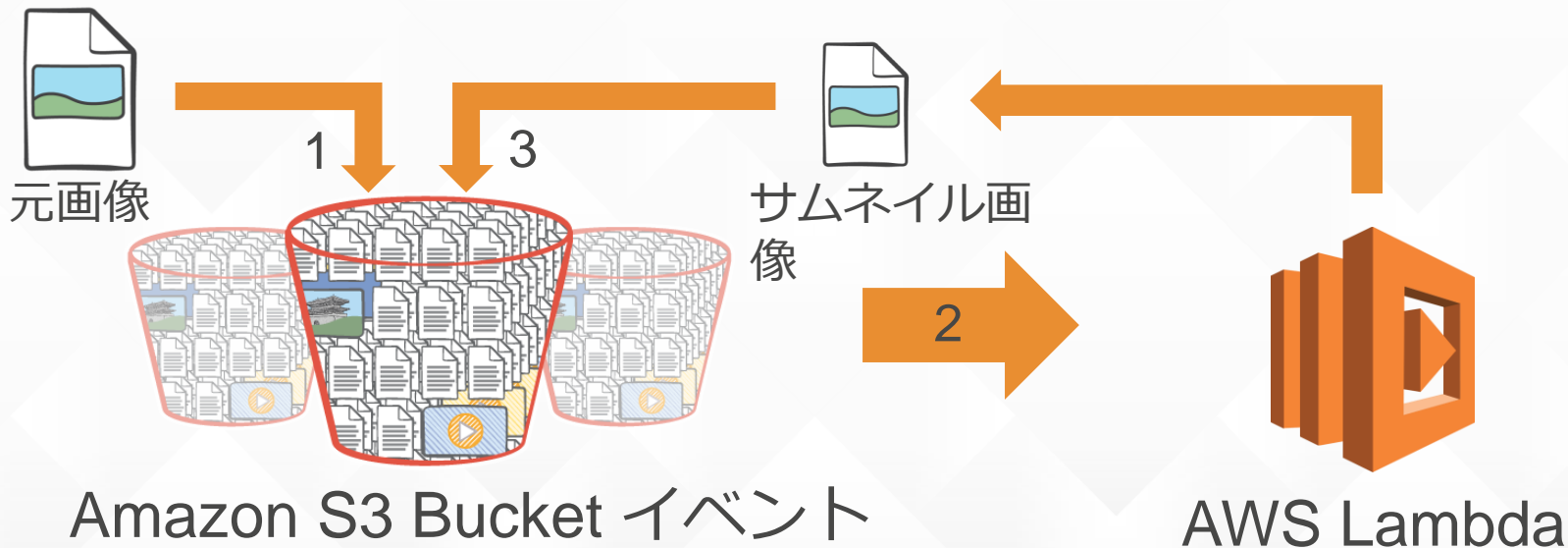
- クラウド側に問い合わせをして状態変更を検知するアプリケーションを実装
- アプリケーションを稼働させるサーバ群を用意
  - OSの設定や言語環境の構築
  - パッチ適用や更新をし続ける必要も
- 予測困難なリクエスト数に対し、スケールや耐障害性を高める仕組みを自身で構成
- キャパシティや状態、セキュリティなどを24時間365日モニタリング



# AWS Lambda以降

# サムネイルの生成やリサイズ

- S3に画像がアップロードされたときにサムネイルの生成やリサイズを実行



# 値チェックや別テーブルへのコピー

- DynamoDBへの書き込みに応じて値チェックをし、別テーブルの更新やプッシュ通知を実行



**「何をするか」**  
**を書くだけでいい**



「何をするか」  
を書くだけでいい

All you need is code.





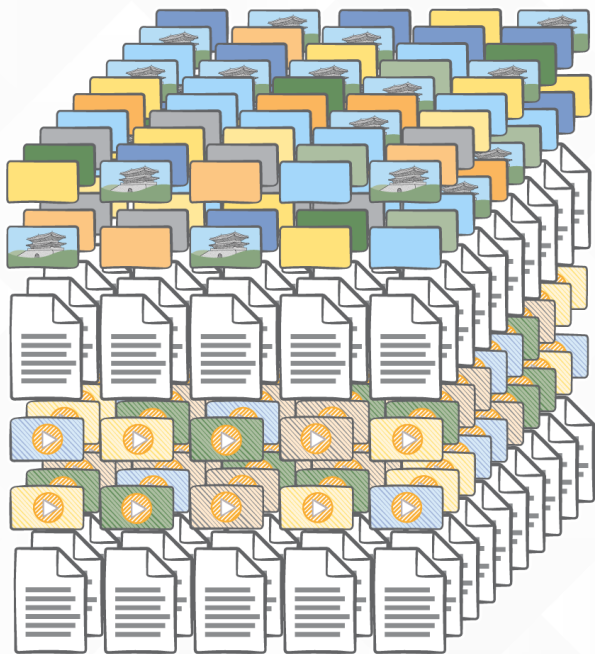
# AWS Lambdaの特徴

# インフラの管理が不要



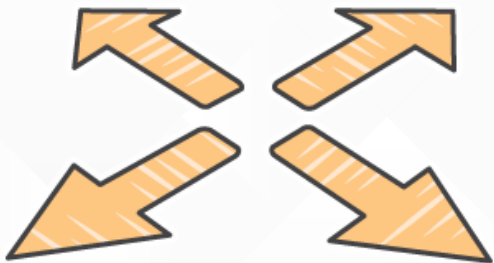
- 実行基盤は全てAWSが管理
- ビジネスロジックにフォーカスできる
- コードをアップロードするだけで、あとはAWS Lambdaが以下をハンドリング
  - キャパシティ
  - スケール
  - デプロイ
  - 耐障害性
  - モニタリング
  - ロギング
  - セキュリティパッチの適用

# オートスケール



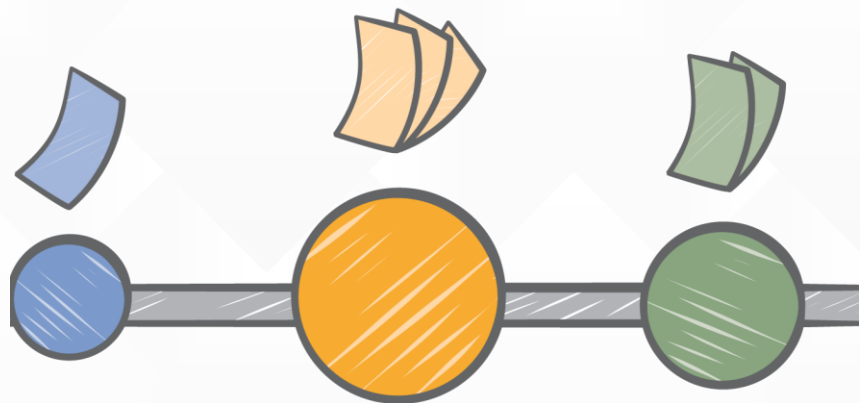
- イベントの発生頻度にあわせて自動でスケール
- プロビジョニング中や完了を気にする必要なし
- コードが稼働した分だけのお支払い

# Bring your own code



- Node.js/Javaで書かれたコードを実行
- コード内では以下も可能
  - スレッド/プロセスの生成
  - バッチスクリプトや何らかの実行ファイルの実行
  - /tmpのread/write
- 各種ライブラリも利用可能
  - ネイティブライブラリも可能
  - 利用するライブラリを一緒にアップロード

# 細やかな料金体系



- 100ミリ秒単位でのコンピュータ時間に対する価格設定
- リクエストに対する低額の課金
- 十分な無料枠
- アイドル状態は一切課金されない

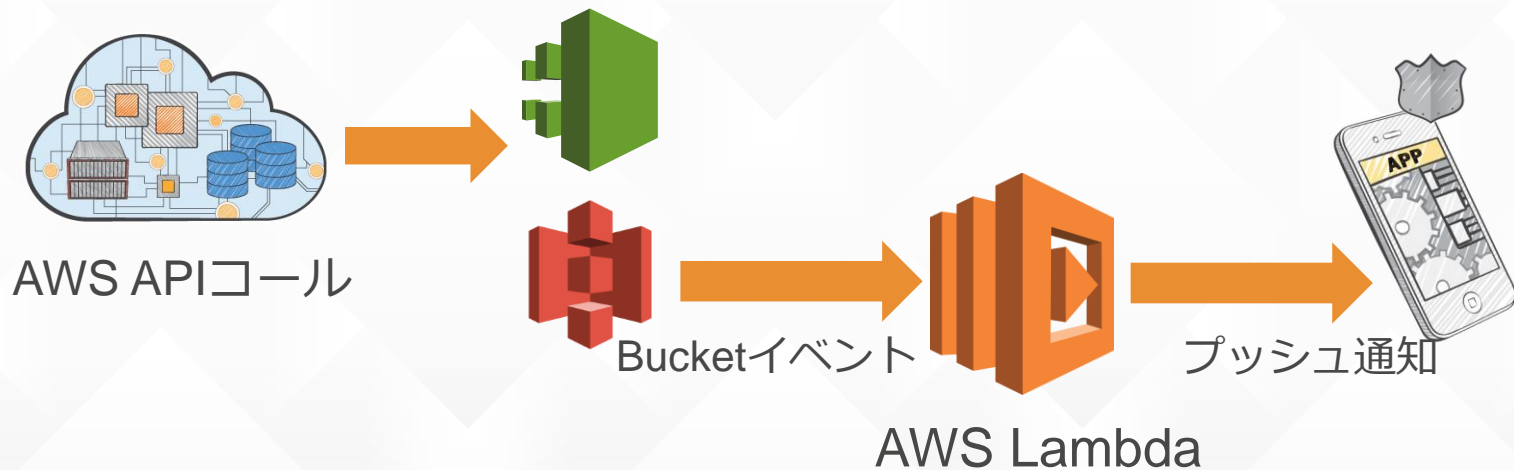


# ユースケース

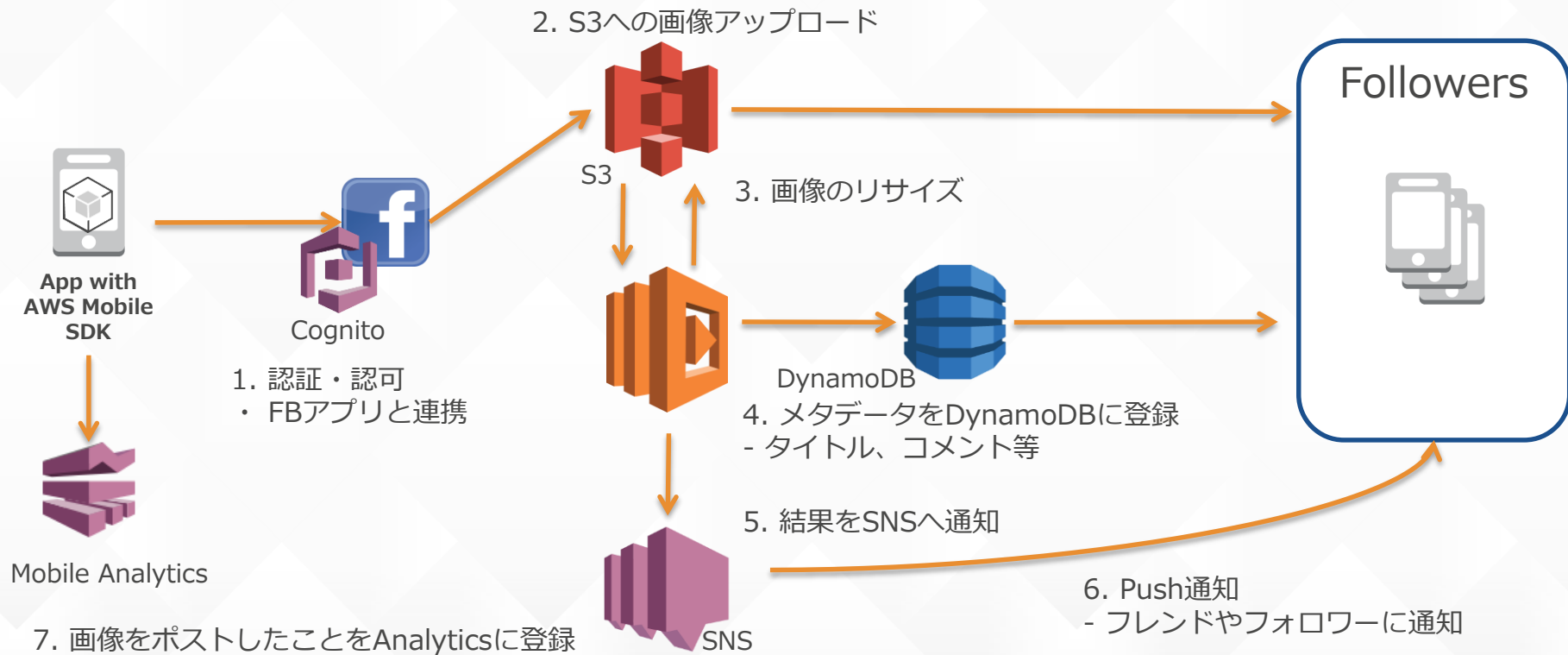
# 監査と通知

- S3に保管されるCloudTrailのログを分析し、怪しい行動や異常を検知したら通知する

AWS CloudTrail Logs



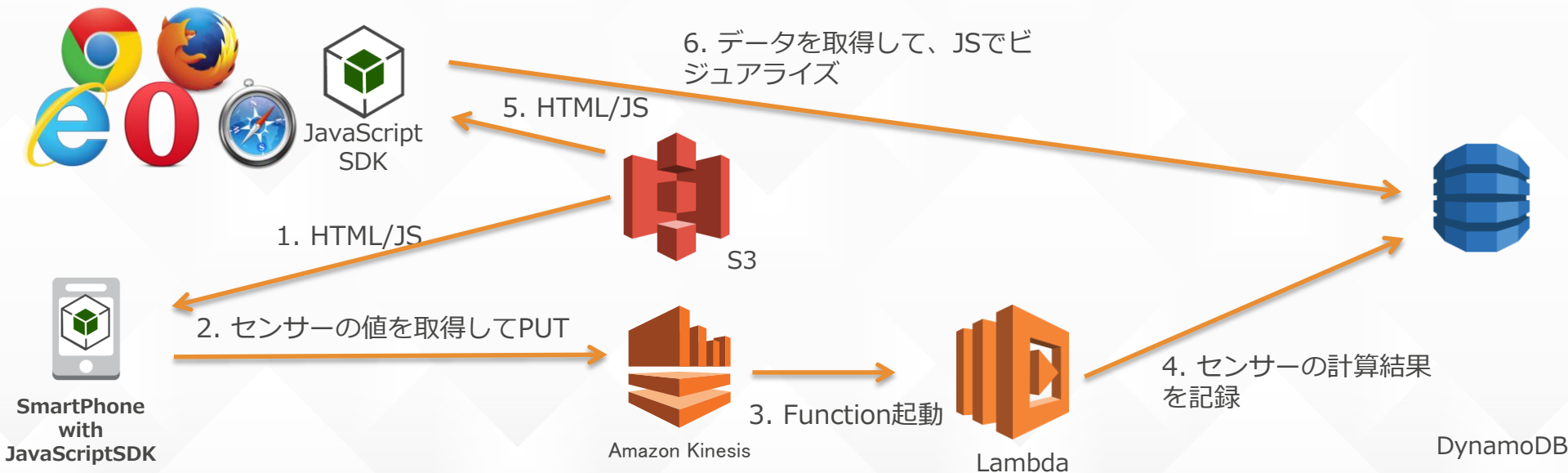
# 写真共有モバイルアプリ





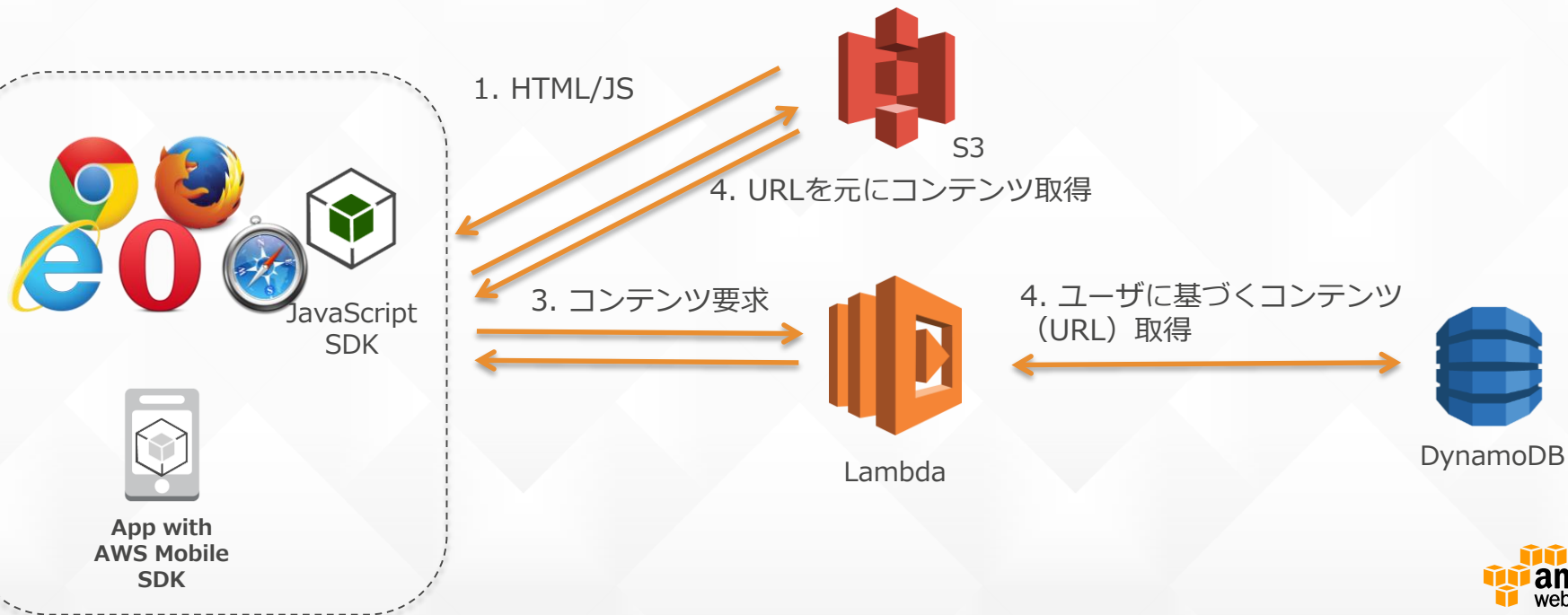
# モーションセンサーを利用した観客参加型ゲーム

- 加速度センサーの値をKinesisに流し込み、Lambdaで計算し、結果をDynamoDBへ
- PCブラウザから結果を取得してリアルタイムにビジュアライズ



# APIサーバの代わりとして

- 例えば、ユーザによってコンテンツの出し分けをしたい場合
- 同期呼び出しで実現





# AWS Lambdaの使い方

# Lambdaファンクション

- コードはJavaScript (Node.js) /Javaで記述
- メモリ容量
  - Node.jsの場合は128MB、Javaの場合は512MBがデフォルト
  - 64MBごとに128MBから1.5GBの間で設定可能
  - 容量に応じてCPU能力なども変動
- タイムアウト
  - Lambdaファンクションの実行時間に関するタイムアウト
  - Node.jsの場合は3秒、Javaの場合は15秒がデフォルト
  - 最大60秒まで
- それぞれが隔離されたコンテナ内で実行される

# イベントソース

- イベントの発生元となるAWSリソース
- サポートするAWSサービス
  - Amazon S3
  - Amazon Kinesis
  - Amazon DynamoDB Streams
  - Amazon Cognito(Sync)
  - Amazon SNS
  - Alexa Skills Kit
  - Amazon SWF

# ユーザアプリケーションからの呼び出し

- モバイルもしくはWebアプリからの呼び出しが可能
  - AWS SDKもしくはAWS Mobile SDKを利用
- 2種類の実行タイプ
  - 非同期実行
    - レスポンス内容はリクエストが正常に受け付けられたかどうかのみ
  - 同期実行
    - 実行完了時にレスポンスが返ってくる。レスポンス内容はLambdaファンクション内でセット

# 同期呼び出しコード例 (JavaScript)

```
var params = {
  IdentityPoolId: "Cognito Identity Pool ID",
};

AWS.config.region = 'us-east-1';
AWS.config.credentials = new AWS.CognitoIdentityCredentials(params);
AWS.config.credentials.get(function(err) {
  if (!err) {
    var event = {
      'key': 'value'
    };
    var params = {
      FunctionName: 'Sample',
      InvocationType: 'RequestResponse',
      LogType: 'Tail',
      Payload: JSON.stringify(event)
    };

    var lambda = new AWS.Lambda();
    lambda.invoke(params, function(err, data) {
      if (err){
        console.log(err, err.stack);
      }else{
        console.log(data.Payload);
      }
    });
  } else {
    console.log("Error:" + err);
  }
});
```

# イベント

- イベントはJSON形式でLambdaに渡される
- Lambdaファンクションはイベントごとに実行される
  - PUSHモデル: Amazon S3、Amazon Cognito、Amazon SNSとカスタムイベント
    - 順不同
    - サービスもしくはアプリケーションが直接実行
    - 3回までリトライ
  - PULLモデル: Amazon DynamoDB と Amazon Kinesis
    - 順序性あり。ストリームに入ってきた順に処理される
    - イベントソースとして登録したストリームに対してLambdaが自動的に取得しに行く
    - イベントごとに複数のレコードを取得可能
    - データが期限切れになるまでリトライ



# イベント例 (S3)

```
{
  "Records": [
    -- 省略 --
    "s3": {
      "s3SchemaVersion": "1.0",
      "configurationId": "testConfigRule",
      "bucket": {
        "name": "sourcebucket",
        "ownerIdentity": {
          "principalId": "A3NL1KOZZKExample"
        },
        "arn": "arn:aws:s3:::mybucket"
      },
      "object": {
        "key": "sourcebucket/HappyFace.jpg",
        "size": 1024,
        "eTag": "d41d8cd98f00b204e9800998ecf8427e"
      }
    }
  ]
}
```

# イベント例 (Kinesis)

```
{
  "Records": [
    {
      "awsRegion": "us-east-1",
      "sequenceNumber": "196800000000000000000374",
      "partitionKey": "2efdb0ea22685b46993e42a67302a001",
      "eventSource": "aws:kinesis",
      "data": "SOME CUSTOM DATA 1"
    },
    {
      "awsRegion": "us-east-1",
      "sequenceNumber": "196800000000000000000571",
      "partitionKey": "2efdb0ea22685b46993e42a67302a003",
      "eventSource": "aws:kinesis",
      "data": "{\"key\": \"value\"}"
    }
  ]
}
```

# パーミッションモデル

- ExecutionパーミッションとInvocationパーミッションの2種類
  - IAMロールとして用意する
- Lambdaファンクション作成時にAWS Lambda側で自動で作成することも可能
- クロスアカウントアクセスも可能

# ExecutionRole

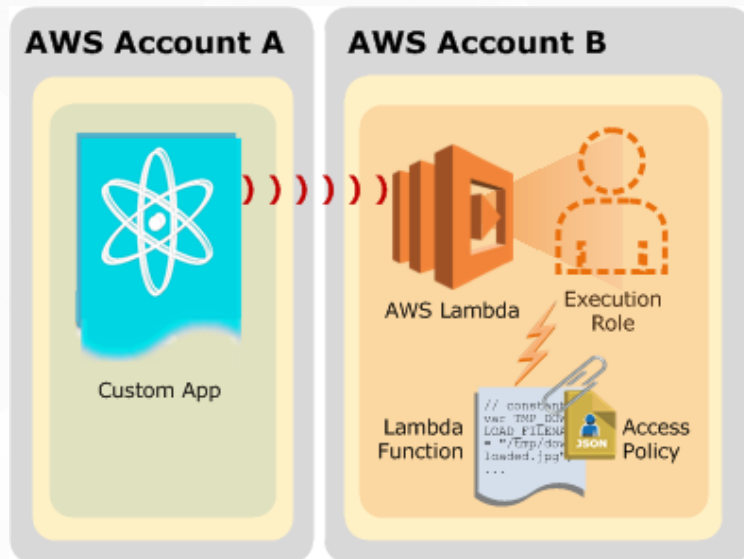
- 必要なAWSリソースへのアクセスを許可するIAMロール
- 指定されたIAMロールにそってAWSのリソースへのアクセスが許可される

# Execution Role例 (イベントソースがKinesisの場合)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:GetRecords",
        "kinesis:GetShardIterator",
        "kinesis:DescribeStream",
        "kinesis:ListStreams",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

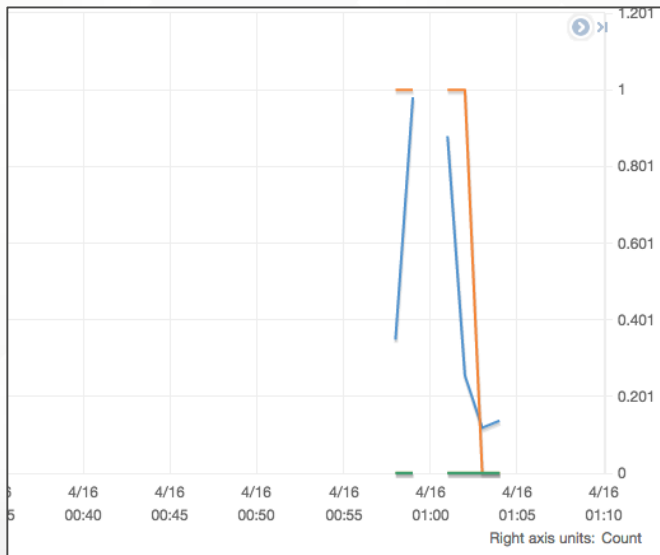
- Kinesisの場合、Pullモデルとなるため、LambdaがKinesisにポーリングできるように権限を与える
- LambdaのInvokeFunctionというアクションの許可も必要

# クロスアカウントアクセス



- 異なるAWSアカウントのアプリケーションからのアクセスが可能
- LambdaファンクションのオーナーがAccess Policyでクロスアカウントアクセスを許可する設定を追加する

# モニタリング



- **ダッシュボード**
  - 全てのLambdaファンクションのリスト
  - 可視化されたメトリクス
- **CloudWatchを用いたMetricsの監視**
  - Invocations
  - Errors
  - Duration
  - Throttle

# デバッグ

```
exports.handler =
  function(event, context) {
    console.log('Received event:');
    var bucket =
      event.Records[0].s3.bucket.name;
    var key =
      event.Records[0].s3.object.key;
    console.log('Bucket: '+bucket);
    console.log('Key: '+key);
    ...
  }
```

- 実行ログがCloudWatchに出力される
  - 各Lambdaファンクションごとのロググループ
- 実行開始/終了と消費したリソースに関するデフォルトのログエントリ
  - メモリ使用量 (Max Memory Used)
  - 実行時間 (Duration)
  - 課金対象時間 (Billed Duration)
- カスタムログエントリの追加も可能
  - ファンクション内でconsole.logを用いて出力

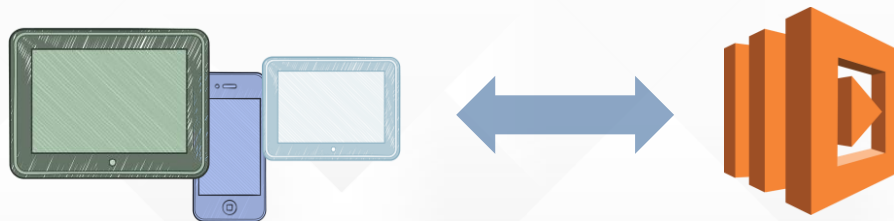


# モバイルバックエンドとしての AWS Lambda



# モバイルバックエンドとしてのAWS Lambda

- AWS Mobile SDKによるサポート
  - AWS Mobile SDK for iOS
  - AWS Mobile SDK for Android
- Lambdaファункциオンの同期呼び出し
  - 簡単・即座に利用可能でスケーラブルなバックエンドとして利用可能



# とてもシンプル

- 1 Lambdaファンクションを用意
- 2 Mobile SDKを使ってモバイルアプリからInvokeする

# 各種データへのアクセス

- モバイルSDK経由で実行すると、デバイスやアプリ、アイデンティティといった情報にアクセス可能
  - SDKが自動的に収集して設定
  - アプリ内の行動に対するレスポンスをパーソナライズすることも可能

プロパティ名	内容
awsRequestId	Lambdaファンクション呼び出しリクエストのID
logStreamName	CloudWatch LogsのLogストリーム名
clientContext	クライアントアプリおよびデバイスに関する情報
identity	Amazon CognitoのIdentity providerに関する情報

# Client Context

プロパティ名	Android	iOS
client.installation_id	初回起動時に生成されたUUID	初回起動時に生成されたUUID
client.app_version_code	Android ManifestのversionCode	CFBundleShortVersionStringの値
client.app_version_name	Android ManifestのversionName	CFBundleVersionの値
client.app_package_name	パッケージ名	CFBundleIdentifierの値
client.app_title	アプリケーションのタイトル	CFBundleDisplayNameの値
env.platform_version	Build.VERSION.RELEASE	systemVersionの値 (OSバージョン)
env.platform	Android (固定)	systemNameの値 (OS名)
env.make	Build.MANUFACTURER	apple(固定)
env.model	Build.MODEL	モデル名
env.locale	Locale.getDefault()で返ってくる値	autoupdatingCurrentLocaleの localeIdentifier



# Lambdaファンクションを モバイルアプリから呼び出す

# Lambdaファンクションを用意する

```
exports.handler = function(payload, context) {  
    console.log("Received event");  
    context.succeed("Hello " + payload.firstName + ". Your user ID is " +  
context.identity.cognitoIdentityId + " and your platform is " +  
context.clientContext.client.platform);  
};
```

- Client ContextはSDKによってデバイス上で収集され、Lambdaファンクションに渡される
  - 特定バージョンだけ処理を分岐するなども可能
- クレデンシャルプロバイダとしてAmazon Cognitoを利用している場合、そのアイデンティティも渡される



# Androidの場合



# Lambdaファンクションに対応するメソッドを定義する

```
public interface MyInterface {  
    @LambdaFunction(functionName = "hello", invocationType =  
        "RequestResponse")  
        String hello(NameInfo nameInfo);  
}
```

- Lambdaで実行したい処理をinterfaceとして定義
- 各メソッドに@LambdaFunctionというアノテーションを付与する
  - ファンクション名やInvocation Type、LogTypeはパラメータで指定
- 例では引数としてNameInfoというオブジェクトが渡されている  
(後述)

# カスタムデータタイプ

```
public class NameInfo {  
  
    String firstName;  
    String lastName;  
  
    public NameInfo(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

- Lambda関数に渡すデータをカスタムデータとして定義
  - 単なるJavaオブジェクトとして実装すればよい
- 標準ではシリアライズ/デシリアライズにLambdaJSONBinderが使われる

# LambdaInvokerFactoryの初期化

```
AWSCredentialsProvider provider = new  
CognitoCachingCredentialsProvider(myActivity, xxxxxxxxxxxxxxxxxxxx,  
Regions.US_WEST_1);
```

```
LambdaInvokerFactory factory = new LambdaInvokerFactory(myActivity,  
Regions.US_WEST_1, provider);
```

- Amazon Cognitoを使ってLambdaInvokerFactoryを初期化
  - Cognitoを使うことでテンポラリで権限の制限されたクレデンシャルが提供される
  - 利用するIdentityPoolIdを指定し、AWSCredentialProviderを初期化する

# Proxyオブジェクトの作成とLambda関クションの呼び出し

```
//Proxyオブジェクトの作成
```

```
MyInterface invoker = factory.build(MyInterface.class);
```

```
//Lambda関クションの呼び出し
```

```
String result = invoker.hello(nameInfo);
```

- 作成したInterfaceのProxyオブジェクトを作成し、それを利用して関クションを呼び出す
- ネットワークコールが発生するのでメインスレッドでは実行しないこと

# エラーハンドリング

```
try {  
    result = invoker.hello(nameInfo);  
} catch (LambdaFunctionException lfe) {  
    // Lambdaファンクション失敗時  
    Log.e(TAG, "Failed to execute echo", lfe);  
} catch (AmazonServiceException ase) {  
    // クレデンシャルの間違い、など  
} catch (AmazonClientException ace) {  
    // ネットワーク起因のエラー  
}
```

- Lambda側でエラーが発生した場合はLambdaFunctionExceptionがthrowされる
  - エラーメッセージおよび呼び出し結果を取得可能
- 不正なクレデンシャルやネットワークなどその他の理由で失敗した場合はLambdaFunctionExceptionはthrowされない

# カスタムデータバインダ

- シリアライズ/デシリアライズを独自に行いたい場合
  - 標準のLambdaJSONBinderを使いたくない場合
- LambdaDataBinderを実装し、Proxyオブジェクト作成時に指定する

# (例) カスタムデータバインダ

```
public class JacksonDataBinder implements LambdaDataBinder {
    private final ObjectMapper mapper;

    public JacksonDataBinder() {
        mapper = new ObjectMapper();
        mapper.setPropertyNamingStrategy(PropertyNamingStrategy.CAMEL_CASE_TO_LOWER_CASE_WITH_UNDERSCORES);
    }

    public T deserialize(byte[] content, Class clazz) {
        try {
            return mapper.readValue(content, clazz);
        } catch (IOException e) {
            throw new AmazonClientException("Failed to deserialize content", e);
        }
    }

    public byte[] serialize(Object object) {
        try {
            return mapper.writeValueAsBytes(object);
        } catch (IOException e) {
            throw new AmazonClientException("Failed to serialize object", e);
        }
    }
}
```

```
MyInterface myInterface = lambdaInvokerFactory.build(MyInterface.class, new JacksonDataBinder());
```



# iOSの場合 (Objective-C)



# AWSCognitoCredentialsProviderのセットアップ

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    AWSCognitoCredentialsProvider *credentialsProvider =
[[AWSCognitoCredentialsProvider alloc] initWithRegionType:AWSRegionUSEast1
identityPoolId:@"YourCognitoIdentityPoolId"];
    AWSServiceConfiguration *configuration = [[AWSServiceConfiguration alloc]
initWithRegion:AWSRegionUSWest2
credentialsProvider:credentialsProvider];
    AWSServiceManager.defaultServiceManager.defaultServiceConfiguration =
configuration;

    return YES;
}
```

# Lambdaファンクションの呼び出し

```
AWSLambdaInvoker *lambdaInvoker = [AWSLambdaInvoker defaultLambdaInvoker];  
[[lambdaInvoker invokeFunction:@"hello"  
    JSONObject:@{@"firstname" : @"YourFirstName",  
                  @"lastname" : @"YourLastName"}  
    ] continueWithBlock:^(BFTask *task)  
{  
    if (task.result) {  
        NSLog(@"Result: %@", task.result);  
        NSString *result = task.result;  
    }  
    return nil;  
}];
```

- invokeFunctionで同期呼び出しを行う
  - パラメータでファンクション名を指定
- パラメータとしてJSONObjectが指定された場合、シリアライズされて送信される
  - LambdaはJSON形式のレスポンスを返し、JSONObjectにデシリアライズされる

# エラーハンドリング

```
if (task.error) {  
    NSLog(@"Error: %@", task.error);  
    NSLog(@"Function error: %@", task.error.userInfo[AWSLambdaInvokerFunctionErrorKey]);  
}
```

- Lambdaの実行に失敗した場合、NSErrorが返る
  - ドメインはAWSLambdaErrorDomain
  - エラーコードは以下の4種類から失敗内容に応じてセットされる
    - AWSLambdaErrorUnknown / AWSLambdaErrorService
    - AWSLambdaErrorResourceNotFound / AWSLambdaErrorInvalidParameterValue
- Lambdaファンクションの実行に失敗した場合
  - ドメインはAWSLambdaInvokerErrorDomain
  - エラーコードは AWSLambdaInvokerErrorTypeFunctionError
  - userInfoにはAWSLambdaInvokerFunctionErrorKeyというキーでLambdaファンクションが返すエラーが含まれる



# Lambdaファンクションの書き方

# 基本

- 各言語のベストプラクティスに従う
  - AWS SDK、ImageMagickは組み込み済みで使えるようになっている
- ステートレス
  - データを永続化するためにはS3、DynamoDBもしくはその他のインターネット経由で利用可能なストレージを利用すること
  - 実際に実行されるサーバは毎回異なり、ログインもできない
  - /tmpへのread/writeは可能だがあくまでも一時的な用途として使用すること
- その他
  - プロセス、スレッド、ソケットを利用可能
  - 各種ライブラリを利用可能
  - インバウンドのソケット接続は不可能

# Lambdaファンクションへの登録

- Zipファイル化してアップロード
  - Javaの場合はJARファイルにも対応
  - Node.js/Javaともに外部ライブラリを含めてパッケージングすること
  - S3にアップロードしてそれを指定することも可能
- Node.jsの場合はManagement Console上のコードエディタも利用可能
  - サンプルテンプレートも用意

## CLIでの実行例

```
aws lambda update-function-code --function-name sample --zip-file  
fileb:///path/to/zipfile/function.zip
```



# Node.jsの場合

# プログラミングモデル

```
exports.handler_name = function(event, context) {  
  console.log("value1 = " + event.key1);  
  console.log("value2 = " + event.key2);  
  ...  
  context.done(null, "some message");  
}
```

- handler\_nameがLambdaが呼び出す関数名となる
  - ファンクション作成時ならびにコードのアップロード時に指定する
- イベントのデータをパラメータとして渡す
  - JSON形式で内容はサービスごとに異なる
  - カスタムイベントの場合はInvoke時に任意のJSON形式のデータを渡す
- 終了時にはcontextを用いていずれかの終了メソッドを呼び出す（後述）



# contextオブジェクト

- Lambda関数の実行環境に関する情報と関数終了するためのメソッドを提供
- context.succeed(Object result)
  - 関数とコールバックを正常終了とする場合
  - 関数の実行結果をJSON.stringifyコンパチの形式でresultとして渡す (オプション)
  - InvocationタイプがEventの場合、CloudWatch LogsのLogストリームにresultをメッセージ出力
  - InvocationタイプがRequestResponseの場合、resultがレスポンスボディとしてセットされる。同時にLogストリームにも出力
- context.fail (Object error)
  - 関数とコールバックの実行結果をエラーとする場合
  - 実行結果をerrorとして渡すことができる (オプション)
  - Errorがnull以外の場合、レスポンスヘッダ (X-Amz-Function-Error-Message) およびレスポンスボディ (errorMessage) としてセットされ、Logストリームにも出力される

# contextオブジェクト

- context.done (String message, Object result)
  - フังก์ションを終了させる場合
  - エラーのサマリをmessageとして渡す (オプション)
  - 実行結果をresultとしてJSON.stringifyコンパチ形式で渡す (オプション)
  - messageに値が入っている場合、Lambdaはエラーとみなす
  - InvocationタイプがEventの場合、自動的にmessageとresultをCloudWatch LogsのLogストリームに出力する
  - InvocationタイプがRequestResponseの場合、以下の挙動となる
    - Messageがnullの場合、resultの文字列表現がレスポンスボディとしてセットされる
    - Messageがnullでない場合、resultの文字列表現をX-Amz-Function-Error-Messageというレスポンスヘッダとしてセットし、レスポンスボディとしてerrorがセットされる
    - 引数としてmessage飲みであった場合、その値がレスポンスヘッダ (X-Amz-Function-Error-Message) およびレスポンスボディとしてセットされる

# contextのプロパティ

- `awsRequestId`
  - Lambdaファンクション呼び出しリクエストのID
- `logStreamName`
  - CloudWatch LogsのLogストリーム名
- `clientContext`
  - モバイルSDK経由で呼び出された場合、クライアントアプリおよびデバイスに関する情報
  - Nullになる場合もあり
- `Identity`
  - MobileSDK経由で呼び出された場合、Amazon CognitoのIdentity providerに関する情報
  - Nullになる場合もあり

# コード例 (Node.js)

```
console.log('Loading function');
var aws = require('aws-sdk');
var s3 = new aws.S3({apiVersion: '2006-03-01'});

exports.handler = function(event, context) {
  console.log('Received event:', JSON.stringify(event, null, 2));
  //渡されたイベントの情報からオブジェクトを取得しContentTypeを出力
  var bucket = event.Records[0].s3.bucket.name;
  var key = event.Records[0].s3.object.key;
  s3.getObject({Bucket: bucket, Key: key}, function(err, data) {
    if (err) {
      console.log("Error getting object " + key + " from bucket " + bucket +
        ". Make sure they exist and your bucket is in the same region as this function.");
      context.fail('Error', "Error getting file: " + err);
    } else {
      console.log('CONTENT TYPE:', data.ContentType);
      context.succeed();
    }
  });
};
```



# Javaの場合

## 2種類のライブラリを利用可能

- aws-lambda-java-core
  - Contextオブジェクトや事前定義済みのインターフェースを提供
  - Lambda関クションのハンドラを定義する際に利用する
- aws-lambda-java-events
  - 各イベントソースに対するイベントの型を提供
  - 自分でシリアライゼーション処理を用意する必要なし
- 必須ではないが利用したほうが便利

# プログラミングモデル

```
public outputType handler-name(inputType input, Context context) {  
    ...  
}
```

- **inputType**
  - イベントソースから発行されたイベントデータもしくは任意のデータオブジェクト
- **outputType**
  - 同期呼び出しをする際のレスポンスとなり、サポートされているあらゆる型を利用可能
  - 非同期呼び出しで利用する場合はvoidにするのが望ましい
- **input/outputで利用可能な型**
  - プリミティブ型 (intなど) およびString、Integer、Boolean、Map、List
  - aws-lambda-java-eventsで提供されるイベントデータ型 (S3Eventなど)
  - 独自のPOJOオブジェクト
    - 自動的にJSON形式にシリアライズ/デシリアライズされる
  - Stream
    - POJOを使いたくない場合、Lambdaのシリアライズ処理では要件を満たさない場合など
- **Context**
  - Lambdaファンクションの実行環境に関する情報が格納されている
  - LambdaLoggerを取得する場合などに利用
  - 省略可能

# Logging

- CloudWatch Logsに出力
- System.out()/System.err()も利用できるが LambdaLogger.log()がオススメ

```
public class Hello {  
    public String myHandler(String name, Context context) {  
        LambdaLogger logger = context.getLogger();  
        logger.log("log data from Lambda logger"); //LambdaLoggerを使ってCloudWatchに出力  
  
        System.out.println("log data from stdout"); //System.outでも出力可能  
        System.err.println("log data from stderr.");  
  
        //LogStreamをContextから取得してreturnで出力することも可能  
        return String.format("Hello %s. log stream = %s", name, context.getLogStreamName());  
    }  
}
```



# コード例 (Java)

```
package com.sample.lambda;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.event.S3EventNotification.S3EventNotificationRecord;
import com.amazonaws.services.s3.model.GetObjectRequest;
import com.amazonaws.services.s3.model.S3Object;

public class S3EventHandler {
    public String handler(S3Event event, Context context) {
        LambdaLogger lambdaLogger = context.getLogger();

        S3EventNotificationRecord record = event.getRecords().get(0);
        String bucket = record.getS3().getBucket().getName();
        String key = record.getS3().getObject().getKey();

        AmazonS3 s3 = new AmazonS3Client();
        S3Object object = s3.getObject(new GetObjectRequest(bucket, key));
        String content_type = object.getObjectMetadata().getContentType();

        lambdaLogger.log("Content-Type: " + content_type);

        return content_type;
    }
}
```

The background of the image consists of several horizontal strips of yellow caution tape. The word "CAUTION" is printed in large, bold, black letters on the tape. The "Empire" logo, which includes a stylized globe icon, is also visible on the tape. The tape is slightly out of focus, creating a sense of depth.

# Lambdaファンクション作成時の注意事項

# Lambdaファンクション作成時の注意事項

- ロールを正しくセットアップする

- Execution rolesはLambdaファンクションがアクセスするAWSリソースに対する権限を与える
- Kinesis、DynamoDBをイベントソースとする場合はLambdaがポーリングできるように権限を与える



- 再帰処理は慎重に

- イベントハンドラ内部でS3やDynamoDBへオブジェクトの書き込みを行うと、別のイベントがトリガーされ無限ループに陥る可能性がある

- Node.jsの場合、処理が非同期で実行されることを意識する

# KinesisのイベントをNode.jsで処理する場合の例

- KinesisやDynamoDBのイベントはBatchサイズを指定することで複数レコードを処理可能
- 1イベントに含まれる複数レコードに対してループ処理を行う場合、ループ内で別の非同期な処理を呼ぶとコールバックを受け取れない
  - async等のライブラリの利用が必須

# KinesisのレコードをDynamoDBへ 1件ずつ登録する例 (Node.js)

```
var AWS = require('aws-sdk');
var async = require('async');
var dynamodb = new AWS.DynamoDB();

exports.handler = function(event, context) {
  async.eachSeries(event.Records, function(v, callback){
    var encodedPayload = v.kinesis.data;
    var payload = new Buffer(encodedPayload, 'base64').toString('ascii');
    var data = JSON.parse(payload);
    var params = {
      Item: {
        someKey: { N: 'STRING_VALUE'},
      },
      TableName: 'STRING_VALUE'
    };
    dynamodb.putItem(params, function(err) {
      if (err) console.log(err, err.stack); callback();
    });
  }, function(err, result){
    if (err) {
      context.fail(err);
    } else {
      context.succeed('success');
    }
  });
}
```

- DynamoDBのputItem()は非同期のため、Array.forEach()ではコールバックを実行できず意図した結果にならない
- asyncを使うことで非同期処理の順序制御が可能





# Deep Dive into AWS Lambda





# コンテナのライフサイクルについて

# コンテナのライフサイクル

- Lambdaファンクションはコンテナとして実行される
- コンテナのライフサイクルとして開始・終了・再利用の3つのサイクルがある



# 開始

- ファンクション作成後、もしくはコードや設定更新後の初回実行時は新たにコンテナが作成され、ファンクション用のコードがコンテナ内にロードされる
- ランタイム側では、ハンドラが最初に呼び出される前に初期化コードがコンテナの生成ごとに一度だけ実行される

# 終了

- 複数の終了パターン
  - Timeout
    - ユーザが指定した実行時間を超えた場合。その時どういう処理が行われているかは関係なく、即時停止される
  - Controlled termination
    - コールバックの1つがcontext.done()を呼び出して終了した場合。この時点で他のコールバックが実行されてるかに関わらず終了する
  - Default termination
    - 全てのコールバックが終了してファンクション自体が終了した場合。この時、context.done()を実行していなくてもログに“Process exited before completing request”と出力される
- クラッシュもしくはprocess.exit()を呼び出すことでも終了
  - 例えば不具合のあるライブラリが含まれていて、セグメンテーション違反を起こした場合、そのコンテナは実行終了となる

# 再利用

- 実行からある程度時間が経過した後に再度実行される場合は新たにコンテナが作成される
  - 初回実行時と同様
- コードの変更がなく前回の実行から時間が立っていない場合は以前のコンテナを再利用することがある
  - ランタイムの初期化処理をスキップできるなどパフォーマンス上のアドバンテージ
  - 再利用された場合、最後に/tmpに書き込んだ内容も残っているがあてにはしないこと



# プロセスの凍結と再開

# プロセスの凍結と再開

- ファンクションの終了時に実行中のバックグラウンドプロセスがある場合、Lambdaはプロセスをfreezeさせ、次回ファンクションを呼び出した際に再開する
  - ただし、コンテナが再利用される場合だけであり保証はされていない
- この場合、バックグラウンドプロセスは残っていても処理は行われていない
  - プロセス再作成のオーバーヘッドを減らせる



---

# 制限事項

# 制限事項（リソース）

リソース	制限
Ephemeral disk (/tmp) の容量	512MB
ファイルディスクリプタ数	1024
プロセス数およびスレッド数(合計)	1024
アカウントあたりの同時リクエスト(*)	100
1リクエストあたりの実行時間	60秒
コードストレージの容量	1.5GB

\*秒間リクエスト数 × 1リクエストあたりの平均実行時間

# 制限事項（インプット）

インプット	制限
デプロイするパッケージのサイズ（zip/jar）	50MB
パッケージ化できるコード/依存ライブラリのサイズ （無圧縮状態のzip/jar）	250MB
Invokeのリクエストボディのpayloadサイズ	6MB
Invokeのレスポンスボディのpayloadサイズ	6MB
アップロード可能なパッケージの合計サイズ（アカウントあたり）	1.5GB





# 料金

# 料金体系

- リクエスト (全リージョン)
  - 月間100万リクエストまでは無料
  - 超過分は\$0.20/100万リクエスト (1リクエストあたり\$0.00000002)
- 実行時間 (全リージョン)
  - 100ms単位で課金
  - 100ms以下は繰り上げで計算
  - メモリー容量により単価および無料時間が異なる
- かなり複雑なのでこちらも参考に
  - <http://qiita.com/Keisuke69/items/e3f79b50b6039175401b>

Memory (MB)	Price per 100ms (\$)	Free tier seconds per month
128	0.000000208	3,200,000
192	0.000000313	2,133,333
256	0.000000417	1,600,000
320	0.000000521	1,280,000
384	0.000000625	1,066,667
448	0.000000729	914,286
512	0.000000834	800,000
576	0.000000938	711,111
640	0.000001042	640,000
704	0.000001146	581,818
768	0.00000125	533,333
832	0.000001354	492,308
896	0.000001459	457,143
960	0.000001563	426,667
1024	0.000001667	400,000



**Run Code in the cloud!**