第2章 コンテナ活用で変わる開発体験 ── Dev Container、GitHub Actions

コーディングからリリースまで、流れを体験しよう

コンテナを活用したアプリケーション開発の流れを体験してみましょう。本章では、Javaのシンプルなwebアプリケーションを題材に、コーディング、デバッグ、テストをコンテナの中で行います。加えて、GitHub Actionsを活用したCI(*Continuous Integration*、継続的インテグレーション)、リリースワークフローも解説します。GitHub Actionsでテスト、コンテナイメージのセキュリティスキャン、リリースを試しましょう。

なお、本章でサンプルコードリポジトリの取得、ワークフローやコンテナレジストリで利用するGitHubのセットアップを行います。次章以降のハンズオンにも必要な作業ですので、意識して読み進めてください。

2.1 開発環境をコンテナ化する選択肢、作成パターン

コンテナを活用した開発環境には、いくつかの選択肢、作成パターンがあります。ハンズオン環境を作る前に、その概要を説明します。どのパターンが、みなさんが利用できる端末やサービスの条件、制約に合うか、また、好みかを考えながら読んでください。

なお、本章ではVisual Studio Code(以降、VS Code)を利用します。VS Codeのほかにも、アプリケーション開発で広く使われているIDEやエディタはあります。ですが、VS Codeは後述するリモート開発やDev Containerなど、開発環境でのコンテナ活用をリードする存在と言ってよいでしょう。

現在VS Codeを利用していない場合には、ぜひ検討の機会としてください。もし利用しない、できない理由があれば、次のテーマまでは参考程度に読み流していただいてかまいません。本章の後半のテーマであるGitHub Actionsを活用したCIとリリースのワークフローは、VS Codeを前提としません。

2.1.1 » VS Codeのリモート開発機能によるUIの分離とそのメリット

まず前提知識として、VS Codeのリモート開発機能を説明します。この機能は公式ドキュメントで

「VS Code Remote Development」[注1]と表現されています。端的に言うと、VS CodeのUI(*User Interface*)部分を分離し、ほかの機能を別のOSやマシンで実行可能にするしくみです。ほかの機能とは、言語ごとの拡張機能、ターミナルプロセス、デバッガなどです。開発環境の本体と呼んでよいでしょう。

リモート開発機能には、以下のような喜びがあります。

- **エディタの操作や日本語入力環境などのUIは、端末のOSで実行(WindowsやmacOS):慣れた環境でうれしい**
- **リモート開発環境は、本番環境と同じOS(例:Linux)を、端末内に仮想マシンとして作成:本番と一貫性があってうれしい**
- **ビルドやテストに時間がかかる開発では、リソースの豊富なサーバ上にリモート開発環境を作成:早く終わってうれしい**
- **端末と開発するアプリケーションのCPUアーキテクチャが違ってもよい(amd64/arm64など):端末を何台も持たなくてよい**
- **端末の移動や別端末からの利用など、接続元のネットワークが変わっても、同じ環境で開発を継続:どこでも開発できてうれしい**

リモート、という名前が付いていますが、リモート開発環境の配置先は離れた場所に限りません。仮想化技術などを活用し、端末内にUIと開発環境を同居できます。

さて、勘のよい読者のみなさんは気付いておられるかもしれません。そうです。このリモート開発環境の作成や維持に、コンテナ技術を活かします。

2.1.2 ≫ リモートOSの選択肢

UIを動かす環境をローカルOS、開発環境の配置先をリモートOSとします。リモートOSや接続方式に応じ、いくつかの選択肢(拡張機能)があります。

- **Dev Container拡張機能**[注2]
- **Remote - SSH拡張機能**[注3]
- **WSL(*Windows Subsystem for Linux*)拡張機能**[注4]
- **GitHub Codespaces**[注5]
- **VS Code Server**[注6]

以降、「Remote - 」は省略します。

注1 https://code.visualstudio.com/docs/remote/remote-overview
注2 「Dev Containers」https://code.visualstudio.com/docs/devcontainers/containers
注3 「Remote - SSH」https://code.visualstudio.com/docs/remote/ssh
注4 「WSL」https://code.visualstudio.com/docs/remote/wsl
注5 https://code.visualstudio.com/docs/remote/codespaces
注6 https://code.visualstudio.com/docs/remote/vscode-server

まずは主役であるDev Container拡張機能を解説します。実はDev Container拡張機能のほかにも、Dev Container拡張機能と連動する拡張機能があるのですが、追って説明します。

Column 「リモート開発」という呼び方で混乱?

筆者は慣れてしまったせいか、「リモート開発」という呼び方に違和感がありません。ですが、「混乱する」という意見もあったようです。次の文章は、VS Codeバージョン1.72のリリースノートからの引用です。

> We've heard your feedback about the naming of the Remote - WSL and Remote - Containers extensions. We intended for Remote in their names to indicate you develop in a "remote" or "separate" WSL distro or development container, rather than traditional local development. However, this is a different use of "remote" than many people use, and it could lead to confusion.
> (筆者訳)Remote - WSLとRemote - Containersという名前について、みなさんからフィードバックをいただきました。私たちは従来のローカル開発ではなく、「リモート」または「分離した」WSLディストロやコンテナで開発することを示すため、それらをRemoteと名付けました。しかし、多くの人が使う「リモート」とは異なり、混乱を招いた恐れは否めません。

このような背景があり、Remote - ContainersとRemote - WSL拡張機能はそれぞれ、「Dev Container拡張機能」「WSL拡張機能」と名称変更されました。みなさんの導入した拡張機能のバージョンによっては、本書で説明する名称と表示が異なる可能性があります。適宜読み替えてください。

◆Dev Container拡張機能のしくみ

図2.1が、リモートOSとしてコンテナを選択した場合、つまりDev Container拡張機能を利用した場合のアーキテクチャです。

図2.1　VS Code - Dev Containerアーキテクチャ

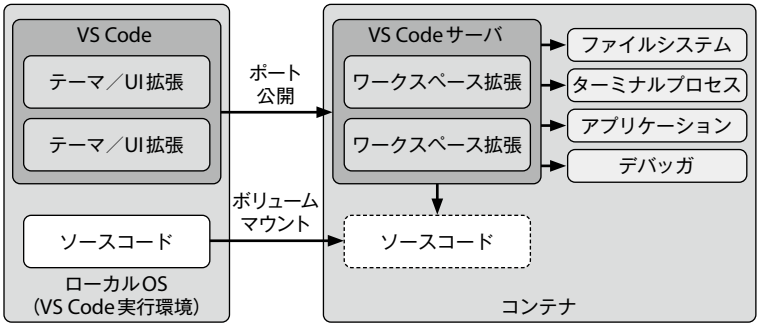


図2.1の左側がローカルOSで、主にVS CodeのUIを担当します。そして右側がVS Codeサーバの動くコンテナです。サーバはポートを公開し、ローカルOS上にあるVS CodeのUIからの接続を受け付けます。VS CodeサーバはVS Codeの拡張機能群とともに、デバッガなどコンテナ内のほかの機能と連動します。また、VS Codeサーバはローカル OSのボリュームをマウントし、ソースコードにアクセスできるようにします。

◆Dev Container拡張機能の前提条件

Dev Container拡張機能を選択する場合、前もってリモートOS側でコンテナが動く環境を準備し、ローカルOSから接続できる必要があります。ローカルOSにDocker Desktopをインストールすれば、リモートOS環境も楽に準備できます。Docker Desktopが、Hyper-VやWSL2など、ローカルOSが持つ仮想化機能を使ってリモートOS環境をセットアップするからです。

◆SSH、WSL拡張機能とDev Container拡張機能を組み合わせる

Docker Desktopを使わずに、SSH拡張機能を利用する手もあります。SSH接続できるマシンにDockerを導入しておき、いったんSSH拡張機能でつないでから、Dev Container拡張機能に切り替える、という方法です[注7]。

なおSSH拡張機能を介する場合、ローカルOSからDev Container拡張機能を使う場合と異なり、ローカルOSのファイルシステムをリモートOSはマウントしません。ローカルOSのファイルシステムを主としてソースコードを管理している場合には、考慮が必要です。リモートOS側でリポジトリから複製するなどの作業がいります。

ソースコードの管理はGitなどリモートリポジトリを主とし、開発環境のソースコードは一時的なもの、というスタイルにおいては、問題にならないでしょう。一方で、SSH拡張機能ではローカルとリモートOS間でファイル共有するオーバーヘッドがないため、良好な性能を得やすいです。多数のファイルを対象としたビルドで、効果を実感できるはずです。

またSSH拡張機能と同様に、WSL拡張機能でWSLに接続した環境から、Dev Container拡張機能へと切り替えることもできます。WSLのフォルダをDev Container拡張機能で開きなおせば、切り替わります[注8]。また、ローカルOSのWindowsからでも、共有名\\wsl$を使ってWSLのファイルシステムが見えるため、そこをDev Container拡張機能で開くこともできます。WSLを活用している開発者には、うれしい選択肢でしょう。

注7 「Open a folder on a remote SSH host in a container」https://code.visualstudio.com/docs/remote/containers#_open-a-folder-on-a-remote-ssh-host-in-a-container

注8 「Open a WSL 2 folder in a container on Windows」https://code.visualstudio.com/docs/devcontainers/containers#_open-a-wsl-2-folder-in-a-container-on-windows

Column

筆者の開発端末

筆者は、非コンテナ環境での開発や検証も行うため、SSH拡張機能を愛用しています。端末に仮想マシンを作るのは面倒なのですが、Ubuntu仮想マシンの作成と利用をサポートするMultipass[注9]を使って、ローカルOSの仮想化機能を抽象化し、作業を省力化しています。MultipassはWindows、macOSどちらにも対応しているため、ローカルOSの種類が違っても同様の手順で作成、利用できる、という利点もあります。

以下の手順で環境を作成、利用しています。

❶Windows、macOS端末にMultipassでLinux仮想マシン（Ubuntu）を作成
❷Linux仮想マシンにDockerを導入
❸Linux仮想マシンにVS Codeリモート開発機能（SSH拡張）で接続
❹SSH拡張機能を通じてVS Codeリモート開発機能（Dev Container拡張機能）を利用（`Reopen in Container`コマンド）

2.2 ハンズオン環境のセットアップとコンテンツの理解

VS Codeのリモート開発機能と選択肢を、おおよそ理解できたでしょうか。ではいよいよ、ハンズオン環境を作りましょう。

本章のサンプルアプリケーションにはJavaを採用しました。Javaは Linux、macOS、Windowsなど複数のOSをサポートする言語ですが、開発体験には言語の動作可否だけでなく、シェルやツールも影響します。本書ですべてのOSを網羅できないため、アプリケーションの実行環境、リモートOSはLinuxに絞ります。ローカルOSはVS Codeのサポート対象から、好きなものを選んでください。

なお、この記事の執筆時点で、Appleシリコンを搭載したMacとDocker Desktopの組み合わせには既知の問題がいくつかあります[注10]。また、arm64アーキテクチャに未対応なコンテナイメージも数多くあります。本章のハンズオンはAppleシリコン搭載Macでの動作を確認していますが、将来これらの問題に該当する場合には、別のプラットフォームを検討してください。Appleシリコン搭載MacをUIに、リモート開発環境を別マシンに作成してSSH拡張機能でつなぐ、またはGitHub Codespacesを使うという解決策もあります。

注9　https://multipass.run/
注10「Known Issues - Docker Desktop for Apple silicon」https://docs.docker.com/desktop/mac/apple-silicon/#known-issues

第 3 章 コンテナ実行環境にPaaSを使うという選択肢 —— Web App for Containers

運用負荷を下げ、開発に専念するための実行環境を選定しよう

前章では、コンテナを活用したアプリケーションの開発の一連の流れをシンプルなWebアプリケーションで体験しました。本章からは、データベースなどを使ったより実践的なWebアプリケーションを実行環境にデプロイし、インターネットに公開してみましょう。

本章では、コンテナの実行環境について取り上げ、マネージドサービスを活用することで、可能な限り実行環境の構築や運用を意識せず、アプリケーションの開発、運用に専念できることを紹介します。

3.1 コンテナ技術の適用に立ちはだかる実行環境の労力

Dev Containerを使ってWebアプリケーションを作成し、イメージをコンテナレジストリに登録する作業も完了しました。あとはそのイメージを実行環境で動かすだけです。それでは、その実行環境はどのように準備し、またアプリケーションとともに運用していくのでしょう。

3.1.1 » 実行環境の構築

作成したイメージを動かすためには当然実行環境の構築が必要です。Webアプリケーションのイメージを動かすためには、サーバを用意し、OSとコンテナランタイムをそれぞれ用意しなければなりません。また、サーバに対するネットワークを疎通し、ドメインやSSL(*Secure Socket Layer*)/TLS(*Transport Layer Security*)サーバ証明書を設定する作業もあります。そのうえ、システムの要件によっては、サーバやネットワークスイッチを複数台用意し、それらを複数のデータセンターに分散させ、冗長構成をとらなければならない場合もあります。このような構築作業はアプリケーション開発の片手間にできる作業では一切なく、専属のエンジニアを要する時間がかかるチャレンジングな業務と筆者は認識しています。

3.1.2 » 実行環境の運用

実行環境を構築したあとには運用が待っています。実行環境の運用と聞いてみなさんは何を思い浮かべるでしょうか。筆者が一番に思い浮かぶのがサーバやネットワークスイッチといったハードウェアの管理と、OSやコンテナランタイムなどのアプリケーションを動かすために必要なソフトウェアコンポーネントの定期的な更新です。セキュリティの観点から定期的なパッチ当てを実行基盤に行う必要があるうえに、保守期限が切れる前にハードウェアやソフトウェアの更新も実行環境の運用として行う必要があります。

また、アプリケーション自体も機能追加のために更新する必要が出てきます。実行環境へスムーズにデプロイするために、コンテナレジストリと同期することも実行環境ではサポートする必要があります。できればステージング環境にいったんデプロイし、最終テストを実施後に実行環境にデプロイする、それも極力ダウンタイムを引き起こさないで行いたい、といったアプリケーションの更新メカニズムの実現も求められます。さらに、アプリケーションを利用するユーザー数の増加に伴ったサーバの追加、サーバを含めた実行基盤自体の死活監視と、実行環境の異常からの復旧も運用に含まれます。

3.1.3 » アプリケーションの開発、運用に専念

以上で述べた実行環境の構築や運用は、アプリケーション自体の開発や運用に専念したいエンジニアにとって悩みの種です。せっかくコンテナ技術の台頭でアプリケーションの配布が容易になったのですから、アプリケーションをコンテナにパッケージングしてマネージドサービスに配布し、実行環境の運用はマネージドサービスに任せたら楽になりそうです。その結果、アプリケーションの開発にエンジニアを集中させ、開発スピードを加速させることができます。

3.2
マネージドサービスを使ってコンテナの開発・運用に専念

実行環境はマネージドサービスから選択すると決めたうえで、コンテナの実行環境としてのマネージドサービスは世の中に多数あります。

3.2.1 » Azureにおける実行環境

たとえば、Microsoft Azureだけみてもコンテナオプションの比較[注1]にあるように、下記マネージドサービスなどが選択肢に挙がります。

❶Azure Container Instances
❷Azure App Service
❸Azure Kubernates Service
❹Azure Container Apps

第2部では、Webアプリケーションを実行環境にデプロイし、運用するまでの一連の流れを体験します。第2部で扱うコンテナは単一コンテナです。そのため複数コンテナを取り扱うために適した実行環境であるAzure Kubernates ServiceやAzure Container Appsなどは実行環境としてオーバースペックであり、第2部のシナリオに対して煩雑さを招くため候補から外れます。

単一コンテナでWebアプリケーションをホスティングする場合、Azure Container InstancesやAzure App Serviceに候補が絞られます[注2]。以降、両サービスの概要を説明し、第2部で取り扱うマネージドサービスを決定します。なお、マイクロサービスなどで複数のコンテナを動作させる実行環境が必要な読者もいると思います。複数のコンテナを1つのマネージドサービスにデプロイするシナリオは第3部で取り扱います。第2部と第3部は基本的に独立していますので、第3部を先に読むことも可能です。

◆Azure Container Instances

Azureアーキテクチャセンターのコンピューティングサービスの選択に記載があるとおり、Azure Container Instances（以降、Container Instances）はAzureで簡単にコンテナを実行する方法です[注3]。Container Instancesは、コンテナグループと呼ばれる、1つあるいは複数のコンテナの集合を仮想マシンで動作させるマネージドサービスととらえることができます。たとえば、Container InstancesにWebアプリケーションのイメージをデプロイすることで、下記URLでインターネットから接続可能なWebアプリケーションを実現できます。

```
<任意の名前>.<リージョン名>.azurecontainer.io:<ポート>
```

他方、そのシンプルさゆえに、SSL/TLSサーバ証明書による通信の保護やロードバランシング、スケールアウトといったWebアプリケーションに求められる機能は、サイドカーコンテナを使ったり、

注1 「Container AppsとほかのAzureコンテナーオプションの比較」https://learn.microsoft.com/ja-jp/azure/container-apps/compare-options

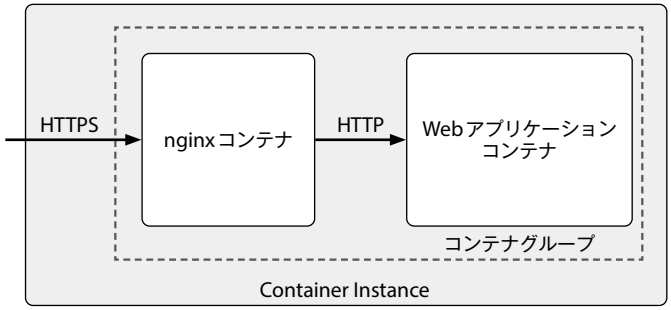
注2 両サービスとも複数コンテナを取り扱うことができます。ただし、Azure App ServiceにおいてはDocker Composeはプレビュー機能です。

注3 「Azureコンピューティングサービスを選択する - 基本的な機能を理解する」https://learn.microsoft.com/ja-jp/azure/architecture/guide/technology-choices/compute-decision-tree#understand-the-basic-features

ほかのマネージドサービスを組み合わせたりして、ユーザー側で構築、運用する必要があります。

たとえば、Container Instancesのドキュメントでは、HTTPSを実現する方法としてサイドカーコンテナの利用が紹介されています[注4]。具体的には**図3.1**にあるように、Webサーバ（nginx）が動作するサイドカーコンテナを使ってHTTPSによってパブリック通信を保護します。SSL/TLSサーバ証明書を設定したnginxをユーザー側で用意し、HTTPエンドポイントで待ち受けているWebアプリケーションコンテナの前段にそのnginxをリバースプロキシとして配置します。その結果、インターネットからnginxまでのパブリック通信はHTTPSで保護され、アプリケーションコードの修正なしにHTTPS接続を実現できます。しかし、nginxおよび証明書の構築、運用はユーザー側の負担となります。

図3.1　サイドカーコンテナを使ったHTTPSによるパブリック通信の保護



◆Azure App Service

Webアプリケーションを構築する場合に、Azure App Service（以降、App Service）は「理想的な選択肢」とあるように[注5]、App Serviceは、HTTP（S）ベースのアプリケーションをホスティングすることに注力したマネージドサービスです。App Service自体に、カスタムドメインやHTTPSによる通信の保護、ロードバランシングやスケールアウトが機能として含まれています。

App ServiceではOSをWindowsとLinuxから選ぶことができ、また複数のホスティングオプションが存在します。一つは、マネージドサービス側があらかじめ言語のランタイムなどを用意し、そのホスティング環境にアプリケーション資材（WARファイルなど）をユーザーが組み込む「コード」ホスティングオプションが挙げられます。一方、ユーザーが用意したイメージを動作させる「コンテナ」ホスティングオプションもあります。「コンテナ」ホスティングオプションのことはWeb App for Containers[注6]と呼ばれ、Webアプリケーションのイメージの実行環境として使えます。第2部では、Web App for Containersを例にとり、マネージドサービスでのコンテナホスティングを説明します。

注4　「サイドカーコンテナーでTLSを有効にする」https://learn.microsoft.com/ja-jp/azure/container-instances/container-instances-container-group-ssl

注5　「Container Appsと他のAzureコンテナーオプションの比較 - Azure App Service」https://learn.microsoft.com/ja-jp/azure/container-apps/compare-options#azure-app-service

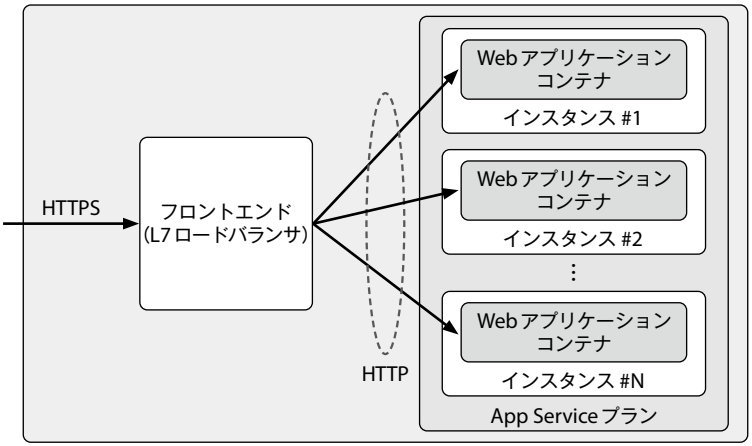
注6　https://azure.microsoft.com/ja-jp/services/app-service/containers/#overview

3.2.2 » App Service（Web App for Containers）でのイメージの動作

Web App for ContainersにてWebアプリケーションコンテナをホスティングした様子を**図3.2**に示します。図3.2にあるように、Web App for Containersにデプロイされたイメージは、App Serviceプランのインスタンスで動作します。インスタンスは、Web Worker、もしくは単にWorkerとも呼ばれ、仮想マシンに相当すると考えられます。

App ServiceではプラットフォームがL7ロードバランサを用意しており、そのロードバランサによってWebアプリケーションコンテナはHTTPSで保護されています。ロードバランサでHTTPSは終端され、HTTPとしてリクエストはコンテナに届きます[注7]。L7ロードバランサがプラットフォーム側で用意されているため、性能や可用性の要件に合わせてインスタンスを複数台用意することもできます。インスタンスが複数台ある場合、各インスタンスで同じイメージに基づきコンテナが動作し、ロードバランサによって各コンテナにHTTPリクエストがラウンドロビンで振り分けられます。

図3.2 Web App for ContainersにWebアプリケーションコンテナをホスティング



3.2.3 » App Service（Web App for Containers）の料金

マネージドサービス全般で言えることですが、Web App for Containersを使ううえで料金が発生します。App Serviceの料金にあるように、主に課金対象となるのはApp Serviceプランの種類とインスタンスの台数です[注8]。筆者は第2部を執筆するにあたり1インスタンスあたり月額13ドル程度のB1プラン（CPU1コア、メモリ1.75GB）1インスタンスを主に使用しました。より高い費用になりますが、P1V3

注7 「Inside the Azure App Service Architecture - Scale Unit Network Configuration」https://learn.microsoft.com/en-us/archive/msdn-magazine/2017/february/azure-inside-the-azure-app-service-architecture#scale-unit-network-configuration

注8 「App Serviceの価格」https://azure.microsoft.com/ja-jp/pricing/details/app-service/linux/

プラン（CPU2コア、メモリ8GB）といったより性能の良いインスタンスを使うことで、より低遅延かつ高スループットなWebアプリケーションにすることもできます。一方、より安価にもなる無料のFreeプランがありますが、1日1時間しかWebアプリケーションを動作させられません。

App Servicプランの種類やインスタンスの台数はいつでも変更できます。そのため料金を抑えるテクニックには、コンテナを動作させる必要がない場合はWeb Appを停止し、こまめにFreeプランにスケールダウンすることが挙げられます。Web Appを停止しても、Azure Virtual Machinesの「停止（割り当て解除）」とは異なり、プランに対する課金が継続発生することに注意が必要です。

3.3 第2部のゴール

第2部においては、Webアプリケーションおよびコンテナイメージをゼロから作り、Web App for Containersでコンテナを運用するまでの一連の流れを身に付けることをゴールにします。

3.3.1 » 各章の概要

第2部の各章の概要を記載します。第5章以降は基本的に独立していますが、特定の章を飛ばさずに、第4章から順に読まれるのが理解しやすいと考えます。

◆第4章

第4章ではサンプルアプリケーションを作成し、コンテナイメージをGitHub Actionsなどを経由してWeb App for Containersにデプロイし、一般公開します。

◆第5章

第5章では、Web App for Containersで動作するコンテナから、データベースなどの外部リソースに接続する方法を説明します。

◆第6章

第6章では、Web App for Containersが提供するIDプロバイダとの連携機能を紹介し、ユーザー認証を簡単に実現できることをデモンストレーションします。

◆第7章

第7章では、可用性の高いWebアプリケーションを作るためのポイントを、Web App for Containersの機能をベースに解説します。

◆第8章

残念ながら永遠に問題なく動き続けるWebアプリケーションを作ることは困難です。有事のための備えとして、第8章では死活監視やアラート、ログ解析といった運用ノウハウを取り扱います。

3.4 まとめ

本章では、実行環境の構築や運用の難点を踏まえ、実行環境としてマネージドサービスであるWeb App for Containersを使うことにしました。次章以降、Web App for ContainersにWebアプリケーションを実際に動作させ、マネージドサービスを使うことによってアプリケーションの開発、運用に専念できることを体験できます。

第10章 Container Appsでのコンテナアプリケーション開発ハンズオン

サンプルを再設計して複数コンテナで動かしてみよう

第2部でアプリケーションの実行環境であるWeb App for Containersを使ったJavaアプリケーション開発の流れとポイントを解説しました。本章では、コンテナアプリケーションを動かす実行基盤としてAzure Container Apps(以降、Container Apps)を使って、第2部で開発したToDoアプリケーションを複数コンテナに拡張したサンプルを用いた実装のポイントとデプロイの方法を具体的に解説します。

10.1 サンプルアプリケーションの機能追加と再設計

まずToDoアプリケーションの機能追加を行うため全体構成について見なおします。

10.1.1 » ToDoアプリケーションへの要望

第2部では、JavaによるToDoアプリケーションをWeb App for ContainersとAzure Database for MySQLで動かしました。マネージドサービスを活用することで、開発者が開発に集中しつつすばやくアプリケーションを開発・運用できることを確認しました。

このアプリケーションをリリースするとたちまち利用者が増え、次のようなフィードバックがあがったと想像してみましょう(**図10.1**)。

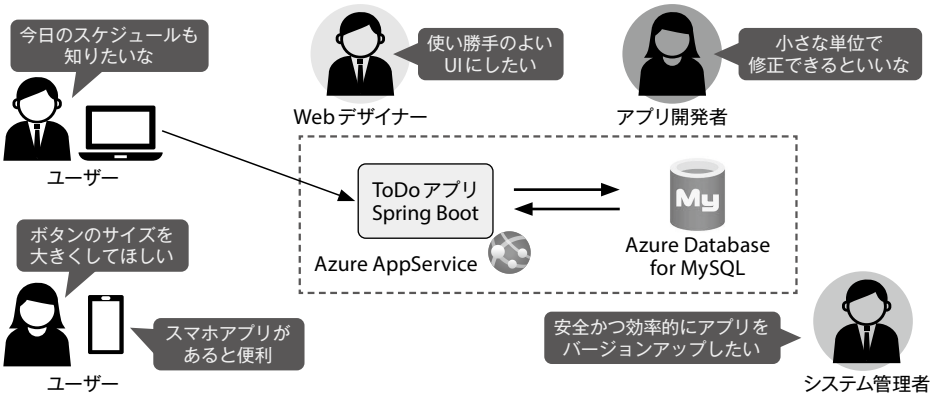
- **本日のスケジュール表示機能があると良い**
- **ユーザーの声を取り込み、使いやすいUIにしてほしい**
- **利用者を拡大して社外ユーザーにも展開したい**
- **モバイルアプリケーションからも利用したい**

スケジュール情報は、別システムで管理されている既存のオフィス統合システムからデータを取得する必要があります。またUIを改善するには、ユーザーの声を聴きながらデザイナーが少しずつ改良していくのがよいでしょう。第2部のアプリケーションはUIとビジネスロジックが1つのコンテナで

実装されているため何か1つでも機能追加や変更を加える場合、アプリケーションすべてをコンパイルしなおしてデプロイする必要があります。

また利用者のモバイルアプリケーションからのリクエスト数が増えると、App Serviceをスケールアップ／スケールアウトをする必要がありますが、リクエストが一時的に増える始業時間だけ一時的にバックエンドのアプリケーションだけ待ち受けを増やしたいなどの非機能要件があがります。

図10.1　サンプルアプリケーションの課題



10.1.2 » アプリケーションアーキテクチャの検討

これらのニーズを満たすためにはどのようにすればよいでしょうか？ まず、アプリケーションの機能分割を検討しましょう。

◆バックエンドとフロントエンドの分離

第2部では、ToDoアプリケーションでデータベースからデータを取得する処理と取得したデータを加工して表示する機能が1つのアプリケーションで動いていました。これをデータを表示するフロントエンド機能とデータを登録／取得／更新／削除する機能に分離します。これによって、UIはフロントエンド開発者、データ操作はアプリケーション開発者が実装を行うことで別々に開発ができます。

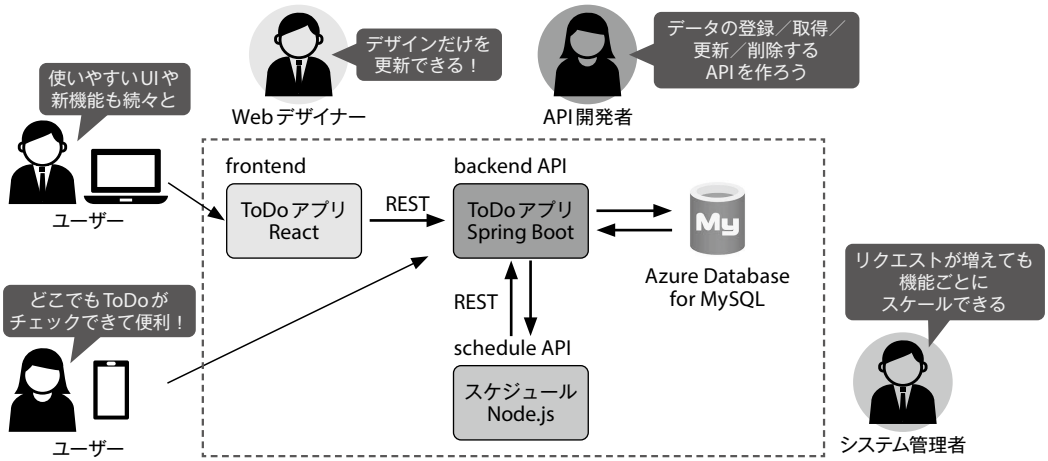
◆アプリケーションのAPI化

データ操作を行う機能について、ブラウザからだけでなくモバイルアプリケーションなどさまざまな環境から呼び出して利用できると便利です。データを登録／取得／更新／削除ができるREST APIを作り、外部から呼び出し可能な実装にします。スケジュール機能を表示する新機能を追加する場合も、外部のAPI（schedule API）を呼び出して必要な情報を返すことができればよいでしょう。

このようにアプリケーションアーキテクチャを変更することで、ざまざまなメリットが出ます。各サービスごとに独立かつ並行して開発ができるため、開発チームの自由度も上がります（**図10.2**）。た

とえば、現状ではbackend APIとschedule APIはともにSpring Bootで実装していますが、将来的にほかの開発言語（PythonやNode.jsなど）に移行したとしても、各サービスから見るとREST APIを呼び出しているだけですので、インタフェースが変わらなければ動作します。つまり、個々の機能開発において影響範囲を限定できるため、新機能のリリースがやりやすくなるでしょう。また、一時的にリクエスト数が増えたときもフロントエンドアプリケーションだけをスケールアウトしてシステム増強ができます。

図10.2 アプリケーションアーキテクチャの検討



10.1.3 » コンテナ実行環境の選択肢

第2部で解説したApp Service（Web App for Containers）やContainer Instancesを使うこともできますが、せっかくなので第3部では第2部で紹介した以外のサービスを検討します。

◆ Azure Functions

小さな関数を動かすのに適したサーバレスのサービスです。IoT（*Internet of Things*）のバックエンドやイベント駆動型アプリケーションを実行するのに向いています。Azure Functionsプログラミングモデルには、イベントに応じて関数の実行をトリガし、ほかのデータソースにバインドすることで、短いコードで機能を実装できるのが特徴です。ただし、選択するプランによっては実行時間の制約などがあります。

◆ Azure Container Apps

Kubernetes環境上でコンテナアプリケーションを動かすマネージドサービスです。Kubernetesのクラスタを管理することなく、マイクロサービスを実行でき、負荷分散やオートスケールなどの機能も

備えています。オープンソーステクノロジとの親和性が高く、分散アプリケーションランタイムDapr（*Distributed Application Runtime*）、L7プロキシEnvoy、イベントドリブンオートスケーラーKEDAをサポートし、GitHubなどを活用したクラウドネイティブな構成が容易にできるという特徴があります。また、Webアプリケーション、APIのホスティング、実行時間の長いバッチ処理などワークロードを問わず幅広く利用できます。

◆ Azure Kubernetes Service

Azure Kubernetes Service（以降、AKS）は、Kubernetesのマネージドサービスです。Kubernetes APIへのアクセスがサポートされるため、Kubernetesクラスタの操作やチューニングなども可能です。ただし、Kubernetesクラスタがサブスクリプション内にデプロイされ、クラスタの構成と運用管理はユーザーの責任範囲内となります。オンプレやほかのクラウドなどですでにKubernetesを使ってアプリケーションを運用している場合はAKSが適しています。

◆ Azure Spring Apps

Springアプリケーションを動かすことに特化したサービスです。監視と診断、構成管理、サービス検出、CI/CD統合、ブルーグリーンデプロイなどを使用して、ライフサイクル管理をできるのが特徴です。

◆ Azure Red Hat OpenShift

Red Hat社が提供するコンテナオーケストレータであるOpenShiftの環境を提供するマネージドサービスです。すでにOpenShiftでの開発と運用実績がある場合に適しています。

Container AppsとほかのAzureコンテナ実行サービスとの比較については公式サイト[注1]にまとまっています。

10.1.4 » システムアーキテクチャの検討

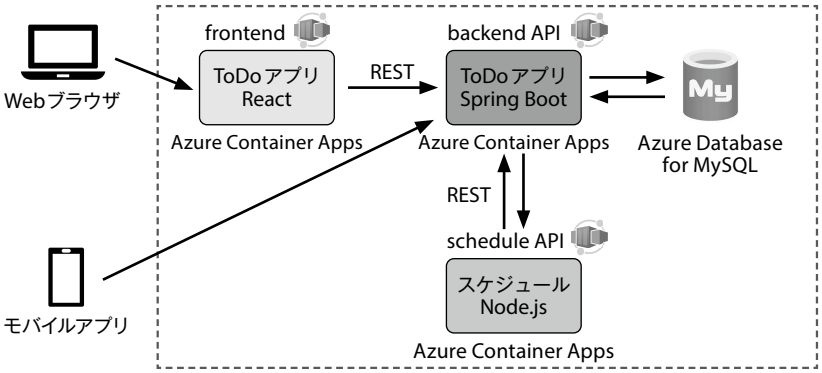
アプリケーションを機能単位で小さく分割した場合はどのようなプラットフォームが適しているでしょうか？ あらためて今回のToDoアプリケーションでは、次のシステム要件があります。

- **複数の言語／テクノロジで書かれたマイクロサービスを運用したい**
- **Webサービスを提供するための負荷分散やオートスケールの機能が欲しい**
- **継続的な開発やリリースを行って新機能を提供したい**
- **インフラの運用管理を最小化して、アプリケーション開発に集中したい**
- **システム運用コストは需要に応じて柔軟に変更したい**

注1　https://learn.microsoft.com/ja-jp/azure/container-apps/compare-options

今回はシステム要件を最も満たす「Azure Container Apps」（以降、Container Apps）を採用することとします（**図10.3**）。

図10.3 サンプルのアーキテクチャ全体像



10.2 Container Appsとは

Container Appsはマイクロサービス型のマルチコンテナをホストするのに適したサービスです。環境を作成する前に、Container Appsをもう少し掘り下げて理解しておきましょう。

コンテナアプリケーションの実行環境として候補に挙がるのがコンテナオーケストレータのデファクトスタンダードであるKubernetesです。Kubernetesはオープンソースとして開発されていて、クラウドネイティブな技術のエコシステムを形成していることもあり利用者も増えています。しかしながら、Kubernetesをうまく使いこなして安定稼働させるにはインフラ技術を深く理解しておく必要があり、専任の運用チームが必要です。また、開発者がKubernetesのしくみを理解して正しく使いこなせるようになるまでの学習コストがかかるというのも実情です。

これらKubernetesで難しい部分をマネージドとして提供し、開発者にとって使いやすい機能のみを提供しているのがContainer Appsです。

Container Appsは2022年5月に開催されたMicrosoftの開発者向けイベントである「Microsoft Build 2022」[注2]で一般提供されました。

クラウドネイティブアプリケーションは、多くの場合、Cloud Native Computing Foundation（CNCF）によって定義されている、疎結合／復元力／管理可能／可観測性を備えた分散マイクロサービスで構成

注2 https://techcommunity.microsoft.com/t5/apps-on-azure-blog/azure-container-apps-general-availability/ba-p/3416885

されます。Container Appsは、開発者がクラウドインフラの管理ではなく、ビジネスの差別化要因となるアプリケーション開発に集中できることを目指して作られたサービスです。Container Appsは、Linuxベースのコンテナにパッケージ化されたアプリケーションを開発者自らで動かすことができる機能を備えています。

Container Appsが提供する機能は次のとおりです。

- **コンテナレジストリからコンテナを実行(第10章で解説)**
- **アプリケーション内で使用するシークレット管理(第10章で解説)**
- **Azure Log Analyticsを使用したアプリケーションログ管理(第10章で解説)**
- **コンテナアプリケーションのバージョン／リビジョンを管理(第11章で解説)**
- **スケールトリガによってアプリケーションを自動スケーリング(第11章で解説)**
- **Ingress(イングレス)／負荷分散機能(第11章で解説)**
- **ブルーグリーンデプロイ／トラフィック分割(第11章で解説)**
- **仮想ネットワーク(VNet)との統合(第11章で解説)**
- **Daprを使用したマイクロサービスの開発(付録で解説)**

Container Appsは、1秒あたりのリソース割り当てとリクエスト数に基づいて課金されます。毎月最初の180,000vCPU秒、360,000GiB秒、200万件のリクエストは無料で利用できます。それ以上の場合は、使用した分だけ秒単位で課金されます。詳しくはドキュメント[注3]で確認してください。

Column

Container Appsの設計思想とオープンソース

Container Appsは、これまでAzureの多くのお客様からいただいた声や技術動向などを踏まえて、「より開発者にうれしいサービスを！」にこだわって作られた新しいサービスです。Container Appsのコンセプトに「Non-opinionated」があります。日本語に訳すと「自らの意見を強調しない＝制約を課さない」のようなニュアンスになります。一般的にアプリケーションPaaS(*Application Platform as a Service*)は、開発言語やランタイムに制約があり、開発者は制約や仕様を意識しつつアプリケーションを開発する必要があります。それに対して、Container Appsはなるべく制約を排除しオープンな技術を活用して、開発者が必要なテクノロジを利用しやすくしたいというコンセプトのもと、さまざまな工夫がなされています。Container AppsはオートスケールにKubernetes Event Driven Autoscaling(KEDA)、分散アプリケーションランタイムであるDistributed Application Runtime(Dapr)、Kubernetesで実行されるL7ロードバランサEnvoyなどのCNCFプロジェクトのオープンソーステクノロジを基盤として構築されています。このオープンソース中心のアプローチのおかげで、アプリケーションの移植性を維持しながら、Kubernetesクラスタ運用の負担なく、クラウドネイティブアプリケーションをAzure上で動かすことができます。

注3 「Azure Container Appsの価格」https://azure.microsoft.com/ja-jp/pricing/details/container-apps/

執筆者プロフィール

真壁 徹（**まかべ とおる**）

日本マイクロソフト株式会社 シニアクラウドソリューションアーキテクト。企業におけるクラウドの可能性を信じ、ユーザーと議論、実装、改善を行う日々。アプリもインフラも好物。趣味はナイスビール。主な著書は『しくみがわかるKubernetes』（翔泳社刊）、『Microsoft Azure実践ガイド』（インプレス刊）など。

URL: https://torumakabe.github.io/
GitHub: torumakabe
Twitter: @tmak_tw

東方 雄亮（**とうぼう ゆうすけ**）

日本マイクロソフト株式会社でPaaS製品であるApp Serviceなどのサポートエンジニアに従事。LinuxやJavaでのアプリケーションの開発や運用、特にトラブルシューティングを得意とする。

URL: https://www.linkedin.com/in/yusuketobo
GitHub: YusukeTobo

米倉 千冬（**よねくら ちふゆ**）

東京エレクトロン株式会社の情シス部門でAzureの活用推進、DXの技術的支援などに従事。以前は日本マイクロソフトにてAzureを含むWebアプリ関連製品の技術サポートを担当。

谷津 秀典（**やつ ひでのり**）

日本マイクロソフト株式会社にてサポートエンジニアとしてPaaS製品であるAzure App Serviceを中心に担当。現職への入社前はSIer企業にてバックエンド開発とプロジェクトリーダーを経験後、フリーランスWeb系エンジニアを経て、AI系ベンチャー企業でのクラウド開発におけるリードエンジニアを担当。多様なバックグラウンドをベースとしてユーザーの技術的な問題解決の支援に従事。

阿佐 志保（**あさ しほ**）

日本マイクロソフト株式会社 自動車／運輸のお客様担当のクラウドソリューションアーキテクト。担当技術領域はAzureのアプリケーション開発／実行環境。趣味は野毛散策。主な著書は『しくみがわかるKubernetes』（翔泳社刊）、『プログラマーのためのVisual Studio Codeの教科書』（マイナビ出版刊）など。

URL: https://asashiho.github.io/
GitHub: asashiho