Internet Week 2017
S7 IoTもおまかせ！　サーバーレスで変わるインフラとの関わり方

サーバーレスアーキテクチャ概論

2017-11-29 @IW2017

株式会社WHERE IoT基盤センター
仲山 昌宏 / @nekoruri

自己紹介

- 株式会社WHERE IoT基盤センター
  サービスプロデューサー（2016-）
- セキュリティ・キャンプ（2015-）
  講師・プロデューサー
- SecHack365 実施協議会（2017-）

- 技術系同人誌サークル「めもおきば」
- ProjectDIVA Arcade LV.624

本セッションについて

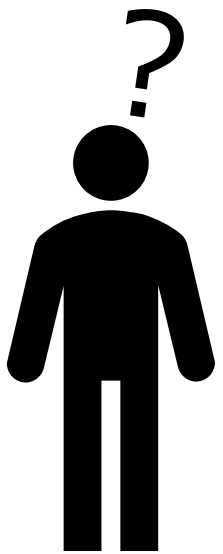
- 「サーバーレス」と呼ばれる技術・ムーブメントについて、

  1. ぶっちゃけ「何」を指しているのか
  2. エンジニアにとってどんな「変化」をもたらすのか
  3. 活用するにはどのような「視点」が必要になるのか

  これらを紹介します。

「サーバーレスアーキテクチャ」？

- 「サーバ」が「無い」アーキテクチャ
  - FAQ：「でもサーバ有るんでしょ？」
  - イメージは人それぞれ
  - いまだに明確な定義はされていない



「サーバーレスアーキテクチャ」の歴史

- 2008年　Google App Engineプレビューリリース
  サーバーレスなPaaSとして一つの完成形
- 2012年　Serverlessというテクニカルターム登場
  「Why The Future Of Software And Apps Is Serverless」
  http://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/
- 2014年　AWS Lambdaリリース
- 2015年　日本語圏で広く知られるきっかけとなる記事
  「サーバーレスアーキテクチャという技術分野についての簡単な調査」
  http://qiita.com/zerobase/items/3bc0d15980b472af841d
- 2016年　Azure Functions、Bluemix OpenWhisk正式リリース
- 2017年　Google Cloud Functionsベータ公開、各社から継続的な機能拡張

サーバーレスアーキテクチャ

- 自分で管理する「サーバ」を無くすための二つの方針

1. フルマネージドなアプリケーション実行環境を活用することで、開発や運用における「サーバ」という単位を廃する

2. クラウド上のコンポーネントをイベント駆動で結びつけて最大限活用していくシステムアーキテクチャ
（そもそも自分で用意する機能そのものを減らす）

1. フルマネージドなアプリケーション実行環境

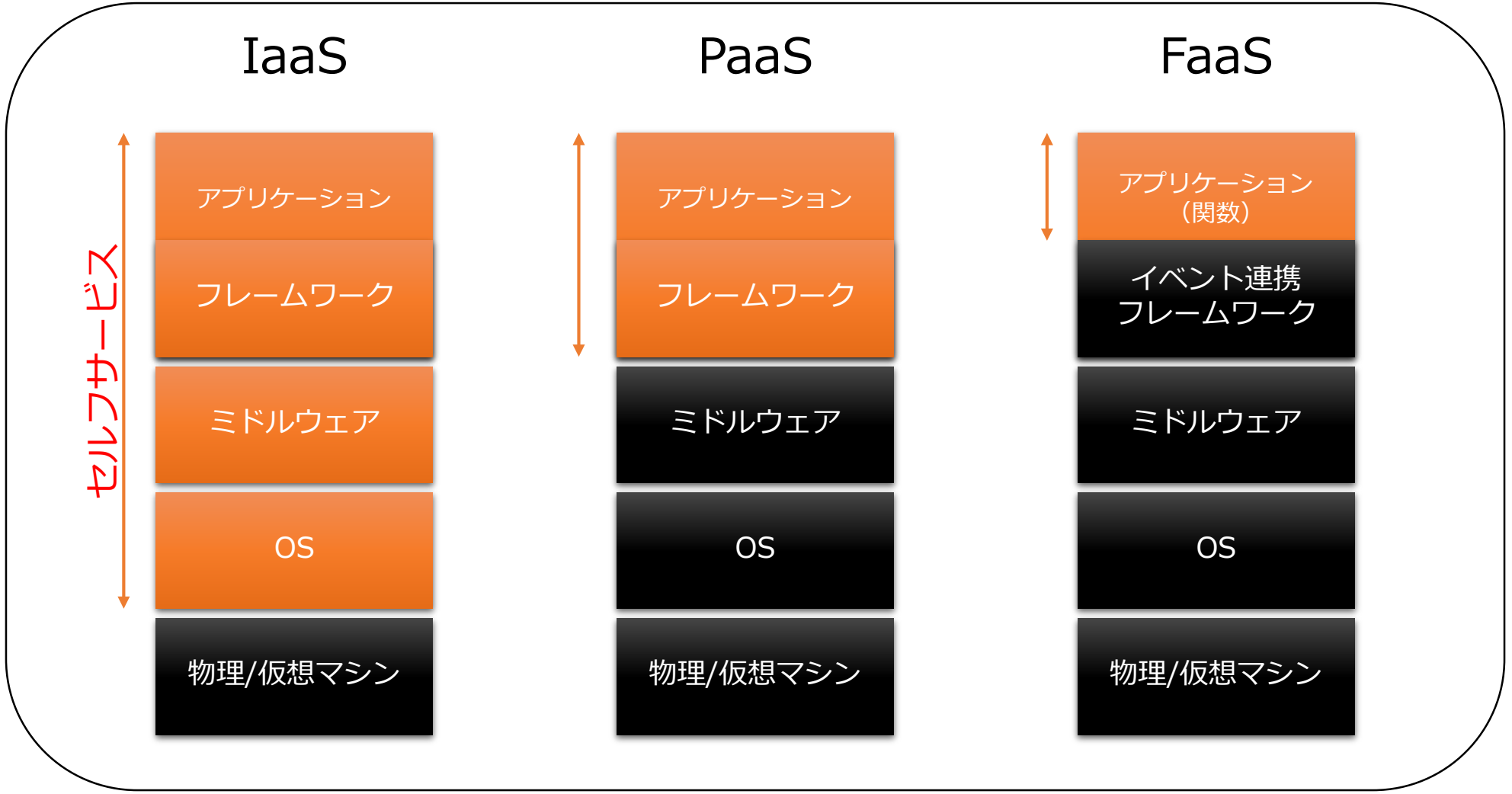
- いわゆる「FaaS（Function as a Service）」
  - 「関数」と呼ばれる小さなコードを動かすマネージドサービス
  - 様々な呼び出し方法を用意
    - HTTPリクエスト（同期呼び出し）
    - メッセージキューやストレージ等からのトリガー（非同期呼び出し）
  - 各社「サーバーレス」の中心人物

    AWS Lambda、Azure Functions

    Google Cloud Functions、Bluemix OpenWhisk

  - FaaS以外の形態もあるが今回は割愛

1. フルマネージドなアプリケーション実行環境



1. フルマネージドなアプリケーション実行環境

- 自分のコードを持ち込む（Bring Your Own Code）だけ
  ＝ 「サーバ」の面倒を自分で見なくてよい
  - 実際のマシン上にコードや依存ライブラリを展開
  - 権限に紐づくアクセスキー等の設定配布
  - コードを動かすアプリケーションプロセスの起動
  - 呼び出し元からのデータ受け取り
  - ログを外部ストレージに転送・保存
  - 需要に応じたマシンの追加・削除とロードバランシング
  - 実際の使用リソースに基づいた課金

1. フルマネージドなアプリケーション実行環境

- 「確保した量」から「使用した量」へのシフト
  - 「所有から利用」の次の段階
  - 確保した**サーバ台数（箱の大きさ）**に課金するのでは無く、
    実際に使用した**実行時間（中身の大きさ）**に課金をする

- （勝手に）クラウド側が自動でスケールさせる
  - サーバ内にファイル置いたりすると消える
    ⇒ The twelve-factor appなど、ステートレスなアプリケーション
    　のための「ベストプラクティスな制約」の普及
  - メガクラウドの物量作戦

参考：The Twelve-Factor App

I. コードベース
II. 依存関係
III. 設定
IV. バックエンドサービス
V. ビルド、リリース、実行
VI. プロセス
VII. ポートバインディング
VIII. 並行性
IX. 廃棄容易性
X. 開発/本番一致
XI. ログ
XII. 管理プロセス

1. フルマネージドなアプリケーション実行環境

- いわゆるPaaS（Heroku等）との違い
  - 明確な定義の違いは無く、スケールのしやすさで区別

https://twitter.com/adrianco/status/736553530689998848

  - 個人的には、1秒ぐらいで上がってくるならいいのでは……？

2. コンポーネントを「のり付け」するアーキテクチャ

- 高機能なクラウド上のコンポーネントの活用
  - Functional SaaS（Software as a Service）
    あるいはBaaS（Backend as a Service）
  - コンポーネント自身が高機能化し、様々な「イベント」を生成
  - イベントからFaaSを呼び出して連携

- フロント側のネイティブアプリ化/SPA化の波
  - アプリから直接データストア等にアクセスできる
  - 「ガチャ」のようなブラックボックスだけクラウド側に実装を持つ
  - アプリの一部としてクラウドとメッセージング連携

2. コンポーネントを「のり付け」するアーキテクチャ

- クラウド時代の「制御の反転」
  - アプリケーションサーバが各コンポーネントを呼び出すのではなく、各コンポーネントを小さな関数が接続する
  - システムアーキテクチャの設計手法の変化
  - マイクロサービス化、コレオグラフィ化の流れの一部

- 背景
  - 高水準なクラウド上のコンポーネントの登場
  - 様々な「イベントトリガ」の整備
  - ID基盤のうえでコンポーネント側だけで細かいアクセス認可
  - そもそも「餅は餅屋」、自分で作らなくて良い部分が増えている。

大手クラウドでのサーバーレスアーキテクチャ

- Amazon Web Services
  - AWS Lambda
- Microsoft Azure
  - Azure Functions
  - Service Fabric
- Google Cloud Platform
  - Google App Engine
  - Google Cloud Functions
- IBM Bluemix
  - OpenWhisk

AWS Lambda

- 主な特徴
  - 2014年末にリリース
  - JavaScript(Node)、Python、Java8、C#(.NET Core)に対応
  - Amazon Linuxベースのコンテナで、Goバイナリなども実行可能
  - 実行プロセスは「良い感じ」に使い回される

- 課金モデル
  - 確保メモリ×実行時間＋実行回数
  - 確保メモリは128MB～1.5GBの範囲で64MB単位であらかじめ指定
  - 実行時間は100ms単位で計測
  - （スケールに伴う起動時間などは、一定時間までは無料）

AWS Lambda

- 主なイベントソース
  - 単独ではAWS APIからのみ実行可能
  - 手動実行、定期実行
  - HTTP API：API Gateway
  - データストア：S3、DynamoDB、Cognito
  - メッセージ配信：Kinesis Streams、Simple Notification Service
  - 外部連携：Simple Email Service、Echo
  - 管理：CloudFormation、CloudWatch Logs/Events、AWS Config
- 実行権限とアクセスコントロール
  - IAM Roleによる権限管理と、リソース側での認可
  - Cognito IdentityでAWS側の一次トークンに紐付け

Azure Functions

- 主な特徴
  - 2016年3月にプレビューリリース
  - C#、JavaScript(Node)、Python、F#、PHP、BAT、Bash、Java
  - 良い感じにスケールする「動的サービスプラン」のほか、App Serviceとして確保したVM上で動かすプランも設置
  - 実行ランタイムがGitHubにてオープンソース化

- 課金モデル
  - 確保メモリ×実行時間＋実行回数
  - 確保メモリは128MB～1.5GBの範囲で128MB単位であらかじめ指定
  - 実行時間は100ms単位で計測

Azure Functions

- 主なイベントソース（入力バインド）
  - 単独でもHTTP APIとして呼び出し可能
  - タイマー
  - HTTPリクエスト（API Management）、Webhook
  - データストア：Storage BLOB、Storage テーブル、DocumentDB、Mobile Apps
  - メッセージ配信：Storage キュー、Service Bus キュー、Event Hub
- 出力バインド：関数の出力をコンポーネントに接続
- 実行権限とアクセスコントロール
  - Azure全体のRBAC準拠
  - Shared Access Signatureでリソースに直接アクセス可能

Google App Engine

- 主な特徴
  - 2008年にプレビューリリースされた元祖サーバレス
  - ……だったはずが、
    正式リリースでは、インスタンス単位課金の一般的なPaaSに……
  - 基本的にはHTTPで呼び出される普通のPaaS

Google Cloud Functions

- 主な特徴
  - 2017年3月にベータリリース
  - JavaScript(Node)が実行可能

- 課金モデル
  - 確保インスタンス単価×実行時間＋実行回数
  - インスタンスは[128MB/200MHz]から[2048MB/2.4GHz]の5タイプ
  - 実行時間は100ms単位で計測

Google Cloud Functions

- 主なイベントソース
  - HTTP リクエスト
  - データストア : Cloud Storage
  - 非同期メッセージング : Cloud Pub/Sub
  - モバイル統合 : Firebase

IBM Bluemix OpenWhisk

- 主な特徴
  - オープンソース実装とそれに基づくパブリッククラウド基盤
  - パブリッククラウドとしては2016年12月に正式リリース
  - JavaScript(Node)、Python、Swift、Dockerコンテナが実行可能

- 主なイベントソース
  - HTTP リクエスト
  - データストア：noSQL DB
  - 非同期メッセージング：Message Hub
  - タイマー

サーバーレス時代の指針：
全てを分散システムの流儀で考える

- リアクティブシステム

- ID管理とリソースへのアクセス制御

リアクティブシステム

- 分散システムのベストプラクティス
  - 素早く、かつ安定した応答時間を保つ（＝リアクティブな）システムを設計するためのベストプラクティス
  - いわばThe Twelve Factor Appのレイヤー高い版

- 4つの「特徴」を定義
  - 目的：即応性
  - 要件１：耐障害性
  - 要件２：弾力性
  - 手段：メッセージ駆動

リアクティブシステム

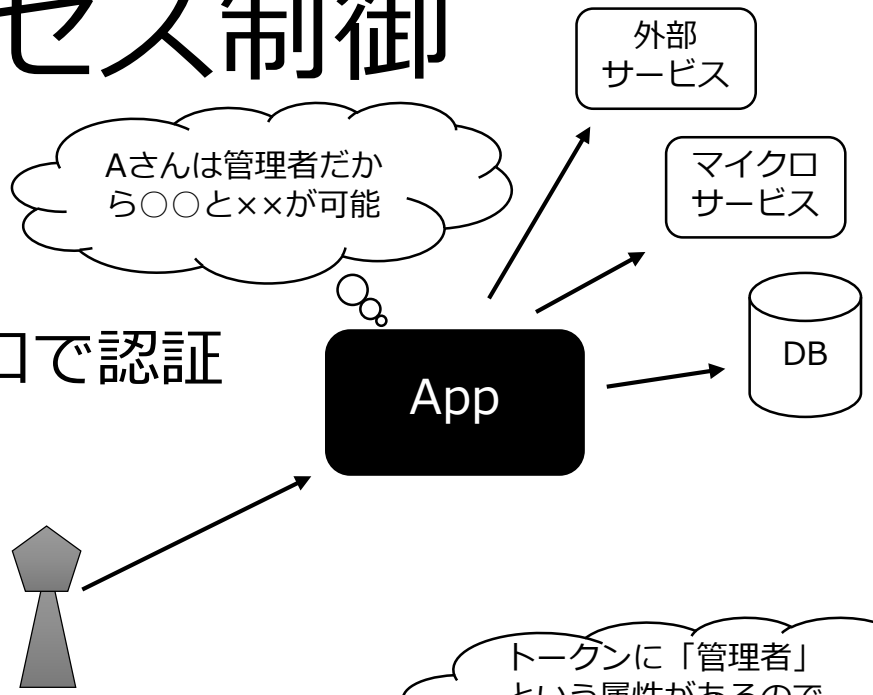
- 目的：即応性
  - システム全体として、素早く、かつ安定した応答時間を保つ
- 要件１：耐障害性
  - 障害が発生しても、それをコンポーネント内部に影響を隔離することで、システム全体としての即応性を保つ。
- 要件２：弾力性
  - 負荷の増減があっても、ボトルネックを排除し、割り当てるリソースを調整することで、即応性を保つ。
- 手段：メッセージ駆動
  - 各コンポーネント間を、非同期なメッセージ配信で疎結合に保つ。

リアクティブシステム

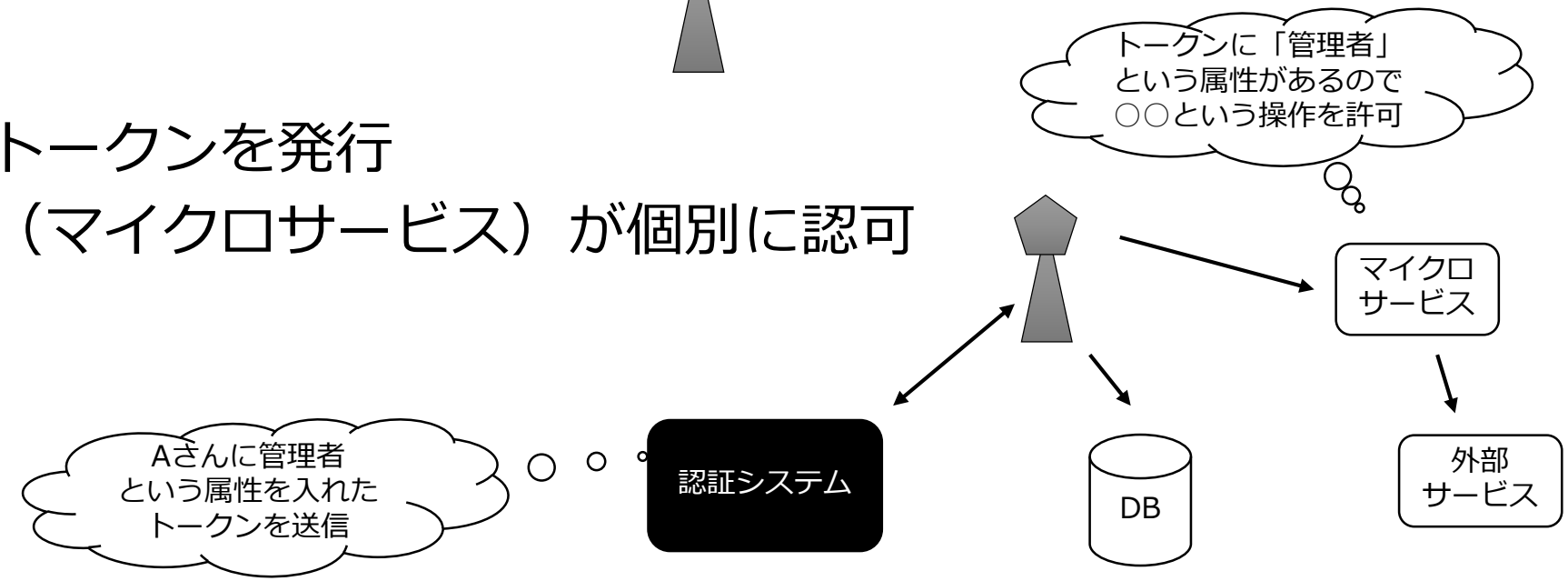
- メッセージ駆動（手段）
  - システム間をキューで非同期に接続する
  - 複数のワーカプロセスがキューから取ってきて処理
- 弾力性（要件2）
  - メッセージが増えてきたらワーカプロセスを増やせばよい
  - 横並びのワーカプロセスに相互依存はないので気軽にスケールアウト・イン
- 耐障害性（要件1）
  - コンポーネントで異常が起きたら自爆して、別のワーカが実行
  - ずっとおかしいメッセージはDead letter queueに積み替えて例外処理
- 即応性（目的）
  - 様々な状況に強いシステムが構築できる

ID管理とリソースのアクセス制御

- これまで：
  - 全ての権限を持つアプリケーションが入口で認証
  - アプリケーション内部で認可制御



- これから
  - 認証システムがトークンを発行
  - 各サブシステム（マイクロサービス）が個別に認可



ID管理とリソースのアクセス制御

- サーバレス時代のマルチクラウド連携はID連携が基本
  - IDを連携し、それに対するリソース側で細かい認可を制御
  - JWT等を引き回すことで、「誰の」「どのアプリが」アクセスしてきたのかをコンポーネント間で連携

- プログラムは「整合性」のための最小限に留まるのでは
  - 「ガチャ」
  - 「複雑なトランザクション」

クライアントアプリからの2-tier構成

- クライアントアプリを経由したID連携
  - mBaaS（CognitoやFirebase）等がすでに確立している

- そもそも高レベルのデータストア等があり、
  ID基盤のうえで、細かいアクセス制御（認可）が可能
  - 「Facebookから連携したIDがあれば、書き込みを許可」
  - 「ID=Aなら、階層Aの下の領域だけ読み書き可能」

主なパターン

- ウェブアプリ・モバイルアプリのバックエンド
  - いわゆるBaaSの発展系
  - サーバ側ロジックが不要ならデータストアに直接アクセス
- クラウド基盤のイベント処理
  - 「ピタゴラ装置」的に動画共有サイトを実現する事例など
  - AWS Lambdaは「イベント」をたくさん用意したのが勝利の鍵
  - 運用の自動化では既に多数の事例
- イベントのストリーミング処理
  - さいきんはやりのLINE Botを手軽に実現する事例など

サーバーレスアーキテクチャの今と未来

- 今
  - 基本機能は揃い始め、既に適用しやすい領域ではメリット大
  - 「小さく始める」領域では、もうデメリットは小さい
  - 管理ツールなどが見えてきた
  - テスト手法、CI/CDのやり方や開発フレームワークの整備はこれから

- 未来
  - ID管理とリソース側の認可だけでできることが増え、
    「サーバが全ての面倒を見る設計じたいが陳腐化していくのでは
  - FaaSがエッジコンピューティング側にも拡がっていく第一歩
    「次の集中から分散への波」

サーバーレス時代のエンジニア像

- 「餅は餅屋」戦略への転換が必要
  - 自分が**所有するべき「得意分野」**である技術領域は何か
  - その領域を最大化するためには何が必要か

- 所有する必要が無い技術の「放棄」
  - 時間は有限、人も有限、**選択と集中**が求められている
  - だいたいのことはクラウドベンダーの方が自分より上手くできる
    ⇒ 「クラウドベンダーのかんがえたベストプラクティス」
  - とはいえ抑えている方が「つぶしはきく」

いわゆる「サーバ」系技術の今後

- サーバの設計・構築・運用系の技術
  - 純粋に必要が無くなる領域もある
    例：ディスク障害時の迅速なベンダー連絡方法
  - 形が変わるだけの領域もある
    例：システムの監視

- プライベートクラウドの需要は当面続く
  - 自らが組織内クラウドベンダーとしてパブリッククラウドと闘う
  - 当然全てのスキルセットが必要となる

Infrastructure as Code普及期

- 準備は整っている
  - API経由で全てが制御できる
  - Hashicorp Terraformなどのツールエコシステムの整備
  - オンプレミス環境でも十分に採用可能

- もはや「手作業」が許されなくなる時代
  - サーバ環境の変更履歴を残すことの重要性が増している
  - 人間は必ずミスをする⇒手作業の余地を減らすのは義務
  - 働き方改革⇒同じ場所から作業するわけですらない

これからの付加価値

- クラウドベンダー特化の知識
  - 膨大に存在するクラウド上のコンポーネントの知識
  - 実際にその構成で「どれくらいいけるのか」という実績・ノウハウ

- 一般的なシステムアーキテクチャの知識
  - クラウドの背後に存在するシステムへの理解を踏まえた最適化
    例：DBやネットワークなど
  - 高い品質のサービスを安価にリスク少なく実現できる技術選択方針

まとめ

- ふたつのサーバーレスアーキテクチャ
  - ステートレスなソフトウェアを前提としたフルマネージドな実行環境
  - クラウドコンポーネントを活用するリアクティブなアーキテクチャ設計
  - どちらも良いシステムを導くための「良い制約」
＝クラウドが提供するベストプラクティスの活用
- 大手クラウドの提供する「サーバーレス」の違い
- サーバーレスなシステムを設計するときの指針
  - リアクティブシステム
  - ID管理とリソース側での細かい認可
- サーバーレス時代のエンジニア像と付加価値