

# Azure Container Apps の利用 💪

[Japan Azure User Group 13周年イベント](#)

by Takekazu Omi(@Baleen.Studio)

2023/9/16 v1.0.0



Takekazu Omi @Baleen.Studio



# 自己紹介

近江 武一 [@takekazuomi](#)

- 所属 [BALEEN STUDIO](#)
- [仲間募集中](#)
  - Go, Flutter など
- 13年、、、、光陰矢の如し  
[Tech Day 2010](#)



## 今日の話

Azure Container Apps は、  って話。

APIの集合体でサービスを作つて、そこにアプリ(UI)を乗せるには最適

- Azure Container Apps(以下ACA) とは
- マイクロサービスアーキテクチャとかは割りとどうでもよい 😂
- Pros and Cons

## Azure Container Apps(ACA)とは

### In short

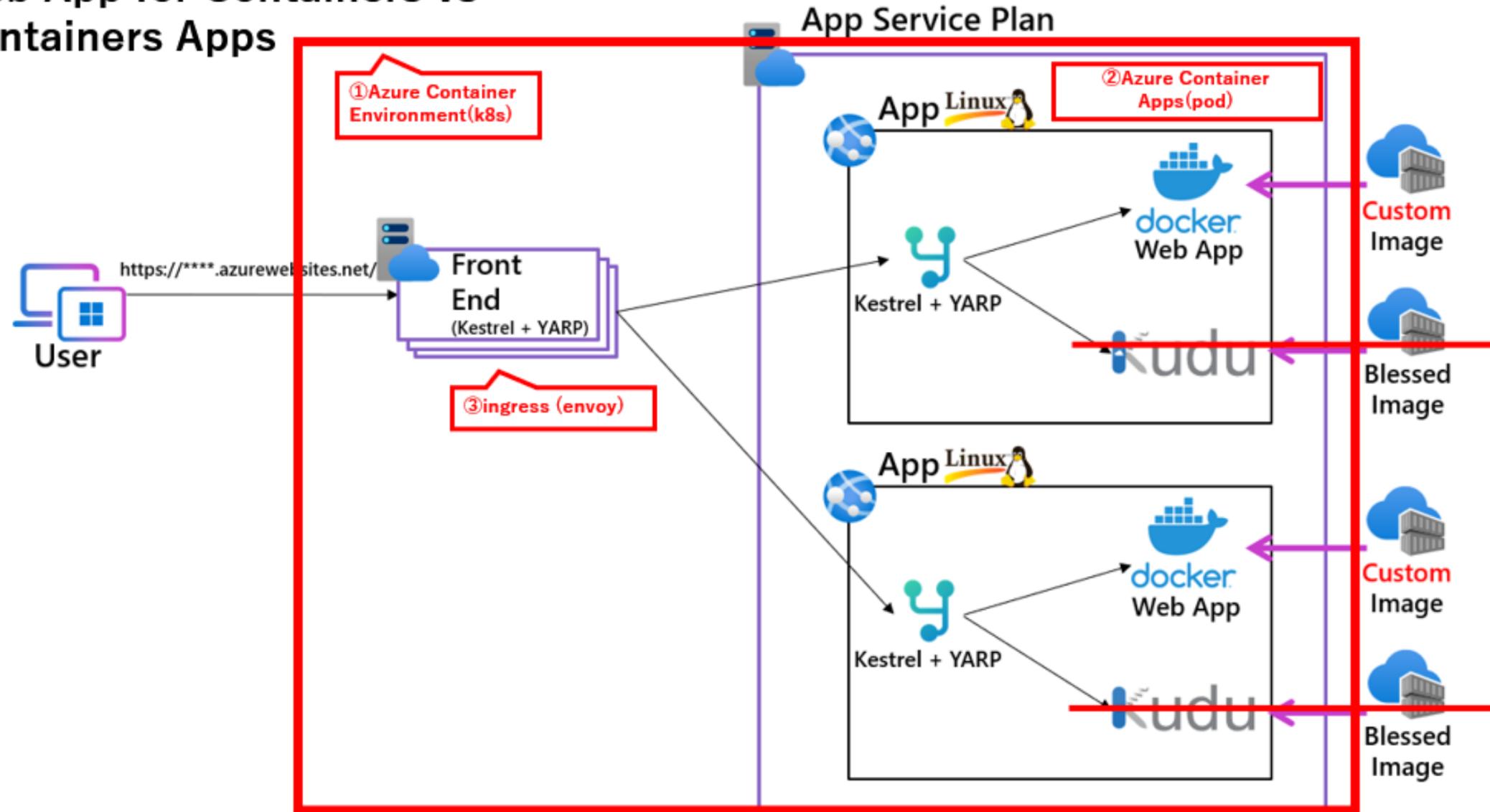
- ACA: Azure Container Environment(以下ACE)を足回りにした、マネージドなコンテナ実行環境
- ACE: Kubernetes(以下k8s)の複雑性を隠ぺいし高いスケーラビリティと可用性を実現
- マイクロサービスアーキテクチャに適した実行環境の提供

## ACA の特徴

- コンテナ向けのPaaS
  - App Service は、Webアプリ向けのPaaS
  - ACA は、コンテナ向けのPaaS
  - App Service PlanがACEで、App Service が、ACAの関係
- KEDAによる動的なスケーリング
  - TODO

※ACAはコンテナを動かすことに特化している

# Web App for Containers vs Containers Apps



元: <https://jpazpaas.github.io/blog/2023/05/10/Appservice-components.html>

# ACAとApp Serviceの構成比較

- App Service は、Web サーバークラスタ上にWebアプリを配布
  - 課金単位は、App Service Plan
  - スケール単位は、App Service Planのノード数
  - Web アプリケーション前提(=http縛り)
- ACAは、コンテナオーケストレーター上にコンテナをデプロイ
  - 課金単位は、コンテナのリソース使用量(+ httpリクエスト)
  - スケール単位は、コンテナが実行されるレプリカの数
  - コンテナ前提 (=http以外も可)

## サービス構成

- アプリは、コンテナでACAで実行
- クレデンシャルはAzure Key Vault
- Azure 内の認証はManaged Identity
- ACEは、VNet統合
- AzureリソースへはPrivateLinkでの接続
- 運用環境へのアクセスはAzure Bastion
- CIはGitHub Actions
- バックエンドの開発はGo

## コードの規模感

- 100万行程度、多分半分はテスト、コードは重複あり
- コンテナ数は、40ぐらい
- API数、grpc サービス数で約30、メソッド、rpc 300弱程度

※ バックエンドのAPI側のみ

## 開発中の雑感

- とりあえず、コードはコンテナに入れとく
  - ローカルでも、リモートでも動く
- 開発はローカルでできるようにする
  - ローカルで動かないと開発効率が悪い
  - ローカルで動かないと、テストが不便
  - エミュレータが無いAzureサービスが辛い
- Goは最高だけど
  - Azure SDKが足りん
  - go mod は、開発中のgo workと、git tagが辛い

## マイクロサービスアーキテクチャ

-  マイクロサービスっぽく作ってみた。分けるのは嫌な面も多いけど、分けてしまえば独立して作れるのは魅力的
-  分けるのは良いけど、くっつけて動くのかという話はある。プラモデル製造は難しい
-  どのように分けるのかって話は、デザインの統一性と拡張性の問題でもある
  - ここはAIPを使うことにした

# API Improvement Proposals(AIP)とは

APIの設計を統一するためのガイドライン。最重要ポイントの1つ 

- APIには型安全とクライアントコード生成が必要
-  REST API + Swagger は煩雑
-  grpc + protobuf は、簡潔、型安全
- もっと高水準のAPI設計ガイドラインが必要
  - バラバラなAPIが300あったらアプリ開発は破綻する

## AIP-121 リソース指向の設計

- API の基本的な構成要素は、個別に名前が付けられたリソース（名詞）と、それらの間に存在する関係と階層
- 少数の標準メソッド（動詞）が、最も一般的な操作のセマンティクスを提供
  - 標準メソッドが適合しない状況ではカスタマイズメソッドを利用
- ステートレス プロトコル：クライアントとサーバー間の各リクエストは独立

※ 一貫性を持たせることで利用者にとってAPIの理解が容易になる

## 書籍

### APIデザインパターン

API Improvement Proposals(AIP)の本では無いが内容は被る、API設計の本のお勧め 

# APIデザイン・パターン

JJ Geewax【著】  
松田晃一【訳】

## API Design Patterns

GoogleのAPI設計ルールに学ぶ  
Web API開発のベストプラクティス集

- Google Cloud PlatformとそのAPI設計に取り組む著者が安全かつ柔軟で再利用可能なAPIパターンを解説
- APIの一貫性、拡張性、可用性を確保する方法について
- TypeScriptで解説。他の開発者から信頼され、便利に使ってもらえるWeb APIを設計しよう

MANNING



## マイクロサービスアーキテクチャ? 😺

- 分割統治は、ソフトウェアの王道
- 分けてしまえば、バラバラに作れる
  - ピザ2枚チームのためにも分割は重要
- スレッショルドと、チームサイズ(7y ago)
  - <https://www.slideshare.net/takekazuomi/service-fabric-and-cloud-design-pattern/20>

## マイクロサービスアーキテクチャ 別視点(1) 😺

### 永続化 ✈️

- DB集約型はスケーラビリティに問題がある
- DBは結局グローバル変数でコントロールが難しい
- DBを頑張って設計するか、APIを頑張るか
- DBもシングルで行ける範囲は限りがあり、分けざる得ない
  - 分け方は垂直、水平。極限では両方必要。
  - 最初は、垂直に分けて。足りなくなったら、その中を水平。
  - 分けてしまふとトランザクションに頼れなくなる

## マイクロサービスアーキテクチャ 別視点(2) 😺

スコープを狭める ✕ ✕

- コードとデータは近くに置く
  - グローバル変数を避けて、モジュールに隠ぺいする
  - DBはグローバル変数化しやすい
  - APIはサービス固有のストレージを持つ
- [polygot persistance](#) は、ほどほどに
  - あまり異なったものを混ぜると、運用が大変に 👎
  - 今回はMySQLに統一。でもDatabaseは別

## マイクロサービスアーキテクチャ 別視点(3) 😺

コードベースを小さく保つ ✈️ ✈️ ✈️

- レイヤーの積み重ねを避け、コードベースをシンプルに保つ
- 個々のAPIの実装は、小機能で完結するようになる＝コードベースが小さくなる
- 従来のMVCモデルのような考え方でさえ、API実装には不要
  - APIの実装では、[Clean Architecture](#)のような、Controller/Use Case/Entityのような階層構造も複雑すぎる
- コードベースを小さく保ち、コードの理解を容易にすることが重要

## マイクロサービスアーキテクチャ 小まとめ

- マイクロサービスアーキテクチャかどうかはあまり重要ではない
- 分割統治、チームサイズが重要
- 分割し、IFを固定することが重要
  - IFを固定することで、独立性が向上する

総じてIFを決める部分がコスト高。とは言ってもこの手のものがあります一般的な手法に。昔のマイクロサービスアーキテクチャとは、大分違う物になりつつあるので別の言葉が欲しい。

## Pros and Cons

-  Pros
  - アプリケーションパッケージとしてのコンテナは使い勝手が良い。
    - 実行再現性が高く、余計なトラブルを減らしてくれる
  - ACAの管理性は高く、PaaSとして優良([NaaS](#)っていうらしい)
  - ACAは、大量のコンテナ(まだ2桁だけど)を動かしたときのリソースコントロールがやりやすい
-  Cons
  - 次のページへ

## Pros and Cons(2)

-  **Cons**
  - 開発に必要がスキルが幅広い。API設計、DB設計など
  - くっつけて動かすときがコスト高（結合コスト高）
  - 常時結合は、頻繁に壊れてしまう
  - GoのApplication Insights 統合が辛い（Observabilityが低い）
  - Batch系は改善が必要。ACA Jobはもっと早く欲しかった
  - 設計上の失敗は、手戻りが大きい（なんでもだけど）

## まとめ

アプリケーションをコンテナにパッケージし、オーケストレーション環境にデプロイするということが割と簡単に実現出来るようになってきた。それにより、分割統治というソフトウェア開発の王道が上位のレイヤーで出来るようになり。結果、更にワンステップ上がった規模のサービス構築が視野に入ってきたいる 😊

- 今回のコンテンツ、
  - GitHub [jazug13y-aca.md](#)
  - pdf [jazug13y-aca.pdf](#)

Powerd by [Marp](#)。ありがとうございました。

JAZUG 13周年

終

