Docker と Kubernetes によるアプリのコンテナ化

Docker と Kubernetes を使ったコンテナのデプロイ、
スケーリング、オーケストレーション、管理

Dr. Gabriel N. Schenker

Packt
www.packt.com

Docker と Kubernetes による アプリのコンテナ化

Docker と Kubernetes を使ったコンテナのデプロイ、
スケーリング、オーケストレーション、管理

Gabriel N. Schenker 博士

Packt

バーミンガム - ムンバイ

Docker と Kubernetes によるアプリのコンテナ化

外部編集者 : Vijin Boricha
編集者 (権利獲得): Shrilekha Inani
編集者 (コンテンツ開発): Ronn Kurien
編集者 (技術): Swathy Mohan
コピー エディター : Safis Editing
プロジェクト コーディネーター : Jagdish Prabhu
校正者 : Safis Editing
索引作製者 : Mariammal Chettiyar
グラフィック : Tom Scaria
プロダクション コーディネーター : Nilesh Mohite

初版 : 2018 年 9 月

プロダクション リファレンス : 1260918

発行 : Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78961-036-9

www.packtpub.com

mapt.io

Mapt は、5,000 を超える書籍やビデオへのフル アクセスをはじめ、利用者個人の成長を促し、キャリアを高める上で役立つ業界最高水準のツールを提供するオンライン デジタル ライブラリです。詳細については、弊社 Web サイトをご覧ください。

購読するメリットは？

- 4,000 名を超える業界の専門家が提供する実用的な e-Book やビデオで、学習時間を短縮してコーディングにかける時間を増やすことができる
- 個別に構築されたスキル プランでより効果的に学習できる
- 毎月無料の e-Book やビデオを入手できる
- Mapt は全件検索が可能
- コンテンツのコピーと貼り付け、印刷、ブックマークへの登録ができる

PacktPub.com

Packt は、出版された全書籍の e-Book 版を PDF と ePub ファイルで提供しています。www.PacktPub.com で e-Book 版にアップグレードできます。また、印刷版をお持ちのお客様には、e-Book 版を割引価格でご購入いただけます。詳しくは、customercare@packtpub.com からお問い合わせください。

www.PacktPub.com では、無料の技術記事を閲覧したり無料ニュースレターに登録いただけるほか、Packt の書籍や e-Book の限定割引や特典もご用意しています。

制作協力

著者について

Gabriel N. Schenker 博士： 独立系コンサルタント、アーキテクト、リーダー、トレーナー、指導者、開発者として 25 年を超える実績を誇ります。現在は Confluent でシニア カリキュラム開発者を務めており、前職の Docker でも同様の役職に就いていました。物理学の博士号のほか、Docker Captain および Certified Docker Associate、ASP Insider の認定も取得しています。プライベートでは、妻の Veronicah と子供たちとの時間を大切にするファミリーマンでもあります。

校閲者について

Xijing Zhang: 南カリフォルニア大学 (USC) 電気工学科卒業後、Docker に入社。現在は同社で技術カリキュラム開発者を務めています。SanDisk の故障解析チームでインターンの経験があるほか、USC と清華大学では複数の研究職を歴任。これまで、空調装置の効率化や原子力安全、単光子放射に関する数々のプロジェクトに取り組んできました。

Peter McKee: Docker, Inc. のソフトウェア アーキテクト兼シニア ソフトウェア エンジニア。Docker Success Center 技術チームのリーダーとして、20 年以上にわたりチームの指導に携わってきました。プライベートでは良き夫であり、7 人の子供の父親でもあります。テキサス州オースティン在住。

企画・原稿募集

Packt 書籍の執筆にご関心がある方は、今すぐ `authors.packtpub.com` からご応募ください。弊社はこれまでに何千人もの開発者や技術専門家と協働し、グローバルなハイテク コミュニティとのインサイトの共有に貢献してきました。一般的な応募だけでなく、弊社が執筆者を募集している特定のトピックでの応募や、独自の企画の提出などもお待ちしています。

目次

序文

コンテナ化は DevOps を実装する最善の方法であるといわれています。Azure 環境にエンドツーエンドのデプロイメント ソリューションを提供することが、本書の主な目的です。

本書では、Docker と Kubernetes を導入して稼動させるとともに、コンテナのデプロイと管理の実装を開始します。次に、Azure のクラウド ソリューションを使用した Docker でのコンテナ管理とオーケストレーションの操作について説明します。また、スケーラビリティの高いアプリケーションをデプロイして管理する方法と、手つかずの環境にある Azure に本番準備の完了した Kubernetes クラスターをセットアップする方法も説明します。本書はマイクロソフトの Docker ツールと Kubernetes ツールを活用して、Azure に迅速にデプロイできるアプリを開発するのにも役立ちます。

本書を読み終わるころには、いくつかの高度なトピックを実際に試して、Docker と Kubernetes に関する知識をさらに深めることができるでしょう。

本書の対象読者

本書は、オンプレミスまたはクラウドで、Docker と Kubernetes を使用してミッション クリティカルなアプリケーションをスケーラブルかつ安全、高い可用性で実行する開発者、システム管理者、DevOps エンジニアを対象としています。パッケージのインストール、ファイルの編集、サービスの管理などの基本的な Linux/Unix スキルがあることが前提とされており、基本的な仮想化の知識がある場合は、さらに効果的です。

本書の内容

第 1 章 コンテナとは何であり、なぜ使用すべきなのか – この章では、ソフトウェア サプライ チェーンとその内部における摩擦に焦点を当てています。次に、この摩擦を減らし、そこにエンタープライズ クラスのセキュリティを追加する手段として、コンテナを紹介します。またこの章では、コンテナとその周囲のエコシステムがどのように組み立てられているかについても見ていきます。具体的には、アップストリームの OSS コンポーネント (Moby) の区別について説明します。これらのコンポーネントは、Docker や他のベンダーのダウンストリーム製品の構成要素となっています。

第 2 章 作業環境の設定 – この章では、Docker コンテナで作業する際に使用できる、開発者、DevOps、オペレーターにとって理想的な環境を設定する方法を詳細に説明しています。

第 3 章 コンテナの操作 – この章では、コンテナの起動、停止、削除方法について説明しています。また、コンテナを検査してその追加メタデータを取得する方法についても説明します。さらに、追加プロセスの実行方法や、すでに実行中のコンテナ内のメイン プロセスへの接続方法も紹介します。また、内部で実行されているプロセスによって生成されたコンテナからログ情報を取得する方法も示しています。最後に、この章では、Linux 名前空間や cgroup などを含むコンテナの内部動作を紹介します。

第 4 章 コンテナ イメージの作成と管理 – この章では、コンテナのテンプレートとして機能するコンテナ イメージの作成方法を説明します。イメージの内部構造と、それがどのようにビルドされるかを紹介します。

第 5 章 データ ボリュームとシステム管理 – この章では、コンテナで実行されているステートフル コンポーネントが使用できるデータ ボリュームを紹介します。また、Docker および基礎となる OS に関する情報の収集に使用されるシステム レベルのコマンドと、孤立したリソースからシステムをクリーンアップするコマンドについても説明しています。最後に、Docker エンジンによって生成されたシステム イベントを紹介します。

第 6 章 分散アプリケーション アーキテクチャ – この章では、分散アプリケーション アーキテクチャの概念を紹介し、分散アプリケーションを正常に実行するために必要なさまざまなパターンとベスト プラクティスについて説明しています。最後に、このようなアプリケーションを本番環境で実行するために必要な追加要件について説明します。

第 7 章 シングルホスト ネットワーキング – この章では、Docker コンテナ ネットワーキング モデルと、そのシングルホスト実装をブリッジ ネットワークの形で紹介します。また、ソフトウェア定義ネットワークの概念と、それらがコンテナ化アプリケーションを保護するためにどのように使用されるかについても説明します。最後に、コンテナ ポートを公開してコンテナ化されたコンポーネントを外部からアクセスできるようにする方法を紹介します。

第 8 章 Docker Compose – この章では、複数のサービスから成るアプリケーションがそれぞれ単一のコンテナで実行されるという概念と、このようなアプリケーションを宣言型アプローチを使って簡単にビルド、実行、スケーリングする Docker Compose の機能について説明します。

第 9 章 オーケストレーター – この章では、オーケストレーターの概念を紹介します。オーケストレーターがなぜ必要なのか、そして概念的にどのように動作するのかについて説明しています。また、よく使われているオーケストレーターの概要を示し、それぞれの長所と短所をいくつか取り上げます。

第 10 章 Kubernetes を使用したコンテナ化アプリケーションのオーケストレーション – この章では、Kubernetes について説明します。現在、コンテナ オーケストレーションの分野を明らかにリードしているのが、Kubernetes です。まず最初に、Kubernetes クラスターのアーキテクチャの概要を紹介し、次に Kubernetes がコンテナ化アプリケーションを定義して実行するために使用する主なオブジェクトについて説明します。

第 11 章 Kubernetes によるアプリケーションのデプロイ、更新、および保護 – この章では、アプリケーションを Kubernetes クラスターにデプロイ、更新、および拡張する方法を説明します。また、ミッション クリティカルなアプリケーションの中断のない更新とロールバックを可能にするために、ゼロ ダウンタイム デプロイメントをどのように実現するかについても説明します。さらに、サービスを構成して機密データを保護する手段として、Kubernetes シークレットを紹介します。

第 12 章 コンテナ化されたアプリケーションをクラウドで実行する – この章では、Azure Kubernetes Service (AKS) を使用してマイクロソフト Azure 上でホストされた Kubernetes クラスターに複雑なコンテナ化アプリケーションをデプロイする方法を説明します。まず、Kubernetes クラスターをプロビジョニングする方法を説明し、次に Azure Container Registry で Docker イメージをホストする方法を示します。最後に、アプリケーションのデプロイ、実行、監視、スケーリング、アップグレードの方法を説明します。また、クラスター内の Kubernetes のバージョンをダウンタイムを発生させることなくアップグレードする方法も示します。

本書を最大限に活用するには

Windows 10 Professional または Mac OS X の最新バージョンがインストールされたノート PC またはパーソナル コンピュータにアクセスできることが理想です。また、一般的な Linux OS がインストールされたコンピュータでも動作します。Mac をお使いの場合は Docker for Mac を、Windows の場合は Docker for Windows をインストールしてください。これらはこちらからダウンロードできます : `https://www.docker.com/community-edition`。

旧バージョンの Windows を使用している場合や Windows 10 Home Edition を使用している場合は、Docker Toolbox をインストールする必要があります。Docker Toolbox は次の場所にあります : `https://docs.docker.com/toolbox/toolbox_install_windows/`。

Mac ではターミナル アプリケーションを、Windows では PowerShell コンソールを使用して、学習するコマンドを試します。また、Google Chrome、Safari、Internet Explorer などのブラウザの最新バージョンが必要です。また、本書で使用するツールやコンテナ イメージをダウンロードするには、インターネットにアクセスする必要があります。

第 12 章「コンテナ化されたアプリケーションをクラウドで実行する」では、マイクロソフトの Azure にアクセスする必要があります。Azure のアカウントがない場合は、こちらで試用アカウントをリクエストできます : `https://azure.microsoft.com/free/`。

EPUB/mobi とサンプル コード ファイルをダウンロードする

本書の EPUB/mobi 版は、Github から無料でご利用いただけます。これらは、コード バンドルと一緒に次のサイトからダウンロードできます : `https://github.com/PacktPublishing/Containerize-your-Apps-with-Docker-and-Kubernetes`。

本書のサンプル コード ファイルは、次のアカウントからダウンロードできます : `http://www.packtpub.com`。本書を他の場所で購入された場合は、`http://www.packtpub.com/support` でご登録いただければ、電子メールで直接ファイルをお送りします。

以下の手順を実行して、コード ファイルをダウンロードできます。

1. `http://www.packtpub.com` でログインまたは登録します。
2. **[サポート]** タブを選択します。
3. **[コードのダウンロードと正誤表]** をクリックします。
4. **[検索]** ボックスに本書の書名を入力し、画面の指示に従います。

ファイルのダウンロードが完了したら、必ず以下の最新バージョンを使用してフォルダを解凍または抽出してください。

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

本書のコード バンドルは、GitHub (`https://github.com/appswithdockerandkubernetes/labs`) でホストされています。

豊富な書籍カタログから他のコード バンドルとビデオもご利用いただけます (`https://github.com/PacktPublishing/`)。ぜひご確認ください。

カラー画像をダウンロードする

本書で使用されているスクリーンショットおよび図表のカラー画像を収録した PDF ファイルも同時にご提供しています。これは `https://www.packtpub.com/sites/default/files/downloads/9781789610369_ColorImages.pdf` からダウンロードできます。

本書で使用されている規則

本書では、数々の表記規則が使用されています。

`CodeInText`: テキスト内のコード ワード、データベースのテーブル名、フォルダ名、ファイル名、ファイル拡張子、パス名、ダミー URL、ユーザー入力、Twitter のハンドルを示します。たとえば、各レイヤーの内容は、ホスト システム上の特別なフォルダにマップされます。通常、これは `/var/lib/docker/` のサブフォルダです。

コード ブロックは次のように設定されます。

```
COPY ./app
COPY ./web /app/web
COPY sample.txt /data/my-sample.txt
ADD sample.tar /app/bin/
ADD http://example.com/sample.txt /data/
```

コード ブロックの特定の部分に着目していただきたい場合、関連する行または項目を太字で表示しています。

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY ./requirements.txt /app/
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

コマンドライン入力または出力は、次のように記述されます。

```
az group create --name pets-group --location westeurope
```

太字 : 新しい用語や重要な単語、画面上に表示される単語を示します。たとえば、本文中ではメニューやダイアログ ボックスは太字で表示されます。例 : **[管理]** パネルから **[システム情報]** を選択します。

警告や重要な注意事項はこのようなボックスに表示されます。

ヒントやコツはこのように表示されます。

お問い合わせ先

読者の皆様からのフィードバックをお待ちしています。

一般的なフィードバック : customercare@packtpub.com 宛にメールを送信してください。その際、メッセージの件名に書名を入れてください。本書について、ご質問がある場合は、customercare@packtpub.com まで電子メールでご連絡ください。

正誤表 : 内容の正確さには細心の注意を払っていますが、それでも間違いが起こることがあります。本書に誤りを見つけられた際には、お手数ですが弊社までお知らせください。http://www.packtpub.com/submit-errata で該当する本を選択し、誤植の提出書式のリンクをクリックして、詳細情報をご入力ください。

海賊行為 : どのような形態であれ、当社の作品の違法コピーをインターネット上で見つけた場合は、お手数ですが直ちにロケーション アドレスまたは Web サイト名を教えてください。資料のリンクを添えて、copyright@packtpub.com までご連絡ください。

執筆にご関心がある場合 : **ご自身の専門知識を扱ったトピックで、本を書いたり寄稿したりすることをお考えの場合は、**http://authors.packtpub.com をご覧ください。

レビュー

購入元のサイトに、本書に対するご意見・ご感想をお寄せください。お客様の公平なご意見は、本書に関心を寄せられている読者の方が購入を決定する際の貴重な参考となると同時に、Packt にとっては弊社製品に対するお客様の声を知るための一助となります。また、弊社の執筆者は皆さまからのレビューを通じて著書の感想を得ることができます。ぜひご協力ください。

Packt の詳細については、`packtpub.com` をご覧ください。

1
コンテナとは何であり、なぜ使用すべきなのか

第 1 章では、コンテナとそのオーケストレーションについて紹介します。本書はコンテナの分野に関する事前知識がない読者を対象としており、このトピックについて非常に実践的に説明していきます。

この章では、ソフトウェア サプライチェーンとその内部における摩擦に焦点を当てます。次にその摩擦を減らし、エンタープライズ クラスのセキュリティを追加する手段として、コンテナを紹介します。またこの章では、コンテナとその周囲のエコシステムがどのように組み立てられているかについても見ていきます。具体的には、コード名 Moby で統一されたアップストリームの**オペレーション サポート システム** (**OSS**) コンポーネントの区別について説明します。これらのコンポーネントは、Docker その他のベンダーのダウンストリーム製品の構成要素となっています。

この章では以下のトピックについて説明します。

- コンテナとは
- コンテナはなぜ重要なのか
- 自分や会社にとってのメリットは何か
- Moby プロジェクト
- Docker 製品
- コンテナのエコシステム
- コンテナのアーキテクチャ

このモジュールを終了すると、次のことができるようになります。

- 関心を持つ初心者に対し、コンテナとは何かということを、物理的なコンテナなどを例えにしながら、いくつかの簡単な文章で説明する
- 関心を持つ初心者に対し、コンテナがなぜそれほど重要なのかを、例えとして物理的なコンテナと従来式の配送、または集合住宅と一戸建て住宅などを比較しながら、納得できるように説明する
- Docker for Mac/Windows など、Docker 製品で使用されるアップストリームのオープンソース コンポーネントを 4 つ以上挙げる
- Docker 製品を 3 つ以上挙げる

技術的要件

この章はこのトピックの理論的導入部になります。そのため、この章には特別な技術的要件はありません。

コンテナとは

ソフトウェア コンテナは非常に抽象的なものなので、多くの読者におなじみの例えを出して説明するとわかりやすいかもしれません。その例えとは、運送業界の輸送コンテナのことです。

日々大量の商品が、電車、船、トラックで輸送されています。こうした商品は目的地で降ろされ、そこからまた別の手段で輸送されることもあります。商品は多様性に富んでおり、扱いが難しいこともよくあります。輸送コンテナが発明される前、ある輸送手段から別の輸送手段へのこの積み替えは、非常に複雑かつ面倒なプロセスでした。たとえば、農夫がリンゴを一杯に積んだ荷車を中央駅まで持ってきて、そこでそれを他の大勢の農夫が持ってきたたくさんのリンゴと一緒に列車に積み込むことを考えてみてください。または、ワイン醸造業者がワインの樽をトラックで港に持ち込み、そこで積み降ろして、海外に輸送する船に積み込むという例でも構いません。あらゆる種類の商品がそれぞれ独自の方法で梱包されていたため、それぞれ独自の方法で取り扱わなければなりませんでした。また包装されていない商品はみな、輸送の最中に盗まれたり損傷したりする危険性がありました。そこにコンテナが登場し、輸送業界に革命を起こすことになったのです。

コンテナは寸法が標準化されたただの金属の箱です。それぞれのコンテナの長さ、幅、高さはみな同じです。これが非常に重要なポイントです。世界がこの標準サイズについて合意しなければ、輸送コンテナがこれほど普及することはなかったでしょう。現在、企業が A 地点から B 地点へ商品を輸送したいと考えた場合は、まずその商品をこの標準化されたコンテナに積み込みます。そしてその輸送を、標準化された輸送手段を用いる配送業者に依頼します。この手段とは、コンテナを積載できるように設計されたトラックや、それぞれの貨車で 1 つまたは複数のコンテナを運べる列車の場合もあります。さらに、大量のコンテナの輸送を専門とする船舶もあります。配送業者は商品を開梱して再梱包する必要はありません。配送業者にとってコンテナは単なるブラック ボックスであり、ほとんどの場合彼らがその中身に関心を持つことはなく、気にする必要もありません。コンテナは標準寸法のただの大きな鉄製の箱です。コンテナへの商品の積み込みは現在、商品の輸送を希望する当事者に完全に委任されており、その商品の取り扱いや梱包の方法に関しては彼らが最もよく知っているはずです。すべてのコンテナは形状と寸法が標準化されており同一なので、配送業者はコンテナの取り扱いに標準のツールすなわちクレーンを利用して、列車やトラックからコンテナを積み下ろして船舶に積み込んだりその逆に積み替えたりできます。1 種類のクレーンがあれば、長期間にわたって到着するすべてのコンテナを扱うことができます。また、コンテナ船、トラック、列車などの輸送手段も標準化できます。こうしたすべての標準化によって、商品の輸送にまつわるすべてのプロセスを標準化することが可能になり、コンテナ以前の時代よりもはるかに効率性が高まりました。

これで輸送コンテナがなぜそれほど重要であり、輸送業界全体に革新をもたらしたのかが理解できたと思います。この例えを選んだのは、これから説明するソフトウェア コンテナが、物品のサプライ チェーンで輸送コンテナが果たすのとまったく同じ役割を、ソフトウェア サプライ チェーンで果たしているからです。

では、過去に開発者が新しいアプリケーションを開発するときにどうしていたかを考えてみましょう。アプリケーションが開発者から見て完成した状態になると、アプリケーションは運用エンジニアに渡され、本番サーバーにインストールされて稼働されることになります。運用エンジニアは運が良ければ、インストール手順を記載した正確な文書を開発者から入手できることもあります。そこまでは問題なく、作業も容易でした。しかし企業内に多数の開発者チームができ、それぞれがまったく異なる種類のアプリケーションを作成して、かつそれらすべてを同じ本番サーバーにインストールして実行し続けなければならなくなると、状況はやや手に負えなくなり始めました。それぞれのアプリケーションには通常、それが構築されたフレームワークや使用するライブラリなど、いくつかの外部依存関係があります。

場合によっては 2 つのアプリケーションが同じフレームワークを使用することもありますが、フレームワークのバージョンが異なり、それらに互換性がない場合もあります。当社の運用エンジニアの生活は長年の間に、以前よりもはるかに困難になりました。サーバーすなわち「船」に、さまざまなアプリケーションを傷つけることなく積み込む方法について、大きな創造力を要求されたからです。特定のアプリケーションの新しいバージョンをインストールすることはそれ自体が複雑なプロジェクトであり、何ヶ月もの計画とテストが必要になることもよくありました。言い換えれば、ソフトウェア サプライ チェーンの中には多くの摩擦が存在していたのです。しかし最近では、企業がソフトウェアへの依存度をますます高めており、リリース サイクルも短くなっています。新リリースは年に 2 回程度では足りません。アプリケーションは数週間または数日での更新が必要になっており、場合によっては 1 日に複数回更新されることもあります。それに対応できない企業は、俊敏性の欠如のため廃業に追いやられる危険性があります。では、どのような解決策があるでしょうか。

まず第 1 のアプローチは、**仮想マシン** (**VM**) を使用することでした。企業は複数のアプリケーションをすべて同じサーバー上で実行する代わりに、VM ごとに 1 つのアプリケーションをパッケージ化して実行するようになりました。それによって互換性の問題はなくなり、すべてがまたうまくいったように思われました。しかし残念なことに、平穏な状態は長く続きませんでした。VM はすべて Linux や Windows Server などの本格的な OS を含んでいるため、それ自体がかなり重たい代物なうえ、そのすべてが 1 つのアプリケーションにしか使えません。これは輸送業界で言えば、バナナを積み込んだ 1 台のトラックを輸送するために巨大な船舶を使用するようなものです。これは大変な浪費であり、まったく利益を上げることができません。この問題に対する究極の解決策は、VM よりはるかに軽量でありながら、輸送すべき物品を完全にカプセル化できるものを利用することでした。ここでいう物品とは、開発者が実際に作成したアプリケーションであり、さらに (重要な点として) フレームワーク、ライブラリ、構成などアプリケーションの外部依存関係もすべて含まれます。このソフトウェア パッケージング メカニズムの究極形態となったのが、Docker コンテナでした。

開発者は Docker コンテナを使用してアプリケーション、フレームワーク、ライブラリをパッケージ化した後、そのコンテナをテスト担当者または運用エンジニアに送ります。テスト担当者や運用エンジニアにとって、コンテナは単なるブラック ボックスに過ぎません。重要なのは、それが標準化されたブラック ボックスであるということです。すべてのコンテナは、内部でどんなアプリケーションが実行されるかにかかわらず、均一に扱うことができます。エンジニアは自分のサーバー上でいずれかのコンテナが実行されれば、他のコンテナも実行できるとわかります。そしてそれは事実であり、一部のまれなケースは常に存在しますが、それは例外的なものです。そのため Docker コンテナは、アプリケーションとその依存関係を標準化された方法でパッケージ化するための手段となっています。Docker はこれにより、「どこにでも構築し、発送し、実行できる」という新たなフレーズを生み出しました。

コンテナはなぜ重要なのか

最近ではアプリケーションの新規リリースの間隔がますます短くなっていますが、ソフトウェア自体はまったく単純化されていません。逆に、ソフトウェア プロジェクトはその複雑さを増しています。そのため問題に対応し、ソフトウェア サプライチェーンを単純化するための方法が必要になっています。

セキュリティの向上

また、サイバー犯罪がどれほど増加しているかについてのニュースも毎日聞かれます。幾多の有名企業がセキュリティ侵害の影響を受けています。社会保障番号やクレジットカード情報など、機密情報の顧客データが盗まれています。侵害されているのは顧客データだけではなく、機密性の高い企業秘密も盗難に遭っています。

コンテナはさまざまな方法で役に立ちます。まず第一に、Gartner は最近のレポートで、コンテナ内で実行されているアプリケーションはそうでないアプリケーションよりも安全であると報じました。コンテナは Linux カーネル名前空間などの Linux セキュリティ プリミティブを使用して、同じコンピュータおよび**コントロール グループ** (**cgroups**) で動作するさまざまなアプリケーションに対しサンドボックス アプローチをとるため、1 つの問題あるアプリケーションがサーバーの利用可能リソースを使い切って他のすべてのアプリケーションを動作不能に追い込むという、「noisy neighbor (迷惑な隣人)」の問題を回避できます。

コンテナ イメージは不変なので、簡単にスキャンして既知の脆弱性やエクスポージャーを確認することができ、それによってアプリケーションの全体的なセキュリティも強化できます。

コンテナを利用する場合にソフトウェア サプライ チェーンの安全性を高めるもう一つの方法は、**コンテンツ トラスト**を使用することです。コンテンツ トラストは基本的に、コンテナ イメージの作成者が自称通りの人物であることを保証するとともに、コンテナ イメージの使用者に対してはイメージが転送中に改ざんされていないことの保証を提供します。この後者は**中間者** (**MITM**) 攻撃と呼ばれています。

ここまでに説明したことはもちろん、コンテナを使用しなくても技術的に可能ですが、コンテナは世界的に認められた標準を導入しているため、こうしたベストプラクティスを実装し実施することがはるかに容易になります。

しかしコンテナが重要であることの理由はセキュリティだけではありません。次の 2 つのセクションでは、その他の理由について説明します。

運用環境に近い環境をシミュレートする

1 つの理由は、コンテナを利用すると、開発者のノート PC でも本番のような環境を簡単にシミュレートできることです。アプリケーションをコンテナ化できるということは、Oracle や MS SQL Server などのデータベースもコンテナ化できるということです。Oracle データベースをコンピュータにインストールしたことのある人なら誰でも、それが容易な作業ではなく、さらにコンピュータの容量の多くが奪われてしまうことを知っています。このような作業を、開発したアプリケーションが実際にエンド ツー エンドで動作するかどうかをテストするためだけに、開発用ノート PC で行うことは避けたいものです。コンテナがあれば、コンテナ内で非常に簡単に完全なリレーショナルデータベースを実行できます。テストが終了したら、コンテナを停止して削除するだけで、データベースはコンピュータに痕跡を残すことなく消え去ります。

コンテナは VM に比べて非常に軽量なので、開発者のノート PC で同時に多くのコンテナが実行されることも珍しくなく、その場合もノート PC が過負荷状態になることはありません。

インフラの標準化

コンテナが重要である第 3 の理由は、オペレーターがようやく自分の本当に得意なこと、すなわちインフラのプロビジョニングと、運用環境でのアプリケーションの実行および監視に集中できるようになることです。運用システム上で実行すべきアプリケーションがすべてコンテナ化されていれば、オペレータはインフラの標準化を開始できます。すべてのサーバーは単なる 1 つの Docker ホストになります。これらのサーバーに特別なフレームワークのライブラリをインストールする必要はなく、OS と Docker などのコンテナ ランタイムがあれば十分です。

またアプリケーションはコンテナに内蔵されて動作するため、オペレータにはアプリケーションの内部構造についての詳細な知識が不要になります。運用エンジニアにとっては、コンテナは輸送業界の人員にとっての輸送コンテナと同様、単なるブラックボックスのように見えます。

自分や会社にとってのメリットは何か

以前ある人が、今日では一定の規模を持つすべての企業が、ソフトウェア会社になる必要性を認識すべきだと言いました。ソフトウェアはすべてのビジネスを動かしています。すべての企業がソフトウェア企業になれば、ソフトウェアのサプライチェーンを確立することが必要になります。企業が競争力を維持するためには、ソフトウェア サプライ チェーンが安全かつ効率的でなければなりません。効率性は徹底した自動化と標準化によって達成できます。しかしコンテナは、セキュリティ、自動化、標準化という 3 つの分野すべてにおいてその優位性を実証しています。

数々の有名な大企業が、既存のレガシー アプリケーション (従来型アプリケーションと呼ばれることが多い) をコンテナ化し、完全自動化されたソフトウェア サプライ チェーンをコンテナに基づいて構築すれば、そうしたミッションクリティカルなアプリケーションのメンテナンスに費やされるコストを 50 ～ 60% 削減し、従来型アプリケーションの新リリースの間隔を最大 90% 短縮できると報告しています。

すなわちこうした企業では、コンテナ技術を採用することで多額の資金を節約すると同時に、開発プロセスのスピードアップと市場投入までの時間短縮も実現しています。

Moby プロジェクト

当初 Docker 社が Docker コンテナを導入したときは、すべてがオープン ソースでした。この時点では、Docker は何も商品を持っていませんでした。同社が開発した Docker エンジンは一体型のソフトウェアであり、コンテナ ランタイム、ネットワーク ライブラリ、RESTful API、コマンドライン インターフェイスなど、数多くの論理的パーツが含まれていました。

その他の Red Hat や Kubernetes などというベンダーやプロジェクトは、独自の製品で Docker エンジンを利用していましたが、ほとんどの場合は機能の一部しか利用していませんでした。たとえば、Kubernetes は Docker エンジンの Docker ネットワーク ライブラリを使用せずに、独自のネットワークキング方法を提供していました。Red Hat では Docker エンジンを頻繁に更新せず、古いバージョンの Docker エンジンに非公式のパッチを適用して、それを **Docker エンジン**と呼んでいました。

こうしたさまざまな理由から、Docker では何らかの方法で Docker のオープン ソース部分を商用部分から明確に切り離さなければならないという考えが生じました。さらに同社は、競合他社が独自の利益のために Docker という名前を使用または悪用することを防止したいと考えました。これが、Moby プロジェクトが誕生した主な理由です。このプロジェクトは Docker が開発し、現在も開発を続けているオープン ソースのコンポーネントの大半をカバーしています。これらのオープン ソース プロジェクトには、もう Docker という名前は使用されていません。

Moby プロジェクトには、イメージ管理、シークレット管理、構成管理、ネットワーキングとプロビジョニング、さらにその他多くの機能用のコンポーネントが含まれています。また Moby プロジェクトの一部は、コンポーネントをアセンブルして実行可能な成果物を作成するためなどに使用される、特別な Moby ツールになっています。

技術的に Moby プロジェクトに属するコンポーネントの一部は、Docker により **Cloud Native Computing Foundation** (**CNCF**) に寄付されたため、コンポーネントのリストにはもう表示されていません。顕著なものとしては `containerd` および `runc` があり、これらはともにコンテナ ランタイムを形成します。

Docker 製品

Docker では現在、製品ラインを 2 つのセグメントに分類しています。1 つは **Community Edition** (**CE**) で、これはクローズド ソースですが完全に無料です。もう 1 つは **Enterprise Edition** (**EE**) で、これもクローズド ソースであり、年単位でのライセンス契約が必要になります。エンタープライズ製品は 24 時間 365 日サポートされており、CE 製品よりもはるかに長い期間バグ修正が提供されます。

Docker CE

Docker Community Edition には、Docker Toolbox、Docker for Mac、Docker for Windows などの製品があります。この 3 つの製品はいずれも、主に開発者をターゲットにしています。

Docker for Mac と Docker for Windows はインストールの容易なデスクトップ アプリケーションであり、これを使用すると Mac および Windows 上で Docker 対応のアプリケーションやサービスを構築、デバッグ、テストできます。Docker for Mac と Docker for Windows は、それぞれのハイパーバイザー フレームワーク、ネットワーキング、およびファイルシステムと深く統合された完全な開発環境です。このツールは Docker を Mac または Windows で実行するための最も高速かつ信頼性の高い方法になります。

CE のカテゴリには、より運用エンジニア向けの 2 つの製品もあります。それが Docker for Azure と Docker for AWS です。

たとえばネイティブの Azure アプリケーションである Docker for Azure を使用すると、マウスを数回クリックするだけで、基盤となる Azure の**サービスとしてのインフラ** (**IaaS**) サービス用に最適化され統合された Docker をセットアップできます。これにより、運用エンジニアは Azure で Docker アプリケーションを構築し実行するのにかかる時間を短縮できます。

Docker for AWS も機能は非常によく似ていますが、これは Amazon の クラウド用です。

Docker EE

Docker EE は 2 つの製品、**Universal Control Plane** (**UCP**) と **Docker Trusted Registry** (**DTR**) で構成されており、これらは両方とも Docker Swarm 上で動作します。これらはいずれも Swarm アプリケーションです。Docker EE は Moby プロジェクトのアップストリーム コンポーネント上に構築されており、**ロールベースのアクセス制御** (**RBAC**)、マルチ テナント、Docker Swarm と Kubernetes の混合クラスタ、Web ベースの UI、コンテンツ トラストなどのエンタープライズグレードの機能に加え、イメージ スキャンの機能も追加します。

コンテナのエコシステム

これまで IT に導入された新技術の中で、コンテナほど急速かつ包括的に普及したものは他にありません。時代に取り残されないよう努める企業なら、コンテナを無視することはできません。業界のすべての部門から寄せられた、コンテナに対する大きな関心により、この分野には多くの革新がもたらされました。無数の企業がコンテナを専門に扱っており、この技術を利用して構築された製品を提供するか、それをサポートするツールを開発しています。

Docker は当初、コンテナ オーケストレーションのためのソリューションを持っていなかったため、他の企業やプロジェクト (オープン ソースかどうかによらず) がこのギャップを埋めようと試みました。その最も顕著なものが、Google によって開始され、その後 CNCF に寄付された Kubernetes です。他のコンテナ オーケストレーション製品には、Apache Mesos、Rancher、Red Hat の Open Shift、そして Docker 独自の Swarm などがあります。

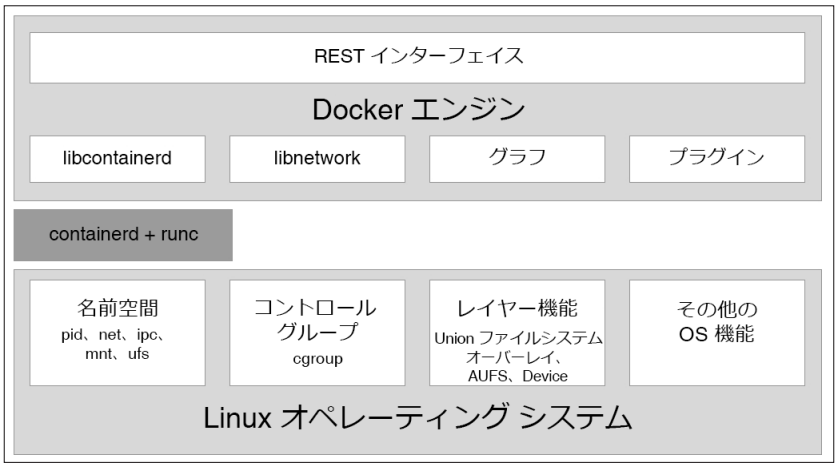
最近では、サービス メッシュがトレンドの先端になってきています。サービス メッシュは新しい流行語となっています。より多くのアプリケーションをコンテナ化し、それらのアプリケーションをよりマイクロサービス指向のアプリケーションにリファクタリングしていくと、単純なオーケストレーション ソフトウェアでは信頼性と拡張性をもって解決できない問題が発生してきます。この分野におけるトピックは、サービス検出、監視、トレース、およびログ集計です。この分野では多くの新しいプロジェクトが出現しており、現時点で最も人気があるものは Istio で、これも CNCF の一部です。

多くの人々が、ソフトウェアの進化における次の段階は機能、より正確にいえば**サービスとしての機能** (**FaaS**) だと言っています。いくつかのプロジェクトはまさにこの種のサービスを提供しており、それらはコンテナ上に構築されています。この顕著な例が OpenFaaS です。

ここまで、コンテナのエコシステムをごく表面的に説明しました。Google、マイクロソフト、Intel、Red Hat、IBM などという大手 IT 企業はみな、コンテナとその関連技術に夢中で取り組んでいます。CNCF は主にコンテナとその関連技術を扱っており、登録プロジェクトが非常に多いため、1 枚のポスターにはすべてが収まりきらなくなっています。今この分野で働くことはとてもエキサイティングです。そして私の意見では、これはまだ単なる始まりにすぎません。

コンテナのアーキテクチャ

では、Docker コンテナを実行できるシステムの設計について詳しく見ていきましょう。次の図は、Docker がインストールされているコンピューターを示したものです。ちなみに Docker がインストールされたコンピューターは、Docker コンテナを実行またはホストできるため、Docker ホストと呼ばれることがよくあります。



Docker エンジンの詳細なアーキテクチャ図

上の図には 3 つの重要な部分があります。

- 一番下に Linux オペレーティング システムがある
- 真ん中の濃いグレーの部分にコンテナ ランタイムがある
- 上部には Docker エンジンがある

コンテナが実現されるのは、Linux OS が名前空間、コントロール グループ、レイヤー機能などのプリミティブを提供しているからです。こうしたプリミティブは、コンテナ ランタイムと Docker エンジンによって非常に特殊な方法で活用されます。Docker は**プロセス ID** (**pid**) 名前空間や**ネットワーク** (**net**) 名前空間などの Linux カーネル名前空間を利用することで、コンテナ内で動作するプロセスをカプセル化またはサンドボックスすることができます。またコントロール グループにより、コンテナは「noisy neighbor」症候群に悩まされずに済みます。これはコンテナ内で実行されている 1 つのアプリケーションが、Docker ホスト全体の利用可能リソースのほとんどまたはすべてを消費できてしまうという現象です。Docker はコントロール グループを利用することで、各コンテナに最大限割り当てられる CPU 時間や RAM 量などのリソースを制限できるようになります。

Docker ホスト上のコンテナ ランタイムは、`containerd` および `runc` で構成されています。`runc` はコンテナ ランタイムの低レベルの機能であり、`runc` をベースとする `containerd` はより高レベルの機能を提供します。どちらもオープン ソースで、Docker が CNCF に寄付したものです。

コンテナ ランタイムはコンテナのライフ サイクル全体を管理します。コンテナ ランタイムは必要に応じて、コンテナのテンプレートであるコンテナ イメージをレジストリから取得し、そのイメージからコンテナを作成し、コンテナを初期化して実行し、後で要求されたらコンテナを停止してシステムから削除します。

Docker エンジンはコンテナ ランタイムに加え、ネットワーク ライブラリやプラグインのサポートなどという追加機能を提供します。また、すべてのコンテナ操作を自動化できる REST インターフェイスも提供します。本書でこれから何度も紹介する Docker コマンドライン インターフェイスは、この REST インターフェイスのコンシューマーの 1 つです。

まとめ

この章では、コンテナがいかにソフトウェア サプライ チェーンの摩擦を大幅に軽減できるだけでなく、サプライ チェーンの安全性も大きく向上させられるかについて説明しました。

次の章では、コンテナにより親しんでいきたいと思います。コンテナを実行、停止、削除する方法や、それ以外の方法で操作する方法を紹介します。また、コンテナの構造についてもわかりやすく説明します。そしていよいよ実際にコンテナを活用してみたいと思いますので、ぜひ楽しみにしていてください。

質問

学習の進捗状況を評価するために、以下の質問にお答えください。

1. 次の文のうち正しいものはどれですか (複数回答可)。
    1. コンテナは一種の軽量な VM である
    2. コンテナは Linux ホスト上でのみ動作する
    3. コンテナは 1 つのプロセスしか実行できない
    4. コンテナ内のメイン プロセスは常に PID 1 となる
    5. コンテナとは、Linux 名前空間によってカプセル化され cgroup によって制限される、1 つ以上のプロセスのことである
2. 関心を持つ初心者に対し、例えなども用いながら自分の言葉で、コンテナとは何かを説明してください。
3. コンテナが IT における画期的要素と見なされているのはなぜですか。理由を 3 ～ 4 つ挙げてください。
4. 「特定のプラットフォームで動作するコンテナはどこでも動作できる」というのはどういう意味ですか。これが本当である理由を、2 ～ 3 つ挙げてください。
5. 次の文は正しいでしょうか、間違っているでしょうか。「Docker コンテナが本当に有用なのは、マイクロサービスに基づく現代的な新しいアプリケーションにおいてのみです。」その答えの理由についても説明してください。
6. 一般的な企業がレガシー アプリケーションをコンテナ化すると、どのくらいのコストを節約できますか。
    1. 20%
    2. 33%
    3. 50%
    4. 75%
7. コンテナの基盤となっている Linux の 2 つの主要概念は何ですか。

参考情報

下記は、この章で説明したトピックの詳細情報を確認できるリンクの一覧です。

- Docker の概要 (https://docs.docker.com/engine/docker-overview/)
- Moby プロジェクト (https://mobyproject.org/)
- Docker 製品 (https://www.docker.com/get-docker)
- Cloud Native Computing Foundation (https://www.cncf.io/)
- containerd – industry standard container runtime at https://containerd.io/

2
作業環境の設定

前章では、Docker コンテナとは何か、なぜ重要であるのかについて学びました。また、現代のソフトウェア サプライ チェーンにおいて、コンテナがどのような問題を解決できるのかについても確認しました。

この章では、Docker を使用して効率的かつ効果的に作業するための、個人的な環境または作業環境を準備します。ここでは、Docker コンテナを使用して作業するための、開発者、DevOps、およびオペレーターにとって理想的な環境を設定する方法について詳しく解説します。

この章で取り上げるトピックは次のとおりです。

- Linux コマンド シェル
- Windows 用 PowerShell
- パッケージ マネージャーの使用
- コード エディターの選択
- Docker Toolbox
- Docker for macOS と Docker for Windows
- Minikube
- ソース コード リポジトリの複製

この章を終了すると、次のことができるようになります。

- Dockerfile、`docker-compose.yml` ファイルなどの単純なファイルを編集可能なエディターをノート PC 上で使用する
- macOS の Bash、Windows の PowerShell といったシェルを使用して、各種 Docker コマンドのほか、フォルダ構造間の移動、フォルダの新規作成などの単純な操作を実行する
- Docker for macOS または Docker for Windows をコンピューターにインストールする
- Docker for macOS または Docker for Windows で、`docker version`、`docker container run` などの単純な Docker コマンドを実行する
- Docker Toolbox をコンピューターに正常にインストールする
- `docker-machine` を使用して VirtualBox に Docker ホストを作成する
- ローカルな Docker CLI を構成し、VirtualBox で動作中の Docker ホストにリモート アクセスする

技術的要件

この章では、macOS または Windows (可能であれば Windows 10 Professional) がインストールされていることが前提となります。また、アプリケーションをダウンロードするための無料インターネット接続と、ダウンロードしたアプリケーションをノート PC にインストールするための権限が必要です。

Linux コマンド シェル

Docker コンテナは、最初に Linux 向けに Linux 上で開発されました。このため、Docker に使用される主要なコマンドライン ツール (シェルとも呼びます) が Unix シェルであるのは自然なことです (Linux は Unix から派生して開発されたものです)。ほとんどの開発者によって使用されているのが、Bash シェルです。**Alpine** など、一部の軽量の Linux ディストリビューションには Bash はインストールされていないため、この場合はより単純な Bourne シェル (単に「**sh**」とも呼びます) を使用する必要があります。コンテナ内や Linux VM 上など、Linux 環境で作業するときはいつでも、`/bin/bash` または `/bin/sh` のうち、使用可能なほうを使用することになります。

macOS X は Linux OS ではありませんが、Linux と OS X はどちらも Unix のフレーバーであり、同じ種類のツールをサポートしています。シェルも、こうしたツールの 1 つです。つまり、macOS で作業する場合は、通常は Bash シェルを使用することになります。

本書では、読者が Bash、あるいは PowerShell (Windows を使用している場合) の最も基本的なスクリプト コマンドについて精通していることを想定しています。まったくの未経験であれば、以下のクイックガイドを参考にして、必要な知識を身に付けておくことを強く推奨します。

- 『Linux Command Line Cheat Sheet』Dave Child　著 (`http://bit.ly/2mTQr8l`)
- 『PowerShell Basic Cheat Sheet』(`http://bit.ly/2EPHxze`)

Windows 用 PowerShell

Windows のコンピューター、ノート PC、またはサーバーには、いくつものコマンドライン ツールが用意されています。最も一般的なのはコマンド シェルです。コマンド シェルは何十年もの間、あらゆる Windows コンピューター上で利用されてきました。非常にシンプルなシェルです。より高度なスクリプト作成のためにマイクロソフトが開発したのが、PowerShell です。PowerShell は非常に強力なシェルで、Windows で作業するエンジニアの間で高い人気を得ています。Windows 10 になり、マイクロソフトが満を持して公開した **Windows Subsystem for Linux** では、Bash や Bourne シェルなど、あらゆる Linux ツールを使用できます。これ以外にも、Git Bash シェルなど、Windows 上に Bash シェルをインストールするための他のツールもあります。本書では、すべてのコマンドで Bash 構文を使用しますが、ほとんどのコマンドは PowerShell でも実行できます。

したがって、Windows 上で Docker を使用する場合は、PowerShell か、または他のいずれかの Bash ツールを使用することをお勧めします。

パッケージ マネージャーの使用

macOS または Windows ノート PC に最も簡単にソフトウェアをインストールする方法は、優れたパッケージ マネージャーを使用することです。macOS で最も一般的に使用されているのは **Homebrew** であり、Windows の場合は **Chocolatey** が最適です。

Homebrew の macOS へのインストール

Homebrew は、簡単な手順で macOS にインストールできます。`https://brew.sh/` の指示に従ってください。

Homebrew をインストールするコマンドは次のとおりです。

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install)"
```

インストールが完了したら、ターミナルに「brew --version」と入力して、Homebrew が動作しているかどうかをテストします。次のように表示されるはずです。

```
$ brew --version

Homebrew 1.4.3

Homebrew/homebrew-core (git revision f4e35; last commit 2018-01-11)
```

これで、Homebrew を使用してツールとユーティリティをインストールする準備が整いました。たとえば Vi テキスト エディターをインストールするには、次のように入力します。

```
$ brew install vim
```

これで、エディターがダウンロードされ、インストールされます。

Chocolatey の Windows へのインストール

Chocolatey パッケージ マネージャーを Windows にインストールするには、https://chocolatey.org/ の指示に従うか、または、自分が管理者として実行している PowerShell ターミナル上で、次のコマンドを実行します。

```
PS> Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.
ps1'))
```

Chocolatey のインストールが完了したら、パラメーターを付けずに choco コマンドを実行してテストします。次のような出力が表示されるはずです。

```
PS> choco
Chocolatey v0.10.3
```

Vi エディターなどのアプリケーションをインストールするには、次のコマンドを使用します。

```
PS> choco install -y vim
```

-y パラメーターを指定すると、再確認が要求されることなくインストールが実行されます。Chocolatey によるアプリケーションのインストールが完了したら、このアプリケーションを使用するための新しい PowerShell ウィンドウを開く必要があります。

コード エディターの選択

Docker を生産的に活用するためには、優れたコード エディターの使用が不可欠です。もちろん、どのエディターが最も優れているかについては判断の難しいところであり、個人の好みも影響します。一例を挙げると、Vim をはじめ、Emacs、Atom、Sublime、**Visual Studio** (**VS**) Code などのエディターは多くの人に使用されています。自分にとってどのエディターが最適かどうか決めかねているのであれば、VS Code を試してみることを強くお勧めします。VS Code は無料で軽量のエディターでありながら、非常に強力で、macOS、Windows、Linux に対応します。ぜひお試しください。VS Code は `https://code.visualstudio.com/download` からダウンロードできます。

既に愛用のコード エディターがある場合は、そちらを使用してかまいません。テキスト ファイルが編集できるのであれば、それで問題ありません。Dockerfiles、JSON、YAML ファイルに対して構文の強調表示機能をサポートしているエディターであれば、さらに理想的です。

Docker Toolbox

Docker Toolbox は、開発の場でここ数年使用され続けているツールです。Docker Toolbox は、Docker for macOS、Docker for Windows といった最近のツールの先駆けです。このツールボックスを使用すると、あらゆる macOS または Windows コンピューター上で、コンテナをきわめて効率的に操作できます。コンテナは Linux ホスト上で実行する必要があります。Windows も macOS も、コンテナをネイティブに実行することはできません。したがって、ノート PC で Linux VM を実行し、その上でコンテナを実行する必要があります。Docker Toolbox により、ノート PC に VirtualBox がインストールされます。VirtualBox は、ここで必要となる Linux VM を実行するために使用します。

Windows ユーザーであれば、Windows 上でネイティブに実行する「Windows コンテナ」の存在について耳にしたことがあるかもしれません。これは事実です。近年、マイクロソフトは Docker エンジンを Windows に移植しました。この結果、現在では VM を使用しなくても、Windows Server 2016 上で Windows コンテナを直接実行できるようになりました。したがって、現在では Linux コンテナ、Windows コンテナの 2 種類のフレーバーが存在することになります。前者は Linux ホスト上のみで実行でき、後者は Windows Server 上のみで実行できます。本書では Linux コンテナに限定して解説していますが、本書の学習内容のほとんどは Windows コンテナにも当てはまります。

では、`docker-machine` を使用して環境を設定しましょう。まず、システムで現在定義されている Docker 対応 VM をすべて列挙します。Docker Toolbox をインストールした直後であれば、次のような出力が表示されます。

```
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
default   -        virtualbox   Running   tcp://192.168.99.100:2376           v18.04.0-ce
$
```

すべての Docker 対応 VM のリスト

実際の IP アドレスはこの出力とは異なるかもしれませんが、`192.168.0.0/24` の範囲内であるはずです。また、この VM にインストールされている Docker のバージョンが `18.04.0-ce` であることが分かります。

何らかの理由で既定の VM が存在しない場合、または VM を誤って削除してしまった場合は、次のコマンドを使用して VM を作成します。

```
$ docker-machine create --driver virtualbox default
```

次のような出力が表示されます。

```
$ docker-machine create --driver virtualbox default
Running pre-create checks...
Creating machine...
(default) Copying /Users/gabriel/.docker/machine/cache/boot2docker.iso to /Users/gabriel/.docker/machine/
machines/default/boot2docker.iso...
(default) Creating VirtualBox VM...
(default) Creating SSH key...
(default) Starting the VM...
(default) Check network to re-create if needed...
(default) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docke
r-machine env default
$
```

VirtualBox による既定の VM の作成

Docker クライアントを、この仮想マシン上で実行されている Docker エンジンに接続する流れを確認するには、次のコマンドを実行します。

```
$ docker-machine env default
```

default として定義した VM の準備が整ったら、この VM への SSH 接続を試します。

```
$ docker-machine ssh default
```

このコマンドを実行すると、boot2docker によるウェルカム メッセージが表示されます。

次のように、コマンド プロンプトで「docker --version」と入力します。

```
docker@default:~$ docker --version
Docker version 18.06.1-ce, build e68fc7a
```

次に、コンテナを実行してみましょう。

```
docker@default:~$ docker run hello-world
```

このコマンドを実行すると、次のような出力が表示されます。

```
docker@default:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

docker@default:~$
```

Docker の Hello World コンテナの実行

Docker for macOS と Docker for Windows

使用しているのが macOS である場合、またはノート PC に Windows 10 Professional がインストールされている場合は、それぞれ Docker for macOS または Docker for Windows をインストールすることを強く推奨します。それぞれのツールを使用することで、コンテナを最大限に活用できます。ただし、旧バージョンの Windows、または Windows 10 Home Edition では Docker for Windows を実行できないので注意が必要です。Docker for Windows では、VM 内でコンテナを透過的に実行するために Hyper-V を使用しますが、Hyper-V は旧バージョンの Windows、または Windows Home Edition には対応していません。

Docker for macOS のインストール

次のリンクから、Docker for macOS をダウンロードします。`https://docs.docker.com/docker-for-mac/install/`

このツールには、Stable バージョン (安定版) と Edge バージョン (最新機能を搭載した評価版) の 2 種類があります。本書ではいくつかの最新機能と Kubernetes を使用しますが、これらは、本書の執筆時点では Edge バージョンのみで利用できます。このため、Edge バージョンを選択してください。

インストールを開始するには、以下の手順を実行します。

1. **[Get Docker for Mac (Edge)]** ボタンをクリックし、画面の指示に従います。
2. Docker for macOS のインストールが正常に完了したら、ターミナルを開きます。Command キーとスペースバーを同時に押して Spotlight を開き、「`terminal`」と入力して、Enter キーを押します。次のように、Apple のターミナルが開きます。

Apple ターミナル ウィンドウ

3. コマンド プロンプトで「`docker --version`」と入力し、Enter キーを押します。Docker for macOS が正しくインストールされていれば、次のような出力が表示されます。

```
$ docker -version
Docker version 18.02.0-ce-rc2, build f968a2c
```

4. コンテナを実行できるかどうか確認するには、ターミナルに次のコマンドを入力して、Enter キーを押します。

```
$ docker run hello-world
```

特に問題がない場合は、次のような出力が表示されます。

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

$
```

Docker for macOS での Hello World コンテナの実行

これで、Docker コンテナで作業する準備がすべて整いました。

Docker for Windows のインストール

Docker for Windows は、Windows 10 Professional または Windows Server 2016 にしかインストールできないことに注意してください。Docker for Windows には Hyper-V が必要となりますが、これは旧バージョンの Windows または Windows 10 Home Edition には対応していません。Windows 10 Home Edition または旧バージョンの Windows を使用している場合は、Docker Toolbox を使用する必要があります。

1. 次のリンクから、Docker for Windows をダウンロードします。`https://docs.docker.com/docker-for-windows/install/`

このツールには、Stable バージョン (安定版) と Edge バージョン (最新機能を搭載した評価版) の 2 種類があります。本書ではいくつかの最新機能と Kubernetes を使用しますが、これらは、本書の執筆時点では Edge バージョンのみで利用できます。このため、Edge バージョンを選択してください。

2. インストールを開始するには、**[Get Docker for Windows (Edge)]** ボタンをクリックして、画面の指示に従います。Docker for Windows では、Linux コンテナと Windows コンテナのどちらも開発、実行、およびテストできます。ただし本書では、Linux コンテナのみについて解説します。
3. Docker for Windows が正常にインストールできたら、PowerShell ウィンドウを開き、コマンド プロンプトに「`docker --version`」と入力します。次のような出力が表示されます。

```
PS> docker --version
Docker version 18.04.0-ce, build 3d479c0
```

Hyper-V を搭載した Windows 上での docker-machine の使用

Docker for Windows がインストールされている Windows ノート PC では、Hyper-V も有効化されています。この場合は VirtualBox が使用されるので、Docker Toolbox は使用できません。Hyper-V と VirtualBox は共存できず、同時に実行できないためです。この場合は、Hyper-V ドライバーとともに `docker-machine` を使用できます。

1. PowerShell コンソールを管理者として開きます。次のように、Chocolatey を使用して `docker-machine` をインストールします。

```
PS> choco install -y docker-machine
```

2. Window の Hyper-V マネージャーを使用して、「DM Internal Switch」という名前の新しい内部スイッチを作成します。「DM」は「**docker-machine**」の頭文字です。
3. 次のコマンドを使用して、Hyper-V で「default」という名前の VM を作成します。

```
PS> docker-machine create --driver hyperv --hyperv-virtual-
switch "DM Internal Switch" default
```

上記のコマンドは管理者モードで実行する必要があり、そうでない場合はコマンドが失敗します。

前述のコマンドを実行することで、次のような出力が表示されます。

```
Running pre-create checks...(boot2docker) Image cache directory
does not exist, creating it at C:\Users\Docker\.docker\machine\
cache...(boot2docker) No default Boot2Docker ISO found locally,
downloading the latest release...(boot2docker) Latest release
for github.com/boot2docker/boot2docker is v18.06.1-ce
....
....
Checking connection to Docker...Docker is up and running!To see
how to connect your Docker Client to the Docker Engine running
on this virtual machine, run: C:\Program Files\Doc ker\Docker\
Resources\bin\docker-machine.exe env default
```

4. Docker クライアントを、この仮想マシン上で実行されている Docker エンジンに接続する流れを確認するには、次のコマンドを実行します。

```
C:\Program Files\Docker\Docker\Resources\bin\docker-machine.
exe env default
```

5. docker-machine によって生成されたすべての VM を一覧表示すると、次のような出力が表示されます。

```
PS C:\WINDOWS\system32> docker-machine ls
NAME          ACTIVE   DRIVER   STATE     URL
SWARM   DOCKER        ERRORS
default .-        hyperv   Running   tcp://[...]:2376
v18.06.1-ce
```

6. 次に、作成した boot2docker VM に対して SSH を実行してみましょう。

```
PS> docker-machine ssh default
```

ウェルカム メッセージが表示されます。

`docker version` コマンドを実行して VM をテストします。この結果は次のようになります。

```
docker@default:~$ docker version
Client:
 Version:           18.06.1-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        e68fc7a
 Built:             Tue Aug 21 17:20:43 2018
 OS/Arch:           linux/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.1-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       e68fc7a
  Built:            Tue Aug 21 17:28:38 2018
  OS/Arch:          linux/amd64
  Experimental:     false
docker@default:~$
```

Docker クライアント (CLI) およびサーバーのバージョン

「`OS/Arch`」の項目で分かるように、これは明らかに Linux VM であり、Docker `18.06.1-ce` がインストールされています。

Minikube

Docker for macOS または Docker for Windows を使用できない場合、あるいは何らかの理由で、Kubernetes 非対応の古いバージョンのツールしか使用できない場合には、Minikube をインストールする方法をお勧めします。Minikube を使用すると、ワークステーション上に単一ノードの Kubernetes クラスターがプロビジョニングされます。これには、Kubernetes での作業に使用されるコマンドライン ツールである **kubectl** を使用してアクセスします。

macOS と Windows での Minikube のインストール

macOS または Windows 用に Minikube をインストールするには、次のリンクに移動します。`https://kubernetes.io/docs/tasks/tools/install-minikube/`

画面の指示をよく読み、これに従ってください。Docker Toolbox をインストールしている場合は、システムには既にハイパーバイザーが入っています。Docker Toolbox インストーラーによって VirtualBox もインストールされているためです。そうでない場合は、まず VirtualBox をインストールすることをお勧めします。

Docker for macOS または Docker for Windows をインストールしている場合は、`kubectl` も同時にインストールされているはずです。したがって、このステップも省略できます。それ以外の場合は、サイト上の指示に従います。

最後に、macOS または Windows 用 Minikube の最新バイナリを選択してインストールします。macOS 用の最新バイナリは「`minikube-darwin-amd64`」であり、Windows 用の最新バイナリは「`minikube-windows-amd64`」という名称です。

Minikube と kubectl のテスト

Minikube がワークステーションに正常にインストールされたら、ターミナルを開いてインストールをテストします。

1. まず、Minikube を起動する必要があります。コマンド ラインに「`minikube start`」と入力します。次のような出力が表示されます。

```
Starting local Kubernetes v1.9.0 cluster...
Starting VM...
Downloading Minikube ISO
 142.22 MB / 142.22 MB [============================================] 100.00% 0s
Getting VM IP address...
Moving files into cluster...
Downloading localkube binary
 162.41 MB / 162.41 MB [============================================] 100.00% 0s
 0 B / 65 B [----------------------------------------------------------]   0.00%
 65 B / 65 B [==============================================================] 100.00% 0sSetting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
$
```

Minikube の起動

2. 次に、「`kubectl version`」と入力して Enter キーを押すと、次のような画面が表示されます。

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"9", GitVersion:"v1.9.0", GitCommit:"925c127ec6b946659ad0fd596fa959be43f0cc05"
, GitTreeState:"clean", BuildDate:"2017-12-15T21:07:38Z", GoVersion:"go1.9.2", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"", Minor:"", GitVersion:"v1.9.0", GitCommit:"925c127ec6b946659ad0fd596fa959be43f0cc05",
GitTreeState:"clean", BuildDate:"2018-01-26T19:04:38Z", GoVersion:"go1.9.1", Compiler:"gc", Platform:"linux/amd64"}
$
```

Kubernetes クライアントおよびサーバーのバージョンの確認

タイムアウトなどの理由でこのコマンドが失敗した場合は、`kubectl` が適切なコンテキストに対して構成されていない可能性があります。`kubectl` は、異なる多数の Kubernetes クラスターに対して実行できます。個々のクラスターを、コンテキストと呼びます。

3. 現在 kubectl がどのコンテキストに対して構成されているのかを確認するには、次のコマンドを実行します。

```
$ kubectl config current-context minikube
```

上の出力が示すように、現在のコンテキストは minikube となります。

4. そうでない場合は、kubectl config get-contexts コマンドを使用して、システムで定義されているすべてのコンテキストを一覧表示します。その後、次のコマンドを使用して、現在のコンテキストを minikube に設定します。

```
$ kubectl config use-context minikube
```

kubectl の構成には各コンテキストが保管されており、通常、この構成の場所は ~/.kube/config となります。この場所を、環境変数「KUBECONFIG」を定義することで上書きすることもできます。コンピューターに既にこの変数が設定されている場合は、この変数の設定を解除する必要があります。

Kubernetes コンテキストの詳細な構成方法および使用方法については、次のリンクを参照してください。https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/

Minikube と kubectl がいずれも適切に機能していると想定して、次に kubectl を使用して、Kubernetes クラスターについての情報を収集します。

5. 次のコマンドを入力します。

```
$ kubectl get nodes
NAME        STATUS    ROLES     AGE       VERSION
minikube    Ready     <none>    47d       v1.9.0
```

出力内容から明らかなように、単一ノードによる 1 つのクラスターがあり、このクラスターには Kubernetes v1.9.0 がインストールされています。

ソース コード リポジトリの複製

本書で使用しているソースコードは、GitHub のリポジトリで一般公開されています。次のリンクを参照してください。https://github.com/appswithdockerandkubernetes/labs このリポジトリを、ローカル マシンに複製します。

最初に、ホーム フォルダ内に新規フォルダを作成します。たとえば、apps-with-docker-and-kubernetes のようなフォルダを作成し、このフォルダに移動します。

```
$ mkdir -p ~/apps-with-docker-and-kubernetes \   cd apps-with-
docker-and-kubernetes
```

このフォルダで次のコマンドを実行して、リポジトリのクローンを作成します。

```
$ git clone https://github.com/appswithdockerandkubernetes/labs.git
```

まとめ

この章では、Docker コンテナで生産的に作業できるように、個人的な環境または作業環境を設定および構成しました。この手順は、開発者、DevOps、運用エンジニアのどの場合でもまったく同じです。このコンテキストで、適切なエディターを使用し、Docker for macOS または Docker for Windows をインストールしました。また、`docker-machine` を使用して VirtualBox または Hyper-V 内に VM を作成し、これらを使用してコンテナを実行およびテストしました。

次の章では、コンテナに関するさまざまな重要事項について学習します。たとえば、コンテナを実行、停止、一覧表示、および削除する手順を確認しますが、さらに、コンテナの構造や成り立ちについても詳細に学習します。

質問

この章で得られた知識に基づいて、以下の質問にお答えください。

1. `docker-machine` は何に使用されますか。考えられる用途を 3 ～ 4 つ挙げてください。
2. 次の文は正しいでしょうか、間違っているでしょうか。Docker for Windows では、Linux コンテナを開発および実行できます。
3. コンテナを生産的に使用するには、優れたスクリプト スキル (Bash、PowerShell など) が不可欠となるのはなぜですか。
4. Docker が動作保証されている Linux ディストリビューションの名前を 3 ～ 4 つ挙げてください。
5. Windows コンテナを実行できる Windows バージョン名をすべて挙げてください。

参考情報

理解を深めるために、次のリンクも参照してください。

- Docker Machine を使用して Hyper-V で Docker を実行する (`http://bit.ly/2HGMPiI`)

3
コンテナの操作

前の章では、Docker を生産的かつスムーズに利用するために最適な作業環境を準備する方法を学習しました。この章では、実際に操作しながら、コンテナを操作する上で重要なあらゆることを学びます。この章で説明するトピックは次のとおりです。

- 最初のコンテナの実行
- コンテナの開始、停止、削除
- コンテナの検査
- 稼働中のコンテナに移動
- 稼働中のコンテナへの接続
- コンテナ ログの取得
- コンテナの徹底分析

この章を終了すると、次のことができるようになります。

- NGINX、busybox、alpine などの既存のイメージに基づいてコンテナを実行、停止、および削除する
- システム上のすべてのコンテナを一覧表示する
- 稼働中または停止中のコンテナのメタデータを検査する
- コンテナ内で稼働中のアプリケーションによって生成されたログを取得する
- 既に稼働中のコンテナ内で `/bin/sh` などのプロセスを実行する
- 既に稼働中のコンテナにターミナルを接続する
- 関心を持つ初心者にコンテナの基礎を自分の言葉で説明する

技術的要件

この章は、Docker for Mac または Docker for Windows がインストールされている必要があります。旧バージョンの Windows を使用している場合や Windows 10 Home Edition を使用している場合は、Docker Toolbox がインストールされて使用できる状態になっている必要があります。macOS ではターミナル アプリケーションを、Windows では PowerShell コンソールを使用して、学習するコマンドを試すことになります。

最初のコンテナの実行

最初に、Docker がシステムに正しくインストールされていて、コマンドを受け入れる準備ができていることを確認します。新しいターミナル ウィンドウを開き、次のコマンドを入力します。

```
$ docker -v
```

正常に動作すれば、ノート PC にインストールされている Docker のバージョンがターミナルに表示されます。執筆時点では、次のように表示されます。

```
Docker version 17.12.0-ce-rc2, build f9cde63
```

このように表示されない場合はインストールが正常に行われていません。前の章の Docker for Mac または Docker for Windows をシステムにインストールする手順を確認してください。

では、実際のアクションを見てみましょう。ターミナル ウィンドウに次のコマンドを入力し、リターンを押してください。

```
$ docker container run alpine echo "Hello World"
```

このコマンドを実行すると、ターミナル ウィンドウに次のような出力が表示されます。

```
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
2fdfe1cd78c2: Pull complete
Digest: sha256:ccba511b...
Status: Downloaded newer image for alpine:latest
Hello World
```

簡単ですね。では、まったく同じコマンドをもう一度実行してみましょう。

```
$ docker container run alpine echo "Hello World"
```

上記のコマンドを 2 回、3 回と繰り返し実行していくと、ターミナルにはこの出力だけが表示されるようになります。

```
Hello World
```

最初にコマンドを実行したときと、それ以降のすべての場合で表示される出力が異なる理由を考えてみてください。わからなくても気にしないでください。この章の今後のセクションで詳しく説明します。

コンテナの開始、停止、削除

前のセクションでは、コンテナを正常に実行することができました。では、何が起こっていたのか、そしてそれはどうしてなのかについて詳しく説明します。先ほどのコマンドをもう一度見てみましょう。

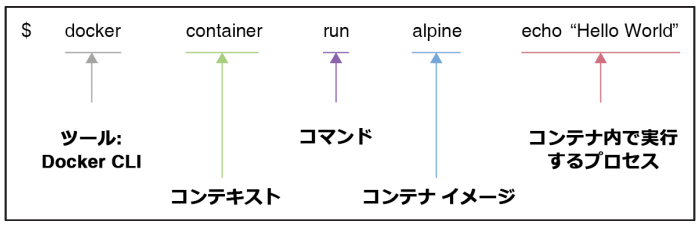
```
$ docker container run alpine echo "Hello World"
```

このコマンドは複数の部分から構成されています。まず、`docker` という単語があります。これが Docker の**コマンド ライン インターフェイス** (CLI) の名前です。これを使用することで、コンテナの実行を担当する Docker エンジンとやり取りします。次に、`container` という単語があります。これは、作業を行っているコンテキストを示しています。コンテナを実行するので、コンテキストは `container` という単語になります。その次は、特定のコンテキストで稼働する実際のコマンドです。すなわち、`run` です。

要するに、`docker container run` という式を実行したわけです。「Docker、コンテナを実行しなさい ...」ということです。

次に、Docker に実行するコンテナを指示する必要があります。この場合、いわゆる `alpine` コンテナと呼ばれるものです。最後に、稼働中のコンテナ内で稼働するプロセスやタスクの種類を定義しなければなりません。ここでは、コマンドの最後の `echo "Hello World"` になります。

これらすべてを次の図にまとめました。



docker container run の式の分析

コンテナを実行するコマンドの各部が理解できたところで、別のコンテナを実行してその内部で別のプロセスを実行してみましょう。ターミナルに次のコマンドを入力します。

```
$ docker container run centos ping -c 5 127.0.0.1
```

ターミナル ウィンドウに次のような出力が表示されます。

```
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
85432449fd0f: Pull complete
Digest: sha256:3b1a65e9a05...
Status: Downloaded newer image for centos:latest
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.022 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.019 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.029 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.030 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.029 ms

--- 127.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4103ms
rtt min/avg/max/mdev = 0.021/0.027/0.029/0.003 ms
```

先ほどと違う点は、今回は使用するコンテナ イメージが `centos` であること、そして `centos` コンテナ内で稼働するプロセスが `ping -c 5 127.0.0.1` であるということです。すなわち、ループバック アドレスを 5 回 ping して、終了するプロセスです。

出力を詳しく見て見ましょう。

- 最初の行は次のとおりです。

  ```
  Unable to find image 'centos:latest' locally
  ```

  これは、Docker がシステムのローカル キャッシュに `entos:latest` という名前のイメージを見つけられなかったことを示しています。すなわち、コンテナ イメージの格納されたレジストリからイメージを取得する必要があると Docker が認識しているということです。既定では、すべてのイメージを `docker.io` の Docker Hub から取得するように Docker 環境が設定されています。このことは、2 行目で次のように表現されています。

  ```
  latest: Pulling from library/centos
  ```

- その後の 3 行の出力は次のとおりです。

```
85432449fd0f: Pull complete
Digest: sha256:3b1a65e9a05...
Status: Downloaded newer image for centos:latest
```

  これは、Docker が Docker Hub からイメージ centos:latest を正常に取得したことを示しています。

それ以降のすべての出力行は、コンテナ内で稼働したプロセスによって生成されています。この場合は ping ツールです。`latest` というキーワードが繰り返し使われていることにお気付きの人もいるかもしれません。各イメージにはバージョン (タグとも呼ばれます) があり、バージョンが明示的に指定されていなければ、Docker は自動的に最新のバージョンを想定します。

先ほどのコンテナをこのシステムでもう一度実行すると、最初の 5 行の出力が表示されなくなります。Docker はローカルにキャッシュされたコンテナ イメージを使用できるので、コンテナ イメージをダウンロードする必要がないからです。実際にやってみましょう。

ランダム引用コンテナの実行

この章のこれ以降のセクションでは、バックグラウンドで連続的に動作し、興味深い出力を生成するコンテナが必要になります。そのため、引用をランダムに生成するアルゴリズムを選択しました。これらの無料のランダムな引用を生成する API は、`https://talaikis.com/random_quotes_api/` にあります。

ここでは、5 秒ごとに新しいランダムな引用を生成して STDOUT に出力するプロセスをコンテナ内で実行してみましょう。次のスクリプトでそれを行うことができます。

```
while :
do
    wget -qO- https://talaikis.com/api/quotes/random
    printf 'n'
    sleep 5
done
```

ターミナル ウィンドウで試してみましょう。スクリプトを停止するには、Ctrl+C を押してください。出力は次のようなものになります。

```
{"quote":"Martha Stewart is extremely talented.Her designs are
picture perfect.Our philosophy is life is messy, and rather than
being afraid of those messes we design products that work the way we
live.","author":"Kathy Ireland","cat":"design"}
```

```
{"quote":"We can reach our potential, but to do so, we must
reach within ourselves.We must summon the strength, the will,
and the faith to move forward - to be bold - to invest in our
future.","author":"John Hoeven","cat":"faith"}
```

それぞれの応答は JSON フォーマットの文字列になっており、引用、その作成者およびカテゴリが含まれています。

では、これを `alpine` コンテナ内でデーモンとしてバックグラウンドで実行してみましょう。そのためには、先ほどのスクリプトをワンライナーに圧縮し、それを `/bin/sh -c "..."` の構文を使用して実行する必要があります。Docker の式は次のようになります。

```
$ docker container run -d --name quotes alpine \
   /bin/sh -c "while :; do wget -qO- https://talaikis.com/api/quotes/
random; printf '\n'; sleep 5; done"
```

この式では、新しいコマンドライン パラメーターを 2 つ使用しています。すなわち、`-d` と `--name` です。`-d` は、コンテナ内で稼働中のプロセスを Linux デーモンとして実行するように Docker に指示するものです。さらに、`--name` パラメーターを使用して、そのコンテナに明示的な名前を付けることができます。先ほどのサンプルでは、`quotes` という名前を付けています。

コンテナの実行時にコンテナ名が明示的に指定されていない場合は、そのコンテナにランダムな一意の名前が Docker によって自動的に割り当てられます。その名前は有名な科学者の名前と形容詞から構成されたものになります。たとえば、`boring_borg` や `angry_goldberg` などです。当社の Docker エンジニアはかなりユーモアがあるのではないでしょうか。

重要なポイントは、システム内でコンテナ名が一意になる必要があるということです。quotes コンテナが起動していることを確認してみましょう。

```
$ docker container ls -l
```

次のようになるはずです。

```
$ docker container ls -l
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS      NAMES
6ce5e46da7ce        alpine              "/bin/sh -c 'while :…"   41 seconds ago      Up 16 seconds                  quotes
$
```

最後に実行したコンテナの一覧表示

この出力の重要な部分は、`STATUS` 列です。ここでは、`Up 16 seconds` となっています。すなわち、このコンテナは 16 秒間稼働しているということです。

最後の Docker コマンドについてはわからなくても気にしないでください。次のセクションで説明します。

コンテナの一覧表示

コンテナの実行を続けていくと、システム内にたくさんコンテナができます。ホスト上で現在稼働中のコンテナを確認するには、コンテナ `list` コマンドを次のように使用できます。

```
$ docker container ls
```

これによって、現在稼働中のすべてのコンテナが一覧表示されます。たとえば、次のような一覧になります。

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
31d719b2f439        nginx:alpine        "nginx -g 'daemon of…"   35 seconds ago      Up 30 seconds       80/tcp              cranky_curie
27b96de70b58        alpine:latest       "ping 127.0.0.1"         23 hours ago        Up 23 hours                             c2
35b8dd512acb        alpine:latest       "/bin/sh"                23 hours ago        Up 23 hours                             c1
$
```

システム上で稼働中のすべてのコンテナの一覧

既定では、7 つの列が Docker から出力されます。それらの意味は次のとおりです。

列	説明
`Container ID`	コンテナの一意の ID。SHA-256 です。
`Image`	このコンテナのインスタンス化の元となっているコンテナ イメージの名前。
`Command`	コンテナ内でメイン プロセスを実行するために使用されるコマンド。
`Created`	コンテナが作成された日時。
`Status`	コンテナのステータス (created、restarting、running、removing、paused、exited、dead)。
`Ports`	ホストにマッピングされているコンテナ ポートの一覧。
`Names`	このコンテナに割り当てられている名前 (複数の名前も可能)。

現在稼働中のコンテナだけではなくmシステム上で定義されているすべてのコンテナを一覧表示する場合は、次のように、コマンドライン パラメーターの `-a` または `--all` を使用できます。

```
$ docker container ls -a
```

これで、`created`、`running`、`exited` などのすべての状態のコンテナが一覧表示されます。

すべてのコンテナの ID だけを一覧表示したい場合もあります。それには、次のようにパラメーターの `-q` を使用します。

```
$ docker container ls -q
```

どういうときに使うのだろうと思うかもしれません。このパラメーターを利用した次のコマンドの例は、次のとおりです。

```
$ docker container rm -f $(docker container ls -a -q)
```

ゆっくりと見てみてください。このコマンドは何をしているのでしょうか。しばらく考えてみましょう。

正解 : このコマンドはシステム上で現在定義されているすべてのコンテナを削除しています。`rm` コマンドは削除を意味しています。これについては後で詳しく説明します。

前のセクションでは、リスト コマンドで `-l` パラメーターを使用しました。Docker のヘルプで `-l` パラメーターの意味を調べてみてください。リスト コマンドのヘルプは次のようにして呼び出すことができます。

```
$ docker container ls -h
```

コンテナの停止と起動

稼働中のコンテナを (一時的に) 停止させたい場合もあります。以前に使用した quotes コンテナで試してみましょう。次のコマンドを使用して、コンテナを再度実行します。

```
$ docker container run -d --name quotes alpine \
   /bin/sh -c "while :; do wget -qO- https://talaikis.com/api/quotes/
random; printf '\n'; sleep 5; done"
```

このコンテナを停止する場合は、次のコマンドを発行することで停止できます。

```
$ docker container stop quotes
```

quotes コンテナを停止しようとすると、このコマンドが実行されるまで少し時間がかかることに気付くことでしょう。正確には、約 10 秒かかります。これはなぜでしょうか。

Docker は Linux `SIGTERM` シグナルをコンテナの内部で稼働しているメイン プロセスに送信します。プロセスがこのシグナルに応答せずに終了した場合、Docker は 10 秒間待機してから `SIGKILL` を送信し、プロセスを強制的に終了してコンテナを終了させます。

先ほどのコマンドでは、コンテナの名前を使用して停止するコンテナを指定しました。しかし、代わりにコンテナ ID を使用することもできたのです。

コンテナの ID はどうすれば取得できるでしょうか。方法はいくつかあります。手動のアプローチとしては、稼働中のすべてのコンテナを一覧表示し、その一覧の中から目的のコンテナを見つける方法があります。そこから ID をコピーします。より自動的なアプローチとしては、いくつかのシェル スクリプトと環境変数を使用する方法があります。たとえば、quotes コンテナの ID を取得するなら、次の式を使用することができます。

```
$ export CONTAINER_ID=$(docker container ls | grep quotes | awk
'{print $1}')
```

これで、コンテナ名の代わりに、この式の `$ CONTAINER_ID` という変数を使用することができます。

```
$ docker container stop $CONTAINER_ID
```

コンテナを停止すると、そのステータスは `Exited` に変わります。

コンテナが停止している場合は、`docker container start` コマンドを使用して再起動することができます。quotes コンテナでやってみましょう。この章の次のセクションで使用しますので、再起動しておきましょう。

```
$ docker container start quotes
```

コンテナの削除

`docker container ls -a` コマンドを実行すると、`Exited` ステータスのコンテナが多数表示されます。これらのコンテナがもう必要ない場合は、メモリから削除しておくとよいでしょう。そうしないと、貴重なリソースが無駄に占有されます。コンテナを削除するコマンドは次のとおりです。

```
$ docker container rm <container ID>
```

次のコマンドもコンテナを削除します。

```
$ docker container rm <container name>
```

ID を使用して終了済みのコンテナを 1 つ削除してみましょう。

コンテナが稼働中で削除できないこともあります。現在の状態に関わらずコンテナを強制的に削除する場合は、コマンドライン パラメーターの `-f` または `--force` を使用できます。

コンテナの検査

コンテナはイメージのランタイム インスタンスであり、その動作を特徴付ける多くの関連データがあります。特定のコンテナに関する詳しい情報については、`inspect` コマンドを使用して取得できます。ここでも、コンテナ ID またはコンテナ名のいずれかを指定して、データを取得するコンテナを特定する必要があります。では、サンプル コンテナを検査してみましょう。

```
$ docker container inspect quotes
```

応答は、詳細な情報が記述された巨大な JSON オブジェクトになります。たとえば、次のようなものです。

```
[
    {
        "Id": "c5c1c68c87...",
        "Created": "2017-12-30T11:55:51.223271182Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "while :; do wget -qO- https://talaikis.com/api/
quotes/random; printf '\n'; sleep 5; done"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            ...
        },
        "Image": "sha256:e21c333399e0...",
        ...
        "Mounts": [],
        "Config": {
            "Hostname": "c5c1c68c87dd",
            "Domainname": "",
            ...
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID": "2fd6c43b6fe5...",
            ...
        }
    }
]
```

読みやすいように、出力は短縮されています。

少し時間を使って、取得した応答を分析してみてください。次のような情報が表示されているはずです。

- コンテナの ID
- コンテナの作成日時
- コンテナがどのイメージからビルドされているかなど

`Mounts` や `NetworkSettings` など、出力のセクションの多くは今のところ意味がよくわからないでしょうが、本書のこれからの章で説明します。ここに表示されているデータは、コンテナの**メタデータ**とも呼ばれています。`inspect` コマンドは、情報源として本書の中でこれから何度も使用することになります。

すべての情報のうちのごく一部だけが必要な場合もあります。そのためには、**grep ツール**や**フィルター**を使うことができます。前者の方法では求める結果が必ず得られるわけではありません。そこで、後者の方法を見ていきましょう。

```
$ docker container inspect -f "{{json .State}}" quotes | jq
```

フィルターの定義には、`-f` または `--filter` パラメーターを使用します。フィルターの式自体には、**Go テンプレート**構文を使用します。この例では、出力全体のうちで状態の部分だけを JSON 形式で表示しようとしています。

出力の形式を整えるために、結果を jq ツールでパイプします。

```
{
  "Status": "running",
  "Running": true,
  "Paused": false,
  "Restarting": false,
  "OOMKilled": false,
  "Dead": false,
  "Pid": 6759,
  "ExitCode": 0,
  "Error": "",
  "StartedAt": "2017-12-31T10:31:51.893299997Z",
  "FinishedAt": "0001-01-01T00:00:00Z"
}
```

稼働中のコンテナに移動

稼働中のコンテナ内で別のプロセスを実行したいこともあります。よくある理由としては、誤動作しているコンテナのデバッグがあります。どうすればよいでしょうか。まず、コンテナ ID またはコンテナ名を知る必要があります。それがわかると、実行するプロセスと実行する内容を定義できます。再び、現在稼働中の quotes コンテナを使用し、次のコマンドを使って内部でインタラクティブにシェルを実行します。

```
$ docker container exec -i -t quotes /bin/sh
```

この `-ii` フラグは、新しいプロセスをインタラクティブに実行するという意味です。`-t` は、コマンドの TTY (ターミナル エミュレーター) を使うことを Docker に指示します。最後に、実行するプロセスの `/bin/sh` があります。

ターミナルで上記のコマンドを実行すると、新しいプロンプトが表示されます。これで、quotes コンテナの内側のシェルに入りました。そのことは、たとえば、`ps` コマンドを実行すると簡単に確認できます。コンテキスト内で稼働中のすべてのプロセスが一覧表示されるからです。

```
# / ps
```

結果は次のようになります。

```
/ # ps
PID   USER     TIME   COMMAND
    1 root       0:00 /bin/sh -c while :; do wget -qO- https://talaikis.com/api
   85 root       0:00 /bin/sh
  110 root       0:00 sleep 5
  111 root       0:00 ps
```

quotes コンテナ内で稼働しているプロセスの一覧

`PID 1` のプロセスが quotes コンテナ内で稼働するように定義したコマンドであることが確認できます。 `PID 1` のプロセスはメイン プロセスとも呼ばれます。

プロンプトに `exit` と入力して、コンテナから脱出します。コンテナ内では、追加プロセスをインタラクティブに実行できるというだけではありません。次のコマンドについて考えてみてください。

```
$ docker container exec quotes ps
```

この出力は、先ほどの出力と非常によく似ているように見えます。

```
$ docker container exec quotes ps
PID   USER     TIME   COMMAND
    1 root       0:00 /bin/sh -c while :; do wget -qO- https://talaikis.com/api
  520 root       0:00 sleep 5
  521 root       0:00 ps
$
```

quotes コンテナ内で稼働しているプロセスの一覧

次のように、`-d` フラグを使用することでプロセスをデーモンとして実行し、`-e` フラグ変数を使用して環境変数を定義することもできます。

```
$ docker container exec -it \
    -e MY_VAR="Hello World" \
    quotes /bin/sh
# / echo $MY_VAR
Hello World
# / exit
```

稼働中のコンテナへの接続

`attach` コマンドを使用すると、コンテナの ID またはコンテナ名を使用して、ターミナルの標準入力、出力、エラー (またはこの 3 つの組み合わせ) を稼働中のコンテナに接続することができます。quotes コンテナでやってみましょう。

```
$ docker container attach quotes
```

この場合、出力中に新しい引用が約 5 秒ごとに表示されます。

停止や強制終了をせずにコンテナを終了するには、Ctrl+P と Ctrl+Q の組み合わせを押してください。これによって、バックグラウンドで動作させたままコンテナから抜けることができます。コンテナを抜けると同時にそのコンテナを停止する場合は、Ctrl+C を押してください。

別のコンテナを実行してみましょう。今度は Nginx Web サーバーです。

```
$ docker run -d --name nginx -p 8080:80 nginx:alpine
```

ここでは、nginx という名前のコンテナ内で Nginx の Alpine バージョンをデーモンとして実行します。-p 8080:80 コマンドライン パラメーターによってホスト上の 8080 ポートが開き、コンテナ内で稼働している Nginx Web サーバーにアクセスできます。構文については今は気にしないでください。この機能については、第 7 章のシングルホスト ネットワーキングで詳しく説明します。

Nginx にアクセスできるかどうかを試してみましょう。curl ツールを使用して、次のコマンドを実行します。

```
$ curl -4 localhost:8080
```

正常に動作すれば、Nginx ウェルカム ページが表示されるはずです。

```
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
installed and
working.Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

では、ターミナルを nginx コンテナに接続して、結果を見てみましょう。

```
$ docker container attach nginx
```

コンテナに接続されても、最初は何も表示されません。しかし、別のターミナルを開き、この新しいターミナル ウィンドウで次のスクリプトを使用して curl コマンドを数回実行してみてください。

```
$ for n in {1..10}; do curl -4 localhost:8080; done
```

次のように、Nginx のロギング出力が表示されるはずです。

```
172.17.0.1 - - [06/Jan/2018:12:20:00 +0000] "GET / HTTP/1.1" 200 612
"-" "curl/7.54.0" "-"
172.17.0.1 - - [06/Jan/2018:12:20:03 +0000] "GET / HTTP/1.1" 200 612
"-" "curl/7.54.0" "-"
172.17.0.1 - - [06/Jan/2018:12:20:05 +0000] "GET / HTTP/1.1" 200 612
"-" "curl/7.54.0" "-"
```

Ctrl+C を押して、コンテナを終了します。これでターミナルが切り離され、同時に nginx コンテナが停止します。

クリーンアップするために、次のコマンドを入力して nginx コンテナを削除します。

```
$ docker container rm nginx
```

コンテナ ログの取得

適切なアプリケーションのベスト プラクティスの 1 つは、開発者とオペレーターがどちらも同じように使用できるロギング情報を生成し、それによって、アプリケーションが特定の時間に何を行っているのかを知ったり、何か問題がないかどうかを知ったり、問題の根本原因を突き止めるのに役立てたりできることです。

コンテナ内で稼働中は、アプリケーションはログ項目をファイルではなく STDOUT と STDERR に出力すべきです。ロギング出力が STDOUT と STDERR に送られることで、Docker がその情報を収集して保管してそれをユーザーや他の外部システムが利用することができます。

特定のコンテナのログにアクセスするには、docker container logs コマンドを使用できます。たとえば、quotes コンテナのログを取得するなら、次の式を使用できます。

```
$ docker container logs quotes
```

これによって、アプリケーションが作成された時点からそのアプリケーションが生成したログをすべて取得できます。

ちょっと待ってください。先ほど述べた、コンテナが作成されてからのログをすべて入手できるというのは正確ではありませんでした。既定では、Docker は、いわゆる `JSON-file` ロギング ドライバーを使用します。このドライバーは、ロギング情報をファイルに保存します。また、ファイル ローリング ポリシーが定義されていたとしたら、`docker container logs` は現在のアクティブ ログ ファイル内の内容だけを取得します。ホスト上にまだあるかないかわからない、ローリングされた過去のファイル内の内容は取得しません。

最新のいくつかのエントリだけを取得する場合は、`-t` または `--tail` パラメーターを次のように使用します。

```
$ docker container logs --tail 5 quotes
```

これによって、生成されたコンテナ内で稼働中のプロセスの最後の 5 項目だけが取得されます。

コンテナによって生成されたログをフォローしたい場合もあります。それには、`-f` または `--follow` パラメーターを使用します。次の式を使うと、最後の 5 つのログ項目が出力され、その後にコンテナ化プロセスによって生成されたログが出力されます。

```
$ docker container logs --tail 5 --follow quotes
```

ロギング ドライバー

Docker には、稼働中のコンテナから情報を取得するのに役立つ複数のロギング メカニズムがあります。これらのメカニズムは、**ロギング ドライバー**と呼ばれています。どのロギング ドライバーを使用するかは Docker デーモン レベルで設定できます。既定のロギング ドライバーは `JSON-file` です。現在ネイティブでサポートされているドライバーの例は、次のとおりです。

ドライバー	説明
`none`	特定のコンテナのログ出力は生成されません。
`json-file`	既定のドライバーです。ロギング情報は JSON 形式でファイルに格納されます。
`journald`	ジャーナル デーモンがホストマシン上で稼働している場合は、このドライバーを使用できます。ロギングは `journald` デーモンに転送されます。
`syslog`	`syslog` デーモンがホスト マシン上で稼働している場合、このドライバーを設定できます。このドライバーは、ログ メッセージを `syslog` デーモンに転送します。
`gelf`	このドライバーを使用すると、ログ メッセージは **Graylog Extended Log Format (GELF)** エンドポイントに書き込まれます。そのようなエンドポイントの一般的な例には、Graylog や Logstash があります。
`fluentd`	このドライバーは、`fluentd` デーモンがホスト システムにインストールされていると仮定し、ログ メッセージをそこに書き込みます。

ロギング ドライバーを変更する場合は、docker container logs コマンドは JSON-file および journald ドライバーのみに利用できることに注意してください。

コンテナ固有のロギング ドライバーの使用

ロギング ドライバーを Docker デーモン構成ファイルにグローバルに設定できることがわかりました。しかし、ロギング ドライバーはコンテナ単位で定義することもできます。次の例では、busybox コンテナを実行し、--log-driver パラメーターを使用して none ロギング ドライバーを設定しています。

```
$ docker container run --name test -it \
    --log-driver none \
    busybox sh -c 'for N in 1 2 3; do echo "Hello $N"; done'
```

次のように表示されるはずです。

```
Hello 1
Hello 2
Hello 3
```

では、先ほどのコンテナのログを取得してみましょう。

```
$ docker container logs test
```

出力は次のとおりです。

```
Error response from daemon: configured logging driver does not support
reading
```

これが予想されたとおりの結果です。none ドライバーはロギング出力を生成しないからです。test コンテナをクリーンアップして削除しましょう。

```
$ docker container rm test
```

高度なトピック - 既定のロギング ドライバーの変更

Linux ホストの既定のロギング ドライバーを変更してみましょう。最も簡単な方法は、実際の Linux ホスト上で行うことです。そのために、Vagrant と Ubuntu イメージを使用します。

```
$ vagrant init bento/ubuntu-17.04
$ vagrant up
$ vagrant ssh
```

Ubuntu VM 内に入ると、Docker デーモン構成ファイルを編集します。`/etc/docker` フォルダに移動し、次のように `vi` を実行します。

```
$ vi daemon.json
```

次の内容を入力します。

```
{
  "Log-driver": "json-log",
  "log-opts": {
    "max-size": "10m",
    "max-file": 3
  }
}
```

ESC を押してから `:w:q` と入力し、最後に Enter キーを押して Vi を保存し、終了します。

先ほどの定義は、ローリングされるまでの最大ログ ファイル サイズが 10 MB で、最も古いファイルがパージされるまでにシステム上に存在できるログファイルの最大数が `3` の `JSON-log` ドライバーを使用するように Docker デーモンに指示しています。

ここで、`SIGHUP` シグナルを Docker デーモンに送信し、構成ファイルの変更を受け取る必要があります。

```
$ sudo kill -SIGHUP $(pidof dockerd)
```

先ほどのコマンドは構成ファイルの再読み込みのみを行います。デーモンの再起動は行わないことに注意してください。

コンテナの徹底分析

誤ってコンテナを VM になぞらえる人は多くいます。しかし、その対比は適切ではありません。コンテナはただの軽量な VM というわけではないのです。では、コンテナの正しい説明は何でしょうか。

コンテナは、ホストシステム上で稼働する、特別にカプセル化されて保護されたプロセスです。

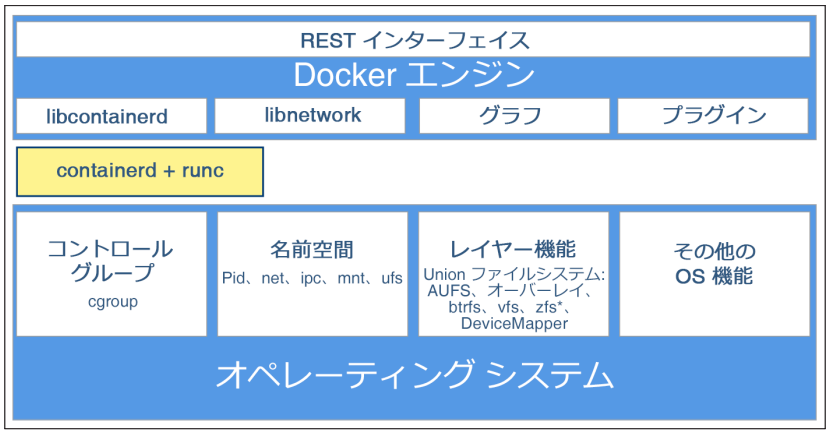
コンテナは、Linux OS で利用できる多くの機能とプリミティブを活用しています。最も重要なのは、**名前空間** と **cgroups** です。コンテナ内で稼働するすべてのプロセスは、基盤となるホスト オペレーティング システムの同じ Linux カーネルを共有します。これが VM と根本的に異なる点です。各 VM にはそれぞれに独自の完全なオペレーティング システムが含まれているからです。

一般的なコンテナの起動時間はミリ秒単位で測定できますが、VM は、通常、起動に数秒から数分かかります。VM は長寿命を前提としています。VM の稼働時間をできるだけ長くすることが各オペレーション エンジニアの大きな目標です。反対に、コンテナは一時的であることを前提としています。短い周期で出現して消失するものです。

まず、コンテナの実行を可能にするアーキテクチャの概要を見てみましょう。

アーキテクチャ

次の図は、全体の組み合わせの示すアーキテクチャ ダイアグラムです。



Docker のハイ レベル アーキテクチャ

この図の下の段は Linux オペレーティング システムで、cgroup、名前空間、レイヤー機能、および特にここで明記する必要のない他の機能を示しています。そして、**containerd** と **runc** から構成される中間層があります。一番上には Docker エンジンがあります。Docker エンジンは、Docker CLI、Docker for Mac、Docker for Window、Kubernetes などの任意のツールでアクセスできる RESTful インターフェイスを外部に提供します。

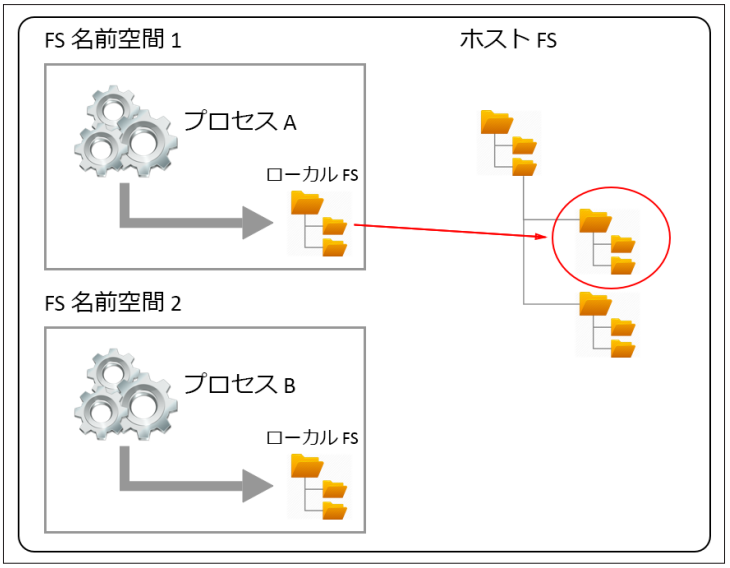
では、主なビルディング ブロックをもう少し詳しく説明しましょう。

名前空間

Linux 名前空間は Docker がコンテナに活用する何年も前から存在していました。名前空間は、ファイルシステム、ネットワーク アクセス、プロセス ツリー (PID 名前空間とも呼ばれます)、システム グループ ID、ユーザー ID などのグローバル リソースを抽象化したものです。Linux システムは、各名前空間タイプの単一インスタンスで初期化されます。初期化が終わると、追加の名前空間を作成したり結合したりすることができます。

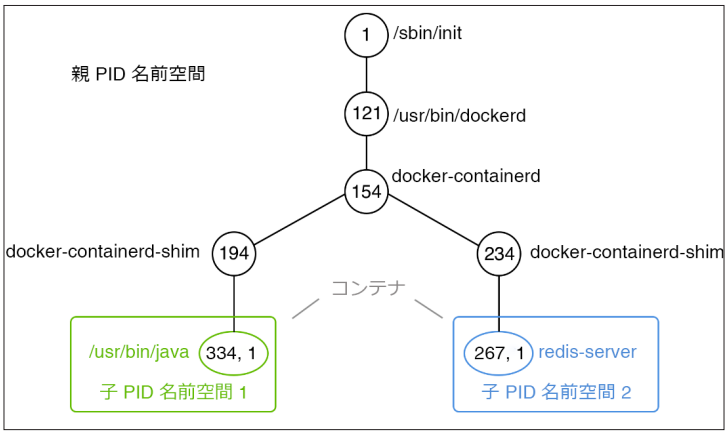
Linux の名前空間は 2002 年の 2.4.19 カーネルから採用されました。カーネルのバージョン 3.8 ではユーザー名前空間が導入され、それによってコンテナで名前空間を使用できるようになりました。

稼働中のプロセスを、たとえばファイル システムの名前空間にラッピングすると、そのプロセスは独自の完全なファイルシステムを所有しているかのように稼働できます。もちろん現実ではありません。仮想 FS にすぎません。ホストから見ると、含まれるプロセスが FS 全体のシールドされたサブセクションを取得していることになります。ファイルシステムの中にあるファイルシステムのようなものです。



このことは、そのための名前空間が存在する他のすべてのグローバル リソースにも当てはまります。別の例としては、ユーザー ID 名前空間があります。ユーザー名前空間があることで、独自の名前空間内にある間はユーザー `jdoe` をシステム内で何度も定義できるようになっています。

PID 名前空間は、あるコンテナ内のプロセスが別のコンテナ内のプロセスを参照することや、それとやり取りすることを防止します。あるプロセスがコンテナ内で見かけ上の PID **1** を持つことがありますが、ホストシステムから見ると、通常の PID、たとえば **334** を持っていることになります。



Docker ホスト上のプロセス ツリー

特定の名前空間では、1 つまたは複数のプロセスを実行できます。コンテナについて語るときにこのことは重要です。先ほど、既に稼働中のコンテナ内で別のプロセスを実行しましたね。

コントロール グループ (cgroups)

Linux の cgroup は、システム上で稼働中のプロセスのコレクションのリソース使用量を制限、管理、隔離するために使用されます。リソースとは、CPU 時間、システム メモリ、ネットワーク帯域幅、またはこれらリソースの組み合わせのことです。

Google のエンジニアは、もともと 2006 年からこの機能を実装していました。cgroup 機能は、2008 年 1 月にリリースされたカーネルのバージョン 2.6.24 で、Linux カーネル mainline に統合されました。

管理者は、cgroup を使用してコンテナの消費できるリソースを制限できます。それによって、たとえば、古くからある noisy neighbor (迷惑な隣人) 問題を防止することができます。すなわち、コンテナ内で稼働している不正なプロセスがすべての CPU 時間を消費したり膨大な量の RAM を予約したりすることによって、ホスト上で稼働している他のプロセスを、それらがコンテナ化されているかどうかに関わらず、すべて停止させてしまうという問題です。

Union ファイルシステム (UnionFS)

UnionFS は、コンテナ イメージと呼ばれるイメージのバックボーンを形成します。コンテナ イメージについては、次の章で詳しく説明します。ここでは、UnionFS の機能とその仕組みについて少し説明します。UnionFS は、主に Linux 上で使用されます。異なるファイルシステムのファイルとディレクトリを重ね合わせて、一貫した単一のファイルシステムを形成できるようにするものです。このコンテキストでは、個々のファイルシステムはブランチと呼ばれます。マージされたブランチ内で同じパスを持つディレクトリの内容は、新しい仮想ファイルシステムのマージされた単一のディレクトリにまとめて表示されます。ブランチをマージするときは、ブランチ間の優先順位が指定されます。したがって、2 つのブランチが同じファイルを含んでいる場合は、より高い優先順位のあるファイルが最終的な FS に表示されます。

コンテナの配管

Docker エンジンが上部にビルドされている基部は、**コンテナの配管**ともいえます。2 つのコンポーネント、**runc** と **containerd** によって形成されています。

Docker はもともとモノリシックにビルドされていたもので、コンテナの実行に必要なすべての機能が含まれていました。時間が経つにつれて硬直性が高くなりすぎたので Docker の機能の一部が独自のコンポーネントに分解されるようになりました。2 つの重要なコンポーネントは runc と containerd になります。

Runc

Runc は軽量でポータブルなコンテナ ランタイムです。SELinux、AppArmor、seccomp、cgroups などの Linux 上で利用できるすべてのセキュリティ機能をネイティブでサポートするとともに、Linux の名前空間を完全にサポートしています。

Runc は **Open Container Initiative** (**OCI**) 仕様に基づいてコンテナを生成および実行するツールです。これは正式に規定された構成形式で、Linux Foundation の下で **Open Container Project** (**OCP**) によって管理されています。

Containerd

Runc はコンテナ ランタイムの低レベル実装です。containerd はその上にビルドされるもので、イメージ転送やストレージ、コンテナの実行、管理、ならびにネットワークとストレージの接続などのより高度な機能を追加します。これによって、コンテナのライフサイクル全体を管理します。Containerd は OCI 仕様のリファレンス実装であり、最も一般的かつ広く使用されているコンテナ ランタイムです。

Containerd は、2017 年に CNCF に寄贈されて受け入れられてきたものです。OCI 仕様の実装は他にもあります。たとえば、CoreOS の rkt、RedHat の CRI-O、Linux Containers の LXD などがあります。しかし、現時点で、containerd がはるかに一般的なコンテナ ランタイムであり、Kubernetes 1.8 以降および Docker プラットフォームの既定のランタイムになっています。

まとめ

この章では、既存のイメージに基づくコンテナの操作方法を学習しました。コンテナを実行、停止、開始、および削除する方法を紹介しました。そして、コンテナのメタデータを検査し、そのログを抽出し、稼働中のコンテナで任意のプロセスを実行する方法を学びました。最後に、これも重要なことですが、コンテナがどのように動作するのか、そして基盤となる Linux オペレーティング システムのどの機能をコンテナが利用するのかについて少し掘り下げて調べました。

次の章では、コンテナ イメージとは何か、そして独自のカスタム イメージをビルドして共有する方法について学習します。カスタム イメージをビルドするときによく使用されるベスト プラクティス (サイズの最小化、イメージ キャッシュの活用など) についても見ていきます。お楽しみに。

質問

学習の進捗状況を評価するために、以下の質問にお答えください。

1. コンテナの状態にはどのようなものがありますか。
2. 現在ホスト上で何が稼働しているかを確認するには、どのコマンドを使用しますか。
3. すべてのコンテナの ID を一覧表示するには、どのコマンドを使用しますか。

参考情報

以下の記事では、この章で説明したトピックに関連する詳しい情報を提供しています。

- Docker コンテナ (`http://dockr.ly/2iLBV2I`)
- コンテナの概要 (`http://dockr.ly/2gmxKWB`)
- ユーザー名前空間を持つコンテナの分離 (`http://dockr.ly/2gmyKdf`)
- コンテナのリソースの制限 (`http://dockr.ly/2wqN5Nn`)

4
コンテナ イメージの作成と管理

前の章では、コンテナの内容、実行方法、停止方法、削除方法、一覧表示方法、および検査方法について学習しました。いくつかのコンテナのロギング情報を抽出し、既に実行中のコンテナ内で他のプロセスを実行し、最後にコンテナの構造について深く掘り下げました。コンテナを実行するたびに、コンテナ イメージを使用してコンテナを作成しました。この章では、これらのコンテナ イメージについての知識を深めていきます。具体的な内容、作成方法、配布方法について詳しく学習します。

この章では以下のトピックをとりあげます。

- イメージとは
- イメージの作成
- イメージの共有または配信

この章を終了すると、次のことができるようになります。

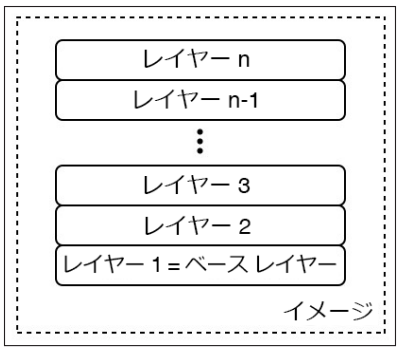
- コンテナ イメージの最も重要な特性を 3 つ挙げる
- インタラクティブにコンテナ レイヤーを変更してコミットすることで、カスタム イメージを作成する
- `FROM`、`COPY`、`RUN`、`CMD`、`ENTRYPOINT` などのキーワードを使用して単純な Dockerfile を作成し、カスタム イメージを生成する
- 既存のイメージを `docker image save` を使用してエクスポートし、`docker image load` を使用して別の Docker ホストにインポートする
- 最終イメージに結果の成果物 (バイナリ) のみを含めることで、結果のイメージのサイズを最小限に抑える 2 ステップの Dockerfile を作成する

イメージとは

Linux では、すべてがファイルです。オペレーティング システム全体が基本的にファイルシステムで、ファイルとフォルダがローカル ディスクに格納されています。これは、コンテナ イメージとは何かを考えるときに覚えておくべき重要な事実です。この後で説明するように、イメージは基本的にはファイルシステムを含む大きな tarball です。具体的には、階層化されたファイルシステムを含んでいます。

階層化されたファイルシステム

コンテナ イメージとは、コンテナの作成元となるテンプレートです。これらのイメージは単一のモノリシック ブロックではなく、多くのレイヤーで構成されています。イメージの最初のレイヤーをベース レイヤーといいます。



レイヤーのスタックとしてのイメージ

各レイヤーにファイルとフォルダが含まれています。各レイヤーには、下位レイヤーに対するファイルシステムの変更のみが含まれます。Docker は union ファイルシステム (第 3 章「コンテナの操作」で説明しました) を使用して、一連のレイヤーから仮想ファイルシステムを作成します。ストレージ ドライバーは、これらのレイヤーが相互にやり取りする方法に関する詳細を処理します。さまざまなストレージ ドライバーを利用できますが、それぞれには異なる状況において長所と短所があります。

コンテナ イメージのレイヤーはすべて変更不可能です。変更不可能とは、一度生成したレイヤーは変更できないという意味です。レイヤーに影響を及ぼすことができる操作は、レイヤーの物理的な削除のみです。このレイヤーの不変性は重要です。というのも、この後で説明するように、不変性によって非常に大きな可能性が生まれるためです。

次のイメージでは、Nginx を Web サーバーとして使用する Web アプリケーションのカスタム イメージがどのように見えるかを確認できます。

Alpine と Nginx に基づくサンプルのカスタム イメージ

ここでのベース レイヤーは Alpine Linux ディストリビューションで構成されています。次に、Alpine の上に Nginx が追加されたレイヤーがあります。最後の 3 番目のレイヤーには、HTML ファイル、CSS ファイル、JavaScript ファイルなどの Web アプリケーションを構成するすべてのファイルが含まれています。

前に述べたように、各イメージはベース イメージから始まります。通常、このベース イメージは、Linux ディストリビューション、Alpine、Ubuntu、CentOS など、Docker Hub にある公式イメージの 1 つです。ただし、最初からイメージを作成することもできます。

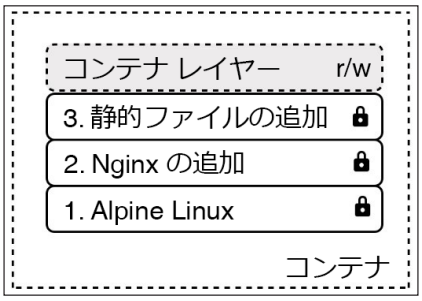
Docker Hub は、コンテナ イメージ用の公開レジストリです。パブリック コンテナ イメージを共有するのに最適な中央ハブです。

各レイヤーは、前のレイヤー セットに関する変更の差のみを含みます。各レイヤーの内容は、ホスト システム上の特別なフォルダにマップされます。通常、これは `/var/lib/docker/` のサブフォルダです。

レイヤーは変化しないので、古くなることなくキャッシュされます。この後で説明するように、これは大きな利点です。

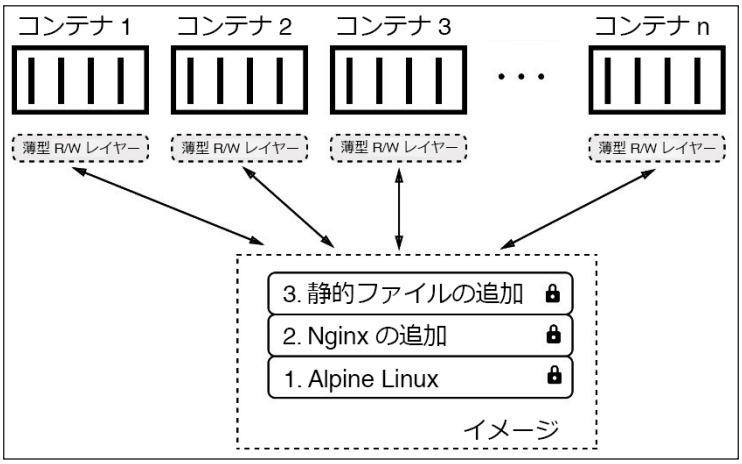
書き込み可能なコンテナ レイヤー

前述したように、コンテナ イメージは不変または読み取り専用レイヤーのスタックでできています。Docker エンジンがそのようなイメージからコンテナを作成する場合、この不変レイヤーのスタックの上に、書き込み可能なコンテナ レイヤーを追加します。スタックは次のようになります。



書き込み可能なコンテナ レイヤー

コンテナ レイヤーは読み取り / 書き込み (r/w) としてマークされます。イメージ レイヤーの不変性のもう 1 つの利点は、このイメージから作成された多くのコンテナでイメージ レイヤーを共有できることです。必要なのは、各コンテナの書き込み可能な薄いコンテナ レイヤーのみです。

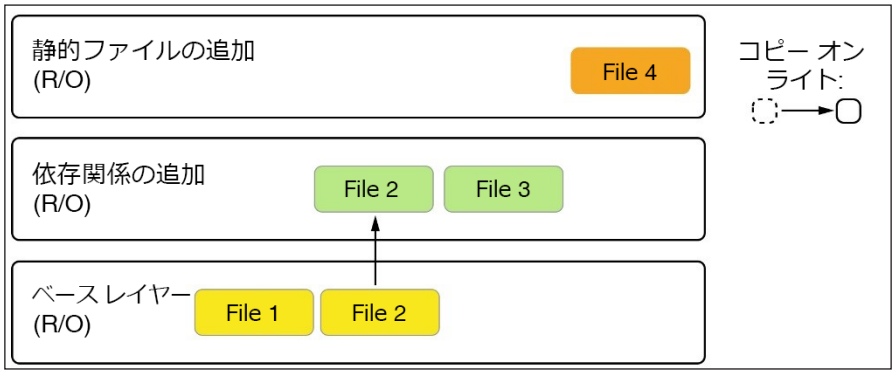


同じイメージ レイヤーを共有する複数のコンテナ

当然ながら、この技術によって、消費されるリソースが大幅に削減されます。さらに、イメージ レイヤーをメモリにロードしたら (これが行われるのは最初のコンテナの場合のみです)、作成する必要があるのは薄いコンテナ レイヤーのみなので、コンテナのローディング時間が短縮されます。

コピーオンライト

Docker はイメージを扱うときにコピーオンライト技術を使用します。コピーオンライトは、効率を最大化するためのファイルの共有およびコピーの戦略です。レイヤーが下のレイヤーのいずれかで使用可能なファイルまたはフォルダを使用する場合は、単にそれを使用します。一方、あるレイヤーが下のレイヤーのファイルを変更する場合は、最初にこのファイルをターゲット レイヤーにコピーしてから変更します。次の図を見れば、これが何を意味するのかがわかります。



コピーオンライト

2 番目のレイヤーでは **File 2** を変更します。このファイルはベース レイヤーに存在しています。そのためコピーしてから変更しました。今度は、上の図の一番上のレイヤーを見てみましょう。このレイヤーは、ベース レイヤーの **File 1** と、2 番目のレイヤーの **File 2** と **File 3** を使用します。

グラフ ドライバー

グラフ ドライバーは、union ファイルシステムを有効にするものです。グラフ ドライバーはストレージ ドライバーとも呼ばれ、階層化されたコンテナ イメージを扱うときに使用されます。グラフ ドライバーは、コンテナのマウント名前空間のために、複数のイメージ レイヤーをルート ファイルシステムに統合します。別の言い方をすれば、ドライバーは Docker ホスト上でのイメージとコンテナの格納・管理方法を制御します。

Docker は、拡張可能なアーキテクチャを使用して、いくつかの異なるグラフ ドライバーをサポートしています。優先するドライバーは overlay2 で、その次が overlay です。

イメージの作成

システムに新しいコンテナ イメージを作成する方法は 3 つあります。第 1 の方法は、希望するすべての追加と変更を含むコンテナをインタラクティブに構築し、その変更を新しいイメージにコミットすることです。第 2 の最も重要な方法は、Dockerfile を使用して新しいイメージの内容を記述し、その Dockerfile をマニフェストとして使用してこのイメージを構築することです。イメージを作成する第 3 の方法は、tarball からシステムにイメージをインポートする方法です。

それでは、これらの 3 つの方法を詳しく見ていきましょう。

インタラクティブなイメージ作成

カスタム イメージを作成する第 1 の方法は、インタラクティブにコンテナを構築することです。つまり、テンプレートとして使用するベース イメージから始めて、それのコンテナをインタラクティブに実行するということです。これが `alpine` イメージだとします。コンテナを実行するコマンドは、次のようになります。

```
$ docker container run -it --name sample alpine /bin/sh
```

既定では、alpine のコンテナには `ping` ツールはインストールされていません。`ping` がインストールされている新しいカスタム イメージを作成するとしましょう。コンテナ内で、次のコマンドを実行できます。

```
/ # apk update && apk add iputils
```

これは Alpine パッケージ マネージャー `apk` を使用して、`iputils` ライブラリをインストールします。`ping` はその一部です。上記のコマンドの出力は、次のようになります。

```
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/main/x86_64/APKINDEX.
tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/community/x86_64/
APKINDEX.tar.gz
v3.7.0-50-gc8da5122a4 [http://dl-cdn.alpinelinux.org/alpine/v3.7/main]
v3.7.0-49-g06d6ae04c3 [http://dl-cdn.alpinelinux.org/alpine/v3.7/
community]
OK: 9046 distinct packages available
(1/2) Installing libcap (2.25-r1)
(2/2) Installing iputils (20121221-r8)
Executing busybox-1.27.2-r6.trigger
OK: 4 MiB in 13 packages
```

これで、実際に ping を使用できるようになります。そのスニペットを次に示します。

```
/ # ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.028 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.044 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.049 ms
^C
--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2108ms
rtt min/avg/max/mdev = 0.028/0.040/0.049/0.010 ms
```

カスタマイズが完了したら、プロンプトで exit と入力するとコンテナを終了できます。docker container ls -a ですべてのコンテナを一覧表示すると、ステータスが Exited であるものの、まだシステム上に存在するサンプル コンテナを確認できます。

```
$ docker container ls -a | grep sample
eff7c92a1b98    alpine    "/bin/sh"    2 minutes ago      Exited (0)
...
```

ベース イメージとの関連でコンテナ内で変更された内容を確認する場合は、docker container diff コマンドを次のように使用します。

```
$ docker container diff sample
```

出力には、コンテナのファイルシステムで行われたすべての変更のリストが表示されます。

```
C /bin
C /bin/ping
C /bin/ping6
A /bin/traceroute6
C /etc/apk
C /etc/apk/world
C /lib/apk/db
C /lib/apk/db/installed
C /lib/apk/db/lock
C /lib/apk/db/scripts.tar
C /lib/apk/db/triggers
C /root
A /root/.ash_history
C /usr/lib
A /usr/lib/libcap.so.2
A /usr/lib/libcap.so.2.25
C /usr/sbin
```

```
C /usr/sbin/arping
A /usr/sbin/capsh
A /usr/sbin/clockdiff
A /usr/sbin/getcap
A /usr/sbin/getpcaps
A /usr/sbin/ipg
A /usr/sbin/rarpd
A /usr/sbin/rdisc
A /usr/sbin/setcap
A /usr/sbin/tftpd
A /usr/sbin/tracepath
A /usr/sbin/tracepath6
C /var/cache/apk
A /var/cache/apk/APKINDEX.5022a8a2.tar.gz
A /var/cache/apk/APKINDEX.70c88391.tar.gz
C /var/cache/misc
```

上記のリストでは、A は追加、C は変更を表しています。削除されたファイルがある場合、それらのファイルには D がプレフィックスされます。

これで docker container commit コマンドを使用して、変更内容を保持し、そこから新しいイメージを作成できるようになります。

```
$ docker container commit sample my-alpine
sha256:44bca4141130ee8702e8e8efd1beb3cf4fe5aadb62a0c69a6995afd49c2e7419
```

上記のコマンドでは、新しいイメージを my-alpine と呼ぶように指定しています。上記のコマンドによって生成された出力は、新しく生成されたイメージの ID に対応します。システム上のすべてのイメージは、次のように一覧表示して確認できます。

```
$ docker image ls
```

このイメージ ID (短縮型) は次のように表示されます。

```
REPOSITORY TAG      IMAGE ID        CREATED               SIZE
my-alpine  latest   44bca4141130    About a minute ago    5.64MB
...
```

my-alpine という名前のイメージには 44bca4141130 という ID が予測され、自動的に latest というタグが割り当てられたことがわかります。このようになるのは、タグを明示的に定義していないためです。この場合、Docker では常にタグが既定で latest になります。

カスタム イメージがどのように構築されているかを確認するには、history コマンドを次のように使用します。

```
$ docker image history my-alpine
```

これにより、イメージを構成しているレイヤーのリストが印刷されます。

```
IMAGE            CREATED          CREATED BY          SIZE
COMMENT
44bca4141130     3 minutes ago    /bin/sh             1.5MB
e21c333399e0     6 weeks ago      /bin/sh -c #...0B
<missing>        6 weeks ago      /bin/sh -c #...4.14MB
```

上記のリストの最初のレイヤーは、`iputils` パッケージを追加することで作成したレイヤーです。

Dockerfile の使用

この章の前のセクションで示したように、カスタム イメージを手動で作成することは、探索、プロトタイプの作成、実現可能性の検討を行う場合に非常に役立ちます。しかしこれには重大な欠点があります。手作業であるため、再現性や拡張性がありません。また、人が手で行う作業と同じようにエラーも発生しやすくなります。そのため、より良い方法が必要です。

ここで、いわゆる Dockerfile が登場します。Dockerfile はテキスト ファイルで、通常、文字通り Dockerfile と呼ばれます。カスタム コンテナ イメージの作成方法に関する命令を含んでいます。イメージを構築するための宣言的な方法です。

宣言型と命令型の違い

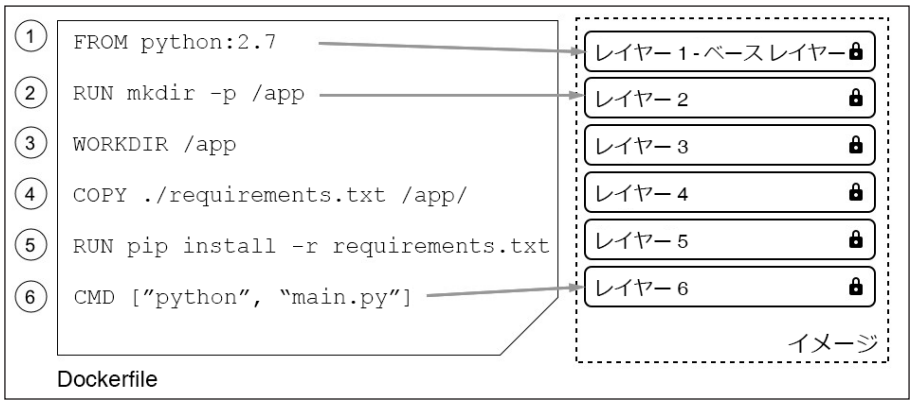
コンピューター サイエンスでは一般的に、また Docker では具体的に、タスクを定義する宣言的な方法を使用することがよくあります。期待される成果を記述して、この目標を達成する方法をシステムに把握させるものです。これに対して、期待される成果を達成する方法について、ステップ バイ ステップの指示をシステムに与えることが命令的アプローチです。

サンプルの Dockerfile を見てみましょう。

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY ./requirements.txt /app/
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

これは Python 2.7 アプリケーションをコンテナ化するために使用される Dockerfile です。ご覧のように、このファイルには 6 行 あり、それぞれが FROM、RUN、COPY などのキーワードで始まります。キーワードをすべて大文字で記述するのが慣習ですが、必須ではありません。

Dockerfile の各行で、結果のイメージにレイヤーが作成されます。次の図のイメージは、レイヤーのスタックとしてイメージを示した前の図と比べて、上下逆さまに描かれています。ここでは、ベース レイヤーが上に示されています。混乱しないでください。実際には、ベース レイヤーは常にスタック内の一番下のレイヤーです。



Dockerfile とイメージのレイヤーとの関係

次は、個々のキーワードをより詳細に見てみましょう。

FROM キーワード

ほとんどの Dockerfile は FROM キーワードで始まります。これを使って、カスタム イメージの構築の始めとなるベース イメージを定義します。たとえば、CentOS 7 から始める場合は、Dockerfile に次の行を記述します。

```
FROM centos:7
```

Docker Hub には、Python、Node JS、Ruby、Go などの多数の重要な開発フレームワークや言語のほか、すべての主要な Linux ディストリビューションのキュレーションされたイメージや公式のイメージがあります。必要に応じて、最も適切なベース イメージを選択する必要があります。

たとえば、Python 2.7 アプリケーションをコンテナ化する場合は、関連する公式の python:2.7 イメージを選択します。

ゼロから始めるのであれば、次のステートメントを使うこともできます。

```
FROM scratch
```

これは、たとえば Hello-World などの単一のバイナリ (静的にリンクされた実際の実行可能ファイル) のみを含む最小限のイメージを構築する状況において役立ちます。スクラッチ イメージは、文字通り空のベース イメージです。

FROM scratch は Dockerfile の no-op (操作なし) に相当し、それ自体は結果のコンテナ イメージにレイヤーを生成しません。

RUN キーワード

次の重要なキーワードは RUN です。RUN の引数は次のような有効な Linux コマンドです。

```
RUN yum install -y wget
```

上記のコマンドは CentOS パッケージ マネージャー yum を使用して、wget パッケージを実行中のコンテナにインストールします。これは、ベース イメージが CentOS または RHEL であることを前提としています。Ubuntu をベース イメージとする場合、コマンドは次のようになります。

```
RUN apt-get update && apt-get install -y wget
```

このようになるのは、Ubuntu が apt-get をパッケージ マネージャーとして使用するためです。同様に、RUN の行をこのように定義できます。

```
RUN mkdir -p /app && cd /app
```

これを行うこともできます。

```
RUN tar -xJC /usr/src/python --strip-components=1 -f python.tar.xz
```

ここで、前者はコンテナに /app フォルダを作成して、そのフォルダに移動し、後者はファイルを指定された場所に展開します。次のような複数の物理行を使用して Linux コマンドをフォーマットすることは、適切であり推奨もされます。

```
RUN apt-get update \
  && apt-get install -y --no-install-recommends \
    ca-certificates \
    libexpat1 \
    libffi6 \
    libgdbm3 \
    libreadline7 \
    libsqlite3-0 \
    libssl1.1 \
  && rm -rf /var/lib/apt/lists/*
```

複数の行を使用する場合は、行の終わりにバックスラッシュ (\) を置いて、コマンドが次の行に続くことをシェルに示す必要があります。

上記のコマンドが何をしているのか調べてみてください。

COPY キーワードと ADD キーワード

最終的にはいくつかのコンテンツを既存のベース イメージに追加してカスタム イメージを作成することになるため、COPY キーワードと ADD キーワードは非常に重要です。たいていの場合、これらは Web アプリケーションのいくつかのソース ファイル、またはコンパイルされたアプリケーションのいくつかのバイナリです。

この 2 つのキーワードは、ファイルやフォルダをホストから構築中のイメージにコピーするために使用されます。2 つのキーワードは非常に似ていますが、ADD キーワードを使用すると、TAR ファイルをコピーして解凍したり、コピーするファイルとフォルダのソースとして URL を提供できる点が異なります。

この 2 つのキーワードの使用方法の例をいくつか見てみましょう。

```
COPY ./app
COPY ./web /app/web
COPY sample.txt /data/my-sample.txt
ADD sample.tar /app/bin/
ADD http://example.com/sample.txt /data/
```

上記のコードで、

- 1 行目は、現在のディレクトリのすべてのファイルとフォルダを再帰的にコンテナ イメージ内の /app フォルダにコピーしています。
- 2 行目は、web サブフォルダの内容をすべてターゲット フォルダ /app/web にコピーしています。
- 3 行目は 1 つのファイル、sample.txt をターゲット フォルダ /data にコピーし、同時に名前を my-sample.txt に変更しています。
- 4 番目のステートメントは、sample.tar ファイルをターゲット フォルダ、/app/bin に展開しています。
- 最後のステートメントは、リモート ファイル、sample.txt をターゲット ファイル、/data にコピーしています。

ソース パスにはワイルドカードを使用できます。たとえば、次のステートメントは、sample で始まるすべてのファイルを、イメージ内の mydir フォルダにコピーします。

```
COPY ./sample* /mydir/
```

セキュリティの観点から、既定では、イメージ内のファイルとフォルダはすべて、**ユーザー ID (UID)** と**グループ ID (GID)** が `0` になります。利点は、`ADD` と `COPY` の両方で、オプションの `--chown` フラグを次のように使用して、イメージ内のファイルの所有権を変更できることです。

```
ADD --chown=11:22 ./data/files* /app/data/
```

上記のステートメントは、`web` という名前で始まるすべてのファイルをコピーしてイメージの `/app/data` フォルダに保存し、同時にユーザー `11` とグループ `22` をこれらのファイルに割り当てます。

ユーザーとグループに数字の代わりに名前を使用することもできますが、その場合はこれらのエンティティがそれぞれ `/etc/passwd` と `/etc/group` でイメージのルート ファイルシステムに既に定義されている必要があります。定義されていないと、イメージの構築は失敗します。

WORKDIR キーワード

`WORKDIR` キーワードは、コンテナをカスタム イメージから実行するときに使用する作業ディレクトリまたはコンテキストを定義します。そのため、コンテキストをイメージ内の `/app/bin` フォルダに設定する場合、Dockerfile の式は次のようになります。

```
WORKDIR /app/bin
```

この行の後にイメージ内で発生するすべてのアクティビティは、このディレクトリを作業ディレクトリとして使用します。Dockerfile の次の 2 つのスニペットが同じではないことに注意してください。

```
RUN cd /app/bin
RUN touch sample.txt
```

上記のコードを次のコードと比較してください。

```
WORKDIR /app/bin
RUN touch sample.txt
```

前者はイメージ ファイルシステムのルートにファイルを作成し、後者は `/app/bin` フォルダの予想される場所にファイルを作成します。イメージのレイヤー全体にコンテキストを設定するのは、`WORKDIR` キーワードのみです。`cd` コマンドだけではレイヤー全体に保持されません。

CMD キーワードと ENTRYPOINT キーワード

CMD キーワードと ENTRYPOINT キーワードは特殊です。Dockerfile に定義されている他のキーワードはすべて、イメージが Docker ビルダーによって作成されるときに実行されますが、この 2 つは実際に、定義したイメージからコンテナを開始するときに起こることを定義しています。コンテナ ランタイムはコンテナを開始するとき、このコンテナ内で実行する必要があるプロセスまたはアプリケーションを知る必要があります。それこそが、CMD と ENTRYPOINT を使用する目的です。つまり、Docker に開始プロセスとそのプロセスの開始方法を伝えることです。

現在、CMD と ENTRYPOINT の差はわずかで、実際のところ、ほとんどのユーザーはこれらを完全に理解していないか、意図した方法で使用していません。幸いにも、ほとんどの場合、これは問題にはならず、コンテナは動作しますが、その処理はそれほど簡単ではありません。

2 つのキーワードの使用法をより深く理解するために、典型的な Linux のコマンドや式がどのようなものかを分析してみましょう。例として、次のような ping ユーティリティを取り上げます。

```
$ ping 8.8.8.8 -c 3
```

上記の式で、ping はコマンドで、8.8.8.8 -c 3 はこのコマンドのパラメーターです。別の式を見てみましょう。

```
$ wget -O - http://example.com/downloads/script.sh
```

同じように上記の式では、wget はコマンドで、-O - http://example.com/downloads/script.sh はパラメーターです。

それでは CMD と ENTRYPOINT に戻ります。ENTRYPOINT は式のコマンドを定義するために使用され、CMD はコマンドのパラメーターを定義するために使用されます。したがって、alpine をベース イメージとして使用し、ping をプロセスとして定義してコンテナ内で実行する Dockerfile は、次のようになります。

```
FROM alpine:latest
ENTRYPOINT ["ping"]
CMD ["8.8.8.8", "-c", "3"]
```

ENTRYPOINT と CMD の両方で、値は文字列の JSON 配列としてフォーマットされます。ここでは、個々の項目は、空白で区切られた式のトークンに対応します。これは、CMD と ENTRYPOINT を定義するための推奨方法です。また、**exec** 形式とも呼ばれます。

あるいは、たとえば **shell** 形式と呼ばれるものを使用することもできます。

```
CMD command param1 param2
```

これで、前述の Dockerfile から次のようにイメージを構築できます。

```
$ docker image build -t pinger .
```

その後、作成した pinger イメージからコンテナを実行できます。

```
$ docker container run --rm -it pinger
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=37 time=19.298 ms
64 bytes from 8.8.8.8: seq=1 ttl=37 time=27.890 ms
64 bytes from 8.8.8.8: seq=2 ttl=37 time=30.702 ms
```

これの利点は、docker container run 式の最後に新しい値を追加して、新しいコンテナを作成するときに、Dockerfile で定義した CMD 部分 (["8.8.8.8"、"-c"、"3"]) を上書きできることです。

```
$ docker container run --rm -it pinger -w 5 127.0.0.1
```

これでコンテナは、ループバックに 5 秒間 ping を実行します。

Dockerfile の ENTRYPOINT に定義されている内容を上書きする場合は、docker container run 式で --entrypoint パラメーターを使用する必要があります。ping コマンドの代わりに、コンテナ内でシェルを実行するとします。これは次のコマンドで行えます。

```
$ docker container run --rm -it --entrypoint /bin/sh pinger
```

このとき、コンテナの中に入っています。exit と入力してコンテナから出ます。

既に述べたように、ベストプラクティスに従って、コマンドを ENTRYPOINT、パラメーターを CMD で定義する必要はありません。代わりに、式全体を CMD の値として入力でき、それは次のように動作します。

```
FROM alpine:latest
CMD wget -O - http://www.google.com
```

ここでは、**CMD** を定義するために、**shell** 形式も使用しています。しかし、ENTRYPOINT が定義されていないこの状況では、実際に何が起こるでしょうか。未定義のままにすると、ENTRYPOINT は既定値の /bin/sh -c になり、CMD の値が何であっても、文字列として shell コマンドに渡されます。これにより、前述の定義では、コンテナ内で実行する次のプロセスが入力されます。

```
/bin/sh -c "wget -O - http://www.google.com"
```

その結果、/bin/sh がコンテナ内で実行されているメイン プロセスであり、新しい子プロセスを開始して wget ユーティリティを実行します。

複雑な Dockerfile

Dockerfile でよく使用される最も重要なキーワードについて説明しました。次は Dockerfile の現実的でやや複雑な例を見てみましょう。興味のある読者なら、この章で説明した最初の Dockerfile と非常に似ていることに気付くかもしれません。これが内容です。

```
FROM node:9.4
RUN mkdir -p /app
WORKDIR /app
COPY package.json /app/
RUN npm install
COPY ./app
ENTRYPOINT ["npm"]
CMD ["start"]
```

ここでは何が起こっているでしょうか。明らかに、これは Node.js アプリケーション用のイメージの構築に使用される Dockerfile です。そのことは、ベース イメージ node:9.4 が使用されているという事実から推測できます。2 行目は、イメージのファイルシステムに /app フォルダを作成するという命令です。3 行目は、イメージ内の作業ディレクトリまたはコンテキストが、この新しい /app フォルダになるように定義しています。4 行目では、package.json ファイルを、イメージ内の /app フォルダにコピーします。その後、5 行目では、コンテナ内で npm install コマンドを実行します。内容は /app フォルダに格納されているため、npm は 4 行目でコピーされた package.json ファイルをそこで見つけます。

すべての Node.js 依存関係がインストールされた後、残りのアプリケーション ファイルをホストの現在のフォルダからイメージの /app フォルダにコピーします。

最後の 2 行で、コンテナがこのイメージから実行されているときの起動コマンドを定義します。このケースでは、起動コマンドは npm start で、Node アプリケーションを起動します。

イメージの構築

イメージを構築するには、次の手順に従ってください。

1. home ディレクトリに FundamentalsOfDocker フォルダを作成し、そこに移動します。

```
$ mkdir ~/FundamentalsOfDocker
$ cd ~/FundamentalsOfDocker
```

2. 上記のフォルダに sample1 サブフォルダを作成し、そこに移動します。

```
$ mkdir sample1 && cd sample1
```

3. お気に入りのエディターを使用して、このサンプル フォルダ内に Dockerfile という名前のファイルを作成し、以下の内容を入れます。

```
FROM centos:7
RUN yum install -y wget
```

4. ファイルを保存してエディターを終了します。ターミナルに戻ると、前述の Dockerfile をマニフェストまたは構築計画として使って、新しいコンテナ イメージを作成できるようになっています。

```
$ docker image build -t my-centos .
```

上記のコマンドの最後にピリオドがあることに注意してください。このコマンドは、Docker ビルダーが現在のディレクトリにある Dockerfile を使用して、my-centos という名前の新しいイメージを作成していることを意味します。ここで、コマンドの最後にあるピリオドは、現在のディレクトリを表しています。上記のコマンドを次のように書くこともできますが、同じ結果が得られます。

```
$ docker image build -t my-centos -f Dockerfile .
```

ただし、-f パラメーターは省略できます。ビルダーは、Dockerfile が文字通り Dockerfile と呼ばれているものと想定するためです。-f パラメーターが必要になるのは、Dockerfile が別の名前を持っているか、現在のディレクトリに存在しない場合のみです。

上記のコマンドでは、この (短縮された) 出力が得られます。

```
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM centos:7
7: Pulling from library/centos
af4b0a2388c6: Pull complete
Digest: sha256:2671f7a3eea36ce43609e9fe7435ade83094291055f1c96d9d1d1d7
c0b986a5d
Status: Downloaded newer image for centos:7
---> ff426288ea90
```

```
Step 2/2 : RUN yum install -y wget
---> Running in bb726903820c
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
* base: mirror.dal10.us.leaseweb.net
* extras: repos-tx.psychz.net
* updates: pubmirrors.dal.corespace.com
Resolving Dependencies
--> Running transaction check
---> Package wget.x86_64 0:1.14-15.el7_4.1 will be installed
...
Installed:
wget.x86_64 0:1.14-15.el7_4.1
Complete!
Removing intermediate container bb726903820c
---> bc070cc81b87
Successfully built bc070cc81b87
Successfully tagged my-centos:latest
```

この出力を分析しましょう。

1. 最初は、以下のような行です。

   ```
   Sending build context to Docker daemon 2.048kB
   ```

   ビルダーが最初に行うことは、現在のビルド コンテキスト内のファイルをパッケージ化することです。ただし `.dockerignore` ファイルに記述されているファイルとフォルダが存在する場合はそれらを除外します。その後、結果の `.tar` ファイルを Docker デーモンに転送します。

2. 次は、以下のような行です。

   ```
   Step 1/2 : FROM centos:7
   7: Pulling from library/centos
   af4b0a2388c6: Pull complete
   Digest: sha256:2671f7a...
   Status: Downloaded newer image for centos:7
   ---> ff426288ea90
   ```

最初の行は、ビルダーが現在実行している Dockerfile のステップを示しています。ここでの Dockerfile にあるのは 2 つのステートメントのみで、ステップ 1/2 です。また、そのセクションの内容も確認できます。ここにはベースイメージの宣言があります。その上にカスタム イメージを作成します。ビルダーが次に行うことは、このイメージがローカル キャッシュでまだ利用できない場合、Docker Hub から引き出すことです。前述のスニペットの最後の行は、作成されたばかりのレイヤーがビルダーによって割り当てられる ID を示します。

3. ここに次のステップがあります。前述のものよりもさらに短縮して、重要な部分に焦点を当てました。

```
Step 2/2 : RUN yum install -y wget
---> Running in bb726903820c
...
...
Removing intermediate container bb726903820c
---> bc070cc81b87
```

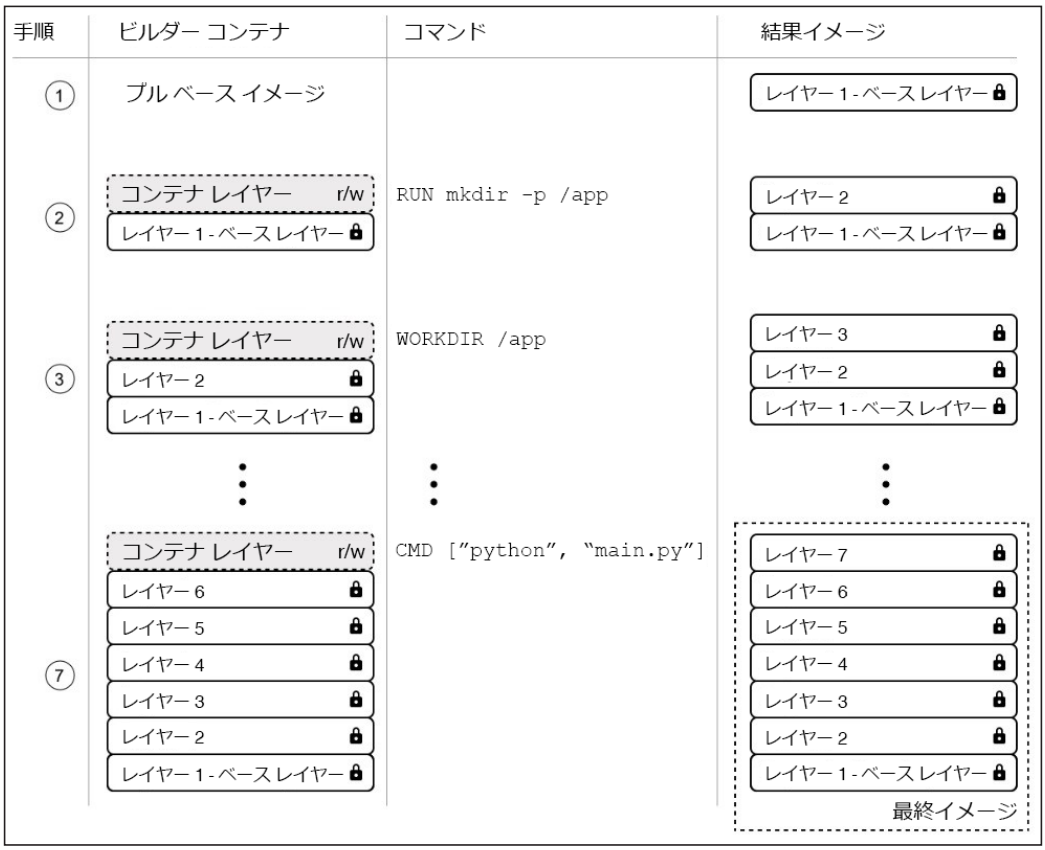
ここでもまた、最初の行はステップ 2/2 であることを示しています。これは、Dockerfile のそれぞれのエントリも表しています。2 行目の `Running in bb726903820c` は、ビルダーが内部に `ID bb726903820c` というコンテナを作成したことを示しています。これは `RUN` コマンドで実行されます。スニペットの `yum install -y wget` コマンドの出力は、このセクションでは重要ではないため省略しています。コマンドが終了すると、ビルダーはコンテナを停止し、新しいレイヤーにコミットして、コンテナを削除します。この特定のケースでは、新しいレイヤーには `IDbc070cc81b87` が割り当てられています。

4. 出力の最後には、次の 2 つの行があります。

```
Successfully built bc070cc81b87
Successfully tagged my-centos:latest
```

これにより、結果として得られるカスタム イメージは `bc070cc81b87` という ID が割り当てられ、`my-centos:latest` という名前でタグ付けされたことがわかります。

そもそもビルダーはどのように動作するのでしょうか。ビルダーはベース イメージから始まります。このベース イメージからローカル キャッシュにダウンロードされると、コンテナを作成し、このコンテナ内の Dockerfile の最初のステートメントを実行します。次に、コンテナを停止し、コンテナに加えられた変更を新しいイメージ レイヤーに保持します。その後でビルダーは、ベース イメージと新しいレイヤーから新しいコンテナを作成し、この新しいコンテナ内で 2 番目のステートメントを実行します。再度、結果は新しいレイヤーにコミットされます。このプロセスは、Dockerfile の最後のステートメントに達するまで繰り返されます。ビルダーは、新しいイメージの最後のレイヤーをコミットした後、このイメージの ID を作成し、`build` コマンドで指定した名前でイメージにタグ付けします。



視覚化したイメージ構築プロセス

マルチステップ ビルド

複数のビルド ステップを持つ Dockerfile が便利な理由を示すため、Dockerfile の例を作ってみましょう。C で書かれた `Hello World` アプリケーションを例として使用します。`hello.c` ファイルには次のようなコードがあります。

```
#include <stdio.h>
int main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

今度は、このアプリケーションをコンテナ化して、この Dockerfile を作成します。

```
FROM alpine:3.7
RUN apk update &&
apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY ./app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
CMD /app/bin/hello
```

それでは、このイメージを構築しましょう。

```
$ docker image build -t hello-world .
```

これはかなり長い出力になります。ビルダーは Alpine SDK をインストールする必要があるためですが、このツールには特に、アプリケーションをビルドするために必要な C++ コンパイラーが含まれています。

ビルドが完了すると、イメージを一覧表示できます。そのサイズは次のように表示されます。

```
$ docker image ls | grep hello-world
hello-world      latest      e9b...2 minutes ago     176MB
```

サイズが 176 MB では、結果として得られるイメージが大きすぎます。結局のところ、単なる `Hello World` アプリケーションです。これが大きくなりすぎているのは、イメージに `Hello World` バイナリだけでなく、ソース コードからアプリケーションをコンパイルしてリンクするためのすべてのツールも含まれているためです。しかしアプリケーションを運用環境で実行する場合には、この状況はまったく望ましいものではありません。理想的なのは、イメージ内に SDK 全体ではなく、バイナリのみを入れることです。

こうした理由から、Dockerfiles を複数の段階として定義する必要があります。いくつかの段階は、最終的な成果物を構築するために使用されるもので、最後の段階は、必要最小限のベース イメージを使用して成果物をその中にコピーするものです。これによってイメージが非常に小さくなります。この改訂された Dockerfile を見てみましょう。

```
FROM alpine:3.7 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY ./app
RUN mkdir bin
RUN gcc hello.c -o bin/hello
FROM alpine:3.7
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

最初の段階には、アプリケーションをコンパイルするために使用されるエイリアス ビルドがあります。2 番目の段階では同じベース イメージ `alpine:3.7` を使用しますが、SDK をインストールせず、`--from` パラメーターを使用して、ビルド段階からバイナリのみをこの最終イメージにコピーします。

イメージを次のように再構築しましょう。

```
$ docker image build -t hello-world-small .
```

イメージのサイズを比較すると、次のような出力が得られます。

```
$ docker image ls | grep hello-world
hello-world-small   latest    f98...20 seconds ago     4.16MB
hello-world         latest    469...10 minutes ago     176MB
```

サイズを 176 MB から 4 MB に減らすことができました。サイズが 40 分の 1 になりました。イメージ サイズを小さくすることには多数の利点があります。ハッカーの攻撃にさらされる部分の縮小、メモリとディスクの消費量の削減、対応するコンテナの起動時間の短縮、Docker Hub などのレジストリからのイメージのダウンロードに必要な帯域幅の削減などは、その一例です。

Dockerfile のベスト プラクティス

Dockerfile を作成するときに考慮すべきいくつかの推奨されるベスト プラクティスを以下に示します。

- **コンテナは一時的なものであることに留意してください**。一時的にすることで、コンテナを停止して破壊することができます。また、必要最小限のセットアップと構成で、新しいコンテナを構築して配置することができます。つまり、コンテナ内で実行されているアプリケーションを初期化して最小限にするのに必要な時間と、アプリケーションの終了またはクリーンアップに必要な時間を確保するよう努力する必要があります。
- **Dockerfile 内のコマンドを個々に出し、可能な限りキャッシュを活用できるようにします**。イメージのレイヤーの構築にはかなりの時間がかかることがあり、時には数分かかる場合もあります。アプリケーションを開発するときには、アプリケーションのコンテナ イメージを複数回構築する必要があります。ビルド時間を最小限に抑えることが求められます。

  以前構築したイメージを再構築するときは、変更されたレイヤーだけが再構築されますが、再構築の必要があるレイヤーが 1 つの場合でも、それに続くすべてのレイヤーもすべて再構築する必要があります。これは非常に重要です。次の例を考えてみましょう。

```
FROM node:9.4
RUN mkdir -p /app
WORKIR /app
COPY ./app
RUN npm install
CMD ["npm", "start"]
```

  この例では、Dockerfile の 5 行目の `npm install` コマンドが通常、最も長い時間を要します。従来の Node.js アプリケーションには多くの外部依存関係があり、これらはすべてこの手順でダウンロードおよびインストールされます。これは完了するまで数分かかることがあります。そのため、イメージを再構築するたびに `npm install` を実行することは避けたいものの、開発者はアプリケーションの開発中に常にソース コードを変更します。つまり、4 行目の `COPY` コマンドの結果が常に変化するため、毎回このレイヤーを再構築する必要があります。しかし既に説明したように、これは後続のレイヤーをすべて再構築する必要があるということを意味し、この場合は `npm install` コマンドが含まれます。これを避けるために、Dockerfile を少し変更して、次のようにすることができます。

```
FROM node:9.4
RUN mkdir -p /app
WORKIR /app
```

```
COPY package.json /app/
RUN npm install
COPY ./app
CMD ["npm", "start"]
```

ここで行ったのは、4 行目で `npm install` コマンドがソースとして必要とする 1 本のファイル、`package.json` ファイルのみをコピーしたことです。このファイルは、一般的な開発プロセスではほとんど変更されません。その結果、`npm install` コマンドも実行する必要があるのは、`package.json` ファイルが変更されたときのみです。残りの頻繁に変更される内容はすべて、`npm install` コマンドの後でイメージに追加されます。

- **イメージを構成するレイヤーの数を比較的少なくします。**イメージのレイヤーが増えるほど、グラフ ドライバーは対応するコンテナの単一のルート ファイルシステムにそのレイヤーを統合するという作業が増えます。もちろんこれには時間がかかるため、イメージのレイヤーが少なくなればなるほど、コンテナの起動時間は速くなります。

  しかしどうすればレイヤーの数を少なくできるでしょうか。Dockerfile では、FROM、COPY、RUN などのキーワードで始まる各行が新しいレイヤーを作成することに注目してください。レイヤーの数を減らす最も簡単な方法は、複数の個別の `RUN` コマンドを 1 つにまとめることです。たとえば Dockerfile に次のようなコマンドがあるとします。

  ```
  RUN apt-get update
  RUN apt-get install -y ca-certificates
  RUN rm -rf /var/lib/apt/lists/*
  ```

  これらのコマンドを次のような 1 つの連結式に結合することができます。

  ```
  RUN apt-get update \
      && apt-get install -y ca-certificates \
      && rm -rf /var/lib/apt/lists/*
  ```

  前者は結果のイメージに 3 つのレイヤーを生成しますが、後者は 1 つのレイヤーしか作成しません。

次の 3 つのベスト プラクティスはすべて、結果のイメージが小さくなります。なぜこれが重要なのでしょうか。イメージが小さくなると、レジストリからイメージをダウンロードするのに必要な時間と帯域幅が少なくなります。また、Docker ホストでコピーをローカルに保存するために必要なディスク容量と、イメージをロードするために必要なメモリも削減されます。最後に、イメージが小さくなると、ハッカーの攻撃にさらされる部分も小さくなります。イメージ サイズを縮小するためのベスト プラクティスは次のとおりです。

- .dockerignore ファイルを使用します。イメージには不要なファイルやフォルダをコピーしないようにして、できるだけスリムにします。.dockerignore ファイルは、Git に精通している人にとっては、.gitignore ファイルとまったく同じ方法で動作します。.dockerignore ファイルでは、イメージを構築するときに、特定のファイルやフォルダをコンテキストに含めないように除外するパターンを設定することができます。
- イメージのファイルシステムに不要なパッケージをインストールしないでください。繰り返しになりますが、これはイメージを可能な限りスリムに保つためです。
- 多段階のビルドを使用することで、結果として得られるイメージが可能な限り小さくなり、アプリケーションまたはアプリケーション サービスの実行に必要な最小限の内容のみが含まれるようにします。

イメージの保存と読み込み

新しいコンテナ イメージを作成する第 3 の方法は、ファイルからのインポートまたは読み込みです。コンテナ イメージは tarball にすぎません。これを実証するために、docker image save コマンドを使用して、既存のイメージを tarball にエクスポートすることができます。

```
$ docker image save -o ./backup/my-alpine.tar my-alpine
```

上記のコマンドは、以前に構築した my-alpine イメージを ./backup/my-alpine.tar ファイルにエクスポートします。

一方、既存の tarball をイメージとしてシステムにインポートする場合は、docker image load コマンドを次のように使用できます。

```
$ docker image load -i ./backup/my-alpine.tar
```

イメージの共有または配信

カスタム イメージを他の環境に配信できるようにするには、まずそれにグローバルで一意な名前を付ける必要があります。このアクションは、一般にイメージのタグ付けと呼ばれます。次に、イメージを中央の場所に公開して、他の利害関係者や当事者が引き出せるようにする必要があります。これらの中央の場所は**イメージ レジストリ**と呼ばれます。

イメージのタグ付け

各イメージにはいわゆる**タグ**が付いています。タグは一般に、イメージのバージョン管理に使用されますが、単なるバージョン番号を超える影響力があります。イメージを操作するときにタグを明示的に指定しないと、Docker は自動的に latest タグを参照します。これは、Docker Hub からイメージを取り出す場合に関係します。たとえば、次のようになります。

```
$ docker image pull alpine
```

上記のコマンドは、Hub から `alpine:latest` イメージを取り出します。タグを明示的に指定する場合は、次のようにします。

```
$ docker image pull alpine:3.5
```

これで、`3.5` とタグ付けされた `alpine` イメージを取り出すようになりました。

イメージの名前空間

これまでに、さまざまなイメージを取り出しましたが、そのイメージの取得元についてはそれほど気にしていませんでした。Docker 環境は既定で、すべてのイメージが Docker Hub から取得されるように設定されています。しかも Docker Hub から取り出したのは、`alpine` や `busybox` などのいわゆる公式なイメージのみでした。

そろそろ範囲を少し広げて、イメージの名前空間の仕組みについて学習しましょう。イメージを定義する最も一般的な方法は、次のような完全修飾名です。

```
<registry URL>/<User or Org>/<name>:<tag>
```

これをもう少し詳しく見てみましょう。

- `<registry URL>`: これは、イメージの取得元となるレジストリの URL です。既定では、`docker.io` です。より一般的には、`https://registry.acme.com` になることがあります。

  Docker Hub 以外にも、イメージを取り出せる公開レジストリが多数あります。以下は、一部の公開レジストリのリストです (順不同)。

  - Google (`https://cloud.google.com/container-registry`)
  - Amazon AWS (`https://aws.amazon.com/ecr/`)
  - Microsoft Azure (`https://azure.microsoft.com/en-us/services/container-registry/`)

    - Red Hat (`https://access.redhat.com/containers/`)
    - Artifactory (`https://jfrog.com/integration/artifactory-docker-registry/`)

- `<User or Org>`: これは、Docker Hub で定義した個人または団体のプライベート Docker ID か、マイクロソフトやオラクルなどのそのことに関する他のレジストリです。
- `<name>`: これは一般にリポジトリとも呼ばれるイメージの名前です。
- `<tag>`: これはイメージのタグです。

例を見てみましょう。

```
https://registry.acme.com/engineering/web-app:1.0
```

ここには、`web-app` というイメージがあり、バージョン `1.0` のタグが付けられています。`https://registry.acme.com` で非公開レジストリの `engineering` という組織に属しています。

また、いくつかの特別な規則があります。

- レジストリ URL を省略すると、自動的に Docker Hub が取得されます
- タグを省略すると、`latest` が取得されます
- Docker Hub にある公式イメージであれば、ユーザーまたは組織の名前空間は必要ありません

表形式のサンプルは次のとおりです。

イメージ	**説明**
`alpine`	Docker Hub にある公式の `alpine` イメージ。`latest` タグ付き。
`ubuntu:16.04`	Docker Hub にある公式の `ubuntu` イメージ。`16.04` タグまたはバージョン付き。
`microsoft/nanoserver`	`Docker Hub` にあるマイクロソフトの `nanoserver` イメージ。`latest` タグ付き。
`acme/web-api:12.0`	`acme` org と関連のある `web-api` イメージ バージョン `12.0`。このイメージは Docker Hub にある。
`gcr.io/gnschenker/sample-app:1.1`	`sample-app` イメージ。`1.1` のタグ付き。Google のコンテナ レジストリにある `gnschenker` ID を持つ個人に属する。

公式イメージ

上の表では、何度か公式イメージという用語を使いました。これには説明が必要です。イメージは Docker Hub レジストリのリポジトリに格納されています。公式リポジトリとは、Docker Hub でホストされているリポジトリのセットで、イメージの中にパッケージ化されているソフトウェアの責任も担っている個人や組織によってキュレーションされています。それが意味することを実例で見てみましょう。Ubuntu Linux ディストリビューションの背後には正式な組織があります。このチームは Ubuntu ディストリビューションを含む Docker イメージの公式バージョンも提供しています。

公式イメージは、必須のベース OS リポジトリ、一般的なプログラミング言語ランタイムのイメージ、頻繁に使用されるデータ ストレージ、およびその他の重要なサービスを提供することを目的としています。

Docker は、Docker Hub の非公開リポジトリにあるすべてのキュレーションされたイメージを見直して公開する役割を担っているチームに資金を提供しています。さらに、Docker はすべての公式イメージの脆弱性をスキャンします。

イメージをレジストリにプッシュする

カスタム イメージを作成することには問題はありませんが、ある時点で、テスト システム、QA システム、本番システムなどのターゲット環境にイメージを実際に配信して共有する必要があります。この場合、通常はコンテナ レジストリを使用します。最も人気のある公開レジストリの 1 つが Docker Hub です。Docker Hub は Docker 環境では既定のレジストリとして設定されており、ここまでのイメージはすべてこのレジストリから取得しています。

レジストリでは通常、個人または組織のアカウントを作成できます。たとえば、Docker Hub の私の個人アカウントは `gnschenker` です。個人アカウントは個人使用に適しています。レジストリを仕事で使用する場合は、Docker Hub で `acme` などの組織アカウントを作成することをお勧めします。後者の利点は、組織が複数のチームを持つことができることです。チームは異なる権限を持つことができます。

イメージを Docker Hub の自分の個人アカウントにプッシュできるようにするには、それに応じてタグ付けする必要があります。たとえば最新のバージョンの `alpine` を私のアカウントにプッシュして、`1.0` というタグを付けるとします。これは次のように行うことができます。

```
$ docker image tag alpine:latest gnschenker/alpine:1.0
```

イメージをプッシュできるようにするには、私のアカウントにログインする必要があります。

```
$ docker login -u gnschenker -p <my secret password>
```

ログインに成功すると、イメージをプッシュできます。

```
$ docker image push gnschenker/alpine:1.0
```

これに似たものがターミナルに表示されるでしょう。

```
The push refers to repository [docker.io/gnschenker/alpine]
04a094fe844e: Mounted from library/alpine
1.0: digest: sha256:5cb04fce... size: 528
```

Docker Hub にプッシュした各イメージについて、自動的にリポジトリが作成されます。リポジトリは非公開にすることも公開することもできます。公開リポジトリからは、誰でもイメージを取得できます。非公開リポジトリからは、レジストリにログインして必要な権限が設定されている場合にのみイメージを取得できます。

まとめ

この章では、コンテナ イメージの概要と、コンテナ イメージの構築・配信方法について詳しく説明しました。これまで見てきたように、イメージを作成するには、手動、自動、または tarball のシステムへのインポートという 3 つの方法があります。また、カスタム イメージを作成するときににによく使用されるベスト プラクティスについても学びました。

次の章では、コンテナの状態を保持するために使用できる Docker ボリュームを紹介します。また、Docker ホストのより詳細な調査や、Docker デーモンによって生成されたイベントの処理、未使用のリソースのクリーンアップに使用できる便利なシステム コマンドも紹介します。

質問

学習の進捗状況を評価するために、以下の質問にお答えください。

1. Ubuntu バージョン 17.04 から継承し、`ping` をインストールし、コンテナが開始されたときに `ping` を実行する Dockerfile をどのように作成しますか。ping の既定のアドレスは `127.0.0.1` です。
2. `alpine:latest` を使用し、`curl` をインストールする新しいコンテナ イメージをどのように作成しますか。新しいイメージに `my-alpine:1.0` という名前を付けます。
3. 複数の手順を使用して、C または Go で書かれた `Hello World` アプリのイメージを最小サイズで作成する Dockerfile を作成してください。
4. Docker コンテナ イメージの重要な特徴を 3 つ挙げてください。

5. foo:1.0 という名前のイメージを Docker Hub にある個人アカウント jdoe にプッシュするとします。次のうち正しいソリューションはどれですか。

    1. $ docker container push foo:1.0
    2. $ docker image tag foo:1.0 jdoe/foo:1.0
       $ docker image push jdoe/foo:1.0
    3. $ docker login -u jdoe -p <your password>
       $ docker image tag foo:1.0 jdoe/foo:1.0
       $ docker image push jdoe/foo:1.0
    4. $ docker login -u jdoe -p <your password>
       $ docker container tag foo:1.0 jdoe/foo:1.0
       $ docker container push jdoe/foo:1.0
    5. $ docker login -u jdoe -p <your password>
       $ docker image push foo:1.0 jdoe/foo:1.0

参考情報

以下の参考文献リストは、コンテナ イメージの作成と構築のトピックをより詳細に説明した資料を示しています。

- Dockerfile 作成のベスト プラクティス (http://dockr.ly/22WiJiO)
- 多段階のビルドを使用する (http://dockr.ly/2ewcUY3)
- Storage ドライバーについて (http://dockr.ly/1TuWndC)
- グラフドライバーのプラグイン (http://dockr.ly/2eIVCab)
- Docker for MAC でのユーザーガイド キャッシング (http://dockr.ly/2xKafPf)

5
データ ボリュームとシステム管理

前章では、独自のコンテナ イメージを構築して共有する方法について学習しました。特に、コンテナ化アプリケーションにとって本当に必要とされる成果物のみを含む、きわめて小さなイメージを構築する方法に焦点を当てました。

この章では、ステートフル コンテナを使用する方法について学習します。ステートフル コンテナは、データを消費し、生産するコンテナです。また、Docker 環境をクリーンに保ち、未使用のリソースが存在しない状態にする方法についても学習します。最後に、Docker エンジンによって生成されるイベント ストリームについても見ていきます。

この章で取り上げるトピックは以下のとおりです。

- データ ボリュームの作成とマウント
- コンテナ間でのデータの共有
- ホスト ボリュームの使用
- イメージでのボリュームの定義
- 包括的な Docker システム情報の取得
- リソース消費量の一覧表示
- 未使用リソースの削除
- Docker システム イベントの消費

この章を読み終わると、次のことができるようになります。

- データ ボリュームを作成、削除、一覧表示する
- 既存データ ボリュームをコンテナにマウントする
- データ ボリュームを使用してコンテナ内から永続データを作成する
- データ ボリュームを使用して複数のコンテナ間でデータを共有する
- データ ボリュームを使用して任意のホスト フォルダをコンテナにマウントする
- データ ボリューム内のデータへのアクセス時に、コンテナのアクセス モード (読み取り / 書き込み、または読み取り専用) を定義する
- 特定のホスト上で Docker リソース (イメージ、コンテナ、ボリュームなど) が消費する領域の量を一覧表示する
- システムから未使用の Docker リソース (コンテナ、イメージ、ボリュームなど) を解放する
- Docker システムのイベントをコンソールにリアルタイムで表示する

技術的要件

この章では、マシンに Docker Toolbox がインストールされているか、ノート PC またはクラウド上で Docker を実行する Linux VM にアクセスする必要があります。この章に付随するコードはありません。

データ ボリュームの作成とマウント

意味のあるアプリケーションはすべて、データを消費または生産します。しかし、コンテナはステートレスであるのが理想的です。この問題に、どのように対処したらよいでしょうか。1 つの方法は、Docker ボリュームを使用することです。ボリュームを使用することで、コンテナでの消費、生産、および状態の変更が可能になります。ボリュームのライフ サイクルは、コンテナのライフ サイクルより長くなります。ボリュームを使用するコンテナが稼働を停止しても、ボリュームは引き続き使用できます。これは、状態の永続性を実現する優れた仕組みです。

コンテナ レイヤーの変更

ボリュームについて学習する前に、まず、コンテナ内のアプリケーションがコンテナのファイル システムに何らかの変更を加えた場合、何が起こるのかを見ていきましょう。このような場合、変更はすべて、書き込み可能なコンテナ レイヤーで発生します。実際の動作を簡単に確認してみましょう。まず、コンテナを実行し、ファイルを新規作成するスクリプトをこのコンテナ内で実行します。

```
$ docker container run --name demo \
    alpine /bin/sh -c 'echo "This is a test" > sample.txt'
```

このコマンドは、demo という名前のコンテナを作成し、このコンテナ内に sample.txt というファイルを作成します。このファイルには「This is a test」と記述されます。この後コンテナは終了しますが、メモリ内には残るので、これを検証します。diff コマンドを使用して、イメージのファイル システムと比較し、コンテナのファイル システムで何が変更されたかを検証します。

```
$ docker container diff demo
```

出力は次のようになります。

```
A /sample.txt
```

予想どおり、コンテナのファイル システムには A という新規ファイルが追加されていることがすぐに分かります。基盤となるイメージ (ここでは alpine) から生成されたレイヤーはすべて変更不可能であり、変更が行われるのは書き込み可能なコンテナ レイヤーのみです。

ここで、メモリからコンテナを削除した場合、そのコンテナ レイヤーも削除されます。これに伴ってすべての変更も削除され、削除を元に戻すことはできません。コンテナの有効期間を超過した後も、この変更を維持したいのであれば、この方法は適切ではありません。こうした場合に、Docker ボリュームという選択肢が役に立ちます。詳しく見ていきましょう。

ボリュームの作成

この時点では、Docker for Mac または Docker for Windows を使用する場合、コンテナは OS X または Windows 上でネイティブに実行されているわけではなく、Docker for Mac または Docker for Windows によって作成された (非表示の) VM 内で実行されています。この場合に最適な方法は、docker-machine を使用して、Docker を実行する明示的な VM を作成および使用することです。ここでは、システムに Docker Toolbox がインストールされていることを前提としています。そうでない場合は、第 2 章「作業環境の設定」に戻り、Docker Toolbox の詳細なインストール方法を参照してください。

docker-machine を使用して、現在 VirtualBox 内で実行されているすべての VM を一覧表示します。

```
$ docker-machine ls
```

node-1 という名前の VM が存在しない場合は、ここで作成します。

```
$ docker-machine create --driver virtualbox node-1
```

node-1 という名前の VM は存在するが、まだ実行されていない場合は、ここで起動します。

```
$ docker-machine start node-1
```

これで、準備がすべて整いました。次に、「node-1」という名前の VM に SSH 接続します。

```
$ docker-machine ssh node-1
```

boot2docker のウェルカム メッセージが表示されるはずです。

データ ボリュームを新規作成するには、`docker volume create` コマンドを使用します。このコマンドで作成した名前付きボリュームを、コンテナにマウントすることで、永続的なデータ アクセスまたはストレージを実現できます。次のコマンドは、既定のボリューム ドライバーを使用して、`my-data` というボリュームを作成します。

```
$ docker volume create my-data
```

既定のボリューム ドライバーは、一般的にはローカル ドライバーと呼ばれ、ホストのファイル システム内にデータをローカルに格納します。データがホスト上のどこに格納されているかを確認する最も簡単な方法は、先ほど作成したボリューム上で `inspect` コマンドを実行することです。実際の場所はシステムによって異なる可能性があるため、この方法では、対象フォルダを最も確実に特定できます。

```
$ docker volume inspect my-data
[
    {
        "CreatedAt": "2018-01-28T21:55:41Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/mnt/sda1/var/lib/docker/volumes/my-data/_
data",
        "Name": "my-data",
        "Options": {},
        "Scope": "local"
    }
]
```

ホスト フォルダを確認するには、出力の `Mountpoint` の項目を調べます。ここでは、VirtualBox で実行される LinuxKit ベースの VM で `docker-machine` を使用しているのであれば、対象フォルダは `/mnt/sda1/var/lib/docker/volumes/my-data/_data` となります。

対象フォルダが保護されている場合も多く、その場合は `sudo` を使用してこのフォルダに移動してから、任意の操作を実行する必要があります。ここでは、`sudo` を使用する必要はありません。

```
$ cd /mnt/sda1/var/lib/docker/volumes/my-data/_data
```

Docker for Mac を使用してノート PC 上にボリュームを作成した場合、作成したボリューム上で `docker volume inspect` を実行すると、`Mountpoint` は `/var/lib/docker/volumes/my-data/_data` のように表示されます。しかし、Mac マシン上にはそのようなフォルダは存在しません。これは、このフォルダが、コンテナを実行するために Docker for Mac が使用する非表示の VM に対する相対パスとなっているためです。現時点では、OS X 上でコンテナをネイティブに実行することはできません。これは、Docker for Windows で作成されたボリュームでも同様です。

プラグインの形でサード パーティから提供されている、他のボリューム ドライバーを使用することもできます。別のボリューム ドライバーを選択するには、create コマンドで --driver パラメーターを指定します。他のボリューム ドライバーは、クラウド ストレージ、NFS ドライブ、ソフトウェア定義ストレージなど、さまざまな種類のストレージ システムを使用してボリュームを支持します。

ボリュームのマウント

名前付きボリュームを作成すると、このボリュームをコンテナにマウントできます。これには、docker container run コマンドで -v パラメーターを指定します。

```
$ docker container run --name test -it \
     -v my-data:/data alpine /bin/sh
```

このコマンドは、my-data ボリュームを、コンテナ内の /data フォルダにマウントします。これで、コンテナ内の /data フォルダにファイルを作成できるようになりました。ファイル作成後に終了します。

```
# / cd /data
# / echo "Some data" > data.txt
# / echo "Some more data" > data2.txt
# / exit
```

ボリューム データを保管するホスト フォルダに移動して、そのフォルダの内容を一覧表示すると、次のように、コンテナ内に作成した 2 つのファイルが表示されます。

```
$ cd /mnt/sda1/var/lib/docker/volumes/my-data/_data
$ ls -l
total 8
-rw-r--r-- 1 root root 10 Jan 28 22:23 data.txt
-rw-r--r-- 1 root root 15 Jan 28 22:23 data2.txt
```

たとえば、2 番目のファイルの内容を出力することもできます。

```
$ cat data2.txt
```

ホストからこのフォルダ内にファイルを作成し、別のコンテナでこのボリュームを使用してみましょう。

```
$ echo "This file we create on the host" > host-data.txt
```

次に、test コンテナを削除し、CentOS ベースの別のコンテナを実行します。ここでは、一歩進めて、ボリュームを別のコンテナ フォルダ「/app/data」にマウントします。

```
$ docker container rm test
$ docker container run --name test2 -it \
    -v my-data:/app/data \
    Centos:7 /bin/bash
```

CentOS コンテナ内では、ボリュームをマウントした /app/data フォルダに移動して、その内容を一覧表示できます。

```
# / cd /app/data
# / ls -l
```

予想どおり、次の 3 つのファイルが表示されます。

```
-rw-r--r-- 1 root root 10 Jan 28 22:23 data.txt
-rw-r--r-- 1 root root 15 Jan 28 22:23 data2.txt
-rw-r--r-- 1 root root 32 Jan 28 22:31 host-data.txt
```

これで、Docker ボリューム内のデータがコンテナの有効期間を超過して永続することが証明されました。また、最初にボリュームを使用したコンテナとは別のコンテナも、同じボリュームを再利用できることが分かりました。

Docker ボリュームをマウントするコンテナ内のフォルダは、Union ファイル システムからは除外されることに注意が必要です。つまり、このフォルダ、およびそのサブフォルダ内で加えられた変更はすべて、コンテナ レイヤーの一部ではないのですが、ボリューム ドライバーの提供するバッキング ストレージ内で永続します。コンテナ レイヤーに対応するコンテナが停止され、システムから削除されると、コンテナ レイヤーも削除されてしまうので、この点は非常に重要です。

ボリュームの削除

ボリュームを削除するには、docker volume rm コマンドを使用します。ボリュームを削除すると、中のデータも破棄され、元に戻すことができないので注意が必要です。このため、このコマンドは危険なコマンドとして認識する必要があります。Docker では、コンテナで使用中のボリュームを削除することはできないため、誤操作の危険がやや緩和されます。ボリュームを削除する場合は必ず、データのバックアップが作成済みであること、あるいはこのデータが本当に不要であることを事前に確認してください。

次のコマンドは、以前に作成した my-data ボリュームを削除します。

```
$ docker volume rm my-data
```

上のコマンドを実行した後、ホスト上のフォルダが削除されたことを再確認してください。

実行中のすべてのコンテナを削除して、システムをクリーンアップするには、次のコマンドを実行します。

```
$ docker container rm -f $(docker container ls -aq)
```

コンテナ間でのデータの共有

コンテナは、実行中のいくつものアプリケーションを囲い込んでいるサンドボックス (砂場) のようなものです。これは、それぞれ異なるコンテナ上で実行されるアプリケーションを相互に保護するために、非常に有益で必要とされる措置です。つまり、あるコンテナ内で実行されるアプリケーションから見えるファイル システム全体は、そのアプリケーション固有のものであり、別のコンテナ内で実行される他のアプリケーションから干渉されることがありません。

とはいえ、コンテナ間でデータを共有しなければならない場合もあります。たとえば、コンテナ A で実行されるアプリケーションが生成したデータを、コンテナ B で実行されるアプリケーションで使用したいとします。どのように対処したらよいでしょうか。もうお分かりですね。Docker ボリュームを使用すれば目的を果たすことができます。この場合、ボリュームを 1 つ作成し、コンテナ A とコンテナ B の両方にマウントします。こうすることで、アプリケーション A と B が同じデータにアクセスできます。

ここでも、複数のアプリケーションやプロセスがデータに同時にアクセスする場合は、従来どおり、不整合を回避するように十分注意する必要があります。競合状態など、同時実行に関する問題を回避するため、データを作成または変更できるアプリケーション (プロセス) は 1 つだけにし、このデータに同時にアクセスする他のすべてのプロセスは読み取り専用であるのが理想的です。コンテナ内で実行されるプロセスの機能を、ボリューム内のデータの読み取りだけに制限するには、このボリュームを読み取り専用としてマウントします。次のコマンドを見てください。

```
$ docker container run -it --name writer \
    -v shared-data:/data \
    alpine /bin/sh
```

ここでは、`writer` という名前のコンテナを作成します。このコンテナには `shared-data` というボリュームがあり、既定の「読み取り / 書き込みモード」でマウントされています。このコンテナ内に、ファイルを作成してみましょう。

```
# / echo "I can create a file" > /data/sample.txt
```

このコマンドは成功するはずです。このコンテナを終了し、次のコマンドを実行します。

```
$ docker container run -it --name reader \
    -v shared-data:/app/data:ro \
    ubuntu:17.04 /bin/bash
```

今度は reader という名前のコンテナを作成し、同じボリュームを読み取り専用 (ro) としてマウントしました。まず、最初のコンテナ内に作成したファイルが表示されることを確認します。

```
$ ls -l /app/data
total 4
-rw-r--r-- 1 root root 20 Jan 28 22:55 sample.txt
```

次に、ファイルを作成してみましょう。

```
# / echo "Try to break read/only" > /app/data/data.txt
```

このコマンドは失敗し、次のメッセージが表示されます。

```
bash: /app/data/data.txt: Read-only file system
```

コマンド プロンプトで「exit」と入力し、このコンテナを終了します。ホストに戻り、すべてのコンテナとボリュームをクリーンアップします。

```
$ docker container rm -f $(docker container ls -aq)
$ docker volume rm $(docker volume ls -q)
```

ここまで完了したら、docker-machine VM を終了します。この場合も、コマンド プロンプトで「exit」と入力します。これで、Docker for Mac または Docker for Windows に戻ります。docker-machine を使用して、VM を停止します。

```
$ docker-machine stop node-1
```

ホスト ボリュームの使用

特定の状況において、たとえば新たなコンテナ化アプリケーションを開発する場合、またはコンテナ化アプリケーションにおいて、たとえばレガシー アプリケーションによって生成された特定のフォルダ内のデータを使用する必要が生じた場合など、特定のホスト フォルダをマウントしたボリュームを使用すると、非常に便利です。次の例を見てみましょう。

```
$ docker container run --rm -it \
    -v $(pwd)/src:/app/src \
    alpine:latest /bin/sh
```

このコマンドは、シェルで alpine コンテナをインタラクティブに起動し、現在のディレクトリの src サブフォルダをコンテナの /app/src にマウントします。ここでは、現在のディレクトリを表す $(pwd) (つまり「pwd」) を使用する必要があります。ボリュームの操作では、常に絶対パスを使用する必要があるためです。

これは、開発者が頻繁に使用する手法です。コンテナ内で実行しているアプリケーションを処理する場合、この手法を使用すれば、コードに加えた最新の変更がコンテナに常に反映されていることを確認できます。変更のつど、イメージを再構築し、コンテナを再実行する必要はありません。

では、これが正しく機能することを実証するサンプルを作成しましょう。Nginx を Web サーバーとして使用して、単純で静的な Web サイトを作成するとします。まず、HTML、CSS、JavaScript ファイルといった Web 資産を格納するための新しいフォルダをホスト上に作成し、ここに移動します。

```
$ mkdir ~/my-web
$ cd ~/my-web
```

次のような単純な Web ページを作成します。

```
$ echo "<h1>Personal Website</h1>" > index.html
```

次に、Dockerfile を追加します。このファイルには、サンプル Web サイトを包含するイメージの構築方法が記述されています。フォルダに、次の内容を含む Dockerfile ファイルを追加します。

```
FROM nginx:alpine
COPY ./usr/share/nginx/html
```

Dockerfile は、最新バージョンの Alpine による Nginx を起動し、現在のホスト ディレクトリ内のすべてのファイルを、コンテナ フォルダ「`/usr/share/nginx/html`」にコピーします。これは、Nginx が Web 資産の格納場所として認識しているフォルダです。次のコマンドを実行し、イメージを構築します。

```
$ docker image build -t my-website:1.0 .
```

最後に、このイメージからコンテナを実行します。コンテナは分離モードで実行します。

```
$ docker container run -d \
   -p 8080:80 --name my-site\
   my-website:1.0
```

`-p 8080:80` パラメーターに注目してください。このパラメーターについてはまだ説明していませんでしたが、第 7 章「シングルホスト ネットワーキング」で詳しく説明します。ここでは、このパラメーターは Nginx が着信要求をリッスンするコンテナ ポート `80` を、ノート PC のポート `8080` にマップし、ここからアプリケーションへのアクセスを可能にする役割を果たすことだけを理解しておいてください。次に、ブラウザのタブを開き、`http://localhost:8080/index.html` にアクセスします。まだ「`Personal Website`」というタイトルしかない Web サイトが表示されるはずです。

次に、お好きなエディターを使用して、index.html ファイルを次のように編集します。

```
<h1>Personal Website</h1>
<p>This is some text</p>
```

ファイルを保存して、ブラウザを更新します。これでは変更が反映されませんね。ブラウザには、タイトルしかない、変更前の index.html が依然として表示されています。今度は、現在のコンテナを停止して削除してから、イメージを再構築し、コンテナを再実行してみましょう。

```
$ docker container rm -f my-site
$ docker image build -t my-website:1.0 .
$ docker container run -d \
   -p 8080:80 --name my-site\
   my-website:1.0
```

今度は、ブラウザを更新すると、新しいコンテンツが表示されました。うまくいきましたが、これでは手間が掛かりすぎです。Web サイトで簡単な変更を加えるたびに、この一連の作業をこなさなければならないとしたらどうでしょうか。あまりにも非現実的です。

今度は、ホストにマウントされたボリュームを使用してみましょう。もう一度、現在のコンテナを削除し、ボリューム マウントで再実行します。

```
$ docker container rm -f my-site
$ docker container run -d \
   -v $(pwd):/usr/share/nginx/html \
   -p 8080:80 --name my-site\
   my-website:1.0
```

さらに、index.html に何らかのコンテンツを追加して、保存します。ブラウザを更新してみましょう。変更が反映されています。まさに、求めていた動作です。これは「エディット コンティニュ」エクスペリエンスとも呼ばれます。Web ファイルに多くの変更を加えても、ブラウザ上ですぐに結果を表示できます。イメージを再構築し、Web サイトを包含するコンテナを再起動する必要はありません。

この場合、更新が双方向に反映されることに注目してください。ホスト上で変更を加えると、この変更はコンテナに反映され、その逆も同様です。もう 1 つ重要な点が、コンテナのターゲット フォルダ /usr/share/nginx/html に現在のフォルダをマウントすると、ターゲット フォルダにそれまで置かれていた内容が、ホスト フォルダの内容に置き換わってしまうことです。

イメージでのボリュームの定義

ここで、第 3 章「コンテナの操作」で学んだことを少し思い出してみましょう。コンテナの起動時、各コンテナのファイル システムは基盤イメージによる変更不可のレイヤーと、このコンテナのみに固有の書き込み可能なコンテナ レイヤーで構成されていました。コンテナ内で実行されているプロセスがファイル システムに加えたすべての変更は、このコンテナ レイヤー内で永続化されます。コンテナが停止され、システムから削除されると、これに対応するコンテナ レイヤーもシステムから削除され、復元されることなく失われます。

データベースなど、コンテナ内で実行するアプリケーションには、コンテナの有効期間が超過した後も、そのデータを永続させたいものもあります。このような場合に、ボリュームを使用します。分かりやすく説明するために、具体的な例で考えてみましょう。MongoDB は、一般的なオープン ソースのドキュメント データベースです。多くの開発者は、アプリケーションのストレージ サービスとして MongoDB を使用しています。MongoDB の管理者はイメージを作成し、このイメージを Docker Hub に公開します。このイメージは、ここではデータベースのインスタンスをコンテナ内で実行するために使用されるとします。このデータベースによって生成されるデータは、長期保存が必要です。しかし、MongoDB の管理者は、このイメージを誰がどのように使用するのかは分かりません。したがって、データベース ユーザーがこのコンテナを起動するために使用する `docker container run` コマンドに対し、MongoDB 管理者が影響を与えることはありません。どのようにボリュームを定義したらよいでしょうか。

幸いにも、Dockerfile 内でボリュームを定義するという方法があります。このためのキーワードが `VOLUME` であり、単一フォルダへの絶対パスを追加することも、複数のパスをコンマで区切って追加することもできます。このようなパスは、コンテナのファイル システムのフォルダを表します。次に、このようなボリューム定義の例をいくつか紹介します。

```
VOLUME /app/data
VOLUME /app/data, /app/profiles, /app/config
VOLUME ["/app/data", "/app/profiles", "/app/config"]
```

1 行目の例では、単一ボリュームを `/app/data` にマウントするように定義しています。2 行目の例では、3 つのボリュームをコンマ区切りのリストとして定義しています。最後の例は 2 行目と同じ定義ですが、ここでは値を JSON 配列として指定しています。

コンテナを起動すると、Docker は自動的にボリュームを作成し、Dockerfile に定義された各パスのとおりに、コンテナの該当ターゲット フォルダにボリュームをマウントします。各ボリュームは Docker に自動作成されるため、ID は「SHA-256」となります。

コンテナ ランタイムでは、Dockerfile でボリュームとして定義されたフォルダは Union ファイル システムからは除外されます。このため、これらのフォルダに加えられた変更はすべて、コンテナ レイヤーには反映されず、それぞれのボリューム内で永続化されます。ボリュームのバッキング ストレージが適切にバックアップされていることを管理するのは、運用エンジニアの責任となります。

Dockerfile で定義されたボリュームに関する情報を収集するには、`docker image inspect` コマンドを使用します。MongoDB から得られる情報を確認しましょう。まず、次のコマンドでイメージを取得します。

```
$ docker image pull mongo:3.7
```

このイメージを検証していきます。`--format` パラメーターを使用して、大量のデータの中から重要な部分だけを抽出します。

```
$ docker image inspect \
   --format='{{json .ContainerConfig.Volumes}}' \
    mongo:3.7 | jq
```

これで、次のような結果が返されます。

```
{
"/data/configdb": {},
"/data/db": {}
}
```

ここから分かるように、MongoDB に対し Dockerfile は `/data/configdb`、`/data/db` に 2 つのボリュームを定義しています。

次のように、MongoDB のインスタンスを実行してみましょう。

```
$ docker run --name my-mongo -d mongo:3.7
```

ここで `docker container inspect` コマンドを使用すると、作成された 2 つのボリュームに関する情報を、その他の情報とともに取得できます。ボリューム情報だけを取得するには、次のコマンドを使用します。

```
$ docker inspect --format '{{json .Mounts}}' my-mongo | jq
```

次のような出力が表示されます。

```
[
  {
    "Type": "volume",
    "Name": "b9ea0158b5...",
    "Source": "/var/lib/docker/volumes/b9ea0158b.../_data",
    "Destination": "/data/configdb",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
```

```
    },
    {
      "Type": "volume",
      "Name": "5becf84b1e...",
      "Source": "/var/lib/docker/volumes/5becf84b1.../_data",
  "Destination": "/data/db",
      "Driver": "local",
      "Mode": "",
      "RW": true,
      "Propagation": ""
    }
]
```

`Name` および `Source` フィールドの値は、読みやすくするために、ここでは一部省略されていることに注意してください。`Source` フィールドには、コンテナ内の MongoDB によって生成されたデータが格納されるホスト ディレクトリへのパスが表示されます。

Docker システム情報の取得

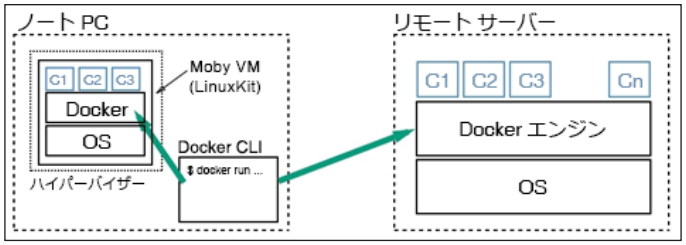
システムのトラブルシューティングが必要になった場合は、このセクションで紹介する各コマンドが重要な役割を果たします。これらのコマンドは、ホストにインストールされた Docker エンジン、およびホスト オペレーティング システムについてのさまざまな情報を提供します。最初に紹介するのは `docker version` コマンドです。このコマンドを使用すると、現在の構成で使用している Docker クライアントおよびサーバーについてのさまざまな情報を取得できます。CLI でこのコマンドを入力すると、次のような結果が表示されます。

```
$ docker version
Client:
 Version:       18.04.0-ce
 API version:   1.37
 Go version:    go1.9.4
 Git commit:    3d479c0
 Built: Tue Apr 10 18:13:16 2018
 OS/Arch:       darwin/amd64
 Experimental:  true
 Orchestrator:  swarm

Server:
 Engine:
  Version:      18.04.0-ce
  API version:  1.37 (minimum version 1.12)
  Go version:   go1.9.4
  Git commit:   3d479c0
  Built:        Tue Apr 10 18:23:05 2018
  OS/Arch:      linux/amd64
  Experimental: true
$
```

Docker に関するバージョン情報

この例では、クライアントとサーバーの両方で、バージョン `18.04.0-ce-rc2` の Docker エンジンが使用されていることが分かります。また、オーケストレーターが Swarm であることなども分かります。

クライアントとサーバーとを明確に理解するため、次の図を見てください。



CLI から複数の Docker ホストにアクセス

クライアントは小規模な CLI であり、ここから Docker ホストのリモート API に Docker コマンドを送信していたことが分かります。Docker ホストは、コンテナをホストするコンテナ ランタイムであり、同一マシン上で CLI として実行されることも、リモート サーバー上で実行されることもあれば、オンプレミスやクラウドで実行されることもあります。CLI を使用して、異なる複数のサーバーを管理できます。これには、`DOCKER_HOST`、`DOCKER_TLS_VERIFY`、`DOCKER_CERT_PATH` などの複数の環境変数を設定します。これらの環境変数が作業マシンに設定されていない状態で、Docker for Mac または Docker for Windows を使用している場合は、自分のマシン上で動作する Docker エンジンを使用していることを意味します。

次に重要なコマンドは、`docker system info` コマンドです。このコマンドは、Docker エンジンが現在動作しているモードの種類 (Swarm モードかどうか)、Union ファイル システムで使用されているストレージ ドライバーの種類、ホスト上の Linux カーネルのバージョンなど、さまざまな情報を提供します。このコマンドを実行して、システムによって生成される出力を注意深く確認してください。どのような情報が表示されるかを分析してみましょう。

```
$ docker system info
Containers: 1
 Running: 0
 Paused: 0
 Stopped: 1
Images: 70
Server Version: 18.04.0-ce
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host ipvlan macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 773c489c9c1b21a6d78b5c538cd395416ec50f88
runc version: 4fc53a81fb7c994640722ac585fa9ca548971871
init version: 949e6fa
Security Options:
 seccomp
  Profile: default
Kernel Version: 4.9.87-linuxkit-aufs
Operating System: Docker for Mac
OSType: linux
Architecture: x86_64
CPUs: 4
Total Memory: 1.952GiB
Name: linuxkit-025000000001
ID: WV5X:CY7N:LHIP:SWJ2:T55W:P5QM:MEYU:MM3V:55OH:RALF:5ZDN:QH7Y
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): true
 File Descriptors: 22
 Goroutines: 42
 System Time: 2018-04-21T12:08:17.962868Z
 EventsListeners: 2
HTTP Proxy: gateway.docker.internal:3128
HTTPS Proxy: gateway.docker.internal:3129
Registry: https://index.docker.io/v1/
Labels:
Experimental: true
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
```

docker system info コマンドの出力

リソース消費量の一覧表示

時間の経過とともに、Docker ホストはイメージ、コンテナ、ボリュームなどの大量のリソースをメモリ内やディスク上に蓄積していく可能性があります。整理整頓の行き届いた家の中のように、作業環境もクリーンに保ち、未使用のリソースは解放して領域を確保する必要があります。そうしないと、いずれは Docker で新たなリソースをこれ以上追加できない状態に陥ります。このような状態では、ディスクやメモリの空き容量が不足しているため、イメージ取得などのアクションが失敗する可能性もあります。

Docker CLI には、手軽に実行できる `system` というコマンドがあります。このコマンドを使用すると、現時点でシステム上で使用されているリソースの量と、解放可能な領域の量を一覧表示できます。コマンドは次のとおりです。

```
$ docker system df
```

システムでこのコマンドを実行すると、次のような出力が表示されます。

```
TYPE            TOTAL   ACTIVE   SIZE      RECLAIMABLE
Images          21      9        1.103GB   845.3MB (76%)
Containers      14      11       9.144kB   4.4kB (48%)
Local Volumes   14      14       340.3MB   0B (0%)
Build Cache                      0B        0B
```

出力の最後の行にある `Build Cache` は、新しいバージョンの Docker のみで表示されます。この情報は最近追加されたものです。以下に、この出力の各項目について説明します。

- この例では、システム上に現在 `21` のイメージがローカルにキャッシュされており、そのうち `9` つのイメージがアクティブに使用されていることが分かります。イメージがアクティブに使用されているとみなされるのは、現時点で、実行中または停止された 1 つ以上のコンテナが、このイメージに基づいている場合です。これらのイメージにより、`1.1` GB のディスク領域が占有されています。技術的には、約 `845` MB の領域を解放できる可能性があります。対応するイメージが現在使用されていないためです。
- また、システム上には `11` の実行中のコンテナがあり、停止されたコンテナが 3 つあり、コンテナの合計数は `14` となっています。この例では、停止されたコンテナによって占有されている領域 `4.4 kB` を解放できます。
- ホスト上でアクティブなボリュームの数は `14` であり、これらによって合計 `340` MB のディスク領域が消費されています。すべてのボリュームが使用中であるため、この時点では領域を解放できません。
- 最後に、この例での `Build Cache` は現時点で空であり、当然ながら、この分の領域は解放できません。

システムのリソース消費について、より詳細な情報を取得するには、このコマンドを詳細モードで実行します。これには、次のように -v フラグを使用します。

```
$ docker system df -v
```

これで、すべてのイメージ、コンテナ、およびボリュームの詳細情報が、それぞれのサイズとともに表示されます。出力は次のようになります。

```
$ docker system df -v
Images space usage:

REPOSITORY                          TAG                 IMAGE ID            CREATED ago         SIZE                SHARED SIZE         UNIQUE SiZE         CONTAINERS
fundamentalsofdocker/ch14-web       1.0                 fe6612f845be        12 days ago ago     72.05MB             72.05MB             1.834kB             0
fundamentalsofdocker/web            2.0                 944523644dd7        12 days ago ago     72.05MB             72.05MB             1.827kB             0
fundamentalsofdocker/ch13-web       2.0                 b47675c8e53c        2 weeks ago ago     72.05MB             72.05MB             1.836kB             0
builder2                            latest              074a85b21f3a        3 weeks ago ago     1.514GB             1.456GB             58.7MB              0
builder                             latest              9864221c5187        3 weeks ago ago     1.464GB             1.456GB             8.051MB             0
ruby                                alpine              b620ae34414c        3 weeks ago ago     55.52MB             4.148MB             51.38MB             1
microsoft/azure-cli                 latest              a52f6e53da4c        4 weeks ago ago     400.4MB             0B                  400.4MB             0
nginx                               alpine              91ce6206f9d8        4 weeks ago ago     18MB                4.148MB             13.86MB             1
perl                                5.26                3c2c4c3b2e15        5 weeks ago ago     879.2MB             879.2MB             0B                  0
fundamentalsofdocker/ch08-web       1.0                 922e085ed002        7 weeks ago ago     72.01MB             68.02MB             3.992MB             0
fundamentalsofdocker/ch08-db        1.0                 4953d5353c17        7 weeks ago ago     39.46MB             4.148MB             35.31MB             0
node                                9.6-alpine          a88ff852e3d4        8 weeks ago ago     68.02MB             68.02MB             0B                  0
alpine                              latest              3fd9065eaf02        3 months ago ago    4.148MB             4.148MB             0B                  0
confluentinc/cp-enterprise-kafka    4.0.0               07d41f8648f5        3 months ago ago    565.1MB             0B                  565.1MB             0
hello-world                         latest              f2a91732366c        5 months ago ago    1.848kB             0B                  1.848kB             0
hseeberger/scala-sbt                latest              da0e1be3bb79        9 months ago ago    925.4MB             0B                  925.4MB             0

Containers space usage:

CONTAINER ID        IMAGE               COMMAND             LOCAL VOLUMES       SIZE                CREATED ago         STATUS                      NAMES
afe0dcab9bc4        nginx:alpine        "ping 8.8.8.8"      0                   0B                  6 seconds ago ago   Up 4 seconds                xenodochial_easley
2a2d742604af        ruby:alpine         "/bin/sh"           0                   16.7MB              5 hours ago ago     Exited (0) 13 minutes ago   keen_lumiere

Local Volumes space usage:

VOLUME NAME         LINKS               SIZE
ch08_pets-data      0                   47.24MB

Build cache usage: 0B

$
```

Docker によって消費されるシステム リソースの詳細な出力

この詳細出力を参照にすることで、システムのクリーンアップを開始する必要があるかどうか、クリーンアップが必要なのはシステムのどの部分か、十分な情報に基づいた判断を下すことができます。

未使用リソースの削除

クリーンアップが必要であると判断した場合、Docker にはいわゆる削除コマンドが用意されています。イメージ、コンテナ、ボリューム、ネットワークなどの個々のリソースに対し、prune コマンドを実行できます。

コンテナの削除

このセクションでは、コンテナを削除することで、未使用のシステム リソースを回復します。まず、次のコマンドを実行します。

```
$ docker container prune
```

このコマンドを実行すると、システムから、状態が `running` ではないすべてのコンテナが削除されます。現在の状態が `exited` または `created` であるコンテナを削除しようとすると、Docker の確認メッセージが表示されます。この確認手順を省略するには、次のように `-f` (または `--force`) フラグを使用します。

```
$ docker container prune -f
```

特定の状況下では、実行中のコンテナを含め、システムからすべてのコンテナを削除したい場合があります。この場合は `prune` コマンドは使用できません。代わりに、次のように、コマンドを組み合わせて実行する必要があります。

```
$ docker container rm -f $(docker container ls -aq)
```

このコマンドの実行には注意が必要です。このコマンドを実行すると、警告が表示されることなく、実行中のコンテナも含めたすべてのコンテナが削除されます。次のセクションに進む前に、上記のコマンドをもう一度よく見て、このコマンドの実行によって何が、どのような理由で行われるのかを説明できるようにしてください。

イメージの削除

次は、イメージの削除です。未使用のイメージ レイヤーによって占有されているすべての領域を解放するには、次のコマンドを使用します。

```
$ docker image prune
```

Docker の確認メッセージに対し、未使用のイメージ レイヤーによる占有領域の解放を確定すると、未使用イメージが削除されます。ここで、未使用のイメージ レイヤーとは何かについて説明する必要があります。前章で学習した内容を思い出してください。1 つのイメージは、変更不可能なレイヤーの積み重ねによって構成されていましたね。カスタム イメージを何度も作成していると、作成中のイメージの対象となるアプリケーションのソース コードなどに変更を加えるたびに、レイヤーを再作成していることになり、このレイヤーの以前のバージョンが孤立してしまいます。これはなぜでしょうか。前章で詳しく説明したように、各レイヤーは変更不可能であるためです。したがって、レイヤーの構築に使用されているソースに何らかの変更が加わると、このレイヤーは再構築されることになり、前のバージョンのレイヤーは破棄されます。

何度もイメージを構築していると、時間が経つにつれ、システム上で孤立したイメージ レイヤーの数が大幅に増加する可能性があります。このように孤立したすべてのレイヤーを削除するには、上述の `prune` コマンドを使用します。

コンテナに対して `prune` コマンドを実行する場合と同じく、次のように force フラグを使用することで、Docker による削除の確認を回避できます。

```
$ docker image prune -f
```

さらに抜本的にイメージを削除する `prune` コマンドもあります。孤立したイメージレイヤーを削除するだけでは不十分で、システムで現時点で使用されていないすべてのイメージを削除しなければならない場合もあります。このような場合には、次のように `-a` (または `--all`) フラグを使用します。

```
$ docker image prune --force --all
```

このコマンドを実行すると、1 つ以上のコンテナによって現在使用されているイメージのみが、ローカル イメージ キャッシュ内に残ります。

ボリュームの削除

Docker ボリュームは、コンテナによるデータへの永続的なアクセスを可能にするために使用されます。アクセス先のデータが重要である場合もあるため、このセクションで説明するコマンドは、特に注意して実行する必要があります。

ボリュームによって占有されているスペースを解放する必要があり、これによって基盤データが破棄され、元に戻すことができないことを理解している場合は、次のコマンドを実行します。

```
$ docker volume prune
```

このコマンドは、1 つ以上のコンテナによって現在使用されていないすべてのボリュームを削除します。

これは破壊的なコマンドであり、元に戻すことはできません。削除対象のデータが今後、確実に不要であることが明らかな場合を除き、ボリュームを削除する前に、このボリュームに関連付けられたデータのバックアップを必ず作成しておいてください。

システムの破損やアプリケーションの誤動作を防ぐため、Docker では、現在 1 つ以上のコンテナによって使用されているボリュームを削除することはできません。これは、停止したコンテナによって使用されているボリュームの場合も同様です。必ず、ボリュームを使用するコンテナから先に削除する必要があります。

ボリュームを削除する場合に便利なのが、`-f` または `--filter` フラグです。これを使用すると、削除の対象とするボリューム セットを指定できます。次のコマンドを見てください。

```
$ docker volume prune --filter 'label=demo'
```

このコマンドでは、`label` の値が「`demo`」であるボリュームだけが削除の対象となります。フィルタリング フラグの形式は「`key=value`」です。複数のフィルターが必要な場合は、次のように複数のフラグを使用できます。

```
$ docker volume prune --filter 'label=demo' --filter 'label=test'
```

フィルター フラグは、コンテナやイメージなどの他のリソースを削除するときにも使用できます。

ネットワークの削除

ネットワークも、削除コマンドによって削除できます。ネットワークについては、第 7 章「シングルホスト ネットワーキング」で詳しく説明します。未使用のネットワークをすべて削除するには、次のコマンドを使用します。

```
$ docker network prune
```

このコマンドにより、現時点でコンテナまたはサービスが接続されていないネットワークが削除されます。ここでは、ネットワークについてはこれ以上触れません。以降の章で詳しく説明するので、そこで理解が深まるはずです。

すべて削除する

複数のコマンドを入力することなく、一度にすべてを削除するには、次のコマンドを使用します。

```
$ docker system prune
```

Docker CLI からの確認メッセージを確定すると、未使用のすべてのコンテナ、イメージ、ボリューム、ネットワークが、適切な順序で一度に削除されます。

ここでも、Docker による確認メッセージを回避するには、コマンドに force フラグを指定します。

Docker システム イベントの消費

Docker エンジンでは、コンテナや他のリソース (ボリューム、ネットワークなど) を作成、実行、停止、削除すると、イベントのログが生成されます。これらのイベントは、情報に基づく意思決定に使用するため、インフラ サービスなどの外部システムによって利用することもできます。このようなサービスの例としては、現在システム上で実行されているすべてのコンテナのインベントリを作成するツールがあります。

このシステム イベント ストリームにフックを設定し、ターミナルなどに出力するには、次のコマンドを使用します。

```
$ docker system events
```

このコマンドはブロッキング コマンドです。したがって、ターミナル セッション内でこのコマンドを実行すると、このセッションがブロックされます。このため、このコマンドを実行する場合は、常に別のウィンドウを開くことをお勧めします。

別のターミナル ウィンドウで前述のコマンドを実行したと仮定すると、次のように、これをテストするためにコンテナを実行できます。

```
$ docker container run --rm alpine echo "Hello World"
```

生成される出力は次のようになります。

```
2018-01-28T15:08:57.318341118-06:00 container create 8e074342ef3b2
0cfa73d17e4ef7796d424aa8801661765ab5024acf166c6ecf3 (image=alpine,
name=confident_hopper)


2018-01-28T15:08:57.320934314-06:00 container attach 8e074342ef3b2
0cfa73d17e4ef7796d424aa8801661765ab5024acf166c6ecf3 (image=alpine,
name=confident_hopper)


2018-01-28T15:08:57.354869473-06:00 network connect c8fd270e1a77
6c5851c9fa1e79927141a1e1be228880c0aace4d0daebccd190f (container=
8e074342ef3b20cfa73d17e4ef7796d424aa8801661765ab5024acf166c6ecf3,
name=bridge, type=bridge)


2018-01-28T15:08:57.818494970-06:00 container start 8e074342ef3b20
cfa73d17e4ef7796d424aa8801661765ab5024acf166c6ecf3 (image=alpine,
name=confident_hopper)


2018-01-28T15:08:57.998941548-06:00 container die 8e074342ef3b20
cfa73d17e4ef7796d424aa8801661765ab5024acf166c6ecf3 (exitCode=0,
image=alpine, name=confident_hopper)
2018-01-28T15:08:58.304784993-06:00 network disconnect c8fd270e1a7
76c5851c9fa1e79927141a1e1be228880c0aace4d0daebccd190f (container=
8e074342ef3b20cfa73d17e4ef7796d424aa8801661765ab5024acf166c6ecf3,
name=bridge, type=bridge)


2018-01-28T15:08:58.412513530-06:00 container destroy 8e074342ef3b
20cfa73d17e4ef7796d424aa8801661765ab5024acf166c6ecf3 (image=alpine,
name=confident_hopper)
```

この出力では、コンテナのライフサイクルを詳細に追跡できます。コンテナは作成され、起動され、最後に破棄されます。このコマンドによって生成される出力の形式を変更したい場合は、コマンドに `--format` パラメーターを指定することで、いつでも変更できます。出力形式の値は、Go テンプレート構文を使用して記述する必要があります。次の例は、イベントのタイプ、イメージ、およびアクションを出力します。

```
$ docker system events --format 'Type={{.Type}} Image={{.Actor.
Attributes.image}} Action={{.Action}}'
```

先ほどとまったく同じ `container run` コマンドを実行すると、今度は次のように出力されます。

```
Type=container  Image=alpine     Action=create
Type=container  Image=alpine     Action=attach
Type=network    Image=<no value> Action=connect
Type=container  Image=alpine     Action=start
Type=container  Image=alpine     Action=die
Type=network    Image=<no value> Action=disconnect
Type=container  Image=alpine     Action=destroy
```

まとめ

この章では Docker ボリュームを紹介しました。Docker ボリュームを使用すると、コンテナによって生成された状態を永続化し、データを永続化できます。ボリュームを使用することで、さまざまなソースからのデータをコンテナに提供することもできます。ボリュームの作成、マウント、使用方法についても学びました。名前で定義する方法、ホスト ディレクトリにマウントする方法、コンテナ イメージ内でボリュームを定義する方法など、ボリュームを定義するためのさまざまな方法を学びました。

この章では、システムのトラブルシューティング、または Docker によって使用されるリソースの管理と排除に役立つ多様な情報を提供する、さまざまなシステム レベル コマンドについても学習しました。最後に、コンテナ ランタイムによって生成されたイベント ストリームを視覚化し、状況に応じてこれを使用する方法について学びました。

次の章では、コンテナ オーケストレーションの基本について説明します。ここでは、1 つあるいは小数のコンテナではなく、クラスター内の多数のノード上にある数百規模のコンテナを管理および実行する場合に、何が必要かについて検証します。このような状況では、解決すべき課題がたくさんあります。ここで出番となるのがオーケストレーション エンジンです。

質問

学習の進捗状況を評価するために、以下の質問にお答えください。

1. 既定のドライバーを使用して、たとえば `my-products` のような名前のついたデータ ボリュームをどのように作成しますか。
2. `alpine` イメージを使用してコンテナを実行し、`my-products` ボリュームを読み取り専用モードで /data コンテナ フォルダにマウントするにはどうすればいいでしょうか。

3. my-products ボリュームに関連付けられたフォルダを特定し、ここに移動するにはどうすればいいでしょうか。また、何らかのコンテンツを設定した sample.txt ファイルを作成するにはどうすればいいでしょうか。
4. 別の alpine コンテナを実行して、my-products ボリュームをこのコンテナの /app-data フォルダに読み取り / 書き込みモードでマウントするにはどうすればいいでしょうか。このコンテナ内の /app-data フォルダに移動して、何らかのコンテンツを設定した hello.txt ファイルを作成してください。
5. たとえば ~/my-project というホスト ボリュームをコンテナにマウントするにはどうすればいいでしょうか。
6. 未使用のすべてのボリュームをシステムから削除するにはどうすればいいでしょうか。
7. システム上で実行されている Linux カーネルおよび Docker の正確なバージョンを確認するにはどうすればいいでしょうか。

参考情報

詳細情報については、以下の記事を参照してください。

- ボリュームの使用 (http://dockr.ly/2EUjTml)
- Docker でのデータ管理 (http://dockr.ly/2EhBpzD)
- PWD での Docker ボリューム (http://bit.ly/2sjIfDj)
- コンテナ ― 自宅のクリーンアップ (http://bit.ly/2bVrCBn)
- Docker システム イベント (http://dockr.ly/2BlZmXY)

6
分散アプリケーション アーキテクチャ

前の章では、Docker ボリュームを使用して、作成または変更された状態を保持する方法、およびコンテナ内で実行されているアプリケーション間でデータを共有する方法を学習しました。また、Docker デーモンによって生成されたイベントを処理し、未使用のリソースをクリーンアップする方法も学習しました。

この章では、分散アプリケーション アーキテクチャの概念を紹介し、分散アプリケーションを正常に実行するために必要なさまざまなパターンとベスト プラクティスについて説明します。最後に、このようなアプリケーションを運用環境で実行するために必要な追加要件について説明します。

この章では以下のトピックをとりあげます。

- 分散アプリケーション アーキテクチャとは
- パターンとベスト プラクティス
- 運用環境での実行

この章を終了すると、次のことができるようになります。

- 分散アプリケーション アーキテクチャの特性を 4 つ以上挙げる
- 運用環境対応の分散アプリケーション用に実装する必要があるパターンを 4 つ以上挙げる

分散アプリケーション アーキテクチャとは

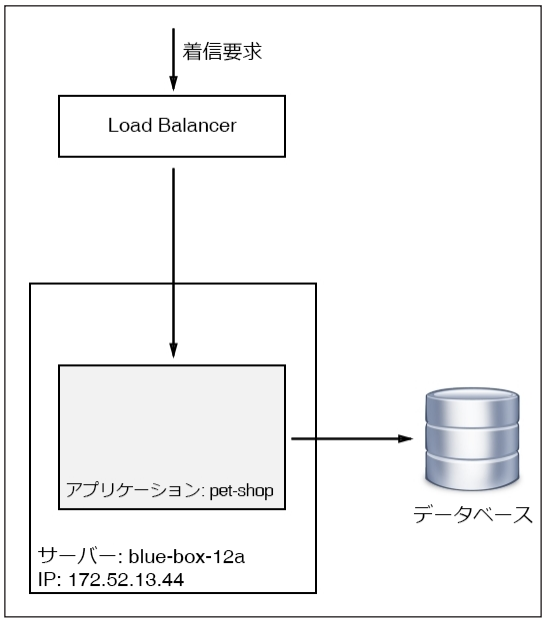
このセクションでは、分散アプリケーション アーキテクチャが何を意味するかについて詳しく説明します。まず、使用するすべての単語や頭字語に意味があることと、その意味がユーザー間で共通していることを確認する必要があります。

用語の定義

この章および後続の章では、すべてのユーザーに知られているとは限らない概念について詳しく説明します。用語の意味を明確に定義するために、これらの概念や言葉の最も重要な側面を簡単に見ていきましょう。

VM	仮想マシンの頭字語。仮想コンピューターです。
ノード	アプリケーションを実行するために使用される個々のサーバー。物理サーバー (多くの場合にベア メタルと呼ばれる) または VM となります。ノードは、メインフレーム、スーパーコンピューター、標準ビジネス サーバー、または Raspberry Pi にすることができます。また、会社のデータ センター内またはクラウド内のコンピューターにすることができます。通常、ノードはクラスターの一部です。
クラスター	分散アプリケーションを実行するために使用されるネットワークによって接続されたノードのグループ。
ネットワーク	クラスターの個々のノードとそれらのノード上で動作するプログラムとの間で、物理的およびソフトウェア的に定義された通信パス。
ポート	Web サーバーなどのアプリケーションが着信要求をリッスンするチャネル。
サービス	きわめて多様な場面で使用される用語であり、その実際の意味は、使用されるコンテキストに依存します。サービスという用語をアプリケーション サービスなどのアプリケーションのコンテキストで使用する場合は、通常、アプリケーションの他の部分で使用される限られた機能セットを実装するソフトウェアであることを意味します。本書の後半のセクションで、わずかに定義の異なるその他のタイプのサービスについて説明します。

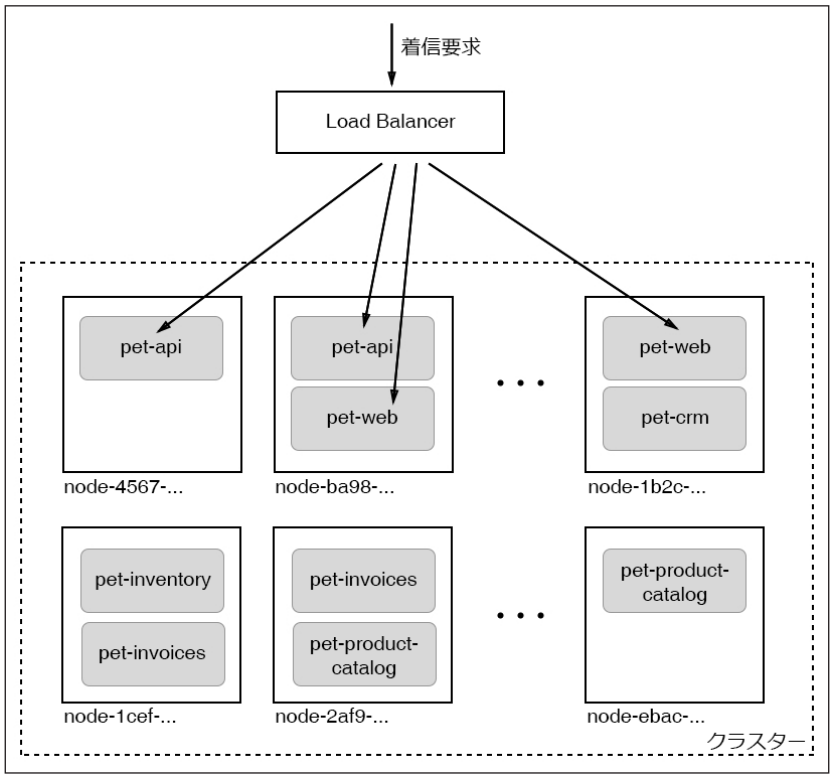
単純に言うと、分散アプリケーション アーキテクチャはモノリシック アプリケーション アーキテクチャの対極にあるものですが、最初にこのモノリシック アーキテクチャを見ていくことは妥当です。従来、ほとんどのビジネス アプリケーションは、データ センター内の名前付きのサーバーで実行される密結合された単一のプログラムとして、結果を見なすことができるように記述されています。そのすべてのコードは、アプリケーションの実行時に共同配置する必要のある、単一のバイナリまたは強力に密結合されたいくつかのバイナリにコンパイルされます。このコンテキストでは、アプリケーションが実行されているサーバー (より一般的にはホスト) が、明確な名前または静的 IP アドレスを持っていることも重要となります。このタイプのアプリケーション アーキテクチャをより明確に示すために、次の図を見てみましょう。



モノリシック アプリケーション アーキテクチャ

前の図には、**pet-shop** という名前のアプリケーションを実行する **blue-box-12a** という名前のサーバー (IP アドレス `172.52.13.44`) が示されています。このアプリケーションは、メイン モジュールおよび密結合されたいくつかのライブラリから成るモノリスです。

では、次の図を見てみましょう。



分散アプリケーション アーキテクチャ

ここからは、名前付きのサーバーを 1 つだけ使用するのではなく、こうした名前付きのサーバーを多数使用するとします。これらのサーバーは、人にとってわかりやすい名前ではなく、**ユニバーサル ユニーク識別子** (**UUID**) のような固有の ID を持っています。また、ここからは、**pet-shop** アプリケーションは単一のモノリシック ブロックで構成されているのではなく、相互作用するが疎結合された多数のサービス (**pet-api**、**pet-web**、**pet-inventory** など) で構成されています。さらに、各サービスは、サーバーまたはホストのこのクラスター内の複数のインスタンスで実行されます。

Docker コンテナに関する本で、なぜこのようなことを議論しているのか疑問に思われるかもしれません。ここで討論するすべての話題は、コンテナがまだ存在しない環境にも同等に適用できますが、コンテナおよびコンテナ オーケストレーション エンジンを活用すれば、すべての問題にはるかに効率的かつ直接的に対処することができます。分散アプリケーション アーキテクチャで解決するのが非常に困難だった問題のほとんどが、コンテナ化された環境ではごく単純なものとなります。

パターンとベスト プラクティス

分散アプリケーション アーキテクチャには多くの魅力的な利点がありますが、モノリシック アプリケーション アーキテクチャよりもはるかに複雑であるという、非常に重大な欠点もあります。この複雑さを解消するために、業界ではいくつかの重要なベスト プラクティスとパターンが考案されました。以下のセクションでは、いくつかの最も重要なベスト プラクティスとパターンを詳しく見ていきます。

疎結合コンポーネント

どのような場合にも、複雑な課題に対処する最良の方法は、より管理しやすい小さな問題に分割することです。一例として、1 ステップで家を建てようとすれば、途方もなく複雑な工程となります。単純な部品を組み合わせて最終的な成果物を完成させる方が、はるかに簡単に家を建てることができます。

ソフトウェア開発にも同じことが当てはまります。アプリケーション全体を構成する、相互運用する小さなコンポーネントへと分割すれば、非常に複雑なアプリケーションをはるかに簡単に開発できるようになります。特に、コンポーネントが単に疎結合している場合、これらのコンポーネントを個別に開発することはきわめて簡単です。これは、コンポーネント A が、たとえばコンポーネント B とコンポーネント C の内部動作については何の予測も行わないが、明確に定義されたインターフェイスを介してそれら 2 つのコンポーネントとどのように通信できるかにのみ関心があるということです。各コンポーネントに、システム内および外部環境の他のコンポーネントとの通信が行われる、明確に定義されたシンプルなパブリック インターフェイスがある場合は、これにより、他のコンポーネントに対する暗黙の依存関係なしに、各コンポーネントを個別に開発することができます。開発プロセスでは、システム内の他のコンポーネントをスタブやモックに置き換えて、コンポーネントをテストすることができます。

ステートレスまたはステートフル

意味のあるすべてのビジネス アプリケーションは、データを作成、変更、または使用します。データはステートとも呼ばれます。永続データを作成または変更するアプリケーション サービスは、ステートフル コンポーネントと呼ばれます。一般的なステートフル コンポーネントは、データベース サービスまたはファイルを作成するサービスです。一方、永続データを作成または変更しないアプリケーション コンポーネントは、ステートレス コンポーネントと呼ばれます。

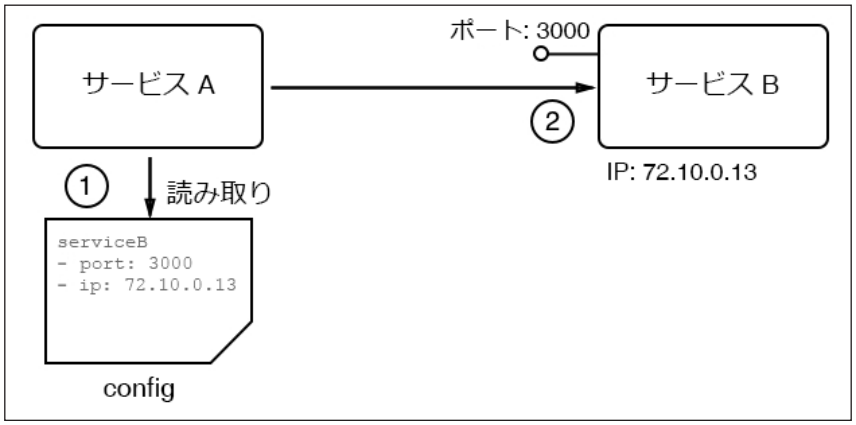
分散アプリケーション アーキテクチャでは、ステートレス コンポーネントはステートフル コンポーネントよりも処理がはるかに簡単です。ステートレス コンポーネントは、簡単に拡大縮小することができます。また、クラスターの完全に異なるノード上で、迅速かつスムーズに断片化し、再起動することもできます。これは、永続データが関連付けられていないためです。

その事実を考慮すると、ほとんどのアプリケーション サービスがステートレスとなるようにシステムを設計すると便利です。すべてのステートフル コンポーネントをアプリケーションの境界にプッシュし、その数を制限することをお勧めします。ステートフル コンポーネントの管理は困難です。

サービス検出

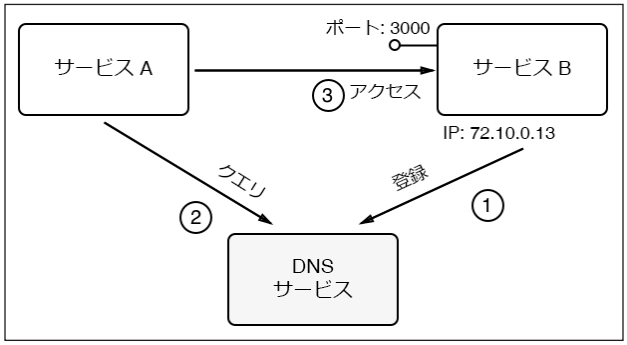
相互に通信する多数の個別のコンポーネントまたはサービスで構成されたアプリケーションをビルドする際には、個別のコンポーネントがクラスター内で互いに認識し合えるようにするメカニズムが必要です。互いに認識し合うということは、通常、ターゲット コンポーネントがどのノード上で実行されているか、およびどのポートで通信をリッスンしているかを知る必要があることを意味します。ほとんどの場合、ノードは IP アドレスとポートで識別されます。これは、明確に定義された範囲内の単なる数字です。

技術的には、ターゲットであるサービス B と通信するサービス A に対して、ターゲットの IP アドレスとポートを知らせることができます。この動作は、たとえば、構成ファイルの項目を介して発生する可能性があります。



固定配線されたコンポーネント

これは、1 つまたは少数の選別された既知のサーバーで動作するモノリシック アプリケーションのコンテキストでは、うまく機能する場合がありますが、分散アプリケーション アーキテクチャでは完全に崩壊します。まず、このシナリオでは多くのコンポーネントが存在し、それらを手動で追跡するのはきわめて困難です。間違いなくスケーラビリティはありません。さらに、サービス A は、通常、クラスターのどのノード上で他のコンポーネントが実行されるかを認識することはありません。アプリケーションの外部のさまざまな理由により、コンポーネント B がノード X から別のノード Y に移動することがあるので、コンポーネントの位置が不安定となる可能性もあります。したがって、サービス A がサービス B を見つけるための別の手段、またはこの問題に対応するその他のサービスが必要です。最も一般的に使用されるのは、いつでもシステムのトポロジーを認識している外部機関です。この外部機関 (サービス) は、クラスターに現在関係しているすべてのノードおよびその IP アドレスを認識しています。つまり、実行中のすべてのサービスおよびそれらの実行場所を認識しています。多くの場合、この種のサービスは **DNS サービス**と呼ばれます。**DNS** は **Domain Name System** を表します。Docker には、基本的なエンジンの一部として DNS サービスが実装されています。また、Kubernetes でも DNS サービスを使用して、クラスター内で実行されているコンポーネント間の通信を容易にします。



外部ロケータ サービスに問い合わせるコンポーネント

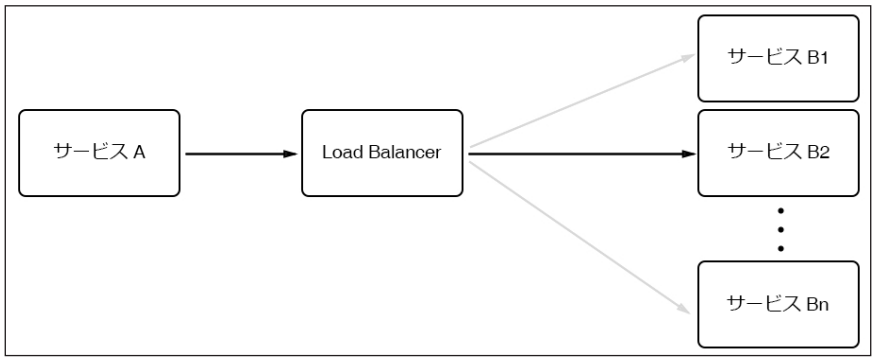
上の図から、サービス A がサービス B とどのように通信するかがわかります。ただし、直接通信することはできません。最初にサービス B の位置に関して、外部機関であるレジストリ サービス (ここでは **DNS サービス**と呼ばれる) に問い合わせる必要があります。レジストリ サービスは、要求された情報で応答し、サービス A がサービス B に到達するために使用する IP アドレスとポート番号を渡します。その後、サービス A はこの情報を使用して、サービス B との通信を確立します。この図は、低レベルで実際に行われている処理を単純化したものですが、サービス検索のアーキテクチャ パターンを理解するには最適です。

ルーティング

ルーティングは、ソース コンポーネントからターゲット コンポーネントにデータのパケットを送信するメカニズムです。ルーティングはさまざまなタイプに分類されます。さまざまなタイプのルーティングを区別するために、いわゆる OSI モデルを使用します (この章の「参考情報」セクションを参照)。コンテナおよびコンテナ オーケストレーションのコンテキストでは、レイヤー 2、3、4、および 7 でのルーティングが関係します。ルーティングの詳細については、以降の章で詳しく説明します。ここでは、レイヤー 2 ルーティングが、MAC アドレスを MAC アドレスに接続する最低レベル タイプのルーティングであり、アプリケーション レベル ルーティングとも呼ばれるレイヤー 7 ルーティングが、最高レベルのルーティングであるとします。後者は、たとえば、URL (`example.com/pets` など) であるターゲット識別子を持つ要求を、システム内の適切なターゲット コンポーネントにルーティングするために使用されます。

負荷分散

次の図に示すように、サービス A がサービス B からのサービスを要求するが、サービス B が複数のインスタンスで実行されている場合は、常に負荷分散が使用されます。



サービス A の要求をサービス B に負荷分散する

システムで、サービス B などのサービスの複数のインスタンスが実行されている場合は、これらのすべてのインスタンスに同量の負荷が割り当てられるようにします。これは一般的なタスクです。つまり、呼び出し元が負荷分散を行う代わりに、外部サービスが呼び出しをインターセプトし、どのターゲット サービス インスタンスに呼び出しを転送するかを決定する役割を引き継ぐようにします。この外部サービスは、**ロード バランサー**と呼ばれます。ロード バランサーは、さまざまなアルゴリズムを使用して、着信呼び出しをターゲット サービス インスタンスにどのように配布するかを決定できます。最も一般的に使用されるアルゴリズムは、ラウンドロビンと呼ばれます。このアルゴリズムは、インスタンス 1、インスタンス 2 の順にインスタンス n まで、単に反復的に要求を割り当てます。最後のインスタンスが処理された後、ロード バランサーはもう一度インスタンス番号 1 から開始します。

防御プログラミング

分散アプリケーションのサービスの開発中は、このサービスがスタンドアロンではなく、他のアプリケーション サービスや、サード パーティによって提供される外部サービス (たとえば、クレジットカード検証サービスや株式情報サービス) に依存することを念頭に置く必要があります。これらのサービスはすべて、開発中のサービスの外部にあります。どの時点においても、それらの正確性や可用性を管理することはできません。したがって、コーディング時には、最悪の事態を想定しながら最善策を講じる必要があります。「最悪の事態を想定する」とは、潜在的な障害を明示的に処理しなければならないことを意味します。

再試行

外部サービスが一時的に利用できくなるか、または応答が十分でない可能性がある場合は、次の手順を使用できます。他のサービスの呼び出しが失敗したりタイムアウトする場合は、短い待機時間の後に同じ呼び出しが繰り返されるように、呼び出しコードを構造化する必要があります。呼び出しが再度失敗した場合は、次の試行までの待機時間が少し長くなります。呼び出しは最大回数まで繰り返され、そのたびに待機時間が長くなります。その後、サービスは試行を断念して、劣化したサービスを提供します。つまり、状況によっては、キャッシュされた古いデータを返したり、データをまったく返さない可能性があります。

ログ

サービス内の重要な操作は、常にログに記録する必要があります。ログ情報を実際に価値のあるものにするには、カテゴリ化する必要があります。一般的なカテゴリのリストは、デバッグ、情報、警告、エラー、および致命的です。ログ情報は、中央ログ集計サービスによって収集され、クラスターの個々のノードには格納されません。集計されたログは、関連情報を得るために容易に解析およびフィルタリングできます。

エラー処理

前述のように、分散アプリケーションの各アプリケーション サービスは他のサービスに依存しています。開発者は、常に最悪の事態を想定し、適切な位置に適切なエラー処理を設置する必要があります。最も重要なベスト プラクティスの 1 つは、すばやく失敗することです。回復不能なエラーができるだけ早期に検出されるようにサービスをコーディングし、そのようなエラーが検出された場合は、すぐにサービスが失敗するようにしてください。ここで、STDERR または STDOUT に有意義な情報を記録することを忘れないでください。この情報は、開発者またはシステム オペレーターが後でシステムの不具合をトレースするために使用できます。また、呼び出しが失敗した理由を可能な限り正確に示す、わかりやすいエラーを呼び出し元に返すようにします。

" すばやく失敗する " 方法の一例は、呼び出し元によって指定された入力値を常にチェックすることです。値は想定範囲内で、完全なものでしょうか。そうでない場合は、処理を続行せずに、ただちに操作を中止します。

冗長性

ミッション クリティカルなシステムは、年中無休で、常時利用可能でなければなりません。ダウンタイムが発生すると、会社の機会や評判が大きく低下する可能性があるため、容認できません。高度に分散されたアプリケーションでは、関与する多数のコンポーネントのうち少なくとも 1 つに障害が発生する可能性を無視できません。問題は、コンポーネントに障害が発生するかどうかではなく、障害がいつ発生するかということです。

システム内の多数のコンポーネントの 1 つに障害が発生した場合にダウンタイムを回避するには、システムの個々の部分を冗長化する必要があります。これには、アプリケーション コンポーネントおよびすべてのインフラ部分が含まれます。つまり、たとえばアプリケーションに支払いサービスが含まれる場合、このサービスを冗長実行する必要があります。その最も簡単な方法としては、クラスターの別々のノードでこのサービスの複数のインスタンスを実行します。エッジ ルーターやロード バランサーについても同じことが当てはまります。これらがダウンすることは許されません。したがって、ルーターやロード バランサーは冗長である必要があります。

ヘルス チェック

多数の部分から成る分散アプリケーション アーキテクチャでは、個々のコンポーネントの障害が発生する可能性は非常に高く、それは時間の問題です。そのため、システムのすべての個々のコンポーネントを冗長実行します。プロキシ サービスは、サービスの個々のインスタンス間でトラフィックを負荷分散します。

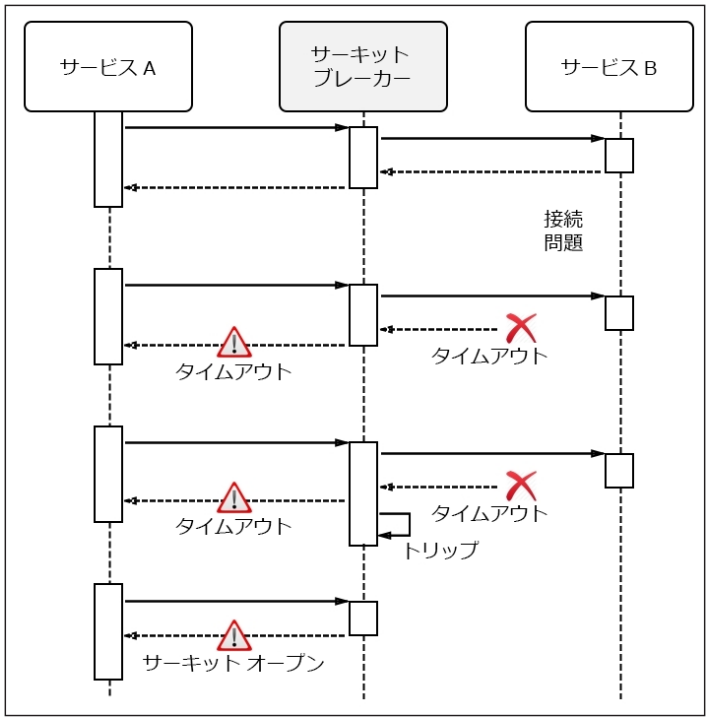
ここで、もう 1 つの問題があります。プロキシやルーターは、特定のサービス インスタンスが利用可能かどうかをどのようにして把握するのでしょうか。サービス インスタンスはクラッシュしているか、応答不能の可能性があります。この問題を解決するには、いわゆるヘルス チェックを使用します。プロキシ、またはプロキシに代わる他のシステム サービスは、定期的にすべてのサービス インスタンスをポーリングし、その正常性をチェックします。基本的な質問は、" まだ存在していますか？正常な状態ですか？ " です。各サービスの回答は " はい " または " いいえ " で、インスタンスがそれ以上応答しない場合、ヘルスチェックはタイムアウトになります。

コンポーネントの回答が " いいえ " であるか、タイムアウトが発生した場合、システムは対応するインスタンスを強制終了し、その場所で新しいインスタンスをスピンアップします。これらすべてが完全に自動化された方法で行われる場合は、自動回復システムを構築していると言えます。

Circuit Breaker パターン

サーキット ブレーカーは、多数の必須コンポーネントの連鎖的な障害のために分散アプリケーションがダウンするのを防ぐために使用するメカニズムです。サーキット ブレーカーは、1 つのコンポーネントに障害が発生したためにドミノ効果によって他の従属サービスが終了することを回避するのに役立ちます。電力線を遮断することで接続機器の障害による火災から家屋を守る、電気系統のサーキット ブレーカーと同様に、分散アプリケーションのサーキット ブレーカーは、サービス B が応答していないか不具合がある場合に、サービス A からサービス B への接続を切断します。

これは、保護されたサービス呼び出しをサーキット ブレーカー オブジェクトにラップすることで実現できます。このオブジェクトは失敗を監視します。障害の数が一定のしきい値に達すると、サーキット ブレーカーが切れます。その後のサーキット ブレーカーのすべての呼び出しは、保護された呼び出しがまったく行われずにエラーを返します。



Circuit Breaker パターン

運用環境での実行

運用環境で分散アプリケーションを正常に実行するには、前のセクションで紹介したベスト プラクティスとパターン以外に、さらにいくつかの点を考慮する必要があります。その重要な領域の 1 つは、イントロスペクションと監視です。これらの最も重要な側面について詳しく見ていきましょう。

ログ

分散アプリケーションが運用環境にある場合、デバッグすることはできません。では、ユーザーによって報告されたアプリケーションの不具合の根本的な原因を特定するには、どうすればよいでしょうか。この問題に対する解決策は、豊富で有意義なログ情報を生成することです。開発者は、エラーが発生した場合や予期しないまたは望ましくない状況に遭遇した場合などに、有用な情報を出力するよう、アプリケーション サービスをインストルメント化する必要があります。多くの場合、この情報は STDOUT と STDERR に出力されます。そこからシステム デーモンによって情報が収集され、ローカル ファイルに書き込まれたり、中央ログ集計サービスに転送されたりします。

ログに十分な情報がある場合、開発者はこれらのログを使用して、報告されたシステム内のエラーの根本原因を追跡することができます。

多数のコンポーネントから成る分散アプリケーション アーキテクチャでは、モノリシック アプリケーションの場合よりもログ記録が重要となります。アプリケーションのすべてのコンポーネントにわたって単一の要求を実行する場合のパスは、非常に複雑になる可能性があります。また、コンポーネントがノードのクラスター全体に分散されていることに注意してください。したがって、重要なすべての項目をログに記録し、たとえば正確な発生時刻、発生元のコンポーネント、コンポーネントが実行されたノードなどの項目を各ログ エントリに追加することは理にかなっています。さらに、ログ情報は、開発者やシステム オペレーターが分析に容易に利用できるように、中央の場所で集計する必要があります。

トレース

トレースは、個々の要求が分散アプリケーションをどのように通過するか、および要求と個々のコンポーネント全体でどれくらいの時間が費やされるかを調べるために使用されます。この情報を収集すると、システムの動作と正常性を示すダッシュボードのソースの 1 つとして使用できます。

監視

オペレーターは、アプリケーションの全体的な正常性を一目で把握できる、システムのライブ キー メトリックを表示するダッシュボードを求めています。これらのメトリックは、メモリや CPU 使用率、システムまたはアプリケーション コンポーネントのクラッシュ数、ノードの正常性などの非機能メトリックである場合と、発注システム内のチェックアウト数やインベントリ サービス内の在庫切れ品目数などの機能メトリック (すなわち、アプリケーション固有のメトリック) である場合があります。

ほとんどの場合、ダッシュボードに使用される数値を集計するために使用される基本データは、ログ情報から抽出されます。これは、主に非機能メトリックで使用されるシステム ログ、または機能メトリックで使用されるアプリケーション レベル ログのどちらかになります。

アプリケーションの更新

企業の競争優位性の 1 つとして、変化する市場状況にタイムリーに対応できることが挙げられます。これには、新しいニーズや変更されたニーズを満たすようにアプリケーションをすばやく調整したり、新しい機能を追加したりできることが含まれます。アプリケーションは更新がすばやいほど、より良いものになります。最近では、多くの企業が新しい機能や変更された機能を 1 日に複数回公開しています。

アプリケーション更新は頻繁に行われるため、こうした更新は中断を伴わないものである必要があります。アップグレード時にメンテナンスのためにシステムを停止させるわけにはいきません。アップグレードはすべてシームレスかつ透過的に行わなければなりません。

ローリング更新

アプリケーションまたはアプリケーション サービスを更新する方法の 1 つとして、ローリング更新の使用が挙げられます。この方法では、更新する必要のある特定のソフトウェアが複数のインスタンスで実行されていることが前提となります。その場合にのみ、このタイプの更新を使用できます。

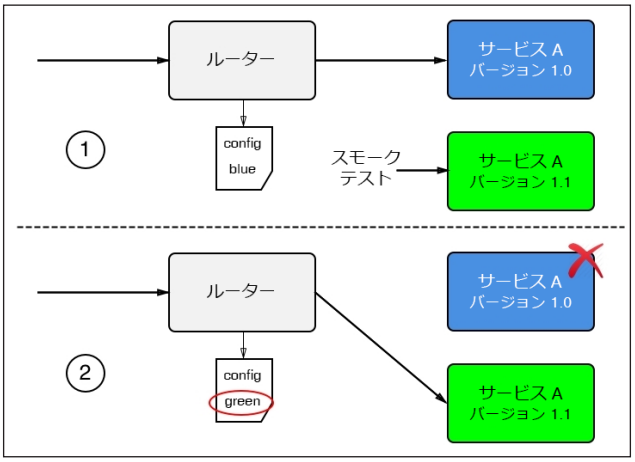
この方法では、システムは現在のサービスの 1 つのインスタンスを停止し、新しいサービスのインスタンスに置き換えます。新しいインスタンスが準備されるとすぐに、トラフィックが処理されます。通常、新しいインスタンスは、期待どおりに動作するかどうかを確認するためにしばらく監視されます。期待どおりに動作する場合は、現在のサービスの次のインスタンスが停止され、新しいインスタンスに置き換えられます。すべてのサービス インスタンスが置き換えられるまで、このパターンが繰り返されます。

どの時点でもいくつかのインスタンス (現在のインスタンスまたは新しいインスタンス) が常に実行中であるため、アプリケーションは常に動作しています。ダウンタイムは必要ありません。

ブルーグリーン デプロイメント

ブルーグリーン デプロイメントでは、アプリケーション サービスの現在のバージョン (" ブルー " と呼ばれる) がすべてのアプリケーション トラフィックを処理します。その後、運用システムに新しいバージョンのアプリケーション サービス (" グリーン " と呼ばれる) をインストールします。この新しいサービスは、アプリケーションの残りの部分にまだ接続されていません。

" グリーン " がインストールされると、この新しいサービスに対してスモーク テストを実行できます。テストが成功した場合は、以前に " ブルー " に流れていたすべてのトラフィックが新しいサービスである " グリーン " に流れるように、ルーターを設定できます。その後、" グリーン " の動作が厳密に観察され、すべての達成基準が満たされれば、" ブルー " は廃止することができます。ただし、何らかの理由で " グリーン " が予期しない動作や望ましくない動作を示した場合は、すべてのトラフィックが " ブルー " に戻されるようにルーターを再設定できます。その後、" グリーン " を削除して修正し、修正されたバージョンで新しいブルーグリーン デプロイメントを実行できます。



ブルーグリーン デプロイメント

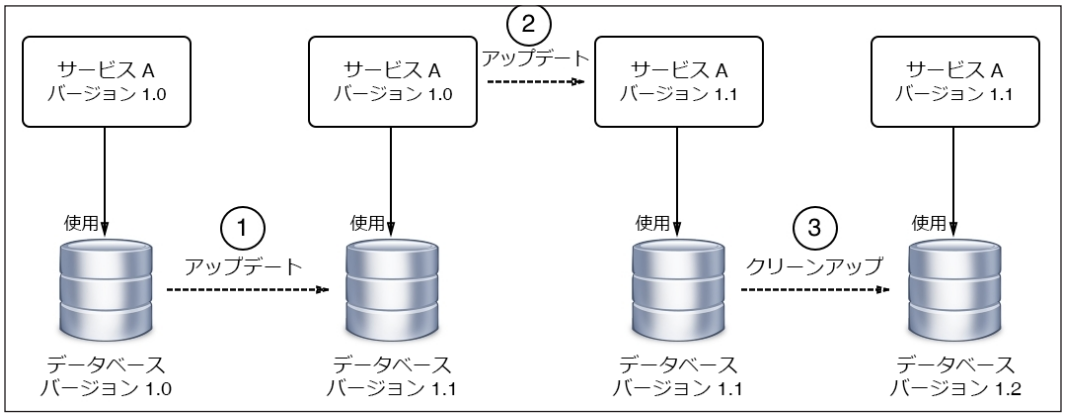
カナリア リリース

カナリア リリースは、現在のバージョンと新しいバージョンのアプリケーション サービスがシステムに並行してインストールされるリリースです。したがって、ブルー グリーン デプロイメントに似ています。最初に、すべてのトラフィックが引き続き現在のバージョンを経由してルーティングされます。次に、トラフィック全体のわずかなパーセンテージ (たとえば 1%) が新しいバージョンのアプリケーション サービスに流れるように、ルーターを設定します。その後、新しいサービスが期待どおりに動作するかどうかを確認するために、その動作が厳密に監視されます。正常な動作のすべての基準が満たされている場合、ルーターは新しいサービスにより、より多くのトラフィック (たとえば 5%) が通過するように設定されます。再び新しいサービスの動作が厳重に監視され、正常に動作している場合は、100% に達するまで、より多くのトラフィックが順次ルーティングされます。すべてのトラフィックが新しいサービスにルーティングされ、しばらくの間安定している場合は、古いバージョンのサービスを廃止できます。

これはなぜカナリア リリースと呼ばれるのでしょう。この名前は、石炭鉱夫が炭鉱の早期警戒システムとしてカナリアを使用していたことにちなみます。カナリアは有毒ガスに特に敏感な鳥で、そのようなカナリアが死んだら、ただちに鉱山を放棄しなければならないことを鉱山者たちは知っていました。

不可逆的なデータ変更

更新プロセスの一環として、補助リレーショナル データベースの不可逆的なスキーマ変更など、状態の不可逆的な変更を実行する場合は、特別な注意を払って対処する必要があります。適切なアプローチを使用すれば、このような変更をダウンタイムなしに実行できます。このような状況では、データの変更と同時にデータ ストア内に新しいデータ構造を必要とするコード変更をデプロイすることはできません。代わりに、更新全体を 3 つの異なるステップに分ける必要があります。最初のステップでは、下位互換性のあるスキーマおよびデータ変更をロールアウトします。このステップが成功したら、2 番目のステップで新しいコードをロール アウトします。このステップも成功したら、3 番目のステップでスキーマをクリーンアップし、下位互換性を削除します。



不可逆的なデータまたはスキーマ変更のロールアウト

ロールバック

運用環境で動作するアプリケーション サービスを頻繁に更新すると、いずれはこうした更新の 1 つで問題が発生します。開発者がバグを修正している間に、新しいバグが発生し、自動化された (あるいは手動の) いずれのテストでも捕捉されなかったために、アプリケーションが誤動作する場合があります。このような場合は、サービスを以前の適切なバージョンにロールバックしなければなりません。つまり、ロールバックとは障害からの回復です。

ここでも、分散アプリケーション アーキテクチャでは、ロールバックが必要かどうかではなく、ロールバックがいつ発生する必要があるかが問題となります。したがって、アプリケーションを構成するすべてのサービスの以前のバージョンに常にロールバックできることを必ず確認しなければなりません。ロールバックは後から考慮することができず、デプロイメント プロセス内の実証済みかつ実績のある部分として組み入れる必要があります。

ブルーグリーン デプロイメントを使用してサービスを更新している場合、ロールバックはかなりシンプルなものになります。新しい " グリーン " バージョンのサービスから以前の " ブルー " バージョンのサービスにルーターを切り替えるだけで済みます。

まとめ

この章では、分散アプリケーション アーキテクチャの概要、および分散アプリケーションを正常に実行するために役立つまたは必要となるパターンとベスト プラクティスについて学習しました。最後に、このようなアプリケーションを運用環境で実行するために必要な追加事項についても学習しました。

次の章では、単一のホストに制限されたネットワーキングについて詳しく説明します。同じホスト上にあるコンテナが互いに通信する方法と、必要に応じて外部クライアントがコンテナ化されたアプリケーションにアクセスする方法について詳しく説明します。

質問

この章の内容に対する理解度を評価するために、以下の質問にお答えください。

1. 分散アプリケーション アーキテクチャのすべての部分をいつ、どのような理由で冗長にする必要がありますか。数行の短い文で説明してください。
2. DNS サービスはなぜ必要なのでしょうか。3 ～ 5 文で説明してください。
3. サーキット ブレーカーとは何ですか。また、なぜ必要なのでしょうか。
4. モノリシック アプリケーションと分散アプリケーション (マルチサービス アプリケーション) との重要な違いは何ですか。
5. ブルーグリーン デプロイメントとは何ですか。

参考情報

以下の記事では、より詳細な情報を提供しています。

- サーキット ブレーカー (http://bit.ly/1NU1sgW)
- OSI モデルの説明 (http://bit.ly/1UCcvMt)
- ブルーグリーン デプロイメント (http://bit.ly/2r2IxNJ)

7
シングルホスト ネットワーキング

前章では、分散アプリケーション アーキテクチャを扱うときに使用される最も重要なアーキテクチャ パターンとベスト プラクティスについて学びました。

この章では、Docker コンテナ ネットワーキング モデルと、そのシングルホスト実装をブリッジ ネットワークの形で紹介します。また、ソフトウェア定義ネットワークの概念と、それらがコンテナ化したアプリケーションを保護するためにどのように使用されるかについても説明します。最後に、コンテナ ポートを公開してコンテナ化したコンポーネントを外部にアクセスできるようにする方法を示します。

この章では以下のトピックをとりあげます。

- コンテナ ネットワーク モデル
- ネットワーク ファイアウォール
- ブリッジ ネットワーク
- ホスト ネットワーク
- Null ネットワーク
- 既存のネットワーク名前空間での実行
- ポート管理

このモジュールを終了すると、次のことができるようになります。

- コンテナ ネットワーキング モデルのドラフトを、すべての必須コンポーネントと共にホワイトボードに作成する
- カスタム ブリッジ ネットワークを作成・削除する
- カスタム ブリッジ ネットワークに接続されたコンテナを実行する
- ブリッジ ネットワークを検査する

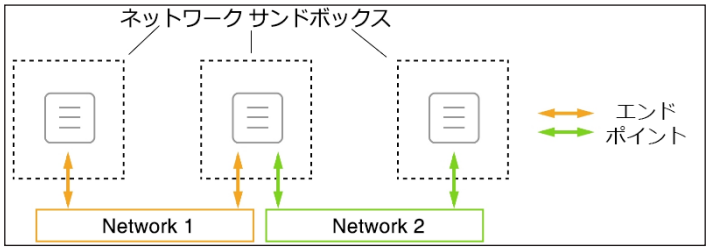
- 異なるブリッジ ネットワーク上でコンテナを実行することによって、コンテナを互いに分離する
- 選択したホスト ポートにコンテナ ポートを公開する

技術的要件

この章では、必要なのは Linux コンテナを実行できる Docker ホストのみです。Docker for Mac、Docker for Windows、Docker Toolbox のいずれかをインストールしたラップトップを使用できます。

コンテナ ネットワーク モデル

ここまでは、単一のコンテナで作業してきました。しかし実際には、コンテナ化されたビジネス アプリケーションはいくつかのコンテナで構成されていて、目標を達成するためにこれらが協力する必要があります。そのため、個々のコンテナが互いに通信する方法が必要です。これは、コンテナ間でのデータ パケットのやりとりに使用できる経路を確立することで実現されます。これらの経路は、**ネットワーク**と呼ばれます。Docker は非常に単純なネットワーキング モデル、いわゆる**コンテナ ネットワーク モデル** (**CNM**) を定義して、コンテナ ネットワークを実装するソフトウェアが満たすべき要件を指定します。以下の図は CNM をグラフィックで示しています。



Docker コンテナ ネットワーク モデル

CNM には 3 つの要素があります。サンドボックス、エンドポイント、ネットワークです。

- **サンドボックス**: サンドボックスはコンテナを外界から完全に分離します。サンドボックス化されたコンテナには、インバウンド ネットワーク接続は許可されません。しかし、システムとの通信がまったく不可能な場合、コンテナがシステム内で何らかの価値を持つ可能性はほとんどありません。この問題を回避するために、2 番目の要素、エンドポイントがあります。

- **エンドポイント**: エンドポイントは、外部から、コンテナを保護するネットワークのサンドボックスへの制御されたゲートウェイです。エンドポイントは、ネットワーク サンドボックス (コンテナではなく) をモデルの 3 番目の要素、ネットワークに接続します。
- **ネットワーク**: ネットワークは、エンドポイントからエンドポイントへの通信、または最終的にコンテナからコンテナへの通信のインスタンスのデータパケットを転送する経路です。

ネットワーク サンドボックスは、ゼロから多数のエンドポイントを持つことがあります。言い換えれば、ネットワーク サンドボックス内の各コンテナが、ネットワークにまったく接続できないこともあれば、同時に複数の異なるネットワークに接続することもあります。上の図では、3 つのネットワーク サンドボックスのうちの中央が、それぞれのエンドポイントを介して Network 1 と Network 2 の両方に接続されています。

このネットワーキング モデルは非常に一般的なもので、互いに通信する個々のコンテナがネットワーク上のどこで実行されるのかを指定しません。たとえば、すべてのコンテナは同じホスト (ローカル) 上で実行することも、ホストのクラスタ全体 (グローバル) に分散することもできます。

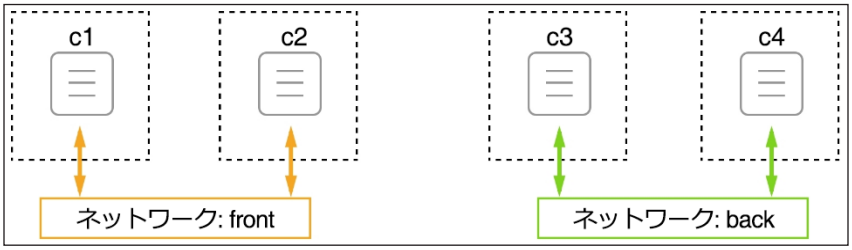
もちろん、CNM は、ネットワーキングがコンテナ間でどのように動作するかを記述するモデルに過ぎません。コンテナでネットワーキングを使用できるようにするには、CNM の実際の実装が必要です。ローカル スコープとグローバル スコープの両方において、CNM の実装は複数あります。次の表に、既存の実装とその主な特性の概要を示します。リストは順不同です。

ネットワーク	会社名	スコープ	説明
ブリッジ	Docker	ローカル	単一のホスト上でネットワーキングを可能にする Linux ブリッジに基づくシンプルなネットワーク
Macvlan	Docker	ローカル	単一の物理ホスト インターフェイス上に複数のレイヤー 2 (つまり MAC) アドレスを設定する
Overlay	Docker	グローバル	**Virtual Extensible LAN** (**VXLan**) に基づくマルチノード対応コンテナ ネットワーク
Weave Net	Weaveworks	グローバル	シンプルで弾力性を備えたマルチホスト Docker ネットワーキング
Contiv Network Plugin	Cisco	グローバル	オープン ソース コンテナ ネットワーキング

Docker によって直接提供されていないネットワーク タイプはすべて、プラグインとして Docker ホストに追加できます。

ネットワーク ファイアウォール

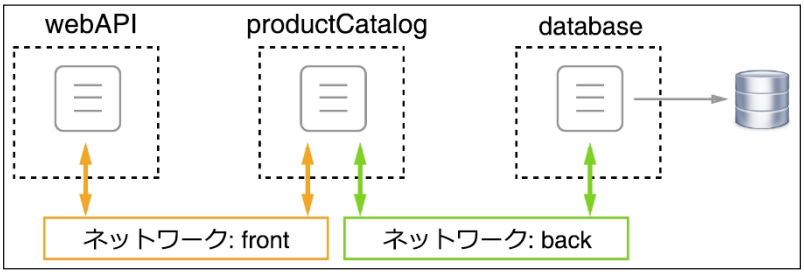
Docker は常に、セキュリティ第一を信念としてきました。この信条は、シングルおよびマルチホストの Docker 環境におけるネットワークの設計・実装方法に直接影響を及ぼしました。ソフトウェア定義のネットワークは簡単かつ安価に作成できるにもかかわらず、このネットワークに接続されているコンテナを、他の接続されていないコンテナや外部からファイアウォールで完全に保護します。同じネットワークに属するすべてのコンテナは互いに自由に通信できる一方、他のコンテナには通信手段がありません。



Docker ネットワーク

上の図には、**front** と **back** という名前の 2 つのネットワークがあります。front ネットワークにはコンテナ **c1** と **c2** が接続されており、back ネットワークにはコンテナ **c3** と **c4** が接続されています。**c1** と **c2** は、**c3** と **c4** と同じように、互いに自由に通信できます。しかし **c1** と **c2** には、**c3** または **c4** と通信する手段がなく、同じように c3 と c4 にも c1 または c2 と通信する手段がありません。

それでは、あるアプリケーションが **webAPI**、**productCatalog**、**database** という 3 つのサービスで構成されている状況はどうでしょうか。**webAPI** は **productCatalog** と通信できるものの database とは通信できないようにし、**productCatalog** は database サービスと通信できるようにします。この状況を解決するには、次の図に示すように、**webAPI** と database を異なるネットワーク上に配置し、**productCatalog** をこの両方のネットワークに接続します。



複数のネットワークに接続されたコンテナ

SDN の作成は安価であり、各ネットワークは不正アクセスからリソースを分離することでセキュリティを強化しているため、複数のネットワークを使用し、互いとの通信が必要不可欠なサービスのみを同じネットワーク上で実行するようにアプリケーションを設計・実行することを強くお勧めします。前述の例では、Web API コンポーネントはデータベース サービスと直接通信する必要がまったくないため、異なるネットワークに配置しています。最悪のシナリオが発生し、ハッカーが Web API を侵害したとしても、まず製品カタログ サービスをハッキングしない限り、そこからデータベースにアクセスすることはできません。

ブリッジ ネットワーク

Docker ブリッジ ネットワークは、コンテナ ネットワーク モデルの中で最初に実装するものです。これについては、これから詳細に説明します。このネットワーク実装は、Linux ブリッジに基づいています。Docker デーモンが初めて実行されると、Linux ブリッジが作成されます。これは `docker0` と呼ばれます。これは既定の動作であり、設定を変えることで変更できます。Docker はこの Linux ブリッジでネットワークを作成し、ネットワーク ブリッジを呼び出します。Docker ホストで作成したコンテナと、別のネットワークに明示的にバインドしていないコンテナはすべて、この Docker ネットワークに自動的に接続されます。
タイプが `bridge` で `bridge` と呼ばれるネットワークが間違いなくホストに定義されたことを確認するには、次のコマンドでホスト上のすべてのネットワークを一覧表示します。

```
$ docker network ls
```

これにより、次のような出力が得られます。

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
928c8ce47bf2        bridge              bridge              local
bdb36adcf70c        host                host                local
af82006f2f2d        none                null                local
$
```

既定で利用可能な Docker の全ネットワークのリスト

このケースでは ID は異なりますが、残りの出力は同じようになります。実際に `bridge` という最初のネットワークでは、ドライバー `bridge` を使用します。`local` というスコープは、このタイプのネットワークが単一のホストに限定され、複数のホストにまたがることはできないことを意味しているだけです。後の章では、グローバル スコープを持つ他のタイプのネットワークについても説明します。このタイプのネットワークは、ホストのクラスタ全体にまたがることができます。

それでは、`bridge` ネットワークとは何なのかについて、もっと詳しく見ていきましょう。これを行うために、Docker の `inspect` コマンドを使用します。

```
$ docker network inspect bridge
```

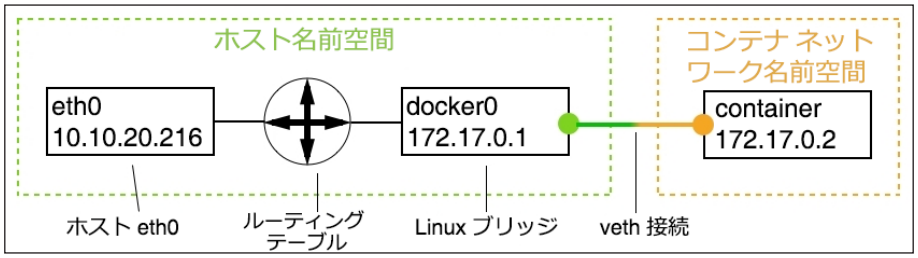
実行すると、該当するネットワークに関する詳細な情報が出力されます。この情報は次のように表示されます。

```
C:\Users\admin>docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "3b08c1c711ada84ae859c4bed48b5af1f45b68db89356ca5045dc7ee8672e946",
        "Created": "2018-04-09T09:47:29.9424652Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

Docker ブリッジ ネットワークを検査したときに生成される出力

すべてのネットワークを一覧表示したときに、既に `ID`、`Name`、`Driver`、`Scope` の各値が表示されていたため、新しいものはありませんが、**IP アドレス管理** (**IPAM**) ブロックを見てみましょう。IPAM は、コンピュータ上で使用される IP アドレスを追跡するために使用されるソフトウェアです。`IPAM` ブロックで重要な部分は、`Subnet` と `Gateway` の値を持つ `Config` ノードです。ブリッジ ネットワークのサブネットは、既定で `172.17.0.0/16` と定義されています。

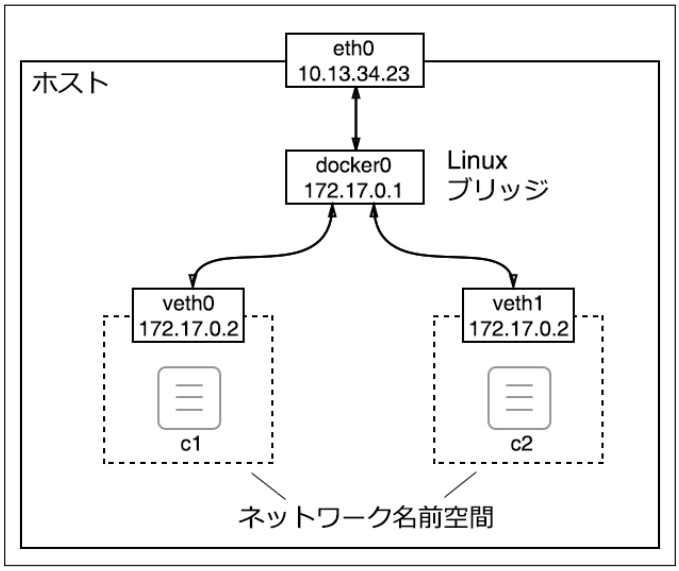
つまり、このネットワークに接続されているすべてのコンテナには、Docker によって IP アドレスが割り当てられます。その値の範囲は、`172.17.0.2` から `172.17.255.255` までになります。`172.17.0.1` アドレスは、このネットワークのルーター用に予約されています。このタイプのネットワークでの役割は、Linux ブリッジによって行われます。Docker によってこのネットワークに接続される最初のコンテナには、`172.17.0.2` アドレスが割り当てられると考えられます。以降のコンテナには、これよりも大きい数字が割り当てられます。次の図はこの事実を示しています。



ブリッジ ネットワーク

上の図には、ホストのネットワーク名前空間が示されており、ホストのエンドポイント `eth0` が含まれています。これは通常、Docker ホストがベアメタルで実行されている場合は NIC で、Docker ホストが VM の場合は仮想 NIC です。ホストへのトラフィックはすべて、`eth0` を通過します。Linux ブリッジは、ホストのネットワークとブリッジ ネットワークのサブネット間のネットワーク トラフィックをルーティングします。

既定では、許可されるのは出力からのトラフィックのみで、入力はすべてブロックされます。つまりコンテナ化されたアプリケーションは、インターネットにはアクセスできますが、外部のトラフィックからはアクセスできません。ネットワークに接続された各コンテナは、ブリッジと独自の**仮想イーサネット** (**veth**) 接続を行います。これを次の図に示します。



ブリッジ ネットワークの詳細

上の図は、ホストの観点からの世界を示しています。この状況がコンテナ内からどのように見えるかについては、このセクションの後半で説明します。

利用できるのは `bridge` ネットワークのみではありません。Docker では独自のカスタム ブリッジ ネットワークを定義することができます。これは単に「あると便利」な機能ではありません。同じネットワーク上ですべてのコンテナを実行するのではなく、追加のブリッジ ネットワークを使用して相互に通信する必要のないコンテナをさらに分離するという、推奨されるベスト プラクティスでもあります。`sample-net` というカスタム ブリッジ ネットワークを作成するには、次のコマンドを使用します。

```
$ docker network create --driver bridge sample-net
```

これを行うと、Docker がこの新しいカスタム ネットワーク用に作成したサブネットを次のように検査することができます。

```
$ docker network inspect sample-net | grep Subnet
```

次の値が返されます。

```
"Subnet": "172.18.0.0/16",
```

見たところ、Docker は IP アドレスの次の空きブロックを新しいカスタム ブリッジ ネットワークに割り当てています。何らかの理由で、ネットワークを作成するときに独自のサブネット範囲を指定する場合は、`--subnet` パラメーターを使用します。

```
$ docker network create --driver bridge --subnet "10.1.0.0/16"
test-net
```

IP アドレスの重複による競合を避けるため、重複しているサブネットでネットワークを作成しないようにしてください。

ブリッジ ネットワークの概要とカスタム ブリッジ ネットワークの構築方法について説明したので、次はこれらのネットワークにコンテナを接続する方法を見ていきます。まず、接続するネットワークを指定せずに Alpine コンテナをインタラクティブに実行しましょう。

```
$ docker container run --name c1 -it --rm alpine:latest /bin/sh
```

別のターミナル ウィンドウで、`c1` コンテナを検査します。

```
$ docker container inspect c1
```

膨大な出力の中で、ネットワーク関連の情報を提供する部分に少し注目しましょう。`NetworkSettings` ノードの下にあります。この内容を次の出力に示しました。


```
},
"NetworkSettings": {
    "Bridge": "",
    "SandboxID": "ae53496fba49de3d0a4727105cc0799b7fbd30746d76700238cb47c611f3eb68",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": {},
    "SandboxKey": "/var/run/docker/netns/ae53496fba49",
    "SecondaryIPAddresses": null,
    "SecondaryIPv6Addresses": null,
    "EndpointID": "c063a725d1f66e867b5769a80d1477cc88d07618860655fa3033a97478e55713",
    "Gateway": "172.17.0.1",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "172.17.0.4",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "MacAddress": "02:42:ac:11:00:04",
    "Networks": {
        "bridge": {
            "IPAMConfig": null,
            "Links": null,
            "Aliases": null,
            "NetworkID": "026e653c2504e464748b4ce9b25cce69d29bc82a52105a25920f2b796663e635",
            "EndpointID": "c063a725d1f66e867b5769a80d1477cc88d07618860655fa3033a97478e55713",
            "Gateway": "172.17.0.1",
            "IPAddress": "172.17.0.4",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "MacAddress": "02:42:ac:11:00:04",
            "DriverOpts": null
        }
    }
}
```


コンテナのメタデータのネットワーク設定セクション

上記の出力で、コンテナが実際に `bridge` ネットワークに接続されていることがわかります。`NetworkID` が `026e65...` と同じで、前述のコードからわかるように、これは `bridge` ネットワークの ID です。コンテナには予想どおり IP アドレス `172.17.0.4` が割り当てられており、ゲートウェイが `172.17.0.1` にあることもわかります。コンテナには `MacAddress` も関連付けられていることに注意してください。Linux ブリッジがルーティングに Mac アドレスを使用するため、これは重要です。

これまでのアプローチは、コンテナのネットワーク名前空間の外側からでした。次は、コンテナ内だけでなく、コンテナのネットワーク名前空間内にあるときの状況を見てみましょう。`c1` コンテナ内で `ip` ツールを使用して、何が起こっているのかを調べてみましょう。次のように `ip addr` コマンドを実行して、生成される出力を観察します。

```
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1
    link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN qlen 1
    link/tunnel6 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00 brd 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
19: eth0@if20: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ #
```

IP ツールから見たコンテナの名前空間

上記の出力で興味深いのは、`eth0` エンドポイントである `19` という数字です。Linux ブリッジがコンテナ名前空間の外で作成した `veth0` エンドポイントは、コンテナ内では `eth0` にマッピングされています。Docker では、常にコンテナのネットワーク名前空間の最初のエンドポイントを `eth0` にマッピングします。これは名前空間の内側から見えるものです。ネットワーク名前空間が追加のネットワークに接続されている場合、そのエンドポイントは `eth1` などにマッピングされます。

この時点では `eth0` 以外のエンドポイントにはあまり関心がないため、より具体的なコマンドのバリアントを使用していた可能性があります。その場合は次のようになります。

```
/ # ip addr show eth0
195: eth0@if196: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500
qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd
    ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope
    global eth0
       valid_lft forever preferred_lft forever.
```

出力では、Docker によってこのコンテナのネットワーク名前空間に関連付けられた MAC アドレス (`02:42:ac:11:00:02`) と IP (`172.17.0.2`) を確認できます。

また、`ip route` コマンドを使用してリクエストをルーティングする方法に関する情報も得られます。

```
/ # ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link src 172.17.0.2
```

この出力から、`172.17.0.1` にあるゲートウェイへのすべてのトラフィックは、`eth0` デバイス経由でルーティングされていることがわかります。

次は、同じネットワーク上で `c2` という別のコンテナを実行しましょう。

```
$ docker container run --name c2 -d alpine:latest ping 127.0.0.1
```

他のネットワークを指定していないため、`c2` コンテナも `bridge` ネットワークに接続されます。その IP アドレスは、サブネットの次の空き IP アドレス `172.17.0.3` になります。次のように簡単にテストできます。

```
$ docker container inspect --format "{{.NetworkSettings.
IPAddress}}" c2
172.17.0.3
```

これで 2 つのコンテナが `bridge` ネットワークに接続されました。このネットワークをもう一度検査して、接続されているすべてのコンテナのリストを出力で確認できます。

```
$ docker network inspect bridge
```

情報は `Containers` ノードの下に表示されます。

```
"ConfigOnly": false,
"Containers": {
    "27b96de70b58cd918d35c235a7c180f56f71df58cf4cec50b8f0103dd529b95f": {
        "Name": "c2",
        "EndpointID": "8883649774c5c4c53063da02598c8d09fe7ee427145b348b1d1703f31213e9ca",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    },
    "35b8dd512acb985647833e1cc52625e129c15e903fd8a0c0ab247932bc910166": {
        "Name": "c1",
        "EndpointID": "28269a9cc630135ab287052fa69c72f28c57a10bd5e7523c451bf2d0976fd1b5",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    }
},
"Options": {
```

docker network inspect bridge の出力の Containers セクション

ここでもまた読みやすさを考慮して、出力を基本要素のみに短縮しました。

ここでは 2 つの追加のコンテナ、c3 と c4 を作成して、test-net に接続しましょう。これには --network パラメーターを使用します。

```
$ docker container run --name c3 -d --network test-net \
    alpine:latest ping 127.0.0.1
$ docker container run --name c4 -d --network test-net \
    alpine:latest ping 127.0.0.1
```

network test-net を検査して、コンテナ c3 と c4 が実際にこのネットワークに接続されていることを確認しましょう。

```
$ docker network inspect test-net
```

このコマンドで次の Containers セクションが出力されます。

```
"Containers": {
    "134295caa6012df5dc7d541436954af1a5264c6f69d5b8012e88f9c12faf40f1": {
        "Name": "c3",
        "EndpointID": "5693cd9329437a9ecec1d27f439887bb0258837b9342a1c32204fa4571298457",
        "MacAddress": "02:42:0a:01:00:02",
        "IPv4Address": "10.1.0.2/16",
        "IPv6Address": ""
    },
    "4a277d33ebfb74f00d31be272d2d74cbfec4b17666e44d88e26cfe83b0a790cc": {
        "Name": "c4",
        "EndpointID": "a1e9ecafebdcf816261883c171434273d9973832d43255b5aa224b081853ed0f",
        "MacAddress": "02:42:0a:01:00:03",
        "IPv4Address": "10.1.0.3/16",
        "IPv6Address": ""
    }
```

docker network inspect test-net コマンドの Containers セクション

次に確認するのは、この 2 つのコンテナ c3 と c4 が互いに自由に通信できるかどうかです。これが事実であることを示すために、コンテナ c3 に exec を行うことができます。

```
$ docker container exec -it c3 /bin/sh
```

コンテナ内に入ると、名前と IP アドレスでコンテナ c4 に ping を試みることができます。

```
/ # ping c4
PING c4 (10.1.0.3): 56 data bytes
64 bytes from 10.1.0.3: seq=0 ttl=64 time=0.192 ms
64 bytes from 10.1.0.3: seq=1 ttl=64 time=0.148 ms
...
```

以下は、コンテナ c4 の IP アドレスを使用した ping の結果です。

```
/ # ping 10.1.0.3
PING 10.1.0.3 (10.1.0.3): 56 data bytes
64 bytes from 10.1.0.3: seq=0 ttl=64 time=0.200 ms
64 bytes from 10.1.0.3: seq=1 ttl=64 time=0.172 ms
...
```

どちらの場合もその答えから、同じネットワークに接続されたコンテナ間の通信が、期待どおりに動作していることを確認できます。コンテナの名前を使って接続できるという事実から、Docker DNS サービスによって提供される名前解決がこのネットワークの中で機能することがわかります。

次は bridge ネットワークと test-net ネットワークが互いにファイアウォールで保護されていることを確認します。これを実証するには、c3 コンテナから c2 コンテナに、その名前または IP アドレスのいずれかで ping を試みることができます。

```
/ # ping c2
ping: bad address 'c2'
```

以下は、代わりにターゲット コンテナ c2 の IP アドレスを使用した ping の結果です。

```
/ # ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
^C
--- 172.17.0.3 ping statistics ---
43 packets transmitted, 0 packets received, 100% packet loss
```

上記のコマンドはハングしたので、Ctrl+C でコマンドを終了させる必要がありました。c2 への ping の答えから、名前解決がネットワーク間で機能しないこともわかります。これは予想どおりの動作です。ネットワークがコンテナに分離レイヤーを追加し、これによりセキュリティが強化されます。

既に、コンテナを複数のネットワークに接続できることを学びました。c5 コンテナを sample-net ネットワークと test-net ネットワークに同時に接続してみましょう。

```
$ docker container run --name c5 -d \
    --network sample-net \
    --network test-net \
    alpine:latest ping 127.0.0.1
```

その後、c5 が c2 コンテナからアクセスできることを、コンテナ c4 と c2 のテストを行ったときと同じようにテストできます。結果は、接続が実際に行われていることを示しています。

既存のネットワークを削除する場合は、docker network rm コマンドを使用しますが、コンテナが接続されているネットワークを誤って削除できないようになっています。

```
$ docker network rm test-net

Error response from daemon: network test-net id 863192... has
active endpoints
```

続行する前に、すべてのコンテナをクリーンアップして削除しましょう。

```
$ docker container rm -f $(docker container ls -aq)
```

その後で、作成した 2 つのカスタム ネットワークを削除します。

```
$ docker network rm sample-net

$ docker network rm test-net
```

ホスト ネットワーク

ホストのネットワーク名前空間でコンテナを実行する場合もあります。これが必要になるのは、ホスト ネットワークのトラフィックの分析またはデバッグに使用されるコンテナでソフトウェアを実行する必要がある場合です。ただし、これは非常に特殊なシナリオであることに注意してください。コンテナでビジネス ソフトウェアを実行する場合、ホストのネットワークに接続されたそれぞれのコンテナを実行することに、正当な理由はありません。セキュリティ上の理由から、運用環境や運用に近い環境で、ホスト ネットワークに接続されたコンテナは実行しないことを強くお勧めします。

それなら、ホストのネットワーク名前空間内でコンテナを実行するにはどうすればよいですか。ただコンテナを host ネットワークに接続するだけです。

```
$ docker container run --rm -it --network host alpine:latest /bin/
sh
```

ip ツールを使用してコンテナ内からネットワーク名前空間を分析すると、ip ツールをホスト上で直接実行した場合とまったく同じ結果が得られることがわかります。たとえば、ホスト上の eth0 デバイスを検査した場合、以下の結果が返ります。

```
/ # ip addr show eth0

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UP qlen 1000

    link/ether 02:50:00:00:00:01 brd
```

```
    ff:ff:ff:ff:ff:ff
    inet 192.168.65.3/24 brd 192.168.65.255 scope
    global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::c90b:4219:ddbd:92bf/64 scope link
       valid_lft forever preferred_lft forever
```

ここで、`192.168.65.3` はホストが割り当てられている IP アドレスであることと、ここに表示されている MAC アドレスもホストの MAC アドレスに対応していることがわかります。

またルートを調べると、次の (短縮した) 結果が返ります。

```
/ # ip route
default via 192.168.65.1 dev eth0 src 192.168.65.3 metric 202
10.1.0.0/16 dev cni0 scope link src 10.1.0.1
127.0.0.0/8 dev lo scope host
172.17.0.0/16 dev docker0 scope link src 172.17.0.1
...
192.168.65.0/24 dev eth0 scope link src 192.168.65.3 metric 202
```

この章の次のセクションに進む前に、`host` ネットワークを使用することは危険であり、可能な場合は避ける必要があることを再度指摘しておきます。

Null ネットワーク

時には、タスクの実行にネットワーク接続をまったく必要としないアプリケーション サービスやジョブを実行する必要があります。そうしたアプリケーションは、`none` ネットワークに接続されているコンテナで実行することを強くお勧めします。このコンテナは完全に分離されているため、外部からアクセスされる心配がありません。そのようなコンテナを実行しましょう。

```
$ docker container run --rm -it --network none alpine:latest /bin/
sh
```

コンテナ内に入ると、利用可能な `eth0` ネットワーク ポイントがないことを確認できます。

```
/ # ip addr show eth0
ip: can't find device 'eth0'
```

利用可能なルーティング情報もないことは、次のコマンドを使って示すことができます。

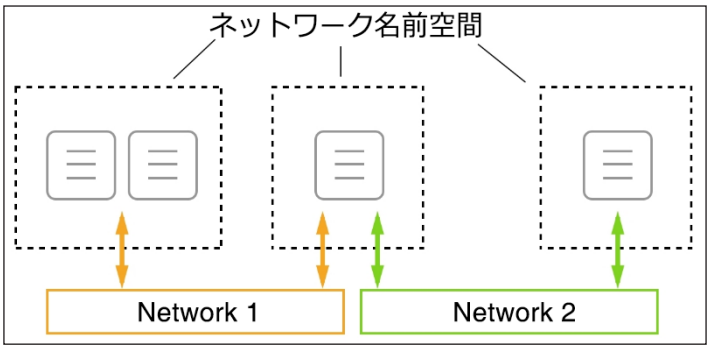
```
/ # ip route
```

これは何も返しません。

既存のネットワーク名前空間での実行

通常 Docker は、実行する各コンテナに対して新しいネットワーク名前空間を作成します。コンテナのネットワーク名前空間は、既に説明したコンテナ ネットワーク モデルのサンドボックスに対応します。コンテナをネットワークに接続するとき、コンテナのネットワーク名前空間と実際のネットワークを接続するエンドポイントを定義します。この方法では、コンテナはネットワーク名前空間ごとに 1 つです。

Docker には、コンテナを実行するネットワーク名前空間を定義する他の方法があります。新しいコンテナを作成するときに、それを接続する必要があるか、または既存のコンテナのネットワーク名前空間に含める必要があるかを指定できます。この手法では、単一のネットワーク名前空間で複数のコンテナを実行できます。



1 つのネットワーク名前空間で実行されている複数のコンテナ

上の図では、一番左のネットワーク名前空間に 2 つのコンテナがあることがわかります。2 つのコンテナは、同じ名前空間を共有しているため、互いにローカルホストで通信できます。次に、ネットワーク名前空間 (個々のコンテナではなく) が **Network 1** に接続されます。

これは、既存のコンテナのネットワークのデバッグを、そのコンテナ内で追加のプロセスを実行せずに行う場合に便利です。特別なユーティリティ コンテナを、検査するコンテナのネットワーク名前空間に接続するだけで行えます。この機能は、Kubernetes がポッドを作成するときにも使用されます。Kubernetes とポッドについては、本書の次の章で詳しく説明します。

では、これがどのように機能するかを見てみましょう。

1. まず、新しいブリッジ ネットワークを作成します。

   ```
   $ docker network create --driver bridge test-net
   ```

2. 次に、このネットワークに接続されたコンテナを実行します。

   ```
   $ docker container run --name web -d --network test-net
   nginx:alpine
   ```

3. 最後に、別のコンテナを実行し、それを `web` コンテナのネットワークに接続します。

   ```
   $ docker container run -it --rm --network container:web
   alpine:latest /bin/sh
   ```

具体的には、ネットワークの定義方法、`--network container:web` に注目してください。これは、新しいコンテナが `web` というコンテナと同じネットワーク名前空間を使用することを Docker に伝えています。

新しいコンテナは、Nginx を実行している web コンテナと同じネットワーク名前空間にあるので、localhost で Nginx にアクセスできるようになりました。これは `wget` ツール (Alpine コンテナの一部) を使用して Nginx に接続することで証明できます。次のように表示されるはずです。

```
/ # wget -qO - localhost


<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

読みやすさを考慮して出力を短くしていることに留意してください。また、同じネットワークに接続された 2 つのコンテナを実行することと、同じネットワーク名前空間で実行されている 2 つのコンテナには、重要な違いがあることにも注意してください。どちらの場合も、コンテナは互いに自由に通信できますが、後者の場合、通信はローカルホスト上で行われます。

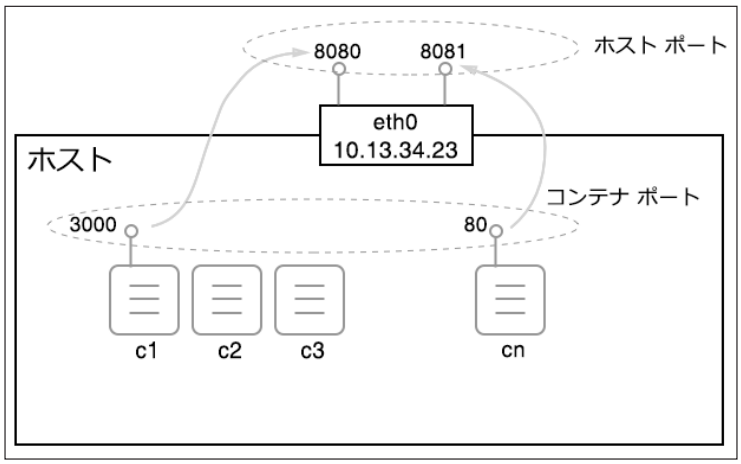
コンテナとネットワークをクリーンアップするには、次のコマンドを使用します。

```
$ docker container rm --force web
$ docker network rm test-net
```

ポート管理

ここまでに、コンテナを異なるネットワーク上に配置することによって、コンテナを分離またはファイアウォールで保護する方法を学び、複数のネットワークにコンテナを接続できるようになりましたが、未解決の問題が 1 つあります。アプリケーション サービスを外部に公開するにはどうすればよいですか。以前から webAPI をホストしている Web サーバーをコンテナで実行するとします。インターネットのお客様がこの API にアクセスできるようにしたいと思います。この API は一般からアクセス可能な API として設計されています。これを実現するには、例えて言うなら、ファイアウォールのゲートを開けて API に外部トラフィックを送り込めるようにする必要があります。セキュリティ上の理由から、ドアを広く開けたくないだけでなく、制御された 1 つのゲートのみからトラフィックが流れるようにしたいと思います。

このようなゲートは、コンテナ ポートをホスト上の利用可能なポートにマッピングすることによって作成できます。また、ポートを公開するためにこのコンテナ ポートを呼び出しています。コンテナには、ホストと同様に独自の仮想ネットワーク スタックがあります。そのため、コンテナ ポートとホスト ポートは完全に独立して存在しており、既定では何も共有していません。ただし、次のスクリーンショットに示すように、コンテナ ポートを空きホスト ポートに配線し、このリンクを介して外部トラフィックを流すことができます。



ホスト ポートへのコンテナ ポートのマッピング

次はコンテナ ポートをホスト ポートに実際にマッピングする方法を示します。これは、コンテナを作成するときに行います。その方法は複数あります。

- まず、コンテナ ポートのマッピング先になるホスト ポートを Docker に決定させることができます。その場合 Docker は、32xxx の範囲で空きホスト ポートの 1 つを選択します。この自動マッピングは、`-P` パラメーターを使用して行います。

```
$ docker container run --name web -P -d nginx:alpine
```

上記のコマンドは、Nginx サーバーをコンテナ内で実行します。Nginx はコンテナ内のポート `80` でリッスンしています。`-P` パラメーターを使用して、公開されているすべてのコンテナ ポートを 32xxx 範囲の空きポートにマップするよう、Docker に指示しています。Docker が使用しているホスト ポートは、`docker container port` コマンドを使用することで確認できます。

```
$ docker container port web
80/tcp -> 0.0.0.0:32768
```

Nginx コンテナがポート `80` のみを公開し、それがホスト ポート `32768` にマッピングされていることがわかります。新しいブラウザ ウィンドウを開いて `localhost:32768` に移動すると、次のスクリーンショットが表示されます。

Nginx ウェルカム ページ

- Docker がコンテナに使用しているホスト ポートを確認するもう一つの方法は、検査です。ホスト ポートは `NetworkSettings` ノードの一部です。

```
$ docker container inspect web | grep HostPort
"HostPort": "32768"
```

- この情報を取得する第 3 の方法は、コンテナを一覧表示することです。

```
$ docker container ls
CONTAINER ID    IMAGE          ...
PORTS                    NAMES
56e46a14b6f7    nginx:alpine   ...0.0.0.0:32768->80/tcp  web
```

上記の出力で、/tcp の部分から、ポートが開かれているのは TCP プロトコルとの通信のためであり、UDP プロトコル用ではないことがわかります。TCP が既定です。UDP 用にポートを開く場合は、それを明示的に指定する必要があります。マッピングの 0.0.0.0 から、どのホスト IP アドレスのトラフィックも、web コンテナのコンテナ ポート 80 に到達できるようになったことがわかります。

時には、コンテナ ポートを非常に特殊なホスト ポートにマッピングする場合もあります。これを行うには、パラメーター -p (または --publish) を使用します。次のコマンドを使って、これがどのように行われるかを見てみましょう。

```
$ docker container run --name web2 -p 8080:80 -d nginx:alpine
```

-p パラメーターの値は、< ホスト ポート >:< コンテナ ポート > の形式です。したがってここでは、コンテナ ポート 80 をホスト ポート 8080 にマッピングしています。web2 コンテナが実行されたら、localhost:8080 に移動することでブラウザ内でテストを行えるようになります。その結果、自動ポート マッピングを扱った前の例と同じ Nginx ウェルカム ページが表示されるはずです。

特定のポートを介した通信に UDP プロトコルを使用する場合、publish パラメーターは -p 3000:4321/udp のようになります。同じポート上で TCP プロトコルと UDP プロトコルの両方で通信できるようにするには、各プロトコルを別々にマッピングする必要があります。

まとめ

この章では、単一のホスト上で実行されているコンテナがどのように互いに通信できるかについて学習しました。最初に、コンテナ ネットワークの要件を定義する CNM を確認してから、ブリッジ ネットワークなどの CNM のいくつかの実装を検討しました。次に、ブリッジ ネットワークがどのように機能しているか、ネットワークとそのネットワークに接続されているコンテナについて Docker がどのような情報を提供しているかを詳細に確認しました。また、2 つの異なる視点を採用し、コンテナの外側と内側の両方から見ることについても学びました。

次の章では Docker Compose を紹介します。個々にコンテナで実行される複数のサービスから成るアプリケーションを作成することと、このようなアプリケーションを宣言型アプローチを使って簡単にビルド、実行、スケーリングできる Docker Compose の機能について学びます。

質問

スキルを評価するために、以下の質問にお答えください。

1. **コンテナ ネットワーク モデル** (**CNM**) の 3 つの主要な要素を挙げてください。
2. たとえば frontend と呼ばれるカスタム ブリッジ ネットワークをどのように作成しますか。
3. frontend ネットワークに接続された 2 つの nginx:alpine コンテナをどのように実行しますか。
4. frontend ネットワークで、次の内容を取得してください。
    1. 接続されたすべてのコンテナの IP。
    2. ネットワークに関連付けられたサブネット。
5. ホスト ネットワークの目的は何ですか。
6. ホスト ネットワークの使用が適切なシナリオを 1 ～ 2 つ挙げてください。
7. non ネットワークの目的は何ですか。
8. none ネットワークはどのようなシナリオで使用すべきですか。

参考情報

ここでは、この章で扱ったトピックを詳しく説明している記事をいくつか紹介します。

- Docker のネットワーキングの概要 (`http://dockr.ly/2sXGzQn`)
- コンテナ ネットワーキング (`http://dockr.ly/2HJfQKn`)
- ブリッジとは (`https://bit.ly/2HyC3Od`)
- ブリッジ ネットワークを使用する (`http://dockr.ly/2BNxjRr`)
- Macvlan ネットワークを使用する (`http://dockr.ly/2ETjy2x`)
- ホスト ネットワークを使用したネットワーキング (`http://dockr.ly/2F4aI59`)

8
Docker Compose

前章では、単一の Docker ホスト上におけるコンテナ ネットワーキングの仕組みについて学び、Docker コンテナ間のあらゆるネットワーキングの基礎となる**コンテナ ネットワーク モデル** (**CNM**) の概要、さらには CNM のさまざまな実装、特にブリッジ ネットワークについて詳しく見ました。

この章では、複数のサービスから成るアプリケーションがそれぞれ単一のコンテナで実行されるという概念と、このようなアプリケーションを宣言型アプローチを使って簡単にビルド、実行、スケーリングする Docker Compose の機能について説明します。

この章では以下のトピックについて説明します。

- 宣言型と命令型の違い
- マルチサービス アプリケーションの実行
- サービスのスケーリング
- アプリケーションのビルドとプッシュ

この章を終了すると、次のことができるようになります。

- アプリケーションの定義および実行に用いる命令型アプローチと宣言型アプローチの主な違いを簡単に説明する
- コンテナと Docker Compose サービスの違いを自分の言葉で説明する
- シンプルなマルチサービス アプリケーション向けの Docker Compose YAML ファイルを作成する
- Docker Compose を使用してシンプルなマルチサービス アプリケーションをビルド、プッシュ、デプロイ、破棄する
- Docker Compose を使ってアプリケーション サービスをスケール アップ / スケール ダウンする

技術的要件

この章に付随するコードは、`https://github.com/appswithdockerandkubernetes/labs/tree/master/ch08` で確認できます。

宣言型と命令型の違い

Docker Compose は Docker が提供するツールで、単一の Docker ホスト上で動作するコンテナを実行およびオーケストレーションする必要がある環境で主に使用されます。これには、開発、**継続的インテグレーション** (**CI**)、自動テスト、手動 QA が含まれますが、これらに限定されません。

Docker Compose は、YAML 形式のファイルを入力として使用します。デフォルトでは、これらのファイルは `docker-compose.yml` となりますが、他の名前も可能です。`docker-compose.yml` のコンテンツは、複数のコンテナから構成される可能性があるコンテナ化アプリケーションを記述および実行する方法が、宣言型だといわれています。

では、宣言型とは何を意味するのでしょうか。

まず、宣言型とは命令型の反対語ですが、これではよくわからないと思いますので、両方の定義を説明しましょう。

- **命令型** : システムが従う必要のある正確な方法を特定することで問題を解決する方法です。

  Docker デーモンなどのシステムにアプリケーションの実行方法を命令的に伝えることは、システムが何をすべきか、また何らかの予期しない状況が起こった場合にどのように対応すべきかを段階的に記述する必要があることを意味します。指示は明確かつ正確である必要があり、すべてのエッジ ケースと対処法を網羅しなくてはなりません。

- **宣言型** : プログラマーが従うべき正確な手順を特定することなく問題を解決する方法です。

  宣言型アプローチでは、アプリケーションの目的の状態を Docker エンジンに指示すると、エンジンはこの状態を達成する方法と、システムがこの状態から逸脱した場合の調整方法を独自に判断しなければなりません。

Docker では、コンテナ化アプリケーションには宣言型アプローチを採用するよう推奨しているため、Docker Compose ツールではこのアプローチを使用しています。

マルチサービス アプリの実行

通常、アプリケーションは単一のモノリシック ブロックではなく、連携する複数のアプリケーション サービスで構成されています。Docker コンテナを使用する場合、各アプリケーション サービスは独自のコンテナで実行されます。このようなマルチサービス アプリケーションを実行する場合は、よく知られている `docker` コンテナの実行コマンドで参加している全コンテナを開始することができますが、これは決して効率的とはいえません。Docker Compose ツールなら、YAML 形式を使用したファイルで宣言的にアプリケーションを定義することができます。

シンプルな `docker-compose.yml` ファイルのコンテンツを見てみましょう。

```
version: "3.5"
services:
  web:
    image: appswithdockerandkubernetes/ch08-
    web:1.0
    ports:
      - 3000:3000
  db:
    image: appswithdockerandkubernetes/ch08-db:1.0
    volumes:
      - pets-data:/var/lib/postgresql/data

volumes:
  pets-data:
```

ファイル内の行は次のように解釈されます。

- `version`: この行では、使用する Docker Compose 形式のバージョンを指定します。この記事の執筆時点では、バージョン 3.5 です。
- `services`: このセクションでは、`services` ブロックのアプリケーションを構成するサービスを指定します。このサンプルには、`web` と `db` の 2 つのアプリケーション サービスがあります。
    - `web`: `web` サービスは Docker Hub から `appswithdockerandkubernetes/ch08-web:1.0` のイメージを使用しており、コンテナ ポート `3000` をホスト ポート `3000` に発行します。
    - `db`: 一方 `db` サービスは、カスタマイズされた PostgreSQL データベースである `appswithdockerandkubernetes/ch08-db:1.0` のイメージを使用しています。ここでは、`pets-data` と呼ばれるボリュームを、`db` サービスのコンテナにマウントしています。

- `volumes`: いずれかのサービスで使用されるボリュームはこのセクションで宣言される必要があります。このサンプルでは、これがファイルの最後のセクションです。アプリケーションの初回実行時、`pets-data` と呼ばれるボリュームが Docker によって作成されます。後続の実行では、ボリュームがまだそこに存在すれば再利用されます。これは、アプリケーションが何らかの理由でクラッシュし、再起動する必要がある場合に重要となることがあります。この場合、以前のデータはまだ存在し、再起動したデータベース サービスによりいつでも使用できます。

`labs` フォルダのサブフォルダ `ch08` に移動し、Docker Compose でアプリケーションを起動します。

```
$ docker-compose up
```

前述のコマンドを入力すると、ツールは現在のディレクトリに `docker-compose.yml` というファイルが存在するとみなし、これを使用して実行します。この場合がまさにこれに当てはまり、アプリケーションが起動します。この場合、出力は次のようになります。

```
$ docker-compose up
Creating network "ch08_default" with the default driver
Creating volume "ch08_pets-data" with default driver
Pulling web (appswithdockerandkubernetes/ch08-web:1.0)...
1.0: Pulling from appswithdockerandkubernetes/ch08-web
605ce1bd3f31: Already exists
d9c1bb40879c: Already exists
d610e8516793: Already exists
66cc3fe80117: Pull complete
e253bc30f8b0: Pull complete
1aeedab40d88: Pull complete
09092f231cdf: Pull complete
Digest: sha256:907f7c06a1ba2ba4b463b1b6c2a002a7fcbef5fc726473e177ed10227a246949
Status: Downloaded newer image for appswithdockerandkubernetes/ch08-web:1.0
Pulling db (appswithdockerandkubernetes/ch08-db:1.0)...
1.0: Pulling from appswithdockerandkubernetes/ch08-db
ff3a5c916c92: Already exists
a503b44e1ce0: Already exists
211706713093: Already exists
8df57d533e71: Already exists
7858f71c02fb: Already exists
55a8ef17ba59: Already exists
3fb44f23d323: Already exists
65cad41156b3: Already exists
5492a5bead70: Already exists
df43dbde4904: Pull complete
Digest: sha256:36a9f43628cfbf1483c7368cd730ef9fbfbefa468fc5c8511b895a27a9391bab
Status: Downloaded newer image for appswithdockerandkubernetes/ch08-db:1.0
Creating ch08_web_1 ... done
Creating ch08_db_1  ... done
Attaching to ch08_db_1, ch08_web_1
```

サンプル アプリケーションの実行、パート 1

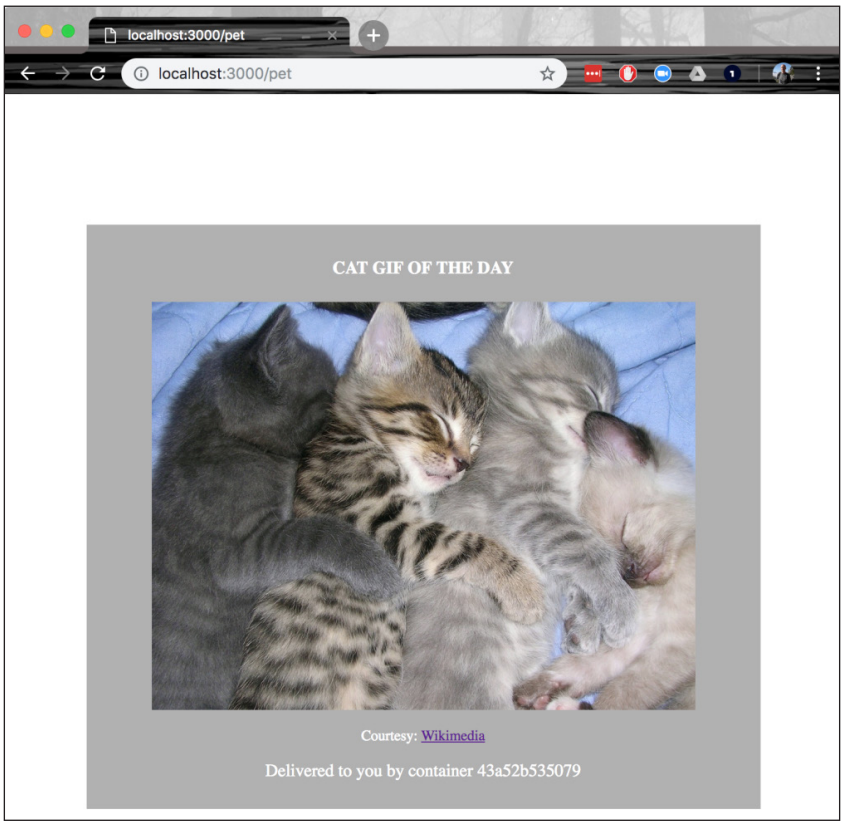
```
db_1   |  done
db_1   | server started
web_1  | Listening at 0.0.0.0:3000
db_1   | CREATE DATABASE
db_1   |
db_1   | CREATE ROLE
db_1   |
db_1   |
db_1   | /usr/local/bin/docker-entrypoint.sh: running /docker-entrypoint-initdb.d/init-db.sql
db_1   | CREATE TABLE
db_1   | ALTER TABLE
db_1   | ALTER ROLE
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   | INSERT 0 1
db_1   |
db_1   |
db_1   | waiting for server to shut down....2018-03-21 12:52:40.709 UTC [34] LOG:  received fast shutdown
 request
db_1   | 2018-03-21 12:52:40.711 UTC [34] LOG:  aborting any active transactions
db_1   | 2018-03-21 12:52:40.712 UTC [34] LOG:  worker process: logical replication launcher (PID 41) exi
ted with exit code 1
db_1   | 2018-03-21 12:52:40.712 UTC [36] LOG:  shutting down
db_1   | 2018-03-21 12:52:40.737 UTC [34] LOG:  database system is shut down
db_1   |  done
db_1   | server stopped
db_1   |
db_1   | PostgreSQL init process complete; ready for start up.
db_1   |
db_1   | 2018-03-21 12:52:40.817 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
db_1   | 2018-03-21 12:52:40.817 UTC [1] LOG:  listening on IPv6 address "::", port 5432
db_1   | 2018-03-21 12:52:40.821 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.543
2"
db_1   | 2018-03-21 12:52:40.832 UTC [49] LOG:  database system was shut down at 2018-03-21 12:52:40 UTC
db_1   | 2018-03-21 12:52:40.835 UTC [1] LOG:  database system is ready to accept connections
```

サンプル アプリケーションの実行、パート 2

以下に、この出力の各項目について説明します。

- 出力の最初の部分で、Docker Compose がアプリケーションを構成する 2 つのイメージをプルしているのがわかります。この後に、ネットワーク `ch08_default`、ボリューム `ch08_pets-data` 、さらに `ch08_web_1` と `ch08_db_1` の 2 つのコンテナ (`web` および `db` の各サービスに 1 つ) が作成されます。すべての名前には、Docker Compose によって親ディレクトリの名前が自動的にプレフィックスされ、この場合は `ch08` と呼ばれます。
- その後、2 つのコンテナによって生成されたログが表示されます。出力の各行にはサービス名がプレフィックスされ、各サービスの出力の色はそれぞれ異なっています。ここでは大半はデータベースによって生成されており、`web` サービスからのものは 1 行のみです。

これでブラウザ タブを開き、`localhost:3000/pet` に移動できます。下記のスクリーンショットのように、可愛らしい猫のイメージと作成元のコンテナに関する情報が表示されます。



ブラウザのサンプル アプリケーション

ブラウザを数回リフレッシュすると、他の猫のイメージが表示されます。アプリケーションは、データベースに URL が格納されている 12 枚のイメージから現在のイメージをランダムに選択します。

アプリケーションはインタラクティブ モードで実行しているため、Docker Compose を実行しているターミナルはブロックされ、Ctrl+C を押すとアプリケーションをキャンセルできます。これを実行すると、以下が表示されます。

```
^CGracefully stopping...(press Ctrl+C again to force)
Stopping ch08_web_1 ... done
Stopping ch08_db_1 ... done
```

データベース サービスは直ちに停止するのに対し、web サービスではこれに約 10 秒かかることがわかります。この理由は、データベース サービスは Docker が送信する `SIGTERM` シグナルを感知し、これに応答する一方で、web サービスはこれを行わないために、Docker が 10 秒後にサービスを終了するためです。

アプリケーションを再度実行すると、出力は次のようにかなり短くなります。

```
$ docker-compose up
Creating network "ch08_default" with the default driver
Creating ch08_web_1 ... done
Creating ch08_db_1  ... done
Attaching to ch08_web_1, ch08_db_1
db_1   | 2018-03-02 01:25:35.874 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
db_1   | 2018-03-02 01:25:35.875 UTC [1] LOG:  listening on IPv6 address "::", port 5432
db_1   | 2018-03-02 01:25:35.877 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
db_1   | 2018-03-02 01:25:35.890 UTC [19] LOG:  database system was shut down at 2018-03-02 01:25:23 UTC
db_1   | 2018-03-02 01:25:35.894 UTC [1] LOG:  database system is ready to accept connections
web_1  | Listening at 0.0.0.0:3000
```

docker-compose up の出力

今回はイメージをダウンロードしたり、データベースを最初から初期化する必要はなく、以前の実行からのボリューム `pets-data` にすでに存在していたデータを再利用しました。

バックグラウンドでアプリケーションを実行することもできます。すべてのコンテナはデーモンとして実行されます。そのためには、下記のコードのような `-d` パラメータを使用する必要があります。

```
$ docker-compose up -d
```

Docker Compose が提供するコマンドはこれだけではありません。それを活用すれば、アプリケーションの一部である全サービスをリストすることが可能です。

```
$ docker-compose ps
   Name                 Command              State          Ports
-------------------------------------------------------------------------
ch08_db_1    docker-entrypoint.sh postgres   Up      5432/tcp
ch08_web_1   /bin/sh -c node src/server.js   Up      0.0.0.0:3000->3000/tcp
$
```

docker-compose ps の出力

このコマンドは、アプリケーションの一部であるコンテナのみをリストするという違いを除いては、`docker container ls` と似ています。

アプリケーションの停止およびクリーンアップには、`docker-compose down` コマンドを使用します。

```
$ docker-compose down
Stopping ch08_web_1 ... done
Stopping ch08_db_1 ... done
Removing ch08_web_1 ... done
Removing ch08_db_1 ... done
Removing network ch08_default
```

データベースのボリュームも削除したい場合は、下記のコマンドを使用できます。

```
$ docker volume rm ch08_pets-data
```

ボリューム名に ch08 のプレフィックスが付いているのはなぜでしょうか。`docker-compose.yml` ファイルでは、`pets-data` を使用するためにボリュームを呼び出していますが、前述したように、Docker Compose は `docker-compose.yml` ファイルの親フォルダ名とアンダースコアをすべての名前にプレフィックスします。この場合、親フォルダは `ch08` と呼ばれます。

サービスのスケーリング

ここでは、サンプル アプリケーションが Web 上にあり、非常にうまくいったとしましょう。大勢の人が可愛らしい動物の画像を見たがっています。しかしここでトラブル発生。アプリケーションの速度が低下し始めたのです。この問題を解決するには、Web サービスの複数のインスタンスを実行する必要があります。Docker Compose なら、これは実に簡単です。

さらに多くのインスタンスを実行することも**スケール アップ**と呼ばれます。このツールを使えば、web サービスを 3 つまでスケーリングできます。

```
$ docker-compose up --scale web=3
```

これから、驚くようなことをお見せします。下記のスクリーンショットのような出力が表示されます。

```
$ docker-compose up --scale web=3
WARNING: The "web" service specifies a port on the host. If multiple containers for this
service are created on a single host, the port will clash.
Starting ch08_web_3 ...
Starting ch08_web_3 ... error

ERROR: for ch08_web_3  Cannot start service web: driver failed programming external conne
ctivity on endpoint ch08_web_3 (534216cc36e0284b775e48c6450e25ff21fe90ff6d7b8b9716f421cb9
8560351): Bind for 0.0.0.0:3000 failed: port is already allocated

ERROR: for web  Cannot start service web: driver failed programming external connectivity
 on endpoint ch08_web_3 (534216cc36e0284b775e48c6450e25ff21fe90ff6d7b8b9716f421cb98560351
): Bind for 0.0.0.0:3000 failed: port is already allocated
ERROR: Encountered errors while bringing up the project.
$
```

docker-compose --scale の出力

Web サービスの 2 番目と 3 番目のインスタンスが起動していません。その理由はエラー メッセージが示しています。つまり、同じホスト ポートは何度も使用できないのです。インスタンス 2 と 3 が起動しようとすると、Docker はポート 3000 がすでに最初のインスタンスに取得されていることを認識します。あとは、各インスタンスに使用するホスト ポートを Docker に判断してもらうしかありません。

compose ファイルの ports セクションで、コンテナ ポートのみを指定してホスト ポートを省略すると、Docker は一時的なポートを自動的に選択します。次のとおりに実行してみましょう。

1. まず、アプリケーションを破棄します。

   ```
   $ docker-compose down
   ```

2. 次に、docker-compose.yml ファイルを次のように修正します。

   ```
   version: "3.5"
   services:
     web:
       image: appswithdockerandkubernetes/ch08-web:1.0
       ports:
         - 3000
     db:
   ```

```
    image: appswithdockerandkubernetes/ch08-db:1.0
    volumes:
      - pets-data:/var/lib/postgresql/data

volumes:
  pets-data:
```

3. これで、アプリケーションを再度起動し、その直後にスケール アップすることができます。

```
$ docker-compose up -d
$ docker-compose scale web=3
Starting ch08_web_1 ... done
Creating ch08_web_2 ... done
Creating ch08_web_3 ... done
```

4. `docker-compose ps` を実行すると、次のスクリーンショットが表示されます。

```
$ docker-compose ps
   Name                 Command              State           Ports
-------------------------------------------------------------------------------
ch08_db_1    docker-entrypoint.sh postgres   Up      5432/tcp
ch08_web_1   /bin/sh -c node src/server.js   Up      0.0.0.0:32769->3000/tcp
ch08_web_2   /bin/sh -c node src/server.js   Up      0.0.0.0:32771->3000/tcp
ch08_web_3   /bin/sh -c node src/server.js   Up      0.0.0.0:32770->3000/tcp
$
```

docker-compose ps の出力

5. ご覧のように、各サービスは異なるホスト ポートに関連付けられています。これがうまく動作するかどうか確認するには、curl などを使用できます。3 番目のインスタンス、`ch08_web_3` をテストしてみましょう。

```
$ curl -4 localhost:32770
Pets Demo Application
```

`Pets Demo Application` という答えから、アプリケーションが期待どおりに動作していることがわかります。その他 2 つのインスタンスについても確認してみてください。

アプリケーションのビルドとプッシュ

docker-compose build コマンドを使用して、基盤となる compose ファイルで定義されたアプリケーションのイメージをビルドすることもできます。ただしこれを行うには、そのビルド情報を docker-compose ファイルに追加する必要があります。フォルダには、docker-compose.dev.yml というファイルがあり、すでに追加された次のような説明が含まれます。

```
version: "3.5"
services:
  web:
    build: web
    image: appswithdockerandkubernetes/ch08-
    web:1.0
    ports:
      - 3000:3000
  db:
    build: database
    image: appswithdockerandkubernetes/ch08-db:1.0
    volumes:
      - pets-data:/var/lib/postgresql/data

volumes:
  pets-data:
```

各サービスの build キーに注目してください。このキーの値は、対応するイメージをビルドする Dockerfile があると Docker が期待しているコンテキストまたはフォルダを示します。

では、そのファイルを使ってみましょう。

```
$ docker-compose -f docker-compose.dev.yml build
```

-f パラメーターは、使用すべき compose ファイルを Docker Compose アプリケーションに指示します。

すべてのイメージを Docker Hub にプッシュするには、docker-compose push を使用します。これを問題なく実行するには Docker Hub にログインする必要があり、そうしないとプッシュ中に認証エラーが発生します。そのため、ここでは以下を実行します。

```
$ docker login -u appswithdockerandkubernetes -p <password>
```

ログインが成功したと仮定して、下記のコードをプッシュします。

```
$ docker-compose -f docker-compose.dev.yml push
```

前述のコマンドは、Docker Hub 上のアカウント `appswithdockerandkubernetes` に 2 つのイメージをプッシュします。これら 2 つのイメージは URL `https://hub.docker.com/u/appswithdockerandkubernetes/` にあります。

まとめ

この章では、`docker-compose` ツールについて説明しました。このツールは主に、単一の Docker ホスト上でマルチサービス アプリケーションを実行およびスケーリングするために使用されます。通常、開発者と CI サーバーは単一のホストを使用し、この 2 つが Docker Compose のメイン ユーザーとなります。このツールは、宣言的な方法でアプリケーションの記述を含む YAML ファイルを入力として使用します。

またこのツールは特に、イメージのビルドおよびプッシュにも使用できます。この章に付随するコードは、`labs/ch08` で確認できます。

次の章では、オーケストレーターについて説明します。**オーケストレーター**は、クラスター内でコンテナ化アプリケーションの実行および管理に使用されるインフラソフトウェアで、これらのアプリケーションが常に目的の状態にあることを確認します。

質問

学習の進捗状況を評価するために、以下の質問にお答えください。

1. デーモン モードでアプリケーションを実行するには `docker-compose` をどのように使用しますか。
2. 実行中のサービスの詳細を表示するには `docker-compose` をどのように使用しますか。
3. どのようにして特定の `web` サービスを 3 つのインスタンスにスケール アップしますか。

参考情報

下記のリンクには、この章で取り上げたトピックに関する追加情報が記載されています。

- YAML 公式サイト : `http://yaml.org/`
- Docker Compose ドキュメント : `http://dockr.ly/1FL2VQ6`
- Compose ファイル バージョン 3 のリファレンス: `http://dockr.ly/2iHUpeX`

9
オーケストレーター

前の章では Docker Compose を紹介しました。Docker Compose は、単一の Docker ホスト上で宣言的な方法で定義されたマルチサービス アプリケーションを扱えるツールです。

この章では、オーケストレーターの概念について説明します。説明する内容は、オーケストレーターがなぜ必要なのか、そして概念的にどのように動作するのかについてです。また、よく使われているオーケストレーターの概要を示し、それぞれの長所と短所をいくつか取り上げます。

この章では以下のトピックをとりあげます。

- オーケストレーターの役割と必要な理由
- オーケストレーターのタスク
- 一般的なオーケストレーターの概要

この章を読み終わると、次のことができるようになります。

- オーケストレーターが実行するタスクを 3 ～ 4 つ挙げる
- 最もよく使われているオーケストレーターを 2 ～ 3 つ挙げる
- 関心を持つ初心者に対し、自分の言葉と適切な例えを使って、コンテナ オーケストレーターが必要な理由を説明する

オーケストレーターの役割と必要な理由

第 6 章「分散アプリケーション アーキテクチャ」では、高度に分散されたアプリケーションの構築、配信、運用を円滑に進めるために一般的に使用されているパターンとベスト プラクティスについて学習しました。現在のところ、高度に分散されたアプリケーションがコンテナ化されている場合でも、コンテナ化されていない分散アプリケーションとまったく同じ問題や課題に直面することになります。このような課題としては、第 6 章「分散アプリケーション アーキテクチャ」で説明したサービス検出、負荷分散、スケーリングなどが挙げられます。

Docker では、コンテナによってソフトウェアのパッケージ化および配信を標準化していますが、これと同様に前述の課題のすべてまたはほとんどを処理するツールやインフラ ソフトウェアがあると便利です。このようなソフトウェアがオーケストレーターであり、オーケストレーション エンジンと呼ばれることもあります。

この説明ではピンとこない方のために、別の視点から説明しましょう。たとえば、楽器を演奏するアーティストがいたとします。それぞれのアーティストは、自身の技術と楽器さえあれば、聴衆に素晴らしい音楽を届けてくれることでしょう。しかし、オーケストラとして構成した場合はどうでしょうか。アーティスト全員に 1 つの部屋に入ってもらい、交響曲の楽譜を渡して演奏するように指示し、部屋を出たとします。非常に才能に溢れた演奏家のグループであっても、指揮者がいなければ調和の取れた演奏ができず、単なる騒音になってしまうことでしょう。心地よく楽しめるオーケストラとしての音楽を奏でられるのは、演奏家のグループをまとめる指揮者がいるオーケストラだけなのです。

コンテナ オーケストレーターとは、オーケストラの指揮者のようなものです

演奏家に相当するのがコンテナです。さまざまな楽器が存在するように、コンテナは運用するコンテナ ホストに対してさまざまな要件を持ちます。また、音楽がさまざまなテンポで演奏されるように、コンテナは特定の方法で相互に通信し、スケール アップしたりスケール ダウンしたりする必要があります。このような点で、コンテナ オーケストレーターの役割はオーケストラの指揮者とほぼ同じといえます。クラスター内のコンテナとその他のリソースが調和して動作できるようにします。

ここまでの説明で、コンテナ オーケストレーターの役割と、それが必要な理由を明確に理解していただければ幸いです。では、オーケストレーターはどのようにして予期される成果、つまりクラスター内のすべてのコンテナが相互に調和して動作する状態を実現するのでしょうか。その答えとなるのが、オーケストレーターが実行する非常に具体的なタスクです。これはオーケストラの指揮者が、オーケストラの演奏に抑揚を付けるために行う一連の動作に似ています。

オーケストレーターのタスク

では、オーケストレーターにかかるコストの対価として期待できるタスクとはどのようなものでしょうか。ここでは、それらについて詳しく説明します。本書の執筆時点で、一般的に企業ユーザーがオーケストレーターから期待できる最も重要なタスクは、以降で示すとおりです。

目的の状態の調整

オーケストレーターを使用する場合は、特定のアプリケーションまたはアプリケーション サービスの運用方法を宣言的に示します。宣言型と命令型の意味については、第 8 章「 Docker Compose」で学習しました。運用するアプリケーション サービスを宣言的な方法で記述する場合は、使用するコンテナ イメージや、そのサービスを実行するインスタンスの数、開放するポートなどの要素を使用します。このようなアプリケーション サービスのプロパティの宣言を目的の状態と呼びます。

したがって、このような新しいアプリケーション サービスを宣言に基づいて作成するようにオーケストレーターに初めて指示すると、オーケストレーターは要求された数のコンテナをクラスター内で使用できるようにスケジュール調整します。コンテナを実行するクラスターのターゲット ノード上でコンテナ イメージを使用できない場合は、まずイメージ レジストリからダウンロード済みであることをスケジューラーが確認します。次に、接続先のネットワークや公開するポートなど、すべての設定が行われた状態でコンテナが起動されます。オーケストレーターは、宣言されている内容を実際のクラスター内でできるだけ正確に実現できるように努めます。

サービスが要求どおりに起動して稼働している場合、つまり目的の状態で実行されている場合でも、オーケストレーターは監視を継続します。オーケストレーターは、サービスの実際の状態と目的の状態との間に相違があることを検出するたびに、再度目的の状態に調整できるよう最善の試みを行います。

では、アプリケーション サービスの実際の状態と目的の状態との間に発生する相違とは、どのようなものでしょうか。たとえば、いずれかのサービスのレプリカ、つまりいずれかのコンテナがバグのためにクラッシュしたとします。この場合、レプリカが 1 つ不足することになるため、オーケストレーターはレプリカ数の点で実際の状態が目的の状態と異なることを検出します。オーケストレーターは、クラッシュしたインスタンスの置き換えとなる新しいインスタンスを別のクラスター ノードで確保できるように直ちにスケジュールします。そのほかの相違としては、実行中のアプリケーション サービスのインスタンス数が多すぎるケースが挙げられ、サービスがスケール ダウンされた場合に発生します。この場合、オーケストレーターは、実際のインスタンス数と目的のインスタンス数が一致するように、必要な数のインスタンスをランダムに終了します。そのほかには、アプリケーション サービスのインスタンスが実行する基盤となるコンテナ イメージのバージョンが誤っている (古い) ことをオーケストレーターが検出したケースなども相違の例として挙げられます。ここまでは理解していただけたでしょうか。

つまり、クラスター内で実行されているアプリケーションのサービスをアクティブに監視し、目的の状態と異なる場合は修正するという従来は人間が行っていた面倒な作業をオーケストレーターに委任するということです。これは、アプリケーション サービスの目的の状態を命令型ではなく宣言的な方法で記述した場合に非常にうまく機能します。

レプリケートされたサービスとグローバルなサービス

クラスター内で実行し、オーケストレーターによって管理するサービスは、大きく 2 種類に分けられます。それは、**レプリケートされたサービス**と**グローバルなサービス**です。レプリケートされたサービスとは、10 など、特定数のインスタンスで実行する必要があるサービスです。一方グローバルなサービスとは、各インスタンスをクラスターの単一のワーカー ノードで実行する必要があるサービスです。ここでワーカー ノードという用語を使いましたが、オーケストレーターが管理するクラスターには、通常、**マネージャー**および**ワーカー**という 2 種類のノードが存在します。通常、マネージャー ノードはクラスターを管理するためにオーケストレーターによって排他的に使用され、他のワークロードを実行しません。一方ワーカーノードは、実際のアプリケーションを実行します。

そのためオーケストレーターは、グローバル サービスについては実行するインスタンス数にかかわらず、すべて単一のワーカー ノードで実行します。インスタンス数を気にする必要はありません。各ノードでは、サービスの単一のインスタンスだけを実行することが保証されます。

繰り返しになりますが、この処理についてはオーケストレーターに完全に依存できます。レプリケートされたサービスでは常に目的のインスタンス数が確保され、グローバル サービスではすべてのワーカー ノードで必ずサービスの単一のインスタンスだけが実行されます。オーケストレーターは、できるだけ目的の状態を実現できるように努めます。

Kubernetes では、グローバル サービスは**デーモン セット**とも呼ばれます。

サービス検出

アプリケーション サービスを宣言的な方法で記述する場合、サービスの各インスタンスを実行するクラスター ノードについてはオーケストレーターに指示しません。タスクに対して最適なノードを決定する処理は、オーケストレーターに任せることができます。

もちろん、非常に決定論的な配置ルールを使用するようにオーケストレーターに指示することは技術的に可能ですが、これはアンチパターンであり、推奨されません。

したがって、アプリケーション サービスの個々のインスタンスを配置する場所についてはオーケストレーション エンジンが完全かつ自由に決定するということと、そのインスタンスがクラッシュし、オーケストレーターによって別のノードに再スケジュールされる可能性があることを考えると、ある時点で個々のインスタンスが実行されている場所を追跡する作業には意味がないことがわかります。重要なことではないため、労力をかけるべきではありません。

では、たとえば 2 つのサービス A および B が存在し、サービス A がサービス B に依存している場合はどうでしょうか。このような場合でも、サービス A の特定のインスタンスが、サービス B のインスタンスの場所を特定できるようにする必要はないのでしょうか。

これについては「必要ない」とはっきり言えます。高度に分散されたスケーラブルなアプリケーションでは、このような種類の情報を処理することは望ましくありません。依存している他のサービス インスタンスに到達するために必要な情報の提供は、オーケストレーターに任せるべきです。これは、旧式の電話システムに例えることができます。昔は友人に直接電話をかけることができず、電話会社の電話局でオペレーターに取り次いでもらう必要がありました。今回のケースでオペレーターの役割を果たすのがオーケストレーターであり、サービス A のインスタンスから送られたリクエストをサービス B の利用可能なインスタンスにルーティングします。このプロセス全体が**サービス検出**と呼ばれます。

ルーティング

これまで、分散アプリケーションでは、多数のサービスどうしがやり取りしていることを学習してきました。サービス A とサービス B のやり取りは、データ パケットの交換を通じて行われます。このようなデータ パケットは、何らかの方法でサービス A からサービス B に到達する必要があります。データ パケットを送信元から宛先に到達させる一連のプロセスは、**ルーティング**とも呼ばれます。このようなルーティングのタスクは、アプリケーションの作成者やオペレーターにとってオーケストレーターに任せたい処理といえます。後続の章で説明しますが、ルーティングはさまざまなレベルで発生する可能性があります。これは実社会で例えることができます。たとえば、ある大手企業の従業員が自社のオフィス ビルの 1 つで勤務しているとします。この企業で社内の他の従業員に文書を転送する必要がある場合は、発送書類入れに文書を入れておけば社内の郵便サービスが回収して、同じビル内の郵便局まで運んでくれます。宛先の人物が同じビルで勤務している場合、その文書は直接届けてもらえます。ただし、同じブロック内の別のビルで勤務している場合は、その宛先のビルの郵便局に文書が転送され、そこから社内の郵便サービスを介して宛先の人物に配達されます。また、文書の宛先が別の都市あるいは別の国に存在する同社の支店で勤務する従業員の場合、その文書は UPS などの社外の郵便サービスに転送され、目的の都市や国まで輸送されます。そこからは、やはり社内の郵便サービスが引き継ぎ、宛先の人物に配達されます。

コンテナ内で実行されているアプリケーション サービス間でデータ パケットをルーティングする場合も、同様の処理が行われています。送信元と宛先のコンテナは同じクラスター ノードに配置されている場合がありますが、これは両方の従業員が同じビル内で勤務している状況に相当します。また、送信先のコンテナが別のクラスター ノードで実行されている場合がありますが、これは 2 人の従業員が同じブロック内の別のビルで勤務している状況に相当します。そして最後に挙げた例は、データ パケットがクラスターの外から送信されていて、クラスター内で実行されている宛先コンテナにルーティングする必要があるケースに相当します。

このような状況をはじめとして、あらゆる状況をオーケストレーターによって処理する必要があります。

負荷分散

高可用性の分散アプリケーションでは、すべてのコンポーネントを冗長化する必要があります。つまり、あるインスタンスで障害が発生した場合でもサービス全体として引き続き運用できるように、すべてのアプリケーション サービスを複数のインスタンスで実行する必要があります。

サービスのすべてのインスタンスが実効的な処理を行い、無駄なアイドル状態のインスタンスが発生しないように、サービスに対する要求がすべてのインスタンスに均等に分散されるようにする必要があります。このようにサービス インスタンス間でワークロードを分散させるプロセスは、**負荷分散**と呼ばれます。ワークロードの分散方法に関するアルゴリズムは、さまざまなものが存在します。通常、ロード バランサーは、いわゆるラウンドロビン アルゴリズムを使用して動作します。このアルゴリズムでは、循環アルゴリズムを使用してワークロードがインスタンスに均等に分散されます。

あるサービスから別のサービス、または外部ソースから内部サービスに要求を負荷分散する処理についても、オーケストレーターに任せることができます。

スケーリング

コンテナ化された分散アプリケーションをオーケストレーターが管理するクラスタ内で実行する場合も、予期される、あるいは予期されないワークロードの増加を容易に処理できる方法が必要です。増加したワークロードを処理する場合、通常は、増加した負荷が発生しているサービスについて追加インスタンスをスケジュールするだけで済みます。その後は、より利用可能性が高いインスタンスにワークロードが分散されるように、ロード バランサーが自動的に構成されます。

ただし現実のシナリオでは、ワークロードは時間の経過とともに変化します。たとえば、Amazon などのショッピング サイトでは、自宅でオンライン ショッピングを楽しむ人が多い夜がピークの時間帯となり、負荷が高くなる可能性があります。さらにブラック フライデーなどの特別な日は、極端な負荷がかかることがあります。その一方で、早朝はトラフィックが非常に少なくなる可能性があります。そのため、サービスはスケール アップするだけでなく、ワークロードが低下した場合はスケール ダウンする必要もあります。

また、オーケストレーターは、スケール アップまたはダウンの際に、有意義な方法でインスタンスを分散する必要があります。たとえば、サービスのすべてのインスタンスを同一のクラスター ノード上で分散する処理は避けるべきです。なぜなら、そのノードがダウンするとサービス全体がダウンしてしまうからです。コンテナを配置する役割はオーケストレーターのスケジューラ―が担いますが、同様に、すべてのインスタンスが同じラックのコンピューターに配置されないように処理する必要があります。これはやはり、ラックの電源装置に障害が発生した場合は、サービス全体が影響を受けるためです。さらに、重要なサービスのサービス インスタンスは、停電による影響を避けるためにデータセンターをまたいで分散する必要があります。このようなすべての判断が、オーケストレーターによって行われます。

自己復旧

近年、オーケストレーターは非常に高度になっており、システムが正常な状態を維持できるように多くの処理を実行することができます。オーケストレーターは、クラスター内で実行されているすべてのコンテナを監視しており、クラッシュしているインスタンスや応答しないインスタンスが存在する場合は別のインスタンスに自動的に置き換えられます。オーケストレーターはクラスター ノードの状態を監視し、ノードが正常に動作していない場合、またはダウンしている場合は、スケジューラ—のループから除外します。そのようなノードに配置されたワークロードは、使用可能な別のノードに自動的に再スケジュールされます。

オーケストレーターが現在の状態を監視し、障害を自動的に修復したり目的の状態に調整したりするすべてのアクティビティは、**自己復旧**システムと呼ばれます。ほとんどの場合、ユーザーが障害の修復に積極的に関与する必要はありません。オーケストレーターが自動的にやってくれます。

ただし、オーケストレーターだけでは対処できず、こちらの支援が必要な状況もある程度存在します。たとえば、コンテナ内であるサービス インスタンスが実行されているとします。そのコンテナは稼動中で、外部からは完全に正常な状態に見えます。しかし、内部のアプリケーションは正常な状態ではありません。アプリケーションはクラッシュしていません。単に設計どおりに動作できなくなっているだけです。こちらから何のヒントも与えずに、オーケストレーターがこの事実を知ることができるでしょうか。それは無理です。正常ではない状態または無効な状態の意味は、アプリケーション サービスごとにまったく異なります。つまり、正常な状態はサービスごとに異なるということです。サービスのコンテキストにおいて何をもって正常とするのかは、そのサービスの作成者またはオペレーターにしかわかりません。

オーケストレーターが定義するのは、アプリケーション サービスが自身の状態をオーケストレーターに伝達するための継ぎ目、つまりプローブです。プローブは、大きく次の 2 種類に分けられます。

- サービスは、自身の状態が正常かどうかをオーケストレーターに伝えることができる
- サービスは、自身の状態が準備完了か一時的に利用不可かどうかをオーケストレーターに伝えることができる

サービスが上記のどちらの回答に該当するのかという判断は、完全にサービスに任されています。オーケストレーターが定義するのは、問い合わせる方法だけです。たとえば、HTTP GET 要求を使用するのか、あるいはどのような種類の応答が予期されるのか (OK または NOT OK など) ということです。

上記の正常かどうか、または利用可能かどうかに答えるロジックがサービスに実装されていれば、真の自己復旧システムを備えているといえます。正常でないサービス インスタンスはオーケストレーターが停止して別の正常なインスタンスに置き換えてくれ、一時的に利用できないサービス インスタンスはロード バランサーのラウンドロビンから除外されるためです。

ダウンタイム ゼロのデプロイメント

近年は、更新が必要でミッションクリティカルなアプリケーションについて、完全なダウンタイムの発生を許容することが困難になってきています。そのような事態になればビジネス機会を失うだけでなく、企業の評判も損なわれることになります。そうした不便な状況が発生すれば、アプリケーションを使用するお客様から受け入れてもらえず、すぐに別の製品に替えられてしまうでしょう。さらに、リリース サイクルもますます短くなっています。これまでは 1 年に 1 回または 2 回の新リリースというペースでしたが、現在では多くの企業がアプリケーションを 1 週間に数回、場合によっては 1 日に数回更新しています。

この問題の解決策は、ダウンタイム ゼロのアプリケーション更新戦略を取り入れることです。オーケストレーターは、個々のアプリケーション サービスをパッチ単位で更新できる必要があります。これは、**ローリング アップデート**とも呼ばれます。この方式では、ある特定の時点において、特定のサービスの全インスタンスのうち 1 つまたは少数のインスタンスだけが停止され、新しいバージョンのサービスに置き換えられます。新しいインスタンスが正常に動作し、予期しないエラーや違反動作が発生していない場合にのみ、次のバッチのインスタンスが更新されます。この動作は、すべてのインスタンスが新しいバージョンに置き換えられるまで繰り返されます。何らかの理由で更新が失敗した場合、オーケストレーターは更新されたインスタンスを自動的に以前のバージョンにロールバックします。

ダウンタイム ゼロのその他のデプロイメントとしては、いわゆる**カナリア リリース**や**ブルーグリーン デプロイメント**が挙げられます。どちらの場合も、新しいバージョンのサービスは、現行のアクティブなバージョンと並行してインストールされます。ただし当初は、新しいバージョンには内部的にしかアクセスできません。インストール後はオーケストレーターが新しいバージョンに対してスモーク テストを実行し、新しいバージョンがうまく動作している場合は、現在のブルーから新しいバージョンのグリーンにルーターが切り替わります (ブルーグリーン デプロイメントの場合)。しばらくの間、新しいバージョンのグリーンのサービスは厳重に監視され、すべて正常であれば、古いブルーのバージョンは廃止されます。新しいグリーンのバージョンが予期したとおりに動作しない場合は、ルーターの設定を古いブルーのバージョンに戻すことで、完全なロールバックが実現されます。

カナリア リリースの場合は、トラフィック全体の中で新しいバージョンのサービスについてはわずかな割合 (たとえば 1%) だけをルーティングし、残りの 99% は引き続き古いバージョンにルーティングするようにルーターを設定します。新しいバージョンの動作は厳重に監視され、古いバージョンの動作と比較されます。すべて正常に動作している場合は、新しいサービスのトラフィックの割合をわずかに増加させます。このプロセスは、トラフィックの 100% が新しいサービスにルーティングされるようになるまで繰り返されます。新しいサービスをしばらくの間実行し、すべて正常に動作している場合は、古いサービスが廃止されます。

ほとんどのオーケストレーターは、ダウンタイム ゼロのデプロイメントとして少なくともローリング アップデートをサポートしており、すぐに利用できます。ブルーグリーン デプロイメントとカナリア リリースは、多くの場合、非常に容易に実装できます。

アフィニティと位置認識

アプリケーション サービスによっては、実行するノード上で専用ハードウェアを必要とする場合があります。たとえば、I/O を伴うサービスが高性能な**ソリッドステート ドライブ** (**SSD**) を備えたクラスター ノードを必要とする場合や、一部のサービスが**アクセラレーテッド プロセッシング ユニット** (**APU**) を必要とする場合があります。オーケストレーターでは、アプリケーション サービスごとにノード アフィニティを定義できます。この場合、そのオーケストレーターは、スケジューラーが必要な基準を満たしているクラスター ノード上のコンテナだけをスケジュールするように処理します。

特定のノードに対するアフィニティの定義は避ける必要があります。そのようにすると単一障害点が発生し、高可用性が損なわれる可能性があります。アプリケーション サービスのターゲットとしては、必ず複数のクラスター ノードのセットを定義します。

一部のオーケストレーション エンジンは、いわゆる**位置認識**または**地理認識**もサポートしています。この機能を使用すると、サービスのインスタンスを複数の場所に均等に分散させるようにオーケストレーターに要求することができます。たとえば、`datacenter` というラベルの可能な値として `west`、`center`、および `east` を定義し、それぞれのノードが配置されている地域に相当する値を持つすべてのクラスター ノードに対して、そのラベルを適用することができます。その後は、そのラベルを特定のアプリケーション サービスで地理認識を行うために使用するように、オーケストレーターに指示します。この場合、サービスのレプリカを 9 つ要求すると、オーケストレーターは west (西)、center (中央)、east (東) の 3 か所のデータセンターのノードに、それぞれ 3 つのレプリカがデプロイされるようにします。

地理認識は階層的に定義することもできます。たとえば、データセンターを最上位レベルの識別子として使用し、その下位に可用性ゾーン、さらに下位にサーバー ラックを配置することができます。

地理認識または位置認識は、電源障害またはデータセンターの停電によってダウンタイムが発生する可能性を低減する目的で使用します。アプリケーション インスタンスを複数のサーバー ラックや可用性ゾーン、さらにはデータセンターにまたがって分散している場合、一度にすべてがダウンする可能性は極めて低く、常にいずれかの地域を利用できることが予期されます。

セキュリティ

近年は、IT におけるセキュリティが大きな話題となっています。サイバー戦争は、かつてない規模で行われています。注目を集める多くの企業がハッカー攻撃による被害を受けた経験を持ち、非常にコストのかかる結果を招いています。**最高情報責任者** (**CIO**) や**最高技術責任者** (**CTO**) にとって最悪の事態は、朝起きたときに自分の会社がハッカー攻撃の犠牲者となり、機密情報が盗まれたり損なわれたりしたというニュースを聞かされることでしょう。

このようなセキュリティ上の脅威に対抗するためには、安全なソフトウェア サプライ チェーンを確立し、徹底したセキュリティ防御を実施する必要があります。エンタープライズ グレードのオーケストレーターに期待できるタスクのいくつかを見てみましょう。

安全な通信とノード識別情報の暗号化

オーケストレーターによって管理されているクラスターに求められることは、第一に安全であることです。クラスターに参加させられるのは、信頼できるノードだけです。クラスターに参加する各ノードはノード識別情報が暗号化され、ノード間のすべての通信を暗号化する必要があります。そのためノードでは、**MTLS** (**Mutual Transport Layer Security**) を使用できます。クラスターのノードを相互に認証するには、証明書を使用します。このような証明書は、証明書が流出した場合でもシステムを保護できるように、定期的または要求に応じて自動的にローテーションされます。

クラスター内で発生する通信は、3 種類に分けることができます。それぞれが通信プレーン、つまり**管理**、**コントロール**、および**データ**の各プレーンに対応しています。

- 管理プレーンは、クラスター マネージャーまたはマスターによって使用され、たとえばサービス インスタンスのスケジュール、正常性チェックの実行、またはクラスター内のその他のリソース (データ ボリューム、シークレット、ネットワークなど) の作成および変更に使用されます。
- コントロール プレーンは、クラスターのすべてのノード間で重要な状態情報を交換するために使用されます。この種の情報は、たとえばルーティング目的で使用されるクラスター上でローカル IP テーブルを更新する目的で使用されます。
- データ プレーンでは、アプリケーション サービスが相互に通信し、データを交換します。

通常、オーケストレーターは主に管理プレーンとコントロール プレーンの保護に対処します。データ プレーンの保護はユーザーが担当しますが、オーケストレーターによって作業が容易になる場合があります。

安全なネットワークとネットワーク ポリシー

アプリケーション サービスの実行時は、必ずしもすべてのサービスがクラスター内の他のすべてのサービスと通信するわけではありません。そのため、各サービスを相互にサンドボックスで隔離できる必要があり、どうしても相互に通信する必要があるサービスだけを同じネットワーキングのサンドボックス内で実行するようにします。他のすべてのサービスと、クラスターの外から送られてくるすべてのネットワーク トラフィックは、サンドボックスで隔離されたサービスにアクセスできないようにする必要があります。

このようなネットワーク ベースのサンドボックスは、少なくとも 2 種類の方法で構成できます。1 つは **SDN** (**Software Defined Network**) を使ってアプリケーション サービスをグループ化する方法で、もう 1 つは単一のフラットなネットワークを定義し、ネットワーク ポリシーを使用して、特定のサービスまたはサービスのグループにアクセスできるリソースとアクセスできないリソースを制御する方法です。

ロール (役割) ベースのアクセス制御 (RBAC)

オーケストレーターによって企業が対応できるようになるタスクの中で、セキュリティに次いで重要なものの 1 つが、クラスターとそのリソースに対してロールベースのアクセス権を提供することです。RBAC では、チームに編成されているシステムのサブジェクト、ユーザー、またはユーザーのグループが、システムにアクセスして操作する方法を定義します。これにより、権限のない人物がシステムに何らかの害を及ぼすのを防いだり、システムで利用可能なリソースの中で、そのような人物に知られたり見られたりすることが望ましくないものを見られないようにしたりできます。

一般的な企業では、Development (開発)、QA (品質保証)、Prod (プロダクト) といったユーザー グループを設定し、それぞれのグループに 1 人以上のユーザーを関連付けています。たとえば開発者の John Doe は Development グループのメンバーであり、開発チーム専用のリソースにはアクセスできますが、Prod チームのリソースにはアクセスできません。一方、Prod チームのメンバーである Ann Harbor は、このリソースにアクセスできますが、開発チームのリソースを操作することはできません。

RBAC を実装する方法の 1 つが、**許可**の定義です。許可とは、サブジェクト、ロール、リソース コレクション間の関連付けです。ロールは、リソースに対する一連のアクセス権限で構成されています。たとえば、コンテナを作成、停止、削除、一覧表示、表示する権限や、新しいアプリケーション サービスをデプロイする権限、クラスター ノードを一覧表示したりクラスター ノードの詳細を表示したりする権限などです。

リソース コレクションは、アプリケーション サービスやシークレット、データ ボリューム、コンテナなど、クラスターの論理的に関連するリソースのグループです。

シークレット

私たちの日常生活にはたくさんの秘密、つまりシークレットがあります。シークレットとは、オンライン銀行口座へのアクセスに使用するユーザー名とパスワードの組み合わせや、携帯電話またはジムのロッカーで使用するコードなど、他人に知らせるべきではない情報のことです。

ソフトウェアを記述するときは、シークレットを使う必要がある場合がよくあります。たとえば、何らかの外部サービスにアクセスする必要があるアプリケーションの場合は、証明書を使用してそのアプリケーションを認証する必要があります。また、他の API にアクセスする場合は、サービスを認証および承認するトークンが必要です。これまで開発者は利便性を考慮して、そのような値を単にハードコーディングしたり、外部構成ファイルにクリア テキストで記述したりしていました。この場合、非常に機密性の高い情報が広範な人々からアクセスできる状態になってしまいますが、実際にはそのような情報を見る機会を与えてしまうような状況があってはなりません。

幸い最近のオーケストレーターには、そのような機密情報を非常に安全な方法で処理できる仕組みとして、いわゆるシークレットが用意されています。シークレットは、権限のある人物または信頼された人物が作成できます。このようなシークレットの値は暗号化され、可用性の高いクラスターの状態データベースに格納されます。シークレットは暗号化されているので、保存時も安全です。権限のあるアプリケーション サービスからシークレットが要求されると、そのシークレットは、その特定のサービスのインスタンスを実際に実行しているクラスター ノードだけに転送されます。シークレットの値がノードで格納されることはなく、`tmpfs` という RAM ベースのボリュームのコンテナにマウントされます。シークレットの値をクリア テキストで利用できるのは、それぞれのコンテナ内部だけです。

先ほど説明したように、シークレットは保存時も安全です。サービスから要求されると、クラスター マネージャーまたはマスターがシークレットを復号化し、回線を介してターゲット ノードに送信します。では、シークレットは転送中も安全なのでしょうか。既に説明したように、クラスター ノードは MTLS を使用して通信を行います。そのため、シークレットはクリア テキストで送信されますが、MTLS によってデータ パケットが暗号化されるため、安全は確保されています。つまり、保存時も転送中もシークレットは安全ということになります。シークレットを使用する権限を持つサービスだけが、そのシークレットの値にアクセスできます。

コンテンツの信頼

セキュリティを強化するため、運用環境のクラスターでは信頼できるイメージだけが実行されるようにする必要があります。オーケストレーターによっては、署名済みのイメージだけを実行するようにクラスターを構成できます。コンテンツの信頼とイメージの署名とは、イメージの作成者が想定どおりであること、つまり信頼できる開発者、あるいは信頼できる CI サーバーであることを確認する処理です。さらにコンテンツの信頼では、取得するイメージが最新であり、古く脆弱性のあるイメージではないことも保証する必要があるほか、悪意のあるハッカーが転送中にイメージを侵害することがないようにする必要があります。このような攻撃は、一般的に **中間者 (MITM)** 攻撃と呼ばれます。

ソース側でイメージに署名し、ターゲット側で署名を検証することにより、実行するイメージが侵害されていないことを保証できます。

稼働時間の巻き戻し

セキュリティの観点から説明する最後の項目は、稼働時間の巻き戻しです。何を意味する言葉でしょうか。たとえば、セキュリティが保護された運用環境のクラスターを構成したとします。このクラスターでは、いくつかのミッションクリティカルなアプリケーションを実行しています。ある日、ハッカーがソフトウェア スタックにセキュリティ ホールを発見し、クラスター ノードの 1 つに対するルート アクセス権を取得してしまいました。それだけで既に重大な問題ですが、さらに悪いことに、そのハッカーはルートを取得したノード上で自身の存在を隠すことができたため、そのノードを起点としてさらに別のクラスターまで攻撃したのです。

Linux など、Unix タイプのオペレーティング システムにおけるルート アクセス権とは、そのシステム上で何でもできることを意味します。ユーザーが所有できる最上位レベルのアクセス権です。Windows では、管理者に相当するロールです。

ただし、コンテナが一時的なものであり、クラスター ノードが迅速に (通常は数分で完全自動で) プロビジョニングされるという事実を利用すれば、結果を変えることができます。一定の稼働時間、たとえば 1 日経過したら各クラスターを終了することができるからです。オーケストレーターにはノードをドレインし、クラスターから除外するように指示します。クラスターがノードから除外されると、そのノードは解体され、新しくプロビジョニングされたノードに置き換えられます。

これによりハッカーは攻撃の起点を失うため、問題が解消されます。この概念はまだ広範に導入されているわけではありませんが、セキュリティ強化に向けた大きな一歩になるものと思われます。また、この分野で働くエンジニアと話す限り、実装は難しくありません。

イントロスペクション

ここまで、オーケストレーターが役割を担うさまざまなタスクについて説明してきました。そのようなタスクは、完全に自律的な方法で実行することができます。しかし、クラスター上で現在実行されている内容や個々のアプリケーションの状態の中には、人間のオペレーターが確認して分析できる必要があるものもあります。つまり、このような対象のすべてについてイントロスペクションを実行できる必要があるため、オーケストレーターは、利用しやすく理解しやすい形式で重要な情報を提示する必要があります。

オーケストレーターは、すべてのクラスター ノードからシステム メトリックを収集し、オペレーターがアクセスできるようにする必要があります。メトリックとしては、CPU、メモリ、ディスクの使用量、ネットワーク帯域幅の消費量などが挙げられます。情報は、ノード単位でも集約された形式でも簡単に利用できなければなりません。

また、サービス インスタンスまたはコンテナによって生成されたログにもアクセスできるようにする必要があります。さらに、適切な権限を持つユーザーには、各コンテナに対する `exec` アクセス権を提供する必要があります。コンテナに対する `exec` アクセス権を持つユーザーは、正常に動作していないコンテナをデバッグできます。

高度に分散されたアプリケーションでは、アプリケーションに対する各リクエストの処理が完了するまでに多数のサービスを経由することになるため、リクエストの追跡は非常に重要なタスクです。オーケストレーターは、追跡戦略の実装を支援してくれたり、戦略への準拠に対する何らかの優れたガイドラインを提供してくれたりすることが理想的です。

最終的には、収集したすべてのメトリックやログ情報および追跡情報をグラフィカルな表現で利用できるようにすることで、人間のオペレータが効率的にシステムを監視することができます。そこで必要になるのがダッシュボードです。最も重要なシステム パラメーターをグラフィカルに表現した基本的なダッシュボードを最低でも数種類は提供していなければ、優れたオーケストレーターとはいえません。

ただし人間のオペレーターの全員が、イントロスペクションを利用したいと考えるわけではありません。外部システムとオーケストレーターを接続して、そのような情報を利用できるようにする必要もあります。クラスターの状態やメトリック、ログなどのデータに外部のシステムからアクセスし、その情報を使用して自動処理を実行するには API が必要です。たとえば、システムが何らかのしきい値を超過した場合は、ポケットベルや電話のアラートを作成したり電子メールを送信したりできるほか、警報サイレンをトリガーすることもできます。

一般的なオーケストレーターの概要

本書の執筆時点で、多数のオーケストレーション エンジンがリリースされ、使用されています。ただし、広く普及しているものは数種類に限られます。その中で覇権を握っているものは間違いなく Kubernetes です。そこから大きく離された 2 番手が Docker 自身が提供する SwarmKit で、その後に Microsoft **Azure Kubernetes Service** (**AKS**)、Apache Mesos、AWS **Elastic Container Service** (**ECS**) などが続きます。

Kubernetes

Kubernetes はもともと Google が設計したもので、その後 **Cloud Native Computing Foundation** (**CNCF**) に寄付されました。Kubernetes は、長年にわたり非常に大規模なコンテナを運用してきた Google 独自の Borg システムをモデルにしています。Kubernetes は、Borg から学んだすべての教訓を組み込んだシステムを設計段階から完全に作り直すという Google の試みによって生まれました。

独自の技術として開発が進められた Borg とは対照的に、Kubernetes は早期の段階からオープンソース化されました。そのため社外から膨大な数の協力者が参加することになり、わずか数年で Kubernetes を中心とした大規模なエコシステムが展開されることになったため、Google によるこの選択は非常に賢明であったといえます。コンテナ オーケストレーションの分野では Kubernetes がコミュニティから最も支持されているオーケストレーターであることに疑いの余地はありません。これまで、ここまで大きな盛り上がりを生み出し、協力者またはアーリー アダプターとしてプロジェクトの成功に有意義な形で積極的に貢献してくれる才能ある多数の人々を引き付けたオーケストレーターはありませんでした。

このような点で、コンテナ オーケストレーションの分野における Kubernetes は、サーバー オペレーティング システムの分野における Linux と非常によく似ています。Linux は、サーバー オペレーティング システムのデファクト スタンダードとなっています。マイクロソフト、IBM、Amazon、RedHat、さらには Docker まで、すべての関連企業が Kubernetes を取り入れています。

Kubernetes は、開発当初から大規模なスケーラビリティを念頭に置いて設計されていることも間違いありません。Google Borg から強い影響を受けて設計されています。

Kubernetes の否定的な側面としては、少なくとも本書の執筆時点ではセットアップと管理が複雑であるという点が挙げられます。初めて使う人にとっては敷居が高く、最初の一歩が険しい道のりとなっています。ただし、しばらく使っていくうちに、すべてが理に適っていることを理解できるでしょう。全体的に深く考えられた設計となっており、非常にうまく機能します。

Kubernetes の最新リリースは 1.10 で、**一般提供** (**GA**) は 2018 年 3 月でした。このリリースでは、Docker Swarm などの他のオーケストレーターと比較して、初期の欠点の大部分が解決されています。たとえばセキュリティと機密性は、補足的な要素ではなく、システムの不可欠な要素となりました。

新しい機能は、非常に速いペースで実装が進められています。新しいリリースは約 3 か月ごと、正確には約 100 日ごとに発生しています。新機能のほとんどは需要主導型です。つまり、Kubernetes を使用してミッションクリティカルなアプリケーションのオーケストレーションを行っている企業は、ニーズを表明することができます。Kubernetes がエンタープライズ対応になっているのはそのためです。このオーケストレーターがスタートアップ企業向けで、リスクを恐れる企業には向いていないという考えは間違いです。実際はその逆なのです。これには根拠があります。たとえば、マイクロソフトや Docker、RedHat といった、大手企業をクライアントに抱える各企業も Kubernetes を完全に取り入れ、自社のエンタープライズ製品に統合して使用されている場合は、エンタープライズ グレードのサポートを提供しているという事実があるからです。

Kubernetes は、Linux コンテナと Windows コンテナの両方をサポートしています。

Docker Swarm

コンテナを普及させ、コモディティ化させたのが Docker であることはよく知られています。Docker はコンテナを発明したわけではありませんが、Docker によって多くの人々がコンテナを利用するようになりました。特に、無料のイメージ レジストリである Docker Hub が大きく貢献しています。当初、Docker は主に開発者と開発ライフサイクルに重点を置いていました。しかし、コンテナを使い始め、その有用性に気付いた企業は、新しいアプリケーションの開発やテストだけでなく、運用環境でのアプリケーションの実行にもコンテナを使用することを望むようになりました。

当初、Docker はその分野に対応したサービスを提供していなかったため、他の企業がその要求に応える形でユーザーへの支援を開始することになります。しかし、シンプルで強力なオーケストレーターに対する大きな需要があることを Docker が認識するまでに、それほど時間はかかりませんでした。Docker が最初の試みとして提供したのは、クラシック Swarm と呼ばれる製品でした。スタンドアロンの製品であり、Docker ホスト マシンのクラスターを作成できます。このクラスターは、可用性が高く自己復旧に対応した方法でコンテナ化されたアプリケーションを実行および拡張する目的で使用できます。

ただし、クラシックの Docker Swarm はセットアップが容易ではありませんでした。多くの複雑な手作業が必要だったのです。製品は評価されていたものの、利用者はその複雑さに手を焼いていました。Docker が改善できると判断したのはそのためです。設計段階から見直し、生まれたのが SwarmKit です。SwarmKit はシアトルで開催された DockerCon 2016 で発表され、最新バージョンの Docker エンジンでは不可欠な要素となりました。この点は現在でも変わっていません。したがって、Docker ホストをインストールすると、自動的に SwarmKit も利用できるようになります。

SwarmKit はシンプルさとセキュリティを念頭に置いて設計されています。Swarm が絶対的な指針として掲げたことは、セットアップに伴う労力がほとんどないことと、特に調整しなくても高いセキュリティが保たれていることです。これは現在でも変わっていません。Docker Swarm は、最小特権を前提として動作します。

可用性の高い Docker Swarm を一通りインストールする作業は、クラスター内の最初のノードで `docker swarm init` コマンドを実行するだけです。このノードがいわゆるリーダーとなり、その後は他のすべてのノードで `docker swarm join <join トークン>` を実行します。`join` トークンは、初期化時にリーダーによって生成されます。最大 10 ノードで構成された Swarm をセットアップするプロセス全体は 5 分もかからずに完了します。自動化されている場合は、さらに短時間で完了します。

先ほど触れたように、Docker が SwarmKit を設計および開発したときに最優先としたことはセキュリティでした。コンテナは、Linux カーネルの名前空間と cgroups に加え、Linux syscall のホワイトリスト (seccomp) および各種 Linux 機能と **Linux セキュリティ モジュール** (**LSM**) のサポートに依存してセキュリティを提供しています。そして SwarmKit では、MTLS および保存時も転送時も暗号化されるシークレットが追加されています。さらに、Swarm ではいわゆる**コンテナ ネットワーク モデル** (**CNM**) を定義しており、Swarm で実行されているアプリケーション サービスのサンドボックス化を提供する SDN が可能になっています。

Docker SwarmKit は、Linux コンテナと Windows コンテナの両方をサポートしています。

Microsoft Azure Kubernetes Service (AKS)

AKS とは完全にホストされ、可用性が高く、スケーラブルでフォールト トレランス性を備えた、マイクロソフトが提供する Kubernetes クラスターです。Kubernetes クラスターのプロビジョニングと管理に伴う労力を引き受けてくれるため、コンテナ化されたアプリケーションのデプロイと実行に注力することができます。AKS を使用すると、運用環境に対応したクラスターを文字どおり数分でプロビジョニングできます。さらに、そのようなクラスター上で実行されているアプリケーションは、Azure が提供する豊富な機能を持つサービスのエコシステム (Log Analytics や ID 管理など) に容易に搭載することができます。

最新バージョンの Kubernetes をベースとした強力なオーケストレーション サービスであり、既に Azure エコシステムに多額の投資を行っている場合は合理的な選択肢といえます。各 AKS クラスターは、Azure ポータル、Azure Resource Manager テンプレート、または Azure CLI を通じて管理できます。アプリケーションは、よく知られている Kubernetes の CLI である `kubectl` を使用してデプロイおよび管理します。

マイクロソフトは、自身の言葉で次のように表明しています。

> AKS を使用すれば、コンテナ オーケストレーションの専門知識がない方でも、コンテナ化されたアプリケーションを迅速かつ容易にデプロイして管理できます。また、アプリケーションをオフラインにすることなく、オンデマンドでリソースをプロビジョニング、アップグレード、スケーリングすることで、継続的な運用とメンテナンスの負担も解消します。

Apache Mesos および Marathon

Apache **Mesos** はオープン ソースのプロジェクトであり、もともとはサーバーまたはノードのクラスターが、外部からは単一の大規模なサーバーに見えるようにする目的で設計されました。Mesos は、コンピューター クラスターの管理をシンプルにするソフトウェアです。Mesos を使用すれば個々のサーバーに注意を払う必要がなくなり、自由に使える膨大なリソースのプールと考えることができるようになります。つまり、クラスター内の全ノードの全リソースが集約されるということです。

IT の世界では、Mesos は他のオーケストレーターと比較して既にかなり古くなっています。最初に公開されたのは 2009 年でしたが、その当時はまだ Docker が存在していなかったため、コンテナを運用する目的で設計されていませんでした。Docker がコンテナを処理する場合と同様に、Mesos は Linux の cgroups を使用して、個々のアプリケーションまたはサービスの CPU やメモリ、ディスク I/O を分離します。

Mesos は実際には、その上に構築された他の有用なサービスの基盤となるインフラです。特にコンテナの観点から考えると、**Marathon** は重要です。Marathon は、Mesos 上で動作するコンテナ オーケストレーターであり、何千という規模のノード数まで拡張できます。

Marathon は Docker や独自の Mesos コンテナなど、複数のコンテナ ランタイムをサポートしています。ステートレスだけでなく、PostgreSQL や MongoDB などのデータベースをはじめとするステートフルなアプリケーション サービスもサポートしています。Kubernetes や Docker SwarmKit と同様に、この章の前半で説明した高可用性、正常性チェック、サービス検出、負荷分散、位置認識などの機能をサポートしています。

Mesos と Marathon の一部はどちらかというと成熟したプロジェクトですが、その適用範囲は比較的限られています。最もよく利用されているのがビッグ データの分野、つまり Spark や Hadoop などのデータ処理サービスの運用と思われます。

Amazon ECS

求めているのがシンプルなオーケストレーターで、AWS エコシステムに既に多額の投資を行っている場合は、Amazon の ECS が最適な選択肢かもしれません。ただし ECS には非常に重大な制限があります。それは、このコンテナ オーケストレーターを購入した場合は、AWS 以外の選択肢がなくなるということです。ECS 上で動作するアプリケーションを別のプラットフォームやクラウドに移植することは容易ではありません。

Amazon は ECS サービスを、Docker コンテナをクラスター上で容易に実行、停止、管理することができる高度にスケーラブルで高速なコンテナ管理サービスとして推進しています。コンテナの実行だけでなく、ECS では、コンテナ内で実行されているアプリケーション サービスから他のさまざまな AWS サービスに直接アクセスできます。このような多数の一般的な AWS サービスとの緊密かつシームレスな統合により、堅牢でスケーラビリティの高い環境で、コンテナ化されたアプリケーションを容易に稼動させることができる方法を求めるユーザーにとって、ECS は魅力的な選択肢となっています。Amazon は独自のイメージ レジストリも提供しています。

AWS ECS を使用すると、基盤となるインフラを Fargate を使用して完全に管理できるため、コンテナ化されたアプリケーションのデプロイに専念できます。ノードのクラスターを作成および管理する方法に気を配る必要はありません。ECS は、Linux コンテナと Windows コンテナの両方をサポートしています。

要約すると、ECS は使いやすく、スケーラビリティが高く、他の一般的な AWS サービスとの統合性に優れていますが、Kubernetes や Docker SwarmKit ほど強力ではなく、Amazon AWS でのみ利用できます。

まとめ

この章では、まずオーケストレーターが必要な理由と概念的にどのように動作するのかについて説明しました。また、執筆時点で最もよく使われているオーケストレーターを示し、各種オーケストレーターの主な共通点と相違点について説明しました。

次の章では、そのようなコンテナ オーケストレーターの中で最も有名な Kubernetes を紹介します。分散型で回復性が高く、堅牢性と可用性に優れたアプリケーションをオンプレミスまたはクラウド内のクラスターでデプロイおよび実行するために Kubernetes が使用するすべての概念とオブジェクトについて詳しく説明します。

質問

学習の進捗状況を評価するために、以下の質問にお答えください。

1. なぜオーケストレーターが必要なのでしょうか。理由を 2 ～ 3 つ挙げてください。
2. オーケストレーターの典型的な役割を 3 ～ 4 つ挙げてください。
3. 少なくとも 2 つのコンテナ オーケストレーターと、それを支える主要なスポンサーを挙げてください。

参考情報

以下のリンクでは、オーケストレーション関連のトピックについてより詳しいインサイトを提供しています。

- Kubernetes - 運用環境グレードのオーケストレーション (`https://kubernetes.io/`)
- Docker Swarm モードの概要 (`https://docs.docker.com/engine/swarm/`)
- Mesosphere - コンテナ オーケストレーション サービス (`http://bit.ly/2GMpko3`)
- コンテナとオーケストレーションの説明 (`http://bit.ly/2DFoQgx`)
- Azure Kubernetes Service (AKS) (`https://bit.ly/2MECYzY`)

10
Kubernetes を使用したコンテナ化アプリケーションのオーケストレーション

前章では、オーケストレーターについて説明しました。オーケストラの指揮者と同様に、オーケストレーターは、コンテナ化されたすべてのアプリケーション サービスがうまく連携し、調和しながら共通の目標に向かっていることを確認します。オーケストレーターの担う責任は重大で、これについてはこれまでに詳しく説明した通りです。また、市場で最も重要なコンテナ オーケストレーターの概要についても簡単に取り上げました。

この章では、**Kubernetes** について解説します。現在、コンテナ オーケストレーションの分野を明らかにリードしているのが、Kubernetes です。まず最初に、Kubernetes クラスターのアーキテクチャの概要を紹介し、次に Kubernetes がコンテナ化アプリケーションを定義して実行するために使用する主なオブジェクトについて説明します。

この章では、次のトピックを取り上げます。

- アーキテクチャ
- Kubernetes マスター
- クラスター ノード
- MiniKube の概要
- Docker for Mac および Docker for Windows での Kubernetes サポート
- ポッド
- Kubernetes ReplicaSet
- Kubernetes デプロイメント
- Kubernetes サービス
- コンテキストベースのルーティング

この章を終了すると、次のことができるようになります。

- メモに Kubernetes クラスターのハイレベル アーキテクチャのドラフトを作成する
- Kubernetes ポッドの主な特徴を 3 ～ 4 つ説明する
- Kubernetes ReplicaSet の役割を 2 ～ 3 つの短い文で説明する
- Kubernetes サービスの主な責任を 2 ～ 3 つ説明する
- Minikube でポッドを作成する
- Docker for Mac または Docker for Windows を構成して Kubernetes をオーケストレーターとして使用する
- Docker for Mac または Docker for Windows でデプロイメントを作成する
- Kubernetes サービスを作成してアプリケーション サービスを内部的 (または外部的に) にクラスターに公開する

技術的要件

コード ファイルへのリンクは、`https://github.com/appswithdockerandkubernetes/labs/tree/master/ch10` をご覧ください。

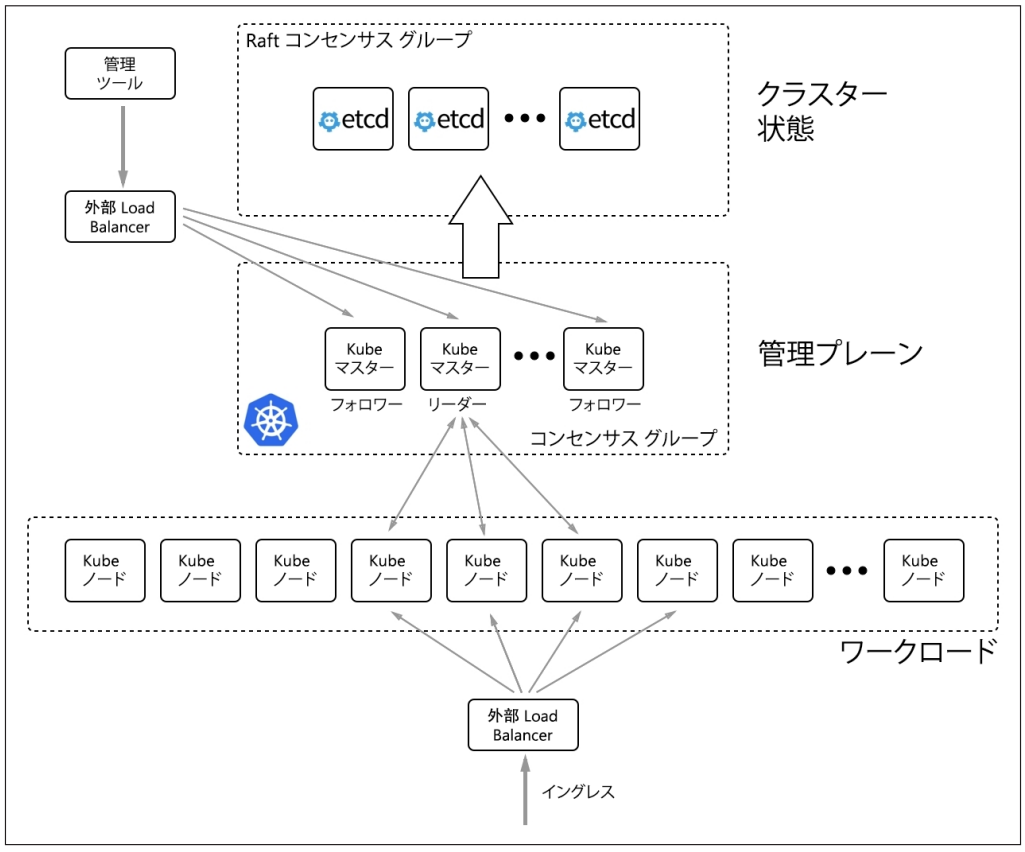
アーキテクチャ

Kubernetes クラスターは、一連のサーバーで構成されています。これらのサーバーは、VM サーバーまたは物理サーバーで、後者は**ベア メタル**とも呼ばれます。クラスターの各メンバーは、Kubernetes マスターもしくは (ワーカー) ノードの 2 つの役割のいずれかを持つことができます。前者はクラスターの管理に使用され、後者はアプリケーションのワークロードを実行します。ワーカーをあえて括弧書きにしているのは、Kubernetes 用語では、ノードはアプリケーションのワークロードを実行するサーバーにしか用いないためです。Docker 用語と Swarm でそれに相当するのがワーカー ノードです。ワーカー ノードの概念は、シンプルなノードよりもサーバーの役割をより的確に説明すると考えられます。

クラスター内には、わずかな奇数のマスターと必要な数のワーカー ノードがあります。小規模のクラスターにはワーカー ノードが少ししかないかもしれませんが、より現実的にはクラスターには数十から数百のワーカー ノードが含まれます。技術的には、クラスターが含むことができるワーカー ノード数に制限はありませんが、実際には、何千ものノードを処理すると管理オペレーションに大幅なスローダウンが発生する場合があります。クラスターのすべてのメンバーは、**アンダーレイ ネットワーク**と呼ばれる物理ネットワークによって接続される必要があります、

Kubernetes はクラスター全体に対してフラットなネットワークを 1 つ定義します。Kubernetes はすぐに利用できるネットワーキングの実装は提供しておらず、サードパーティのプラグインに依存しています。Kubernetes は **コンテナ ネットワーク インターフェイス** (**CNI**) のみを定義し、実装は他に委任します。CNI は実にシンプルです。基本的には、クラスター内で実行されている各ポッドは、ポッド間の**ネットワーク アドレス変換** (**NAT**) なく、同じくクラスター内で実行されている他のどのポッドにも到達できる必要があることを宣言しています。クラスター ノードとポッド間にも同じことが当てはまるはずです。つまり、クラスター ノード上で直接実行されているアプリケーションやデーモンはクラスター内の各ポッドに、またはその逆にも到達できる必要があるということです。

次図では、Kubernetes クラスターのハイレベル アーキテクチャについて説明します。



Kubernetes のハイレベル アーキテクチャ図

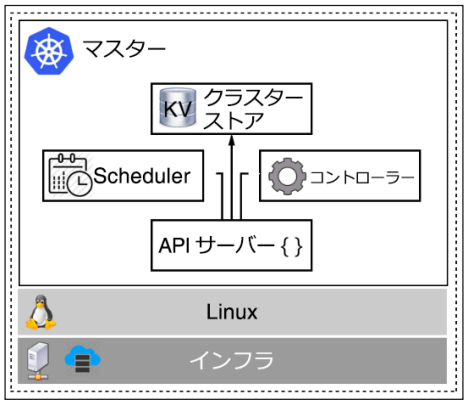
上図の説明は、次のとおりです。

- 上部中央にあるのは **etcd** ノードのクラスターです。etcd は分散型キー値ストアで、Kubernetes クラスターでは、クラスターのすべての状態を保存するのに使用します。etcd ノードの数は、**Raft** コンセンサス プロトコルにより奇数と指定されており、これを使用してノードの調整を行います。クラスターの状態とは、クラスター内で実行中のアプリケーションが生成または消費するデータではなく、クラスターのトポロジーに関するあらゆる情報、実行中のサービス、ネットワーク設定、使用するシークレットなどが該当します。ただし、この etcd クラスターはそのクラスターにとって非常にミッション クリティカルなものであるため、本番環境や高い可用性が求められる環境では、決して 1 つの etcd サーバーのみ実行すべきではありません。
- etcd ノードと同じく、コンセンサス グループも形成する Kubernetes マスター ノードのクラスターがあります。マスター ノード数も奇数であることが必要です。単一のマスターでクラスターを実行することはできますが、本番環境やミッションクリティカルなシステムでは実行すべきではありません。マスター ノードは必ず 3 つ以上必要です。マスター ノードはクラスター全体の管理に使用されるため、管理プレーンについても説明します。マスター ノードは、etcd クラスターをバッキング ストアとして使用します。`https://admin.example.com` のようによく知られた**完全修飾ドメイン名 (FQDN)** の **Load Balancer (LB)** をマスター ノードの前に配置することが推奨されます。Kubernetes クラスターの管理に使われるすべてのツールは、いずれかのマスター ノードのパブリック IP アドレスではなく、この LB から Kubernetes にアクセスする必要があります。これは、上図の左上に示されています。
- 図の一番下の方にあるのがワーカー ノードのクラスターです。ノード数は最少が 1 つで、上限はありません。Kubernetes のマスター ノードとワーカー ノードは相互に通信します。これは双方向通信で、Docker Swarm の場合とは異なります。Docker Swarm では、マネージャー ノードだけがワーカー ノードと通信し、その逆のパターンは絶対にありません。クラスター内で実行されているアプリケーションにアクセスするすべてのイングレス トラフィックは、別のロード バランサーを経由する必要があります。これは、アプリケーションのロード バランサーかリバース プロキシです。外部トラフィックがいずれかのワーカー ノードに直接アクセスするのは好ましくありません。

Kubernetes クラスターのハイレベル アーキテクチャについて簡単に説明しましたが、ここからもう少し深く掘り下げ、さらには Kubernetes マスターとワーカー ノードについても見ていきます。

Kubernetes マスター ノード

Kubernetes マスター ノードは、Kubernetes クラスターの管理に使用されます。次は、このようなマスターをハイレベルな図で示したものです。



Kubernetes マスター

上図の一番下にあるインフラは、オンプレミスの VM、クラウド、またはサーバー (通称**ベア メタル**) にあります。現在、Kubernetes マスターは Linux 上でのみ動作し、RHEL、CentOS、Ubuntu など最も一般的な Linux ディストリビューションがサポートされています。この Linux マシンでは、少なくとも次の 4 つの Kubernetes サービスを実行しています。

- **API サーバー** : Kubernetes へのゲートウェイ。クラスター内のリソースのリスト、作成、変更、削除を求めるすべてのリクエストは、このサービスを経由する必要があります。これは、`kubectl` などのツールがクラスターやクラスター内のアプリケーションの管理に使用する REST インターフェイスを公開します。
- **コントローラー** : より厳密にはコントローラー マネージャー。API サーバーを介してクラスターの状態を確認し、現在の状態や有効状態を目的の状態に近づけるために変更を行うコントロール ループです。
- **Scheduler**: リソース要件やポリシー、サービス品質要件といったさまざまな境界条件を考慮して、ワーカー ノード上のポッドをスケジュールするために最大限の能力を発揮するサービスです。
- **クラスター ストア** : etcd のインスタンスで、クラスターの状態に関するすべての情報を保存するために使用されます。

より正確にいえば、クラスター ストアとして使用される etcd は、必ずしも他の Kubernetes サービスと同じノードにインストールする必要はありません。前セクションのアーキテクチャ図にもあるように、Kudnetes クラスターが etcd サーバーのスタンドアロン クラスターを使用するよう構成されていることがありますが、どのバリアントを使用するかは高度な管理上の決定であり、本書の範囲外です。

少なくとも 1 つのマスターが必要ですが、高可用性を達成するには 3 つ以上のマスター ノードが必要です。これは、Docker Swarm のマネージャー ノードについて学んだことと非常によく似ており、この点においては、Kubernetes マスターは Swarm マネージャー ノードと同等です。

Kubernetes マスターは決してアプリケーションのワークロードを実行しません。その唯一の目的は、クラスターを管理することです。Kubernetes マスターは、**Raft コンセンサス グループ**を作成します。Raft プロトコルは、メンバーのグループが意思決定を下さなければならない状況で使用される標準プロトコルで、MongoDB、Docker SwarmKit、Kubernetes など、一般的な数多くのソフトウェア製品で使用されています。Raft プロトコルの詳細については、「参考情報」セクションのリンクを参照してください。

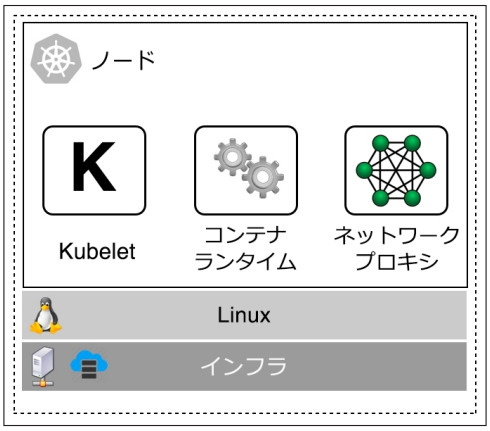
前セクションで述べたように、Kubernetes クラスターの状態は etcd に格納されています。Kubernetes クラスターに高い可用性が求められる場合は、etcd も高可用性モードで設定する必要があり、これは通常、少なくとも 3 つの etcd インスタンスが異なるノードで動作していることを意味します。

クラスター全体の状態は etcd に格納されていることをもう一度言っておきましょう。これには、全クラスター ノード、全レプリカ セット、デプロイメント、シークレット、ネットワーク ポリシー、ルーティング情報などに関するあらゆる情報が含まれます。そのため、このキーと値のストアに堅牢なバックアップ戦略を策定しておくことが極めて重要です。

次に、クラスターの実際のワークロードを実行するノードを見てみましょう。

クラスター ノード

クラスター ノードは、Kubernetes がアプリケーションのワークロードをスケジュールするノードで、クラスターで中心的な役割を果たします。1 つの Kubernetes クラスターに、わずか数個から数千にも及ぶクラスター ノードを含むことができます。Kubernetes は、高いスケーラビリティを考慮してゼロから構築されており、数万ものコンテナを何年も稼働させてきた Google Borg をモデルにしていることを忘れないでください。



Kubernetes ワーカー ノード

ワーカー ノードは、VM やベア メタル、オンプレミス、またはクラウド上で実行できます。ワーカー ノードはもともと、Linux 上でのみ構成できましたが、Kubernetes のバージョン 1.10 以降、Windows Server 2010 でも実行できるようになりました。クラスター間で Linux と Windows のワーカー ノードが混在しても何の問題もありません。

各ノードには、次のように実行する必要のあるサービスが 3 つあります。

- **Kubelet**: これが何よりも優先されるサービスで、**プライマリ ノード エージェント**と呼ばれます。Kubelet サービスは、ポッドの仕様を使用して、対応するポッドの全コンテナが実行中でありかつ健全であることを確認します。ポッドの仕様は、**YAML** ファイルまたは JSON 形式で書かれており、宣言的にポッドを記述します。ポッドの詳細については、次のセクションで説明します。PodSpec は主に API サーバーを介して Kubelet に提供されます。
- **コンテナ ランタイム**: 各ワーカー ノードに存在する必要がある 2 つ目のサービスはコンテナ ランタイムです。バージョン 1.9 以来、Kubernetes は `containerd` をコンテナ ランタイムとしてデフォルトで使用しています。それ以前は、Docker デーモンを使用しており、rkt や CRI-O など他のコンテナ ランタイムも使用できます。コンテナ ランタイムは、ポッドの各コンテナの管理と実行を行います。
- **kube-proxy**: 最後は `kube-proxy` です。デーモンとして実行される kube-proxy は、特定のノード上で実行されている全アプリケーション サービスのシンプルなネットワーク プロキシとロード バランサです。

ここまで Kubernetes のアーキテクチャとマスター ノードおよびワーカー ノードについて学習しました。次は、Kubernetes を対象としたアプリケーションの開発に活用できるツールをご紹介します。

Minikube の概要

Minikube は、VirtualBox または Hyper-V (その他のハイパーバイザーにも対応) に単一ノードの Kubernetes クラスターを作成し、コンテナ化アプリケーションの開発中に使用できるようにするツールです。「第 2 章、作業環境の設定」では、Mac や Windows のノート PC に Minikube およびツール `kubectl` をインストールする方法について説明しました。前述したように、Minikube は単一ノードの Kubernetes クラスターであるため、ノードは同時に Kubernetes マスターでありワーカー ノードとなります。

Minikube が次のコマンドで実行されていることを確認しましょう。

```
$ minikube start
```

Minikube の準備ができたら、`kubectl` を使って単一ノードのクラスターにアクセスします。すると、次のようなスクリーンショットが表示されるはずです。

```
$ kubectl get nodes
NAME       STATUS    ROLES     AGE       VERSION
minikube   Ready     <none>    2d        v1.9.0
$
```

Minikube の全ノードのリスト

前述のように、単一のノード クラスターには `minikube` と呼ばれるノードが含まれます。`ROLES` の列に `<none>` の値がありますが混乱しないでください。このノードはワーカー ノードであり同時にマスター ノードでもあります。

では、このクラスターにポッドをデプロイしてみましょう。ポッドについてはこの章でさらに詳しく説明しますので、今の時点で正確に知らなくても心配する必要はありません。

`labs` フォルダの `ch10` サブフォルダにある `sample-pod.yaml` ファイルを使って該当のポッドを作成します。コンテンツは以下のとおりです。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
```

```
    image: nginx:alpine
    ports:
    - containerPort: 80
    - containerPort: 443
```

kubectl と呼ばれる Kubernetes CLI を使って、このポッドをデプロイしてみましょう。

```
$ kubectl create -f sample-pod.yaml
pod "nginx" created
```

全ポッドをリストにすると、次のようになります。

```
$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          51s
```

このポッドにアクセスするには、サービスを作成する必要があります。下記のコンテンツを含む、sample-service.yaml ファイルを使ってみましょう。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: LoadBalancer
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    app: nginx
```

先ほども言いましたが、サービスの詳細については後ほど詳しく説明しますので、この時点で正確に知っていなくても大丈夫です。ここではとりあえず、このサービスを作成しましょう。

```
$ kubectl create -f sample-service.yaml
```

curl を使ってサービスにアクセスします。

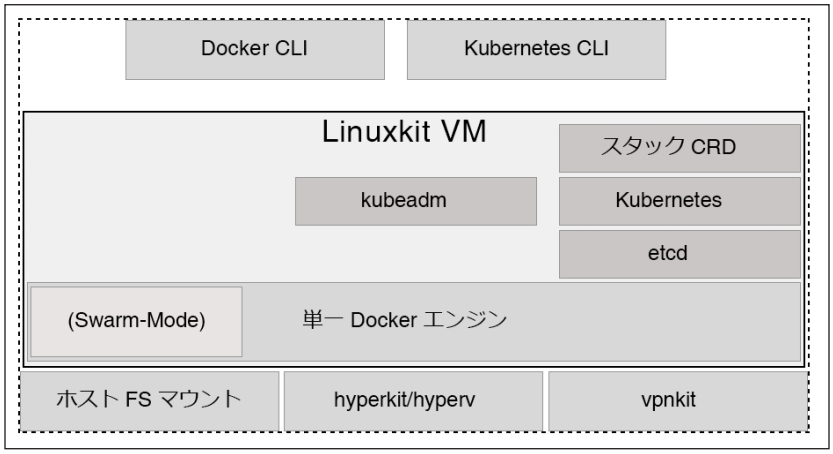
```
$ curl -4 http://localhost
```

すると、Nginx ウェルカム ページが送信されるはずです。先に進む前に、先ほど作成した 2 つのオブジェクトを削除します。

```
$ kubectl delete po/nginx
$ kubectl delete svc/nginx-service
```

Docker for Desktop での Kubernetes サポート

バージョン 18.01-ce から、Docker for Mac および Docker for Windows で、Kubernetes のサポートをすぐに利用できるようになりました。コンテナ化アプリケーションを Kubernetes にデプロイする開発者は、このオーケストレーターを SwarmKit の代わりに使用することができます。Kubernetes のサポートはデフォルトでは無効になっているため、設定で有効にしてください。Kubernetes の初回有効時、Docker for Mac または Docker for Windows で、単一ノードの Kubernetes クラスターを作成するのに必要な全コンポーネントをダウンロードする必要があります。同じく単一ノードのクラスターである Minikube の場合とは異なり、Docker ツールが使用するバージョンでは、Kubernetes 全コンポーネントのコンテナ化されたバージョンを使用します。



Docker for Mac および Docker for Windows での Kubernetes サポート

上図は、Kubernetes サポートが Docker for Mac および Docker for Windows に追加される仕組みを大まかに示したものです。Docker for Mac はハイパーキットを使って LinuxKit ベースの VM を実行し、Docker for Windows は Hyper-V を使って同じく VM を実行します。VM 内には、Docker エンジンがインストールされています。エンジンの一部は SwarmKit で、これで Swarm モードが有効になります。Docker for Mac または Docker for Windows は、kubeadm ツールを使って Kubernetes をその VM 内で設定および構成します。ここで、特筆すべき事実を 3 つご紹介します。

- Kubernetes はそのクラスター状態を etcd に格納するため、etcd はこの VM 上で実行される。
- Kubernetes を構成する全サービスを利用できる。

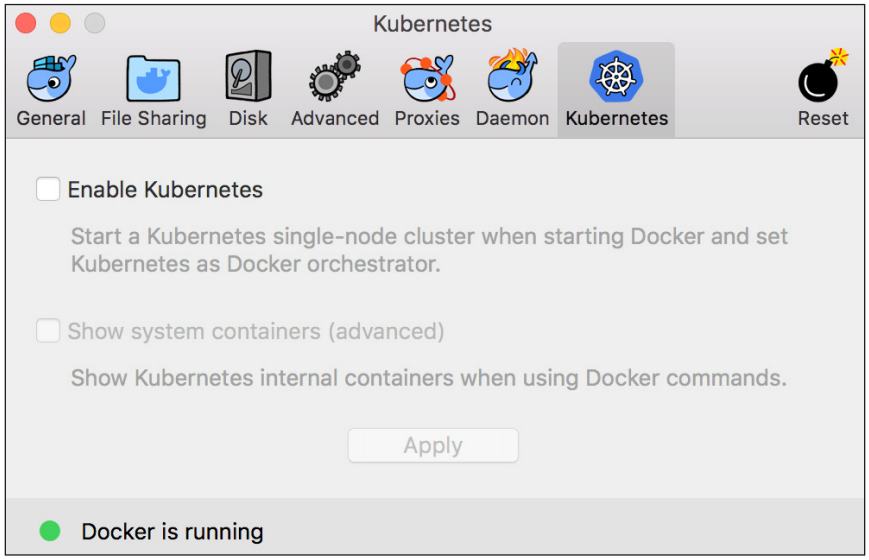
- Docker CLI 上の Docker スタックのデプロイメントをサポートするいくつかのサービスが Kubernetes には統合されている。このサービスは、Kubernetes の公式ディストリビューションの一部ではなく、Docker 固有のもの。

Kubernetes の全コンポーネントは、LinuxKit ベースの VM 内のコンテナで実行されています。これらのコンテナは、Docker for Mac または Docker for Windows の設定から非表示にできます。お使いのノート PC で実行している Kubernetes システム コンテナの全リストについては (Kubernetes のサポートを有効にしている場合)、このセクションの後半でご確認ください。重複を避けるために、ここからは Docker for Mac や Docker for Windows ではなく、**Docker for Desktop** についてのみ説明します。以降の説明は、どちらにも同じように該当します。

Minikube 上で有効になっている Kubernetes 搭載の Docker for Desktop の大きなメリットのひとつは、開発者が単一のツールを使用して Kubernetes を対象とするコンテナ化アプリケーションをビルド、テスト、実行できることです。また Docker Compose ファイルを使用すれば、Kubernetes にマルチサービス アプリケーションをデプロイすることも可能です。

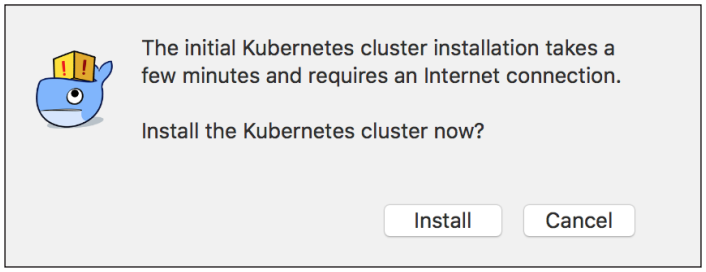
それでは実際にやってみましょう。

1. まず、Kubernetes を有効にします。
2. Mac の場合は、メニュー バーの Docker アイコンをクリックし、**[設定]** を選択します。
3. ダイアログ ボックスが表示されたら、下のスクリーンショットのように **[Kubernetes]** のオプションを選択します。



Docker for Mac での Kubernetes の有効化

4. 次に、**[Kubernetes の有効化]** チェックボックスを選択します。また、**[システム コンテナの表示]** (詳細) のチェックもオンにします。
5. **[適用]** ボタンをクリックします。Kubernetes のインストールと設定に数分かかるという下記のような警告メッセージが表示されます。



Kubernetes のインストールと設定に数分かかることを示す警告メッセージ

6. **[インストール]** ボタンをクリックするとインストールが始まります。その間、お茶でも飲みながら、ゆっくり休憩しておいてください。

インストールが完了したら (設定ダイアログに表示される緑色のステータス アイコンが完了の合図)、テストを行います。ノート PC で実行中の Kubernetes クラスターが 2 つ (Minikube と Docker for Mac) あるため、後者にアクセスするには `kubectl` を構成する必要があります。まず、コンテキストをすべてリストアップしましょう。

```
$ kubectl config get-contexts
CURRENT   NAME                 CLUSTER                      AUTHINFO             NAMESPACE
          docker-for-desktop   docker-for-desktop-cluster   docker-for-desktop
*         minikube             minikube                     minikube
$
```

kubectl のコンテキストのリスト

私のノート PC に、前述した 2 つのコンテキストがあるのがわかります。現在のところ、Minikube のコンテキストはアクティブで、アスタリスクで `CURRENT` 列に表示されています。次のコマンドを使うと、`docker-for-desktop` コンテキストに切り替えることができます。

```
$ kubectl config use-context docker-for-desktop
Switched to context "docker-for-desktop".
$
```

Kubernetes CLI のコンテキストの変更

これで、Docker for Mac が先ほど作成したクラスターに Kubectl を使ってアクセスすることができます。表示は次のようになります。

```
$ kubectl get nodes
NAME                 STATUS    ROLES     AGE       VERSION
docker-for-desktop   Ready     master    15m       v1.9.2
$
```

Docker for Mac が作成した単一ノードの Kubernetes クラスター

これ何だか見覚えがありませんか？実はこれ、Minikube を使用した時に表示されたものとほぼ同じなんです。Docker for Mac が使用している Kubernetes のバージョンは、`1.9.2` です。ノードがマスター ノードであることもわかります。

Docker for Mac で実行中のコンテナをすべてリストすると、下記のスクリーンショットのようなリストが表示されます (`--format` 引数を使用してコンテナの `Container ID` および `Names` のみを出力しています)。

```
$ docker container ls --format "table {{.ID}}\t{{.Names}}"
CONTAINER ID        NAMES
0cddff5e5a86        k8s_compose_compose-5d4f4d67b6-gbrjh_docker_c3f07e06-2e3b-11e8-860f-025000000001_0
0edd323f6cbb        k8s_compose_compose-api-7bb7b5968f-f98jk_docker_c3e89385-2e3b-11e8-860f-025000000001_0
218514b4fc00        k8s_POD_compose-5d4f4d67b6-gbrjh_docker_c3f07e06-2e3b-11e8-860f-025000000001_0
af8a64fc9f7e        k8s_POD_compose-api-7bb7b5968f-f98jk_docker_c3e89385-2e3b-11e8-860f-025000000001_0
f64fbcd5070c        k8s_sidecar_kube-dns-6f4fd4bdf-9b8kn_kube-system_ab66aab5-2e3b-11e8-860f-025000000001_0
4b6138bd34e7        k8s_dnsmasq_kube-dns-6f4fd4bdf-9b8kn_kube-system_ab66aab5-2e3b-11e8-860f-025000000001_0
bf1394d8e48a        k8s_kubedns_kube-dns-6f4fd4bdf-9b8kn_kube-system_ab66aab5-2e3b-11e8-860f-025000000001_0
a16b63a8f614        k8s_kube-proxy_kube-proxy-p4cf8_kube-system_ab6e9881-2e3b-11e8-860f-025000000001_0
655f8dca4a1c        k8s_POD_kube-proxy-p4cf8_kube-system_ab6e9881-2e3b-11e8-860f-025000000001_0
108b5a2fe05c        k8s_POD_kube-dns-6f4fd4bdf-9b8kn_kube-system_ab66aab5-2e3b-11e8-860f-025000000001_0
23f1808a6f8a        k8s_kube-scheduler_kube-scheduler-docker-for-desktop_kube-system_3a369b3ba7d6d3b6fa014295eab94925_0
89a1032beee7        k8s_kube-controller-manager_kube-controller-manager-docker-for-desktop_kube-system_b098d9f7b8b45512f23bcb04fe3f64f5_0
bb25965301d8        k8s_etcd_etcd-docker-for-desktop_kube-system_7278f85057e8bf5cb81c9f96d3b25320_0
126d0edc29f0        k8s_kube-apiserver_kube-apiserver-docker-for-desktop_kube-system_8d19d05a3d7b137bafa35348cb849dd5_0
d87992c3ea6e        k8s_POD_kube-scheduler-docker-for-desktop_kube-system_3a369b3ba7d6d3b6fa014295eab94925_0
063fdf120ea5        k8s_POD_kube-controller-manager-docker-for-desktop_kube-system_b098d9f7b8b45512f23bcb04fe3f64f5_0
22a1c70f6c4e        k8s_POD_kube-apiserver-docker-for-desktop_kube-system_8d19d05a3d7b137bafa35348cb849dd5_0
91ec502f1467        k8s_POD_etcd-docker-for-desktop_kube-system_7278f85057e8bf5cb81c9f96d3b25320_0
$
```

Kubernetes システム コンテナ

このリストで、現在知られている次のような Kubernetes の構成コンポーネントをすべて特定できます。

- API サーバー
- etcd
- Kube プロキシ
- DNS サービス
- Kube コントローラ
- Kube スケジューラ

また、「compose」という言葉を含むコンテナもあります。これらは Docker 固有のサービスであり、Docker Compose アプリケーションを Kubernetes にデプロイするために使用されます。Docker は Docker Compose 構文を変換し、デプロイメント、ポッド、サービスなど必要な Kubernetes オブジェクトを暗黙に作成します。

通常、これらのシステム コンテナでコンテナのリストが煩雑になるのは避けたいので、Kubernetes の設定で **[システム コンテナの表示]** ボックスのチェックを外しておきます。

では、Docker Compose アプリケーションを Kubernetes にデプロイしましょう。`labs` フォルダの `ch10` サブフォルダに移動します。`docker-compose.yml` ファイルを使って、アプリをスタックとしてデプロイします。

```
$ docker stack deploy -c docker-compose.yml app
```

次のように表示されます。

```
$ docker stack deploy -c docker-compose.yml app
Stack app was created
Waiting for the stack to be stable and running...
 - Service db has one container running
 - Service web has one container running
Stack app is stable and running
```

Kubernetes へのスタックのデプロイ

たとえば `curl` を使ってアプリケーションをテストすると、想定どおりに動作していることがわかります。

```
$ curl localhost:3000/pet
<html>
<head>
    <link rel="stylesheet" href="main.css">
</head>
<body>
    <div class="container">
        <h4>Cat Gif of the day</h4>
        <img src="http:&#x2F;&#x2F;ak-hdl.buzzfed.com&#x2F;static&#x2F;2013-10&#x2F;en
        <p><small>Courtesy: <a href="http://www.buzzfeed.com/copyranter/the-best-cat-g
        <p>Delivered to you by container web-5c5964c9b8-b5jq9<p>
    </div>
</body>
$
```

Kubernetes で実行中のペット アプリケーション (Docker for Mac)

`docker stack deploy` コマンドの実行時、Docker が一体何を行ったか知りたいと思われているのではないでしょうか。`kubectl` を使ってこれを解明してみましょう。

```
$ kubectl get all
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/web   1         1         1            1           9m

NAME               DESIRED   CURRENT   READY     AGE
rs/web-5c5964c9b8  1         1         1         9m

NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/web   1         1         1            1           9m

NAME               DESIRED   CURRENT   READY     AGE
rs/web-5c5964c9b8  1         1         1         9m

NAME             DESIRED   CURRENT   AGE
statefulsets/db  1         1         9m

NAME                     READY     STATUS    RESTARTS   AGE
po/db-0                  1/1       Running   0          9m
po/web-5c5964c9b8-b5jq9  1/1       Running   0          9m

NAME               TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
svc/db             ClusterIP      None            <none>        55555/TCP        9m
svc/kubernetes     ClusterIP      10.96.0.1       <none>        443/TCP          45m
svc/web            ClusterIP      None            <none>        55555/TCP        9m
svc/web-published  LoadBalancer   10.111.43.147   localhost     3000:32590/TCP   9m
$
```

Docker スタックのデプロイメント コマンドによって作成した Kubernetes 全オブジェクトのリスト

Docker は、`web` サービスのデプロイメントと `db` サービスのステートフル セットを作成しました。また、`web` および `db` に対する Kubernetes サービスを自動的に作成し、クラスター内でアクセスできるようにします。さらに、Kubernetes サービスの `svc/web-published` も外部アクセス用に作成します。

これによって、オーケストレーターとして Kubernetes を採用しているチームの開発プロセスの摩擦が大幅に低減するため、画期的だといわざるを得ないでしょう。

次に進む前に、クラスターからスタックを削除してください。

```
$ docker stack rm app
```

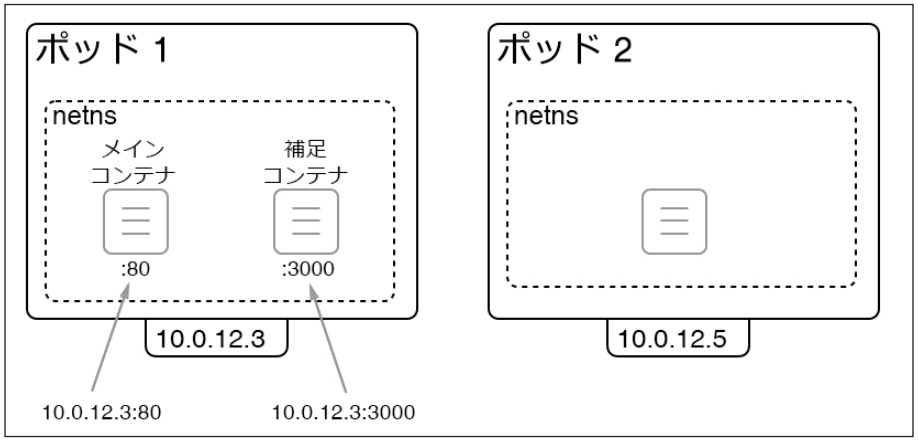
Minikube はこの章のすべてのサンプルに使いますので、`kubectl` のコンテキストを Minikube にリセットすることも忘れないでください。

```
$ kubectl config use-context minikube
```

Kubernetes クラスターで最終的に実行するアプリケーションの開発に活用できるツールの概要をご覧いただいたところで、次はこれらのアプリケーションの定義と管理に使用する重要な Kubernetes オブジェクトについて学習しましょう。まずは**ポッド**からです。

ポッド

Docker Swarm では可能ですが、Kubernetes クラスターではコンテナは直接実行できません。Kubernetes クラスターでは、ポッドのみ実行できます。ポッドは Kubernetes のデプロイメントの原子単位です。ポッドは、ネットワーク名前空間など、同じカーネル名前空間を共有する、1 つまたは複数の共同配置されたコンテナの抽象化で、これに相当するものは Docker SwarmKit にはありません。複数のコンテナを同一場所に配置し、同じネットワーク名前空間を共有するというのは非常に強力な概念です。2 つのポッドについて説明した下図をご覧ください。



Kubernetes ポッド

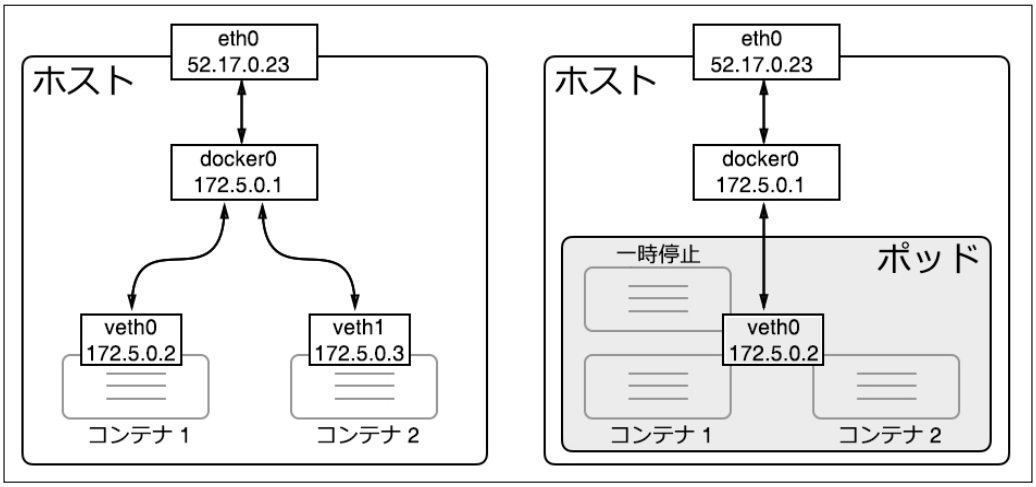
上図には、**ポッド 1** と**ポッド 2** の 2 つのポッドがあります。最初のポッドには 2 つのコンテナ、2 つ目のポッドには 1 つのコンテナのみが含まれます。Kubernetes によって割り当てられた、Kubernetes クラスター全体で固有の IP アドレスが各ポッドにあります。ここでは、IP アドレスは `10.0.12.3` と `10.0.12.5` で、どちらも Kubernetes ネットワーク ドライバーが管理するプライベート サブネットの一部です。

ポッドは、1 つまたは複数のコンテナを含むことができます。これらのコンテナはすべて同じカーネル名前空間、具体的にはネットワーク名前空間を共有します。これは、コンテナを囲む破線の四角形で示されています。同じポッドで実行されている全コンテナはネットワーク名前空間を共有しており、単一のネットワーク名前空間ではポートの重複が許容されないため、各コンテナは必ず独自のポートを使用する必要があります。ここでは、メイン コンテナの**ポッド 1** はポート `80`、補足コンテナはポート `3000` を使用しています。

他のポッドまたはノードからの要求は、ポッドの IP アドレスと対応するポート番号と組み合わせたものを使って個々のコンテナにアクセスします。たとえば、**ポッド 1** のメイン コンテナで実行中のアプリケーションには `10.0.12.3:80` からアクセスできます。

Docker コンテナと Kubernetes ポッドのネットワーキングの比較

次に、Docker のコンテナ ネットワーキングと Kubernetes ポッドのネットワーキングを比較してみましょう。下図では、前者を左側に、後者を右側に示しています。



ネットワーク名前空間を共有するポッドのコンテナ

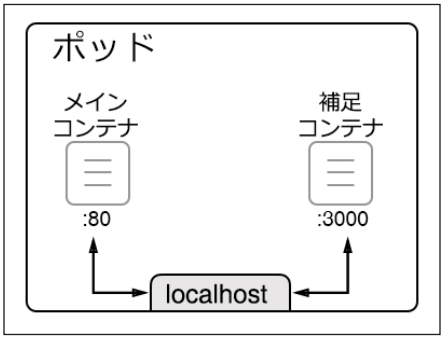
Docker コンテナが作成され、特定のネットワークが指定されていない場合、Docker エンジンは**仮想イーサネット** (**veth**) エンドポイントを作成します。最初のコンテナが **veth0**、次のコンテナが **veth1** という順に取得していきます。これらの仮想イーサネット エンドポイントは、Docker がインストール時に自動的に作成する、Linux ブリッジ **docker0** に接続されています。トラフィックはブリッジ docker0 から、接続されているすべての veth エンドポイントにルーティングされます。すべてのコンテナには独自のネットワーク名前空間がありますが、同じコンテナ内で実行中のアプリケーションを互いに隔離させるという意図により、名前空間が同じコンテナは 2 つと存在しません。

Kubernetes ポッドの場合、状況は異なります。新しいポッドを作成する際、まず Kubernetes はいわゆる**一時停止コンテナ**を作成します。このコンテナの唯一の目的は、ポッドがすべてのコンテナと共有する名前空間を作成、管理することです。それ以外、何の有用な機能も果たしません。一時停止コンテナは veth0 を介してブリッジ docker0 に接続されています。ポッドの一部となる後続のコンテナは、既存のネットワーク名前空間を再利用できる Docker エンジンの特別な機能を使用しています。これを実行する構文は次のようになります。

```
$ docker container create --net container:pause ...
```

重要な部分は `--net` 引数で、これは `container:<container name>` を値として使用します。この方法で新しいコンテナを作成すると、Docker は新しい veth エンドポイントを作成しませんが、コンテナは一時停止コンテナと同じものを使用します。

同じネットワーク名前空間を共有する複数のコンテナに関するもう 1 つの重要なポイントは、相互通信の方法です。2 つのコンテナを含むポッドを例にしてご説明しましょう。1 つのコンテナはポート 80 で、もう一方のコンテナはポート 3000 でリッスンしています。



localhost を介して通信するポッド内のコンテナ

同じ Linux カーネル ネットワーク名前空間を使用する 2 つのコンテナは、localhost を介して相互に通信することができます。これは、同じく localhost を介した相互通信が可能という点で、同一ホスト上で 2 つのプロセスが実行されている状況とよく似ています。これについては、上図で説明しています。メイン コンテナ内部のコンテナ化アプリケーションは、`http://localhost:3000` を経由して、メイン コンテナから補足コンテナ内部で実行中のサービスに到達することができます。

ネットワーク名前空間の共有

すべての理論をご覧いただいたところで、Kubernetes によってポッドがどのように作成されるのか知りたいと思われているかもしれませんね。Kubernetes は Docker が提供するものだけを使用します。そこで疑問となるのが、どうやってネットワーク名前空間を共有するのかです。まず、Kubernetes は前述したいわゆる一時停止コンテナを作成します。このコンテナには、該当ポッドのカーネル名前空間を予約し、それを使用可能な状態に維持する以外の機能はありません。これは、ポッド内で実行中のコンテナが他にない場合でも同様です。では、ポッドの作成をシミュレートしてみましょう。そのために、まず一時停止コンテナを作成し、Nginx を取得します。

```
$ docker container run -d --name pause nginx:alpine
```

今度は main と呼ばれる 2 番目のコンテナを、同じネットワーク名前空間に pause コンテナとしてアタッチして追加します。

```
$ docker container run --name main -dit \
    --net container:pause \
    alpine:latest /bin/sh
```

pause およびサンプル コンテナはどちらも同じネットワーク名前空間の一部であるため、localhost を介して相互に到達できます。これを説明するには、まず exec をメイン コンテナに入れます。

```
$ docker exec -it main /bin/sh
```

これで、pause コンテナで実行され、ポート 80 でリッスンしている Nginx への接続状態をテストできます。このテストに wget ユーティリティを使用すると、下記が表示されます。

```
/ # wget -qO - localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
/ #
```

同じネットワーク名前空間を共有する 2 つのコンテナ

この出力から、`localhost` 上の Nginx にアクセスできることが示されています。これは、2 つのコンテナが同じ名前空間を共有している証拠です。これでも十分でない場合は、`ip` ツールを使用して両コンテナ内にある `eth0` を表示させてみてください。結果はまったく同じです。具体的にいうと、IP アドレスが同一であり、これがすべてのコンテナが同じ IP アドレスを共有しているポッドの特性のひとつです。

```
/ # ip a show eth0
11: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ #
```

ip ツールを使用した eth0 のプロパティの表示

ブリッジ ネットワークをよく見てみると、`pause` コンテナのみがリストされているのがわかります。もう一方のコンテナは、`pause` コンテナのエンドポイントを拒否しているため、`Containers` リストのエントリを取得していません。

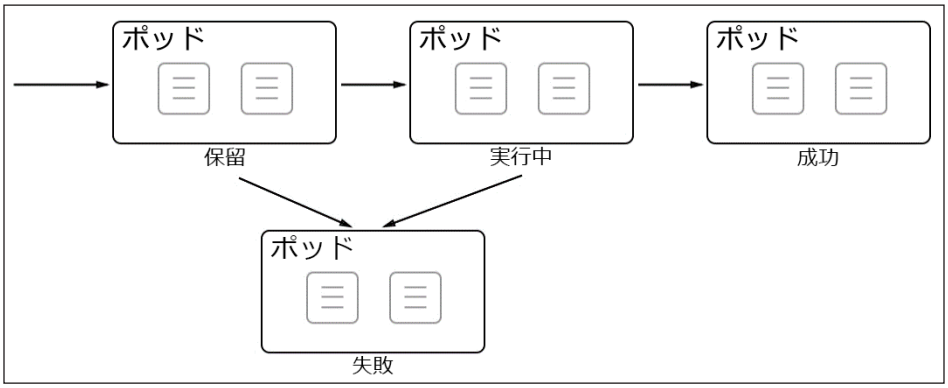

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "41909c08794041cabc3a9d2e034426f2344f5310bd1cbfcbae65c5f25a05f541",
        "Created": "2018-03-26T22:16:44.790966007Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "8965ec65ca4a1de1f1d9c987b68e888c1115cf64f44ba3842953d29a2b9a0ea8": {
                "Name": "pause",
                "EndpointID": "890fc0527f7cb6484d24b7886772db23bb5a0502fe34269fc306277ea7a6f95e",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
$
```


Docker デフォルト ブリッジ ネットワークの詳細

ポッドのライフサイクル

コンテナにライフ サイクルがあることは、本書の前半で学習しました。コンテナは初期化され、実行され、最終的に終了します。コンテナは、終了コード 0 で正常終了するか、ゼロ以外の終了コードに相当するエラーで終了することができます。

同様に、ポッドにもライフ サイクルがあります。1 つのポッドに複数のコンテナを含むことができるため、そのライフ サイクルは単一コンテナのライフ サイクルよりも若干複雑です。次の図で、ポッドのライフ サイクルを見てみましょう。



Kubernetes ポッドのライフ サイクル

ポッドがクラスター ノード上に作成されると、まず**保留**状態となり、ポッドの全コンテナが稼動すると、**実行**状態に入ります。すべてのコンテナが正常に実行された場合にのみ、ポッドはこの状態になります。ポッドは終了を要求されると、すべてのコンテナにも終了を要求します。全コンテナが終了コード 0 で終了すると、ポッドは「成功」状態になります。これがハッピー パスです。

では、ポッドが `failed` 状態となるシナリオをいくつか見てみましょう。次の 3 つのシナリオが考えられます。

- ポッドの起動時、少なくとも 1 つのコンテナを実行できず、失敗した (ゼロ以外の終了コードで終了した) 場合、ポッドは**保留**状態から**失敗**状態に移行します。
- ポッドが実行状態で、いずれかのコンテナが突然クラッシュしたり、ゼロ以外の終了コードで終了した場合、ポッドは**実行**状態から失敗状態へと移行します。
- ポッドが終了を要求され、シャットダウン中に少なくとも 1 つのコンテナがゼロ以外の終了コードで終了した場合は、ポッドも失敗状態となります。

ポッドの仕様

Kubernetes クラスターのポッドを作成する場合、**命令型**または**宣言型**のアプローチを使用できます。本書の前半ではこの 2 つのアプローチの違いについて説明しましたが、重要なポイントを言い換えると、宣言的なアプローチを使用することは、達成したい最終状態に関するマニフェストを示すということです。その詳しい方法についてはオーケストレーターに任せましょう。達成したい終了状態は、**目的の状態**とも呼ばれます。一般に、宣言型アプローチは確立されたすべてのオーケストレーターで強く推奨されており、Kubernetes も例外ではありません。

そのため、この章では宣言型アプローチについてのみ取り上げます。ポッドのマニフェストまたは仕様は、YAML または JSON 形式のいずれかで記述できます。この章では、人間にとってより読みやすい `YAML` に着目します。サンプル仕様を見てみましょう。ここに示したのは、`pod.yaml` ファイルのコンテンツで、これは `labs` フォルダの `ch10` サブフォルダにあります。

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
  - name: web
    image: nginx:alpine
    ports:
    - containerPort: 80
```

Kubernetes の各仕様は、バージョン情報から始まります。ポッドはかなりの古株なので、API バージョンは v1 です。2 行目には、定義したい Kubernetes オブジェクトまたはリソースのタイプを指定します。もちろんここでは、ポッドを指定します。次の行はメタデータを含むブロックで、最低限ポッドには名前を付ける必要があります。ここでは、`web-pod` とします。次のブロックは仕様ブロックで、これはポッドの仕様を含みます。最も重要な部分 (この単純なサンプルでは 1 つのみ) は、このポッドの一部である全コンテナのリストです。ここには 1 つのコンテナしかありませんが、複数のコンテナが可能です。コンテナに選択した名前は web で、コンテナイメージは `nginx:alpine` です。最後に、コンテナが公開しているポートのリストを定義します。

このような仕様を作成したら、Kubernetes CLI `kubectl` を使用してクラスターに適用します。ターミナルで、`ch10` サブフォルダに移動し、次のコマンドを実行します。

```
$ kubectl create -f pod.yaml
```

これによって、作成した `pod "web-pod"` が出力されます。これで、`kubectl get pods` のクラスター内の全ポッドをリストすることができます。

```
$ kubectl get pods
NAME       READY    STATUS     RESTARTS   AGE
web-pod    1/1      Running    0          2m
```

想定どおり、実行状態のポッドが 1 つあります。定義したとおり、このポッドの名前は `web-pod` です。実行中のポッドに関する詳細を表示するには、`describe` コマンドを使用します。

```
$ kubectl describe pod/web-pod
Name:         web-pod
Namespace:    default
Node:         minikube/192.168.99.105
Start Time:   Sun, 25 Mar 2018 22:47:49 -0500
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           172.17.0.3
Containers:
  web:
    Container ID:   docker://e8784dfc2e3fcf1de4bfb9ab1508176799b6024b96d9447126e1db5dd5e2201f
    Image:          nginx:alpine
    Image ID:       docker-pullable://nginx@sha256:17c4704e19a11cd47545fa3c17e6903fc88672021f7f907f212d6663baf6ab57
    Port:           80/TCP
    State:          Running
      Started:      Sun, 25 Mar 2018 22:47:50 -0500
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fhdsm (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  default-token-fhdsm:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-fhdsm
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     <none>
Events:
  Type    Reason                 Age   From               Message
  ----    ------                 ----  ----               -------
  Normal  Scheduled              5m    default-scheduler  Successfully assigned web-pod to minikube
  Normal  SuccessfulMountVolume  5m    kubelet, minikube  MountVolume.SetUp succeeded for volume "default-token-fhdsm"
  Normal  Pulled                 5m    kubelet, minikube  Container image "nginx:alpine" already present on machine
  Normal  Created                5m    kubelet, minikube  Created container
  Normal  Started                5m    kubelet, minikube  Started container
$
```

クラスターで実行中のポッドの記述

先ほどの `describe` コマンドの表記、`pod/web-pod` に注目してください。たとえば `pods/web-pod` や `po/web-pod` など他のバリアントも可能です。`pod` および `po` は `pods` のエイリアスです。`kubectl` ツールはさまざまなエイリアスを定義してくれるため便利です。

describe コマンドは、ポッドが影響を受けたイベントのリストをはじめ、ポッドに関する有用な情報を多数提供してくれます。リストは出力の最後に表示されます。

Containers セクションの情報は、docker container inspect の出力に表示される情報と非常によく似ています。

Volumes セクションのタイプには、Secret のエントリもあります。Kubernetes のシークレットについては、次の章で説明します。次に、ボリュームについて説明します。

ポッドとボリューム

コンテナに関する章では、永続的なデータにアクセスして保存するためのボリュームとその目的について学びました。コンテナがボリュームをマウントできるように、ポッドもボリュームをマウントできます。実際には、ボリュームをマウントするのはポッド内のコンテナですが、それは意味的詳細にすぎません。まず、Kubernetes でボリュームを定義する方法を見てみましょう。Kubernetes は膨大なボリュームタイプをサポートしていますが、これについてはあまり深く掘り下げません。my-data-claim と呼ばれる PersistentVolumeClaim を定義し、ローカル ボリュームを暗黙に作成してみましょう。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-data-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

2GB のデータを求める要求を定義しました。この要求を作成しましょう。

```
$ kubectl create -f volume-claim.yaml
```

kubectl (pvc は PersistentVolumeClaim のショートカット) を使って要求をリストします。

```
$ kubectl get pvc
NAME           STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
my-data-claim  Bound     pvc-aac3bb2c-3224-11e8-a07f-080027c10823   2Gi        RWO            standard       14m
$
```

クラスター内の PersistentStorageClaim オブジェクトのリスト

出力では、要求が pvc-<ID> というボリュームを暗黙に作成しているのがわかります。これで、要求によって作成されたボリュームをポッドで使用できるようになりました。以前使用したポッド仕様を修正したものを使用しましょう。この仕様の更新版は、ch10 フォルダの pod-with-vol.yaml ファイルにあります。この仕様をもう少し詳しく見てみましょう。

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
  - name: web
    image: nginx:alpine
    ports:
    - containerPort: 80
    volumeMounts:
    - name: my-data
      mountPath: /data
  volumes:
  - name: my-data
    persistentVolumeClaim:
      claimName: my-data-claim
```

最後 4 行の volumes のブロックで、このポッドに使用するボリュームのリストを定義します。ここにリストされているボリュームは、ポッドのどのコンテナでも使用できます。ここでは、ボリュームは 1 つしかありません。先ほど作成した要求名を持つ永続ボリュームの要求である、my-data ボリュームがあると定義します。コンテナ仕様にある volumeMounts のブロックでは、使用するボリュームと、そのボリュームがマウントされるコンテナ内の (絶対) パスを定義します。ここでは、コンテナ ファイルシステムの /data フォルダにボリュームをマウントします。このポッドを作成しましょう。

```
$ kubectl create -f pod-with-vol.yaml
```

次に、下記のようにコンテナに exec し、/data フォルダに移動してボリュームがマウントしたことをダブルチェックしたら、そこでファイルを作成してコンテナを終了します。

```
$ kubectl exec -it web-pod -- /bin/sh
/ # cd /data
/data # echo "Hello world!"> sample.txt
/data # exit
```

間違いがなければ、このコンテナのデータはポッドのライフ サイクルが過ぎても持続するはずです。ポッドを削除してから再作成し、そこで exec してデータがまだ残っていることを確認します。このような結果が表示されます。

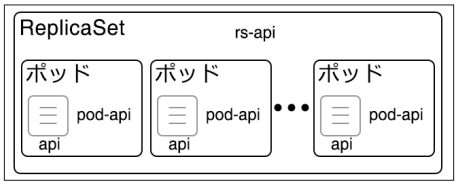
```
$ kubectl delete po/web-pod
pod "web-pod" deleted
$ kubectl create -f pod-with-vol.yaml
pod "web-pod" created
$ kubectl exec -it web-pod -- /bin/sh
/ # cat /data/sample.txt
Hello world!
/ #
```

ボリュームに格納されたデータのライフ サイクルは再作成のポッドより長い

Kubernetes ReplicaSet

可用性要件の高い環境内にポッドが 1 つだけでは不十分です。ポッドがクラッシュしたら？サービスの中断を許容することなく、ポッド内で実行中のアプリケーションを更新する必要がある場合は？これらの質問は、ポッドだけでは不十分であること、そして同一ポッドの複数のインスタンスを管理できるより高度な概念が必要であることを示しているにすぎません。Kubernetes では、**ReplicaSet** は異なるクラスター ノードで実行されている同一ポッド群を定義および管理するために使用されます。**ReplicaSet** は特に、ポッド内で実行されているコンテナによって使用されるコンテナ イメージや、クラスター内で実行されるポッドのインスタンス数を定義します。これらのプロパティや他の多くのプロパティは、**目的の状態**と呼ばれます。

ReplicaSet は、実際の状態が目的の状態から逸脱している場合、常に目的の状態になるよう調整する役割を担います。下記が Kubernetes ReplicaSet です。



Kubernetes ReplicaSet

上図では、**rs-api** と呼ばれる ReplicaSet がいくつかのポッドを管理しています。ポッドは **pod-api** と呼ばれます。ReplicaSet は、常に目的の数のポッドが実行されていることを確認する役割を担います。何らかの理由でいずれかのポッドがクラッシュした場合、ReplicaSet は空きリソースを持つノード上に新しいポッドをスケジュールします。目的の数よりポッドが多い場合、ReplicaSet は余分なポッドを削除します。つまり ReplicaSet によって、自己復旧可能でスケーラブルなポッド セットを得られるわけです。ReplicaSet を構成するポッドの数に制限はありません。

ReplicaSet の仕様

ポッドの場合と同様に、Kubernetes では ReplicaSet も命令的または宣言的に定義し、管理できます。ほとんどの場合、宣言的なアプローチの方が断然推奨されるため、このアプローチについて重点的に説明します。下記は、ReplicaSet のサンプル仕様です。

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 3
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        ports:
        - containerPort: 80
```

これは、先ほど紹介したポッドの仕様に驚くほどよく似ています。では今度は、違いに目を向けてみましょう。まず 2 行目を見ると、Pod だったオブジェクトの種類 (kind) が、今は ReplicaSet になっています。6 ～ 8 行目にはセレクター (selector) があり、ここで ReplicaSet の一部となるポッドを特定します。ここでは、すべてのポッドが app のラベルに web の値を持ちます。次に 9 行目で、実行するポッドの複製数を定義します。この場合は 3 つです。最後に、template セクションで最初に metadata を定義し、次に spec でポッド内で実行するコンテナを定義します。ここでは、イメージ (image) が nginx:alpine、エクスポート ポートが 80 の単一のコンテナがあります。

極めて重要な要素は、複製数、そして ReplicaSet によって管理されるポッド セットを指定するセレクターです。

ch10 フォルダには、前述の仕様をそのまま含む replicaset.yaml というファイルがあります。ではこのファイルを使って、ReplicaSet を作成してみましょう。

```
$ kubectl create -f replicaset.yaml
replicaset "rs-web" created
```

クラスター内の ReplicaSet をすべてリストすると、下記が出力されます (rs は replicaset のショートカット)。

```
$ kubectl get rs
NAME      DESIRED   CURRENT   READY   AGE
rs-web    3         3         3       51s
```

上記の出力では、目的の状態が 3 (ポッド) である rs-Web という単一の ReplicaSet があるのがわかります。現在の状態では 3 つのポッドがあり、3 つすべて準備が整っています。システム内のすべてのポッドをリストすることもでき、そうすると出力は次のようになります。

```
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
rs-web-6qzld   1/1     Running   0          4m
rs-web-frj2m   1/1     Running   0          4m
rs-web-zd2kt   1/1     Running   0          4m
```

ここには、期待される 3 つのポッドが表示されています。ポッド名には ReplicaSet の名前が使用されており、各ポッドに固有の ID が付加されています。READY 列には、ポッドで定義されているコンテナ数と、準備が整っているコンテナ数が表示されています。ここでは、ポッドあたりのコンテナは 1 つのみで、いずれも準備ができています。したがって、ポッドの全体的な状態は Running となります。また、各ポッドに必要な再起動回数も示されており、ここではまだゼロであることがわかります。

自己復旧

ではここで、ランダムにポッドをどれか削除し、ReplicaSet がその後どんな自己復旧の力を発揮するか見てみましょう。まず最初のポッドをリストから削除します。

```
$ kubectl delete po/rs-web-6qzld
pod "rs-web-6qzld" deleted
```

次に、すべてのポッドをもう一度リストします。表示されるのは 2 つのポッドだけのはずですよね？下をご覧ください。

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
rs-web-frj2m  1/1     Running   0          22h
rs-web-q6cr7  1/1     Running   0          41s
rs-web-zd2kt  1/1     Running   0          22h
$
```

ReplicaSet のポッドを 1 つ削除した後のポッドのリスト

AGE 列を見ればわかるように、リストの 2 番目のポッドが明らかに再作成されています。これがまさに自己復旧の力です。ReplicaSet を記述した場合を見てみましょう。

```
$ kubectl describe rs/rs-web
Name:         rs-web
Namespace:    default
Selector:     app=web
Labels:       app=web
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=web
  Containers:
   nginx:
    Image:        nginx:alpine
    Port:         80/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  4m    replicaset-controller  Created pod: rs-web-q6cr7
```

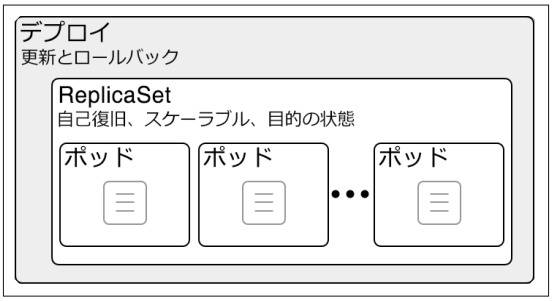
ReplicaSet の記述

たしかに Events の下に入力があります。つまり、ReplicaSet が rs-web-q6cr7 という新しいポッドを作成したということです。

Kubernetes デプロイメント

Kubernetes は単一責任原則に基づいており、Kubernetes の全オブジェクトは、1 つのことだけを極めて適切に行うよう設計されています。この点に関して、Kubernetes の ReplicaSet と Deployment を理解する必要があります。ここまで学習したように、ReplicaSet はアプリケーション サービスの目的の状態を達成し、調整する役割を担います。つまり、ReplicaSet は一連のポッドを管理します。

ローリング アップデートやロールバックの機能により ReplicaSet を拡張するのがデプロイメントです。Docker Swarm では、swarm サービスは ReplicaSet と Deployment の両方の機能を統合しており、この点においては、SwarmKit は Kubernetes よりもはるかにモノリシックです。次の図は、Deployment と ReplicaSet の関係を示したものです。



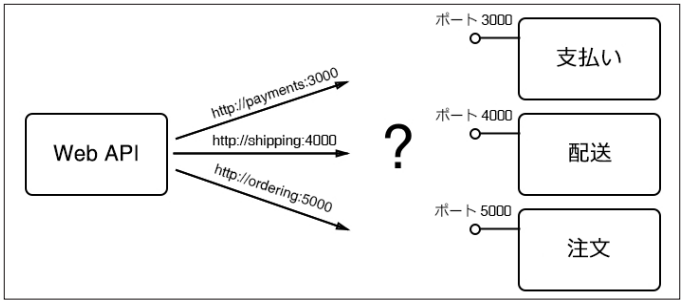
Kubernetes デプロイメント

上図では、**ReplicaSet** は同一のポッドのセットを定義および管理しています。ReplicaSet の主な特徴は、自己復旧可能でスケーラブルであり、常に目的の状態に調整するために最大限の機能を発揮することです。これに対し、Kubernetes デプロイメントはローリング アップデートとロールバックの機能を追加します。このような点で、デプロイメントは ReplicaSet のラッパー オブジェクトだといえます。

ローリング アップデートとロールバックについては、本書の次の章で詳しく説明します。

Kubernetes サービス

複数のアプリケーション サービスから成るアプリケーションを使い始めた瞬間から必要になるのが、サービス検出です。この問題を示した下の図をご覧ください。

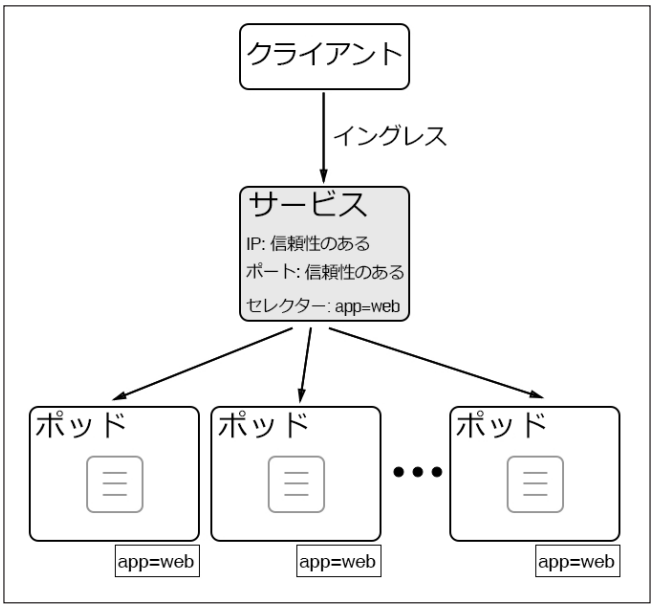


サービス検出

この図には、**支払い**、**配送**、**注文**という他の 3 つのサービスへのアクセスが必要な **Web API** サービスがあります。これらの 3 つのサービスを、どのように、どこで見つけるかは Web API の役割範囲外です。API コードで使用したいのは、到達したいサービスの名前とそのポート番号だけです。サンプルとして挙げられるのが URL `http://payments:3000` で、これが支払いサービスのインスタンスへのアクセスに使用されます。

Kubernetes では、支払いアプリケーション サービスはポッドの ReplicaSet で示されます。高度な分散システムの性質上、ポッドに安定したエンドポイントがあるとは想定できません。ポッドは一瞬にして移り変わることができます。しかし、内外のクライアントから該当のアプリケーション サービスにアクセスする必要がある場合には、これが問題となります。ポッドのエンドポイントの安定性を当てにできなければ、手の打ちようがありません。

ここで出番となるのが Kubernetes サービスです。下図が示すように、Kubernetes サービスは、安定したエンドポイントを ReplicaSet や Deployment に提供することを意図しています。

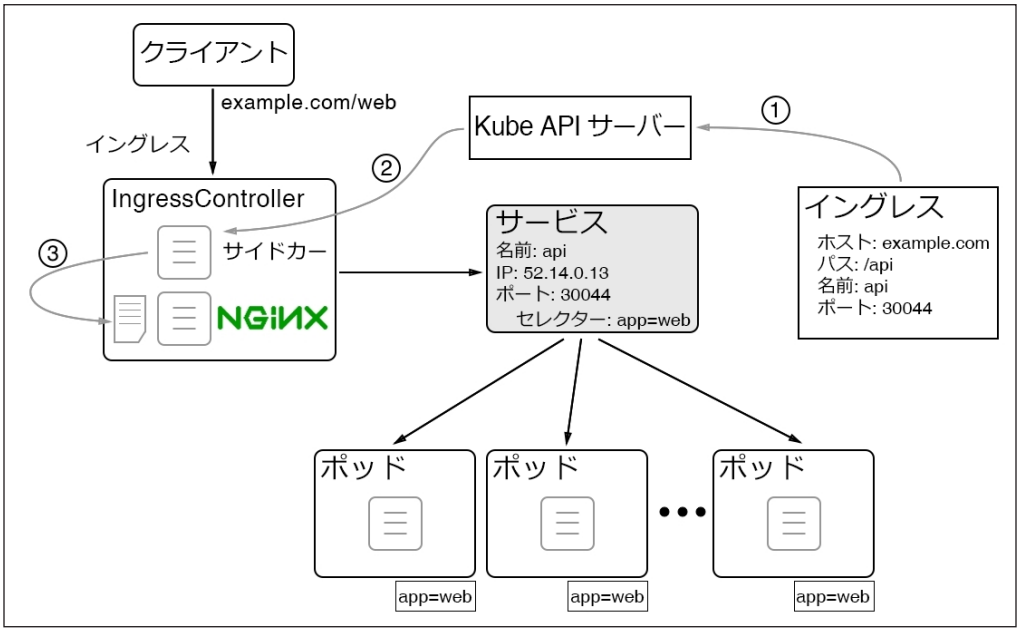


クライアントに安定したエンドポイントを提供する Kubernetes サービス

上図の中央にあるのが Kubernetes サービスで、**仮想 IP** (**VIP**) とも呼ばれる信頼性の高いクラスター全体の IP アドレスと、クラスター全体で固有の信頼性の高いポートを提供します。 Kubernetes サービスがプロキシするポッドは、サービス仕様で定義されているセレクターによって決定されます。セレクターは常にラベルに基づいており、Kubernetes の各オブジェクトには割り当てられた多数のラベル (ゼロを含む) があります。この場合、セレクターは **app=web** で、これはつまり **app** というラベルと web の値を持つすべてのポッドがプロキシされるということです。

コンテキストベースのルーティング

通常、Kubernetes のクラスターにはコンテキストベースのルーティングを構成します。Kubernetes ではこれをさまざまな方法で行うことができますが、現時点で推奨される最も拡張性の高い方法は、**IngressController** を使用するという方法です。下記は、イングレス コントローラーの仕組みを図示したものです。



Kubernetes イングレス コントローラーを使用したコンテキストベース ルーティング

Nginx などのイングレス コントローラーを使用したコンテキストベース (またはレイヤー 7) ルーティングの仕組みがこの図から確認できます。ここでは、web と呼ばれるアプリケーション サービスのデプロイメントがあり、このアプリケーション サービスのポッドにはすべて **app=web** のラベルが付いています。次に、これらのポッドに安定したエンドポイントを提供する web という Kubernetes サービスがあります。このサービスには `52.14.0.13` の (仮想) IP があり、ポート `30044` を公開します。つまり、その名前の web とポート `30044` に対して Kubernetes クラスターのいずれかのノードに要求が送信されると、それがこのサービスに転送され、サービスがいずれかのポッドにその要求をロード バランシングします。

ここまでは順調ですが、クライアントから URL `http[s]://example.com/web` へのイングレス要求は、web サービスにどのようにルーティングされるのでしょうか。まず、コンテキストベースの要求から対応する `<service name>/<port>` 要求へのルーティングを定義する必要があります。これは Ingress オブジェクトを通じて行います。

1. Ingress オブジェクトでは、ソースおよび (サービス) 名前としてホストとパス、ターゲットとしてポートを定義します。Kubernetes API サーバーによってこの Ingress オブジェクトが作成されると、IngressController でサイドカーとして実行されるプロセスがこの変更をピックアップします。
2. Nginx リバース プロキシの構成ファイルを変更します。
3. 新しいルートを追加すると、Nginx はその構成をリロードするよう要求され、`http[s]://example.com/web` への着信要求を正しくルーティングできるようになります。

まとめ

この章では、Kubernetes の基本について学習しました。アーキテクチャの概要や、Kubernetes クラスター内のアプリケーションの定義および実行に使用される主なリソースの概要、Docker for Mac/Docker for Windows での Minikube および Kubernetes のサポートについても取り上げました。

次の章では、Kubernetes クラスターにアプリケーションをデプロイします。次に、ダウンタイム ゼロ戦略を用いてこのアプリケーションのいずれかのサービスを更新し、最後はシークレットを用いて、Kubernetes で動作する機密データを含むアプリケーション サービスを測定します。

質問

この章で学習した内容を踏まえ、以下の質問に回答してください。

1. Kubernetes マスターの役割を簡単に説明してください。
2. Kubernetes の各 (ワーカー) ノードになくてはならない要素を挙げてください。
3. 「Kubernetes クラスター内では個々のコンテナを実行できない」という文は、正しいでしょうか、間違っているでしょうか。
4. ポッドのコンテナが localhost を使って相互通信できる理由を説明してください。
5. ポッド内の一時停止 `container` の目的とは何ですか。

6. 「僕らのアプリケーションは、`web`、`inventory`、`db` の 3 つの Docker イメージで構成されている。単一の Kubernetes ポッド内で複数のコンテナを実行できるわけだから、アプリケーションの全サービスを単一のポッドにデプロイしよう」とボブが提案しています。この考えが好ましくない理由を 3 ～ 4 つ挙げてください。
7. Kubernetes ReplicaSet が必要な理由を自分の言葉で説明してください。
8. Kubernetes デプロイメントが必要になるのは、どのような状況ですか。
9. Kubernetes サービスのタイプを 3 つ以上挙げ、その目的と違いを説明してください。

参考情報

下記の記事では、この章で取り上げた各種トピックに関する詳細情報を紹介しています。

- Raft コンセンサス アルゴリズム (`https://raft.github.io/`)
- Docker for Desktop での Docker Compose と Kubernetes (`https://dockr.ly/2G8Iqb9`)

11
Kubernetes によるアプリケーションのデプロイ、更新、および保護

最後の章では、コンテナ オーケストレーターである Kubernetes の基本について学びました。Kubernetes のアーキテクチャの概要を理解し、Kubernetes がコンテナ化されたアプリケーションを定義して管理するために使用する重要なオブジェクトについて多くのことを学びました。

この章では、アプリケーションを Kubernetes クラスターにデプロイ、更新、およびスケールする方法について説明します。また、ミッション クリティカルなアプリケーションの中断のない更新とロールバックを可能にするために、ゼロ ダウンタイム デプロイメントをどのように実現するかについても説明します。さらに、この章では、サービスを構成して機密データを保護する手段として、Kubernetes シークレットを導入します。

この章では以下のトピックについて説明します。

- 最初のアプリケーションのデプロイ
- ゼロ ダウンタイム デプロイメント
- Kubernetes シークレット

この章を読み終わると、次のことができるようになります。

- マルチサービス アプリケーションを Kubernetes クラスターにデプロイする
- ダウンタイムを発生させることなく Kubernetes で動作中のアプリケーション サービスを更新する
- Kubernetes クラスターでシークレットを定義する
- Kubernetes シークレットを使用するようにアプリケーション サービスを構成する

技術的要件

この章では、ローカル コンピューターで Minikube を使用します。Minikube のインストール方法と使用方法については、「第 2 章 作業環境の設定」を参照してください。

この章のコードは、`labs` フォルダの `ch11` サブフォルダにあります。「第 2 章 作業環境の設定」の説明に従って、`https://github.com/appswithdockerandkubernetes/labs` にある GitHub リポジトリが複製されていることを確認してください。

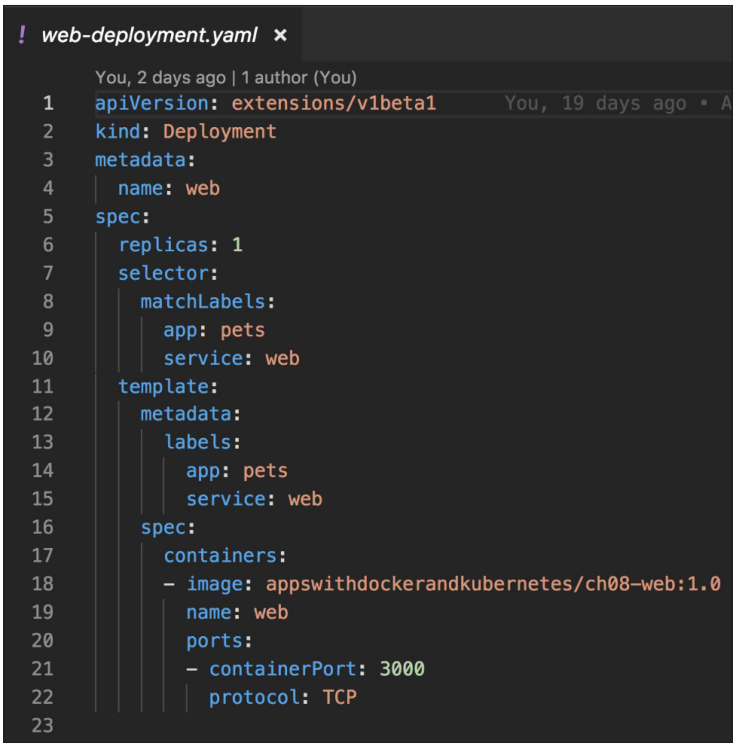
ターミナルで、フォルダ `labs/ch11` に移動します。

最初のアプリケーションのデプロイ

「第 8 章 Docker Compose」で初めて導入したペット アプリケーションを取得して、Kubernetes クラスターにデプロイします。使用するクラスターは、ご存じのように、単一ノード クラスターの Minikube です。ただし、デプロイメントの観点では、クラスターの規模や設置場所 (クラウド、会社のデータ センター、またはパーソナル ワークステーション) はそれほど重要ではありません。

Web コンポーネントのデプロイ

確認ですが、使用するアプリケーションは、Node.js ベースの Web コンポーネントとバッキング PostgreSQL データベースという 2 つのアプリケーション サービスで構成されています。前の章では、デプロイするアプリケーション サービスごとに Kubernetes `Deployment` オブジェクトを定義する必要があることを学びました。まずは、Web コンポーネント用のオブジェクトを定義しましょう。いつものように、オブジェクトを定義する宣言的な方法を選択します。Web コンポーネント用の `Deployment` オブジェクトを定義する YAML を以下に示します。



Web コンポーネント用の Kubernetes デプロイメント定義

前述のデプロイメント定義は、labs フォルダ `ch11` 内の `web-deployment.yaml` ファイル内にあります。コード行は次のとおりです。

- **4 行目** : `Deployment` オブジェクトの名前を `web` **として定義します。**
- **6 行目** : `web` コンポーネントのいずれかのインスタンスを実行する必要があることを宣言します。
- **8 ～ 10 行目** : デプロイメントの一部になるポッド、つまり、ラベルが `app` と `service` で、それぞれの値が `pets` と `web` のポッドを定義します。
- **11 行目** : 11 行目から始まるポッドのテンプレートで、各ポッドに 2 つのラベルの `app` と `service` が適用されることを定義します。
- **17 行目以降** : ポッドで実行される単一のコンテナを定義します。コンテナ用のイメージはよく知られている `appswithdockerandkubernetes/ch08-web:1.0` イメージで、コンテナの名前は `web` になります。
- **ポート** : 最後に、コンテナが TCP 型トラフィック用の**ポート** `3000` **を公開することを宣言します。**

`kubectl` のコンテキストが `Minikube` に設定されていることを確認してください。この方法の詳細については、「第 2 章 作業環境の設定」を参照してください。

この `Deployment` オブジェクトは `kubectl` を使用してデプロイできます。

```
$ kubectl create -f web-deployment.yaml
```

再度、Kubernetes CLI を使用してデプロイメントが作成されたことを二重チェックすることができます。次の出力を確認する必要があります。

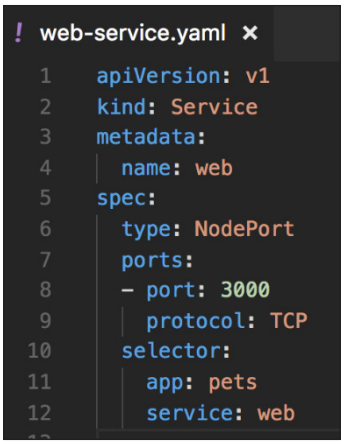
```
$ kubectl get all
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/web   1         1         1            1           5m

NAME               DESIRED   CURRENT   READY     AGE
rs/web-769b88f67   1         1         1         5m

NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/web   1         1         1            1           5m

NAME               DESIRED   CURRENT   READY     AGE
rs/web-769b88f67   1         1         1         5m

NAME                     READY     STATUS    RESTARTS   AGE
po/web-769b88f67-4fccx   1/1       Running   0          5m

NAME             TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   8d
$
```

Minikube で実行中のすべてのリソースの一覧表示

執筆時点で、Minikube または `kubectl` に、コマンド `kubectl get all` を使用すると特定のリソースが 2 回表示されるというバグがあるようです。重複した出力は無視してください。

前述の出力では、Kubernetes が 3 つのオブジェクト (デプロイメント、関係する `ReplicaSet`、および 1 つのポッド (1 つのレプリカのみが必要だと指定)) を作成したことが確認できます。現在の状態は 3 つのオブジェクトすべてが望ましい状態にあることを示しているため、ここまでは順調だと言えます。

ここで、Web サービスを公開する必要があります。そのために、`NodePort` タイプの Kubernetes `Service` オブジェクトを定義する必要があります。`labs` フォルダ `ch11` 内の `web-deployment.yaml` ファイル内にある定義を以下に示します。



Web コンポーネント用の Service オブジェクトの定義

前述のコード行は次のとおりです。

- **4 行目** : この `Service` オブジェクトの名前を `web` に設定します。
- **6 行目** : 使用する `Service` オブジェクトのタイプを定義します。`web` コンポーネントはクラスターの外部からアクセスできる必要があるため、`ClusterIP` タイプの `Service` オブジェクトにすることはできず、`NodePort` と `LoadBalancer` のどちらかのタイプにする必要があります。前の章でさまざまなタイプの Kubernetes サービスについて説明したため、ここでは詳細には触れません。この例では、`NodePort` タイプのサービスを使用します。
- **8 行目と 9 行目** : TCP プロトコルを介してアクセスするポート `3000` を公開するように指定します。Kubernetes は、コンテナ ポート `3000` を自動的に 30,000 から 32,768 の範囲の空きホスト ポートにマップします。Kubernetes が実際に選択するポートは、作成されたサービスに対して `kubectl get service` コマンドまたは `kubectl describe` コマンドを使用して決定できます。
- **10 ～ 12 行目** : このサービスが安定したエンドポイントになるためのポッドのフィルター基準を定義します。このケースでは、ラベルが `app` と `service` で、それぞれの値が `pets` と `web` のポッドを使用します。

`Service` オブジェクトに対するこの指定は、`kubectl` を使用して実現できます。

```
$ kubectl create -f web-service.yaml
```

すべてのサービスを一覧表示することで、前述のコマンドの結果を確認できます。

```
$ kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          9d
web          NodePort    10.103.113.40   <none>        3000:30125/TCP   3m
$
```

Web コンポーネント用に作成された Service オブジェクト

出力では、`web` という名前のサービスが作成されています。一意の `clusterIP` `10.103.113.40` がこのサービスに割り当てられ、コンテナ ポート `3000` がすべてのクラスター ノード上のポート `30125` で公開されています。

このデプロイメントをテストするには、まず、Minikube の IP アドレスを調べて、その IP アドレスを使用して `web` サービスにアクセスする必要があります。これを行うためのコマンドを以下に示します。

```
$ IP=$(minikube ip)
$ curl -4 $IP:30125/
Pets Demo Application
```

いいですね。応答は想定どおりの Pets Demo Application です。`web` サービスが Kubernetes クラスター内で稼働しています。次は、データベースをデプロイします。

データベースのデプロイ

データベースは、ステートフルなコンポーネントであり、Web コンポーネントのようなステートレスなコンポーネントとは異なる扱いをする必要があります。「第 6 章 分散アプリケーション アーキテクチャ」と「第 9 章 オーケストレーター」で、分散アプリケーション アーキテクチャにおけるステートフル コンポーネントとステートレス コンポーネントの違いについて詳しく説明されています。

Kubernetes は、ステートフル　コンポーネント用の特殊なタイプの `ReplicaSet` オブジェクトを定義しました。このオブジェクトは `StatefulSet` という名前です。このタイプのオブジェクトを使用してデータベースをデプロイします。定義は `labs/ch11/db-stateful-set.yaml` ファイル内にあります。詳細は次のとおりです。

db-stateful-set.yaml

```
You, 2 days ago | 1 author (You)
1  apiVersion: apps/v1      You, 19 days ago • Adding exis
2  kind: StatefulSet
3  metadata:
4    name: db
5  spec:
6    selector:
7      matchLabels:
8        app: pets
9        service: db
10   serviceName: db
11   template:
12     metadata:
13       labels:
14         app: pets
15         service: db
16     spec:
17       containers:
18       - image: appswithdockerandkubernetes/ch08-db:1.0
19         name: db
20         ports:
21         - containerPort: 5432
22         volumeMounts:
23         - mountPath: /var/lib/postgresql/data
24           name: pets-data
25   volumeClaimTemplates:
26   - metadata:
27       name: pets-data
28     spec:
29       accessModes:
30       - ReadWriteOnce
31       resources:
32         requests:
33           storage: 100Mi
34
```

DB コンポーネント用の StatefulSet

少し驚いたかもしれませんが、そうでもありません。Web コンポーネント用のデプロイメントの定義より少し長いだけです。これは、PostgreSQL データベースにデータを格納するためのボリュームも定義する必要があるためです。ボリューム要求定義は 25 から 33 行目にあります。名前が `pets-data` で、最大サイズが 100 MB のボリュームを作成します。22 から 24 行目で、このボリュームを PostgreSQL が想定している `/var/lib/postgresql/data` にあるコンテナにマウントします。21 行目で、PostgreSQL がポート `5432` でリッスンすることも宣言します。

いつものように、`kubectl` を使用して `StatefulSet` をデプロイします。

```
$ kubectl create -f db-stateful-set.yaml
```

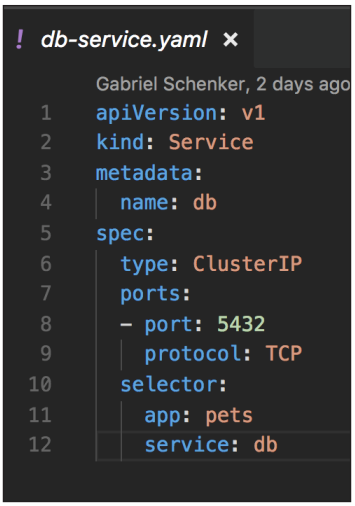
クラスター内のすべてのリソースを一覧表示すると、作成された追加のオブジェクトを確認できます。

```
$ kubectl get all
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/web   1         1         1            1           27m

NAME               DESIRED   CURRENT   READY     AGE
rs/web-769b88f67   1         1         1         27m

NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/web   1         1         1            1           27m

NAME               DESIRED   CURRENT   READY     AGE
rs/web-769b88f67   1         1         1         27m

NAME              DESIRED   CURRENT   AGE
statefulsets/db   1         1         49s

NAME                    READY     STATUS    RESTARTS   AGE
po/db-0                 1/1       Running   0          49s
po/web-769b88f67-qd2xf  1/1       Running   0          27m

NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
svc/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          10d
svc/web          NodePort    10.103.113.40   <none>        3000:30125/TCP   27m
$
```

StatefulSet とそのポッド

1 つの `StatefulSet` と 1 つのポッドが作成されています。両方の現在の状態が望ましい状態を示しているため、システムは正常だと言えます。ただし、この時点では、`web` コンポーネントがデータベースにアクセスできるかどうかはわかりません。ここまでは、サービス検出が機能していません。`web` コンポーネントは、`db` という名前で `db` サービスにアクセスする必要があることを思い出してください。

クラスター内でサービス検出を機能させるには、データベース コンポーネント用の Kubernetes `Service` オブジェクトも定義する必要があります。データベースはクラスター内からしかアクセスできないため、必要な `Service` オブジェクトのタイプは `ClusterIP` です。`labs/ch11/db-service.yaml` ファイル内の指定を以下に示します。



データベース用の Kubernetes Service オブジェクトの定義

データベース コンポーネントは、この `Service` オブジェクトによって表現され、4 行目で定義されているサービスの名前 `db` によって到達できます。また、データベース コンポーネントは公開する必要がないため、`ClusterIP` タイプの `Service` オブジェクトを使用できます。10 から 12 行目のセレクターは、このサービスが、対応するラベル、つまり、`app: pets` と `service: db` が定義されたすべてのポッドの安定したエンドポイントを表していることを定義しています。

次のコマンドを使用して、このサービスをデプロイします。

```
$ kubectl create -f db-service.yaml
```

これで、アプリケーションをテストする準備ができました。今回はブラウザを使用して、面白い猫の画像を楽しむことができます。

Kubernetes で動作するペット アプリケーションのテスト

`192.168.99.100` は、自分の Minikube の IP アドレスです。コマンド `minikube ip` を使用してアドレスを確認します。ポート番号 `30125` は、Kubernetes が Web `Service` オブジェクト用として自動的に選択した番号です。この番号を Kubernetes がサービスに割り当てたポートに置き換えます。コマンド `kubectl get services` を使用して番号を取得します。

これで、単一ノード Kubernetes クラスターである Minikube にペット アプリケーションが正常にデプロイされました。このために、次の 4 つの成果物を定義する必要がありました。

- Web コンポーネント用の `Deployment` オブジェクトと `Service` オブジェクト
- データベース コンポーネント用の `StatefulSet` オブジェクトと `Service` オブジェクト

クラスターからアプリケーションを削除するには、次の短いスクリプトを使用できます。

```
kubectl delete svc/web
kubectl delete deploy/web
kubectl delete svc/db
kubectl delete statefulset/db
```

デプロイメントの合理化

ここまでは、クラスターにデプロイする必要のある 4 つの成果物を作成しました。また、これは 2 つのコンポーネントからなる非常に単純なアプリケーションです。もっと複雑なアプリケーションがあると想像してみてください。それは、すぐにメンテナンスの悪夢に変わるはずです。幸い、デプロイメントを簡素化するためのオプションがいくつか用意されています。ここで説明する方法では、Kubernetes 内のアプリケーションを構成するすべてのコンポーネントを 1 つのファイルで定義することができます。

本書では取り上げなかったその他のソリューションには、**Helm** などのパッケージマネージャーの使用が含まれます。

アプリケーションが `Deployment` オブジェクトや `Service` オブジェクトなどの複数の Kubernetes オブジェクトで構成されている場合は、そのすべてを 1 つのファイルにまとめて、個々のオブジェクト定義を 3 つのダッシュで区切ることができます。たとえば、Web コンポーネント用のデプロイメント定義とサービス定義を 1 つのファイルにまとめたい場合は、次のように指定します。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pets
      service: web
  template:
    metadata:
      labels:
        app: pets
        service: web
    spec:
      containers:
      - image: appswithdockerandkubernetes/ch08-web:1.0
        name: web
        ports:
        - containerPort: 3000
          protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  type: NodePort
  ports:
```

```
    - port: 3000
      protocol: TCP
  selector:
    app: pets
    service: web
```

`labs/ch11/pets.yaml` ファイルからペット アプリケーションに関する 4 つのオブジェクト定義のすべてを収集しました。このアプリケーションは 1 回でデプロイすることができます。

```
$ kubectl create -f pets.yaml
deployment "web" created
service "web" created
statefulset "db" created
service "db" created
$
```

単一のスクリプトを使用したペット アプリケーションのデプロイ

同様に、Kubernetes クラスターからペット アプリケーションのすべての成果物を削除するスクリプト `labs/ch11/remove-pets.sh` を作成しました。

```
$ ./remove-pets.sh
deployment "web" deleted
service "web" deleted
statefulset "db" deleted
service "db" deleted
$
```

Kubernetes クラスターからのペットの削除

「第 8 章 Docker Compose」で導入したペット アプリケーションを取得して、そのアプリケーションを Kubernetes クラスターにデプロイするために必要なすべての Kubernetes オブジェクトを定義しました。各ステップで、想定された結果が得られたことを確認し、クラスター内にすべての成果物が生成された段階で、実行中のアプリケーションが表示されました。

ダウンタイム ゼロのデプロイメント

ミッション クリティカルな環境では、アプリケーションが中断しないことが重要です。最近は、ダウンタイムを発生させることができなくなっています。Kubernetes はこれを実現するためのさまざまな手段を提供しています。ダウンタイムを発生させないクラスター内のアプリケーションの更新を**ゼロ ダウンタイム デプロイメント**と言います。この章では、これを実現する 2 つの方法を紹介します。以下を参照してください。

- ローリング更新
- ブルーグリーン デプロイメント

ローリング更新の説明から始めましょう。

ローリング更新

前の章で、Kubernetes Deployment オブジェクトは、ReplicaSet オブジェクトの機能にローリング更新とロールバックが追加されている点が異なることを学びました。これを実証するために、web コンポーネントを使用します。見たところ、web コンポーネントのデプロイメントのマニフェストまたは記述を変更する必要があります。

前のセクションと同じデプロイメント定義を使用します。重要な違いは、web コンポーネントの 5 つのレプリカが実行中だということです。次の定義は、labs/ch11/web-deploy-rolling-v1.yaml ファイル内にもあります。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 5
  selector:
    matchLabels:
      app: pets
      service: web
  template:
    metadata:
      labels:
        app: pets
        service: web
    spec:
      containers:
      - image: appswithdockerandkubernetes/ch08-web:1.0
        name: web
        ports:
        - containerPort: 3000
          protocol: TCP
```

これで、通常どおりこのデプロイメントを作成し、それと同時に、コンポーネントをアクセス可能にするサービスも作成することができます。

```
$ kubectl create -f web-deploy-rolling-v1.yaml
$ kubectl create -f web-service.yaml
```

ポッドとサービスをデプロイしたら、次のコマンドを使用して web コンポーネントをテストできます。

```
$ PORT=$(kubectl get svc/web -o yaml | grep nodePort | cut -d' ' -f5)
$ IP=$(minikube ip)
$ curl -4 ${IP}:${PORT}/
Pets Demo Application
```

ご覧のように、アプリケーションが稼働中で、想定されたメッセージ「Pets Demo Application」が返されます。

ここで、開発者が web コンポーネントの新しいバージョン 2.0 を作成しました。w eb コンポーネントの新しいバージョンのコードが labs/ch11/web/src フォルダにあり、変更するのは server.js ファイルの 12 行目だけです。

```
 9  app.set('views', __dirname);
10
11  app.get('/',function(req,res){
12      res.status(200).send('Pets Demo Application v2\n');
13  });
14
```

w eb コンポーネントのバージョン 2.0 のコード変更

開発者が次のような新しいイメージをビルドしました。

```
$ docker image build -t appswithdockerandkubernetes/ch11-web:2.0 web
```

また、その後で、そのイメージを Docker Hub にプッシュしました。

```
$ docker image push appswithdockerandkubernetes/ch11-web:2.0
```

ここで、web Deployment オブジェクトの一部であるポッドで使用されるイメージを更新する必要があります。これを行うには、kubectl の set image コマンドを使用します。

```
$ kubectl set image deployment/web \
    web=appswithdockerandkubernetes/ch11-web:2.0
```

アプリケーションを再度テストすると、実際に更新が行われたことが確認されます。

```
curl -4 ${IP}:${PORT}/
Pets Demo Application v2
```

ここで、この更新中にダウンタイムが発生しなかったことを確認するにはどうしたらいいでしょうか。更新は実際にローリング形式で行われたのでしょうか。ローリング更新が意味するものとは何でしょうか。調査してみましょう。まず、rollout status コマンドを使用して、Kubernetes から実際にデプロイメントが行われ、成功したことを確認できます。

```
$ kubectl rollout status deploy/web
deployment "web" successfully rolled out
```

deploy/web を記述する kubectl を使用してデプロイメント web を記述すると、出力の最後に次のイベントのリストが表示されます。

```
Events:
  Type    Reason             Age              From                   Message
  ----    ------             ----             ----                   -------
  Normal  ScalingReplicaSet  12m              deployment-controller  Scaled up replica set web-769b88f67 to 5
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled up replica set web-55cdf67cd to 1
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled down replica set web-769b88f67 to 4
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled up replica set web-55cdf67cd to 2
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled down replica set web-769b88f67 to 3
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled up replica set web-55cdf67cd to 3
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled down replica set web-769b88f67 to 2
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled up replica set web-55cdf67cd to 4
  Normal  ScalingReplicaSet  3m               deployment-controller  Scaled down replica set web-769b88f67 to 1
  Normal  ScalingReplicaSet  3m (x2 over 3m)  deployment-controller  (combined from similar events): Scaled down replica
 set web-769b88f67 to 0
$
```

web コンポーネントのデプロイメント記述の出力に表示されたイベントのリスト

最初のイベントは、デプロイメントの作成時に、5 つのレプリカを含む `ReplicaSet web-769b88f67` が作成されたことを示しています。その後で、update コマンドが実行されます。リストの 2 つ目のイベントは、初期状態で 1 つのレプリカしかない `web-55cdf67cd` という名前の新しい `ReplicaSet` が作成されたことを示しています。したがって、この時点では、5 つの初期ポッドと 1 つの新しいバージョンのポッドの 6 つのポッドがシステム上に存在していることになります。しかし、`Deployment` オブジェクトに必要なのは 5 つのレプリカだけなので、Kubernetes は古い `ReplicaSet` を 4 つのインスタンスにスケールダウンします。このことが 3 つ目のイベントで確認できます。その後で、再び、新しい `ReplicaSet` が 2 つのインスタンスにスケールアップされてから、古い `ReplicaSet` が 3 つのインスタンスにスケールダウンされます。この動作が、5 つの新しいインスタンスが作成され、すべての古いインスタンスが使用停止になるまで繰り返されます。これが行われた正確な時間 (3 分以外) はわかりませんが、イベントの順序が更新全体がローリング形式で行われたことを示しています。

短い間に、一部の Web サービスへの呼び出しに対して古いバージョンのコンポーネントから応答が返され、それ以外の呼び出しに対して新しいバージョンのコンポーネントから応答が返されています。しかし、決してサービスはダウンしていません。

クラスター内の `Recordset` オブジェクトを一覧表示することもできるため、前のセクションで説明した内容を確認できます。

```
$ kubectl get rs
NAME            DESIRED   CURRENT   READY     AGE
web-55cdf67cd   5         5         5         27m
web-769b88f67   0         0         0         36m
$
```

クラスター内のすべての Recordset オブジェクトを一覧表示する

新しいレコードセットで 5 つのインスタンスが実行中であることがわかります。古いレコードセットは 0 インスタンスにスケールダウンされています。古い `Recordset` オブジェクトがまだ残っている理由は、Kubernetes が更新をロールバックする可能性があるからであり、その場合は、`Recordset` が再利用されます。

新しいコードに未知のバグが含まれている場合にイメージの更新プログラムをロールバックするには、rollout `undo` コマンドを使用できます。

```
$ kubectl rollout undo deploy/web
deployment "web"
$ curl -4 ${IP}:${PORT}/
Pets Demo Application
```

実際にロールバックが行われたことを確認するために、前のスニペットで curl を使用して test コマンドを一覧表示しました。レコードセットを一覧表示すると、次の出力が表示されます。

```
$ kubectl get rs
NAME            DESIRED   CURRENT   READY     AGE
web-55cdf67cd   0         0         0         36m
web-769b88f67   5         5         5         45m
$
```

ロールバック後の RecordSet オブジェクトの一覧表示

これは、古い `RecordSet` (`web-769b88f67`) オブジェクトが再利用され、新しいオブジェクトが 0 インスタンスにスケールダウンされたことを示しています。

ただし、新しいバージョンと古いバージョンが混在する状態を許容できないまたは許容したくない場合があります。オールオアナッシングの戦略を取る必要があります。これは、次に説明するブルーグリーン デプロイメントの出番です。

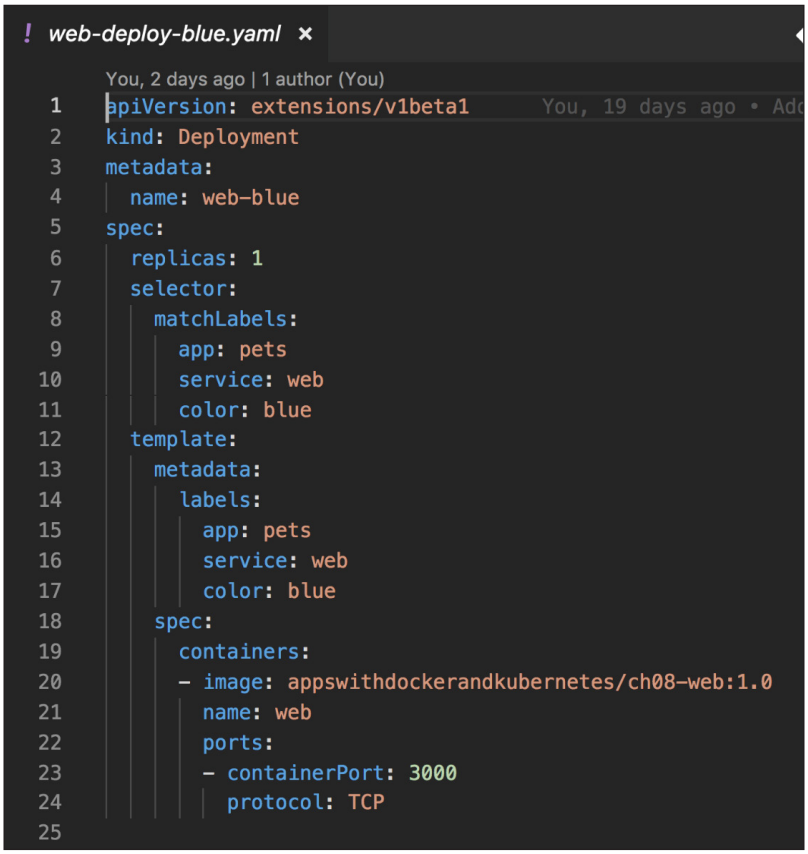
ブルーグリーン デプロイメント

ペット アプリケーションのコンポーネント web のブルーグリーン スタイルのデプロイメントを実施する場合は、ラベルを創造的に使用することによって実現できます。まずは、ブルーグリーン デプロイメントの機能を振り返っておきましょう。大まかな段階的手順を以下に示します。

1. コンポーネント `web` の最初のバージョンを `blue` としてデプロイします。そのために、ポッドに `color: blue` というラベルを付けます。
2. セレクター セクションで、ラベル `color: blue` が付けられたこれらのポッド用の Kubernetes サービスをデプロイします。
3. ここで、web コンポーネントのバージョン 2 をデプロイできますが、今回は、ポッドにラベル `color: green` を付けるだけです。

4. サービスの green バージョンが想定どおりに機能するかどうかをテストできます。
5. ここで、`web` コンポーネント用の Kubernetes サービスを更新することによって、トラフィックを `blue` から `green` に切り替えます。ラベル `color: green` を使用するようにセレクターを変更します。

バージョン 1 の blue 用の `Deployment` オブジェクトを定義しましょう。



web コンポーネントのデプロイメント blue の指定

前述の定義は、`labs/ch11/web-deploy-blue.yaml` ファイル内にあります。次のデプロイメント `web-green` と区別するために、デプロイメントの名前を `web-blue` と定義している 4 行目に注目してください。また、11 行目と 17 行目でラベル `color: blue` が追加されていることにも注目してください。それ以外は前回と同じです。

ここで、web コンポーネント用の Service オブジェクトを定義します。これは、次のスクリーンショットからもわかるように、以前使用したもののマイナー チェンジです。

ブルーグリーン デプロイメントをサポートする web コンポーネント用の Kubernetes サービス

この章の前半で使用したサービスの定義との唯一の違いは、セレクターにラベル color: blue が追加されている 13 行目です。前述の定義は、labs/ch11/web-svc-blue-green.yaml ファイル内にあります。

これで、次のコマンドを使用して、web コンポーネントの blue バージョンをデプロイできます。

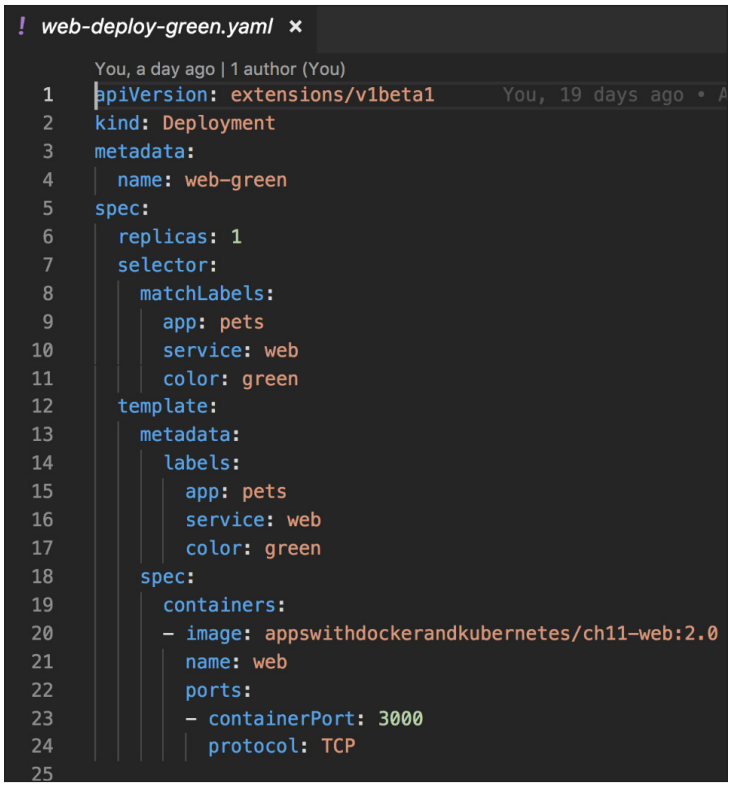
```
$ kubectl create -f web-deploy-blue.yaml
$ kubectl create -f web-svc-blue-green.yaml
```

サービスが稼働すると、IP アドレスとポート番号を特定してそれをテストすることができます。

```
$ PORT=$(kubectl get svc/web -o yaml | grep nodePort | cut -d' ' -f5)
$ IP=$(minikube ip)
$ curl -4 ${IP}:${PORT}/
Pets Demo Application
```

想定どおり、「Pets Demo Application」という応答が返されます。

これで、web コンポーネントの green バージョンをデプロイできます。その Deployment オブジェクトの定義は labs/ch11/web-deploy-green.yaml ファイル内にありますが、それを以下に示します。



web コンポーネントのデプロイメント green の指定

興味深いコード行は次のとおりです。

- **4 行目** : `web-green` という名前を使用して、`web-blue` と区別し、並列インストールを許可する
- **11 行目と 17 行目** : 色 `green` を設定する
- **20 行目** : イメージのバージョン `2.0` を使用する

これで、この `green` バージョンのサービスをデプロイする準備が整いました。`blue` サービスとは別に実行する必要があります。

```
$ kubectl create -f web-deploy-green.yaml
```

両方のデプロイメントが共存していることを確認できます。

```
$ kubectl get deploy
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
web-blue    1         1         1            1           23h
web-green   1         1         1            1           3s
$
```

クラスター内で実行中の デプロイメント オブジェクトのリストの表示

想定どおり、`blue` と `green` の両方が実行中です。`blue` がまだアクティブなサービスであることを確認できます。

```
$ curl -4 ${IP}:${PORT}/
Pets Demo Application
```

ここから、興味深い部分に入ります。web コンポーネント用の既存のサービスを編集することによって、トラフィックを `blue` から `green` に切り替えることができます。そのために、次のコマンドを実行します。

```
$ kubectl edit svc/web
```

ラベル色の値を `blue` から `green` に変更します。次に、エディターを保存して終了します。Kubernetes CLI が自動的にサービスを更新します。再度 Web サービスに問い合わせると、次のような応答が返されます。

```
$ curl -4 ${IP}:${PORT}/
Pets Demo Application v2
```

これは、トラフィックが実際に `web` コンポーネントの `green` バージョンに切り替えられたことを示しています (`curl` コマンドに対する応答の最後にある `v2` に注目してください)。

green デプロイメントで障害が発生し、新しいバージョンに不具合があることが判明した場合は、Web サービスを再度編集してラベル色の値を `green` から `blue` に切り替えることによって、簡単に `blue` バージョンに戻すことができます。このロールバックは、瞬間的に行われ、必ず機能するはずです。その後で、不具合のある `green` デプロイメントを削除して、コンポーネントを修復します。問題を修正したら、`green` バージョンを再度デプロイすることができます。

コンポーネントの `green` バージョンが想定どおりに起動して正常に動作していれば、`blue` バージョンを使用停止できます。

```
$ kubectl delete deploy/web-blue
```

新しいバージョン 3.0 をデプロイする準備ができたら、それが blue バージョンになります。それに応じて、`labs/ch11/web-deploy-blue.yaml` ファイルを更新してデプロイします。その後で、Web サービスを `green` から `blue` に切り替えたりします。

これで、ペット アプリケーションのコンポーネント web を使用したデモを終了します。Kubernetes クラスターでブルーグリーン デプロイメントが実現できました。

Kubernetes シークレット

Kubernetes クラスター内で動作するサービスで、パスワード、シークレット API キー、証明書などの機密データを使用しなければならない場合があります。この機密情報には許可されたサービスや専用サービス以外からはアクセスできないことを確認する必要があります。クラスター内で実行中の他のサービスは、このデータにアクセスできないようにする必要があります。

このために、Kubernetes シークレットが導入されました。シークレットはキーと値のペアで、キーはシークレットの一意の名前、値は実際の機密データです。シークレットは `etcd` に保存されます。Kubernetes は、シークレットが保存中、つまり、`etcd` 内にあるときと、送信中、つまり、シークレットが回線上をマスター ノードからこのシークレットを使用しているサービスのポッドが動作しているワーカー ノードに移動中に暗号化されるように構成できます。

シークレットの手動定義

シークレットは、Kubernetes で他のオブジェクトを作成したのと同じ方法で宣言的に作成できます。このようなシークレットの YAML を以下に示します。

```
apiVersion: v1
kind: Secret
metadata:
  name: pets-secret
type: Opaque
data:
  username: am9obi5kb2UK
  password: c0VjcmV0LXBhc1N3MHJECg==
```

前述の定義は、`labs/ch11/pets-secret.yaml` ファイル内にあります。ここで、値を確認する必要があります。実際の (暗号化されていない) 値でしょうか。いいえ。そうではありません。実は、暗号化された値ではなく、`base64` でエンコードされた値です。つまり、base64 でエンコードされた値は簡単にクリア テキスト値に戻すことができるため、全く安全ではありません。これらの値はどうすれば取得できますか。簡単です。

```
$ echo "john.doe" | base64
am9obi5kb2UK
$ echo "sEcret-pasSw0rD" | base64
c0VjcmV0LXBhc1N3MHJECg==
$
```

シークレットの base64 でエンコードされた値の作成

シークレットを作成して、それを記述することができます。

```
$ kubectl create -f pets-secret.yaml
secret "pets-secret" created
$ kubectl describe secrets/pets-secret
Name:         pets-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  16 bytes
username:  9 bytes
$
```

Kubernetes シークレットの作成と記述

シークレットの記述では、値が隠され、長さのみが指定されます。これでシークレットは安全なのでしょうか。いいえ。必ずしもそうではありません。このシークレットは、`kubectl get` コマンドを使用して簡単にデコードできます。

```
$ kubectl get secrets/pets-secret -o yaml
apiVersion: v1
data:
  password: c0VjcmV0LXBhc1N3MHJECg==
  username: am9obi5kb2UK
kind: Secret
metadata:
  creationTimestamp: 2018-03-31T20:36:05Z
  name: pets-secret
  namespace: default
  resourceVersion: "154786"
  selfLink: /api/v1/namespaces/default/secrets/pets-secret
  uid: 22d818bd-3523-11e8-a3cb-080027c10823
type: Opaque
$
```

デコードされた Kubernetes シークレット

前述のスクリーンショットからわかるように、元のシークレット値に戻されています。これらをデコードできます。

```
$ echo "c0VjcmV0LXBhc1N3MHJECg==" | base64 --decode
sEcret-pasSw0rD
```

そのため、結果として、この Kubernetes の作成方法は、機密データを扱わない開発以外の環境では使用すべきではありません。その他の環境では、シークレットを扱うより良い方法が必要です。

kubectl を使用したシークレットの作成

シークレットを定義するもっと安全な方法は、kubectl を使用することです。まず、前のセクションで行ったのと同様に base64 でエンコードされたシークレット値を含むファイルを作成しますが、今回は、その値を一時ファイルに保存します。

```
$ echo "sue-hunter" | base64 > username.txt
$ echo "123abc456def" | base64 > password.txt
```

ここで、次のように、kubectl を使用して、これらのファイルからシークレットを作成できます。

```
$ kubectl create secret generic pets-secret-prod \
    --from-file=./username.txt \
    --from-file=./password.txt
secret "pets-secret-prod" created
```

そうすれば、シークレットを手動で作成されたシークレットと同じように使用できます。

なぜ、この方法は他の方法より安全なのでしょう。まず、シークレットを定義して、多くの人がシークレットにアクセスして表示したりデコードしたりできるソース コード バージョン管理システム (GitHub など) に保存される YAML が存在しません。シークレットを知ることを許可された管理者だけがそれらの値を確認し、それを使って、(実稼働) クラスターで直接シークレットを作成します。クラスター自体はロールベースのアクセス制御によって保護されているため、許可されていない人物は、それにアクセスすることも、クラスターで定義されたシークレットをデコードすることもできません。

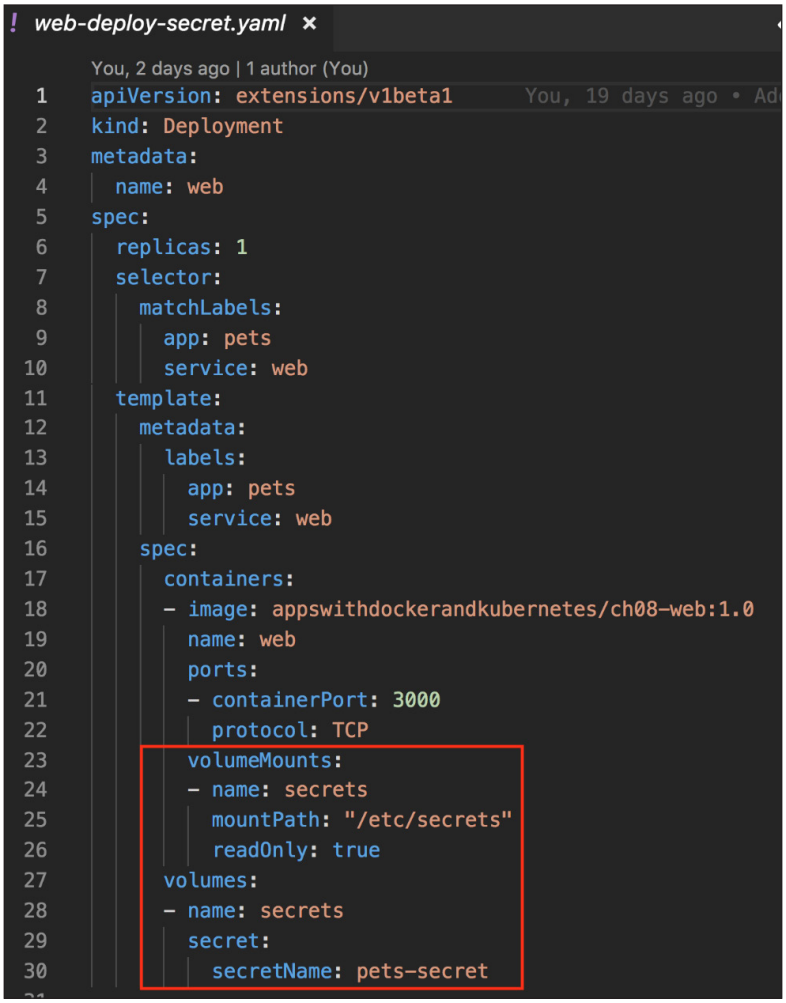
ここでは、定義したシークレットを実際にどのように使用するかを見てみましょう。

ポッドでのシークレットの使用

web コンポーネントが前のセクションで導入された pets-secret という名前のシークレットを使用する Deployment オブジェクトを作成する必要があるとします。次のコマンドを使用して、クラスター内にシークレットを作成します。

```
$ kubectl create -f pets-secret.yaml
```

`labs/ch11/web-deploy-secret.yaml` ファイルで、`Deployment` オブジェクトの定義が見つかります。23 行目以降の部分を `Deployment` オブジェクトの元の定義に追加する必要があります。



シークレットを含む web コンポーネントの Deployment オブジェクト

27 から 30 行目で、シークレット pets-secret から secrets という名前のボリュームを定義しています。その後で、23 から 26 行目に記述されているように、このボリュームをコンテナ内で使用します。シークレットを /etc/secrets にあるコンテナ ファイル システムにマウントして、読み取り専用モードでボリュームをマウントします。これにより、secret 値に上記フォルダ内のファイルとしてのコンテナからアクセスできます。ファイルの名前はキー名に対応し、ファイルの内容は対応するキーの値になります。値は、コンテナ内で実行中のアプリケーションに暗号化されていない形式で提供されます。

このケースでは、シークレット内のキー username と password から、コンテナ ファイル システム内の /etc/secrets フォルダにある username と password という名前の 2 つのファイルが見つかります。ファイル username には値 john.doe を含める必要があり、ファイル password には値 sEcret-pasSw0rD を含める必要があります。確認メッセージを以下に示します。

```
$ kubectl exec -it web-597b7f7749-87mq5 -- /bin/sh
/app # cd /etc/secrets/
/etc/secrets # ls -l
total 0
lrwxrwxrwx    1 root     root            15 Apr  2 01:26 password -> ..data/password
lrwxrwxrwx    1 root     root            15 Apr  2 01:26 username -> ..data/username
/etc/secrets # cat username && cat password
john.doe
sEcret-pasSw0rD
/etc/secrets #
```

シークレットがコンテナ内で利用できることの確認

前述の出力の 1 行目で、web コンポーネントが動作するコンテナに移動しています。次に、2 から 5 行目で、/etc/secrets フォルダ内のファイルを一覧表示し、最後に、6 から 8 行目で、シークレット値がクリア テキストになっている 2 つのファイルの内容を表示しています。

どんな言語で書かれたアプリケーションでも単純なファイルは読み取ることができるため、シークレットを使用するこのメカニズムは高い下位互換性があります。古い COBOL アプリケーションでさえ、ファイル システムからクリア テキスト ファイルを読み取ることができます。

しかし、アプリケーションによっては、環境変数内のシークレットにアクセスできることを想定している場合があります。このような場合に、Kubernetes はどう対処するかを見てみましょう。

環境変数内のシークレット値

web コンポーネントが、環境変数 PETS_USERNAME 内のユーザー名と PETS_PASSWORD 内のパスワードを想定している場合は、デプロイメント YAML を次のように変更することができます。

web-deploy-secret-env.yaml

```
1   apiVersion: extensions/v1beta1
2   kind: Deployment
3   metadata:
4     name: web
5   spec:
6     replicas: 1
7     selector:
8       matchLabels:
9         app: pets
10        service: web
11    template:
12      metadata:
13        labels:
14          app: pets
15          service: web
16      spec:
17        containers:
18        - image: fundamentalsofdocker/ch08-web:1.0
19          name: web
20          ports:
21          - containerPort: 3000
22            protocol: TCP
23          env:
24            - name: PETS_USERNAME
25              valueFrom:
26                secretKeyRef:
27                  name: pets-secret
28                  key: username
29            - name: PETS_PASSWORD
30              valueFrom:
31                secretKeyRef:
32                  name: pets-secret
33                  key: password
34
```

環境変数へのシークレット値のデプロイメント マッピング

23 から 33 行目で、2 つの環境変数 PETS_USERNAME と PETS_PASSWORD を定義し、それらに pets-secret の対応するキーと値のペアをマップしています。

これ以上ボリュームは必要ありませんが、pets-secret の個別のキーをコンテナ内で有効な対応する環境変数に直接マップしていることに注意してください。次のコマンド シーケンスは、それぞれの環境変数内のコンテナでシークレット値が実際に使用できることを示しています。

```
$ kubectl exec -it web-694f958cd4-6zq89 -- /bin/sh
/app # echo $PETS_USERNAME && echo $PETS_PASSWORD
john.doe
sEcret-pasSw0rD
/app #
```

環境変数にマップされたシークレット値

このセクションでは、Kubernetes クラスターでシークレットを定義する方法と、デプロイメントのポッドの一部として実行中のコンテナでそれらのシークレットを使用する方法を示しました。シークレットがコンテナ内でどのようにマップされるかに関する 2 つのバリエーションを示しました。1 つはファイルを使用し、もう 1 つは環境変数を使用します。

まとめ

この章では、アプリケーションを Kubernetes クラスターにデプロイする方法と、そのアプリケーションのアプリケーション レベルのルーティングを設定する方法を学習しました。また、ダウンタイムを発生させることなく Kubernetes クラスター内で動作中のアプリケーション サービスを更新する方法を学びました。最後に、シークレットを使用して、クラスター内で動作中のアプリケーション サービスに機密情報を渡しました。

次の最終章では、マイクロソフトの Azure Kubernetes Service (AKS) オファリングを使用して、クラウドでコンテナ化されたサンプル アプリケーションをデプロイ、実行、監視、デバッグする方法を学習します。お楽しみに

質問

学習の進捗状況を評価するために、以下の質問にお答えください。

1. 2 つのサービスからなるアプリケーションがあります。1 つは Web API で、もう 1 つは Mongo などの DB です。このアプリケーションを Kubernetes クラスターにデプロイする必要があります。その手順を簡単に説明してください。
2. アプリケーションのレイヤー 7 (またはアプリケーション レベル) ルーティングを確立するために必要なコンポーネントを自分の言葉で簡潔に説明してください。
3. 単純なアプリケーション サービス用のブルーグリーン デプロイメントを実装するために必要な主なステップを列挙してください。簡潔にまとめてください。

4. Kubernetes シークレット経由でアプリケーション サービスに提供する情報を 3 から 4 種類挙げてください。
5. シークレットの作成時に Kubernetes が受け入れるソースを挙げてください。

参考情報

この章で取り上げるトピックに関する追加情報へのリンクを以下に示します。

- ローリング更新の実行 (`https://bit.ly/2o2okEQ`)
- ブルーグリーン デプロイメント (`https://bit.ly/2r2IxNJ`)
- Kubernetes 内のシークレット (`https://bit.ly/2C6hMZF`)

12
コンテナ化されたアプリケーションをクラウドで実行する

前章では、マルチサービス アプリケーションを Kubernetes クラスターにデプロイする方法を学習しました。そこでは、当該アプリケーションのアプリケーションレベル ルーティングを構成し、ダウンタイムを回避する戦略を使用してサービスを更新しました。最後に、Kubernetes シークレットを使用して実行中のサービスに機密データを提供しました。

この章では、コンテナ化された複雑なアプリケーションを Microsoft Azure 上のホストされた Kubernetes クラスターにデプロイする方法を示します。ここでは、いわゆる **Azure Kubernetes Service** (**AKS**) オファリングを使用します。AKS は Kubernetes クラスターを完全に管理するため、ユーザーはアプリケーションのデプロイ、実行、さらにアップグレード (必要な場合) に集中できます。

この章では次のようなトピックを取り上げます。

- Azure で、完全に管理された Kubernetes クラスターを作成する
- Docker イメージを Azure コンテナ レジストリにプッシュする
- アプリケーションを Kubernetes クラスターにデプロイする
- Pets アプリケーションをスケーリングする
- クラスターとアプリケーションを監視する
- ダウンタイムを発生させずにアプリケーションをアップグレードする
- Kubernetes をアップグレードする
- AKS で実行中のアプリケーションをデバッグする
- クリーンアップする

この章を読み終えると、次のことができるようになります。

- 完全に管理された Kubernetes クラスターを AKS でプロビジョニングする
- 複数コンテナをベースにしたアプリケーションをクラスターにデプロイする
- Kubernetes クラスターと、そこで実行されているアプリケーションの正常性を監視する
- AKS でアプリケーションのローリング アップデートを実行する
- AKS の Kubernetes クラスターで動作する Node.js アプリケーションをインタラクティブにデバッグする

技術的要件

ここでは、Microsoft Azure でホストされている AKS を使用します。そのため、Azure のアカウントが必要です。既存のアカウントをお持ちでない場合は、`https://azure.microsoft.com/en-us/services/kubernetes-service/` から無料の試用アカウントをリクエストできます。

Azure を利用するには、さらに Azure CLI が必要です。お使いのコンピューターで Azure CLI の最新バージョンを使用できることを確認してください。本書はコンテナに関する書籍なので、CLI をネイティブにインストールするのではなく、CLI のコンテナ化されたバージョンを使用します。

さらに、GitHub の `labs` リポジトリにある `ch12` および `ch12-dev-spaces` フォルダも使用します。コード ファイルへのリンクは `https://github.com/appswithdockerandkubernetes/labs/tree/master` にあります。

Azure で、完全に管理された Kubernetes クラスターを作成する

これまでは、ローカルの node-1 Kubernetes クラスターを使用して、アプリケーションをデプロイしてきました。これは、Kubernetes 上で実行される予定のコンテナ化されたアプリケーションの開発、テスト、およびデバッグには適しています。ただし、私たちの最終目標は、完成したアプリケーションを本番環境にデプロイすることです。この目標を達成するにはいくつかの方法があります。たとえば、可用性が高くスケーラブルな Kubernetes クラスターをセルフホスティングすることはできますが、それは簡単ではありません。この章では、可用性と拡張性の高い Kubernetes クラスターのプロビジョニングとホスティングという難しい作業をマイクロソフトに委託し、Azure 上で動作する AKS を使用します。

次に、Pets アプリケーションをこのクラスターにデプロイします。デプロイ後は、クラスターとクラスター上で動くアプリケーションの動作を監視する必要があります。その後、アプリケーションの Web パーツをスケーリングして更新し、ローリング アップデートを開始します。このアップデートではダウンタイムは発生しません。

最後に、AKS の Dev Spaces を使用して、Node.js フロントエンドなどのアプリケーションをクラウドで実行しながらリモート デバッグする方法について説明します。

AKS は、3 つの異なる方法で使用および操作できます。真っ先に挙げられるのが、Azure ポータルが提供するグラフィカルな Web UI を使用する方法です。次の 2 つは、自動化に適しています。

- Azure CLI を使用して、Kubernetes クラスターを `kubectl` と組み合わせてプロビジョニングおよび管理し、クラスター内でアプリケーションのデプロイ、スケーリング、アップデートなどを行う方法。
- Terraform や Azure リソース テンプレートなどのツールを使用して同じことを行う方法。

この章では、Azure CLI と `kubectl` を重点的に取り上げます。

Azure CLI の実行

前述のように、Microsoft Azure を利用するには Azure CLI が必要です。AKS 関連のすべてのリソースにアクセスするには、CLI の新バージョンまたは最新バージョンを使用する必要があります。今回はコンピューターで CLI をネイティブに実行するのではなく、CLI のコンテナ化されたバージョンを使用します。マイクロソフトは、CLI がインストールされたコンテナを作成しています。このケースでは、コンテナ内にある Docker (`docker`) と Kubernetes CLI (`kubectl`) も使用する必要がありますが、これらのツールはデフォルトではインストールされていません。そこで、マイクロソフトのイメージを基に、ツールがインストールされる新しい Docker イメージの作成に使用できる Dockerfile を作成しました。次の手順に従ってください。

1. 新しいターミナル ウィンドウを開き、`labs` フォルダの `ch12` サブフォルダに移動します。

```
cd ~/labs/ch12
```

2. 次のコマンドを実行して、カスタマイズされた Azure CLI コンテナを作成します。

```
docker image build -t custom-azure-cli:v1 azure-cli
```

3. このイメージが作成されたら、次のコマンドを実行することで、Azure、Docker、および Kubernetes CLI の最新バージョンでコンテナをインタラクティブに実行できます。

```
docker container run --rm -it \
 -v /var/run/docker.sock:/var/run/docker.sock \
    -v $(pwd):/src \
 custom-azure-cli:v1
```

コンテナ内から任意の Docker コマンドを実行できるように Docker ソケットをコンテナにマウントする方法に注目してください。また、現在のフォルダをコンテナ内の `/src` フォルダにマウントして、この章のソース コードにアクセスできるようにします。

コンテナ インスタンスが開始されると、次のプロンプトが表示されます。

```
bash-4.4#
```

コンテナをテストするには、次のコマンドを使用します。

```
bash-4.4# docker version
Client:
 Version: 18.06.0-ce
 API version: 1.38
 Go version: go1.10.3
 Git commit: 0ffa825
 Built: Wed Jul 18 19:04:39 2018
 OS/Arch: linux/amd64
 Experimental: false
Server:
 Engine:
 Version: 18.06.0-ce
 API version: 1.38 (minimum version 1.12)
 Go version: go1.10.3
 Git commit: 0ffa825
 Built: Wed Jul 18 19:13:46 2018
 OS/Arch: linux/amd64
 Experimental: true
```

`kubectl` コマンドを実行すると、次のように出力されます。

```
bash-4.4# kubectl version
Client Version: version.Info{Major:" 1" , Minor:" 11" , GitVersion:"
v1.11.2" , GitCommit:" bb9ffb1654d4a729bb4cec18ff088eacc153c239" ,
GitTreeState:" clean" , BuildDate:" 2018-08-07T23:17:28Z" , GoVersion:"
go1.10.3" , Compiler:" gc" , Platform:" linux/amd64" }
Server Version: version.Info{Major:" 1" , Minor:" 9" , GitVersion:"
v1.9.9" , GitCommit:" 57729ea3d9a1b75f3fc7bbbadc597ba707d47c8a" ,
GitTreeState:" clean" , BuildDate:" 2018-06-29T01:07:01Z" , GoVersion:"
go1.9.3" , Compiler:" gc" , Platform:" linux/amd64" }
```

この出力からは、クライアントで Docker `18.06.0-ce` と Kubernetes `v1.11.2` が実行され、サーバーでは `v1.9.9` が実行されていることがわかります。

Azure CLI を使用して Microsoft Azure のアカウントで作業するには、まずアカウントにログインする必要があります。そのためには、`bash-4.4#` コマンド プロンプトで次のように入力します。

```
az login
```

すると、次のようなメッセージが表示されます。

```
To sign in, use a web browser to open the page https://microsoft.com/
devicelogin and enter the code CMNB2TSND to authenticate. (サインイン
するには、Web ブラウザで https://microsoft.com/devicelogin を開き、コード
CMNB2TSND を入力して認証を受けてください。)
```

認証に成功すると、ターミナル ウィンドウに次のような応答が表示されます。

```
[
 {
" cloudName" :" AzureCloud" ,
" id" :" 186760ad-9152-4499-b317-xxxxxxxxxxxx" ,
" isDefault" : true,
" name" :" xxxxxxxxx" ,
" state" :" Enabled" ,
" tenantId" :" f5e90e29-00df-4ea6-b8a4-xxxxxxxxxxxx" ,
" user" : {
" name" :" gnschenker@xxxxx.xxx" ,
" type" :" user"
 }
 }
]
```

これで Azure 全体、特に AKS で作業を開始する準備が整いました。

Azure リソース グループ

Azure では、**リソース グループ**というコンセプトが重要です。リソース グループはコンテナの一種で、そこには論理的な共通点があればどのような種類のクラウド リソースでも入れることができます。本書で取り上げるケースでは、AKS でプロビジョニングする Kubernetes クラスターを構成するすべての要素が同じリソース グループに属しています。つまり、最初の段階はこのようなグループを作成することです。それには Azure CLI を使用します。グループに `pets-group` という名前を付け、`westeurope` リージョンに置きます。リソース グループを作成するには、次のコマンドを使用します。

```
az group create --name pets-group --location westeurope
```

このコマンドに対する応答は、次のようになります。

```
{
" id" :" /subscriptions/186760ad-9152-4499-b317-xxxxxxxxxxxx/
resourceGroups/pets-group" ,
" location" :" westeurope" ,
" managedBy" : null,
" name" :" pets-group" ,
" properties" : {
" provisioningState" :" Succeeded"
 },
```

サービス グループを作成したら、AKS クラスターが他の Azure リソースとやり取りできるように、サービス プリンシパルを作成する必要があります。そのためには、次のコマンドを使用します。

```
az ad sp create-for-rbac \
    --name pets-principal \
    --password adminadmin \
    --skip-assignment
```

この結果は次のように出力されます。

```
{
 " appId" :" a1a2bdbc-ba07-49bd-ae77-fb8b6948869d" ,
 " displayName" :" azure-cli-2018-08-27-19-26-20" ,
 " name" :" http://pets-principal" ,
 " password" :" adminadmin" ,
 " tenant" :" f5e90e29-00df-4ea6-b8a4-ce8553f10be7"
}
```

この出力では `appId` と `password` が重要です。以後のセクションに出てくるコマンドでも使用できるように、この 2 つを書き留めておいてください。

Kubernetes クラスターのプロビジョニング

やっと、AKS で Kubernetes クラスターをプロビジョニングする準備ができました。プロビジョニングには、次のコマンドを使用します。

```
az aks create \
  --resource-group pets-group \
  --name pets-cluster \
  --node-count 1 \
  --generate-ssh-keys \
  --service-principal <appId> \
  --client-secret <password>
```

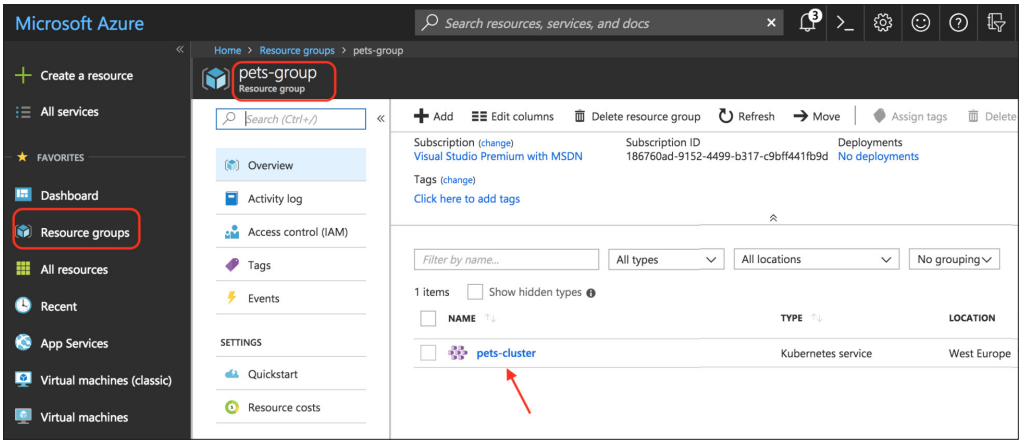
<appId> と <password> はプレースホルダーなので、サービス プリンシパルの作成後にメモした値で置き換える必要があります。また、ここで作成する Kubernetes クラスターには当面はワーカー ノードが 1 つだけであることにも注意してください。後で、Azure CLI を使用してこのクラスターをスケールアップ (またはスケールダウン) することができます。

前述のコマンドに対する応答は次のようになります。

```
SSH key files '/root/.ssh/id_rsa' and '/root/.ssh/id_rsa.pub' have been
generated under ~/.ssh to allow SSH access to the VM. (~/.ssh の下に
SSH キーファイル /root/.ssh/id_rsa] と [/root/.ssh/id_rsa.pub が生成され、
VM への SSH アクセスが許可されました。)If using machines without permanent
storage like Azure Cloud Shell without an attached file share, back up
your keys to a safe location
 - Running .. (Azure Cloud Shell のような永続ストレージを持たないマシンを添付
ファイルを共有せずに使用している場合は、キーを安全な場所にバックアップしてください。
- 実行中 ..)
```

プロビジョニングが完了するには数分かかります (その間は [Running] (実行中) のステータスが表示されます)。

コマンドが完了するのを待っている間に、ブラウザの新規ウィンドウを開いて Azure ポータル (https://portal.azure.com) に移動し、Azure アカウントにログインすることができます。認証に成功すると、[**リソース グループ**] (Resource groups) オプションに移動して、リソース グループのリストに pets-group リソース グループがあるのを確認できます。このグループをクリックすると、グループ内のリソースとして pets-cluster がリストに表示されます。



AKS の Kubernetes クラスター

クラスターが作成されると、ターミナル ウィンドウの最終出力は次のようになります。

```
{ " aadProfile" : null,
  " addonProfiles" : null,
  " agentPoolProfiles" : [
     {
```

```
      " count" : 1,
      " maxPods" : 110,
      " name" :" nodepool1" ,
      " osDiskSizeGb" : null,
      " osType" :" Linux" ,
      " storageProfile" :" ManagedDisks" ,
      " vmSize" :" Standard_DS1_v2" ,
      " vnetSubnetId" : null
    }
  ],
  " dnsPrefix" :" pets-clust-pets-group-186760" ,
  " enableRbac" : true,
  " fqdn" :" pets-clust-pets-group-186760-d706beb4.hcp.westeurope.
azmk8s.io" , " id" :" /subscriptions/186760ad-9152-4499-b317-
c9bff441fb9d/resourcegroups/pets-group/providers/Microsoft.
ContainerService/managedClusters/pets-cluster" ,
  " kubernetesVersion" :" 1.9.9" ,
  " linuxProfile" : {
    " adminUsername" :" azureuser" ,
    " ssh" : {
      " publicKeys" : [
        {
          " keyData" :" ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDMp
2BFCRUo7v1ktVQa57ep7zLg7HEjsRQAkb7UnovDXrLg1nBuzMslHZY3mJ5ulxU00
YWeUuxObeHjRh+ZJHc4+xKaDV8M6GmuHjD8HJnw5tsCbV8w/A+5oUOECaeJn5sQMCkmS
DovmDQZchAjLjVHQLSTiEqjLYmjjqYmhqYpO2vRsnZXpelRrlmfNWoSV5J3L7/
hayI2fg35X/H4xnx1sm403O9pwyEKYYBFfNzCXigNnqyBvxOqwURZUW/caIpTqAhS6
K+D1xPa2w7y1A5qcZS++SnJOHCHyRKZ3UQ4BVZTSejBhxYTr5/dgJE+LEvLk2i
YUo4kUmbxDSVssnWJ"
        }
      ]
    }
  },
  " location" :" westeurope" ,
  " name" :" pets-cluster" ,
  " networkProfile" : {
    " dnsServiceIp" :" 10.0.0.10" ,
    " dockerBridgeCidr" :" 172.17.0.1/16" ,
    " networkPlugin" :" kubenet" ,
    " networkPolicy" : null,
    " podCidr" :" 10.244.0.0/16" ,
    " serviceCidr" :" 10.0.0.0/16"
  },
  " nodeResourceGroup" :" MC_pets-group_pets-cluster_westeurope" ,
  " provisioningState" :" Succeeded" ,
  " resourceGroup" :" pets-group" ,
  " servicePrincipalProfile" : {
    " clientId" :" a1a2bdbc-ba07-49bd-ae77-fb8b6948869d" ,
    " secret" : null
  },
```

```
  " tags" : null,
  " type" :" Microsoft.ContainerService/ManagedClusters"
}
```

クラスターにアクセスするには、次のコマンドで kubectl を構成する必要があります。

```
az aks get-credentials --resource-group pets-group --name pets-cluster
```

コマンドが正常に機能すると、ターミナルに次のような応答が生成されます。

```
Merged" pets-cluster" as current context in /root/.kube/config
```

これで、kubectl を使用してクラスターのすべてのノードを取得することができます。

```
kubectl get nodes
```

この結果は次のように表示されます。

```
NAME                          STATUS    ROLES     AGE       VERSION
aks-nodepool1-54489083-0      Ready     agent     13m       v1.9.9
```

このクラスターは 1 つのワーカー ノードで構成され、その Kubernetes バージョンは 1.9.9 で、起動後 13 分経過しているということがわかります。注意深い読者は、Kubernetes のバージョンが少し古くなっていることに気づいたかもしれません。執筆時に AKS で入手可能な最新のバージョンは 1.11.2 です。これはすばらしいことで、クラスターを Kubernetes の新しいバージョンにアップグレードする方法が表示される可能性があります。

Docker イメージを Azure Container Registry (ACR) にプッシュする

コンテナ化されたアプリケーションを Azure 上でホストされている Kubernetes クラスターにデプロイする際、基礎となる Docker イメージを、同じセキュリティー コンテキストのクラスターに近い場所に格納することにはメリットがあります。つまり、ACR は良い選択肢です。アプリケーションをクラスターにデプロイする前に、必要な Docker イメージを ACR にプッシュします。

ACR の作成

ここでも Azure CLI を使用して、Azure 上に Docker イメージ用のレジストリを作成します。このレジストリは、この章の「Azure リソース グループ」セクションで作成した pet-group リソース グループにも属します。次のコマンドを使用すると、<registry-name> という名前のレジストリが作成されます。

```
az acr create --resource-group pets-group --name <registry-name>
--sku Basic
```

レジストリの名前は <registry-name> で、これは Azure 上で一意である必要があります。つまり、読者はそれぞれ固有の名前を考えなければならず、著者と同じ名前を使うことはできません。そのため、ここでは実際の名前の代わりに <registry-name> というプレースホルダーを使用しています。私の場合は、レジストリに gnsPetsRegistry という名前を付けました。

前記のコマンドは少し時間を要しますが、やがて次のような答えが返されます。

```
{
" adminUserEnabled" : false,
" creationDate" :" 2018-08-27T19:57:01.434521+00:00" ,
" id" :" /subscriptions/186760ad-9152-4499-b317-xxxxxxxxxxxx/
resourceGroups/pets-group/providers/Microsoft.ContainerRegistry/
registries/gnsPetsRegistry" ,
" location" :" westeurope" ,
" loginServer" :" gnspetsregistry.azurecr.io" ,
" name" :" gnsPetsRegistry" ,
" provisioningState" :" Succeeded" ,
" resourceGroup" :" pets-group" ,
" sku" : {
" name" :" Basic" ,
" tier" :" Basic"
 },
" status" : null,
" storageAccount" : null,
" tags" : {},
" type" :" Microsoft.ContainerRegistry/registries"
}
```

この出力の loginServer エントリに注目してください。このキーの値は、Docker イメージのプレフィックスに必要な URL に対応しています。このイメージにタグを付けると、それを ACR にプッシュできます。

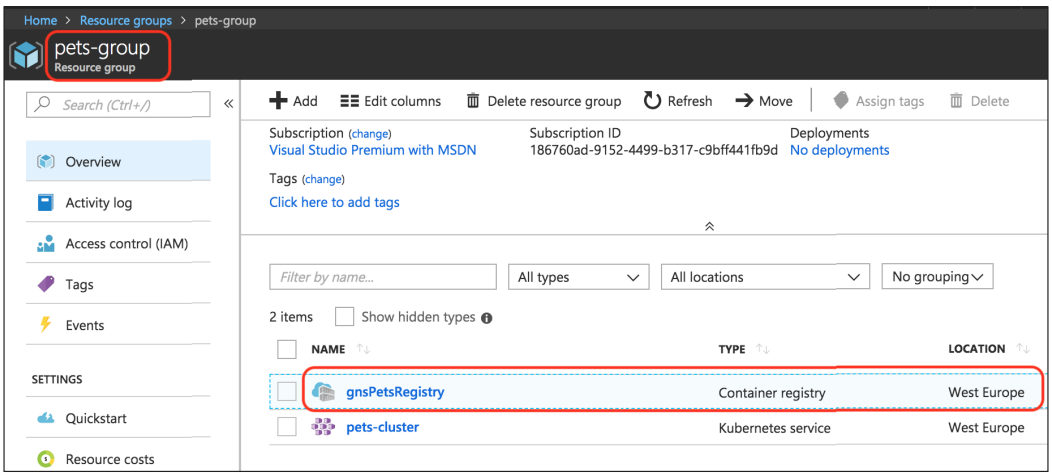
レジストリを作成したら、次のコマンドを使用してレジストリにログインできます。

```
az acr login --name <registry-name>
```

答えは次のようになります。

```
Login Succeeded
```

Azure ポータルでは、今作成したレジストリが `pets-group` の追加リソースとしてリストされているはずです。



ACR の Pets レジストリ

これで、ACR に画像をプッシュする準備が整いました。

Docker イメージのタグ付けとプッシュ

これまでの章で、この章で使用する Docker イメージを作成しました。しかし、それらのイメージを ACR にプッシュできるようにするためには、まず、ACR 用の正しい URL プレフィックスでイメージにタグを付け直す必要があります。このプレフィックス (別名 **ACR ログイン サーバー**。この名前はすでに、レジストリの作成に使用したコマンドの出力で確認しています) を取得するためには、次のコマンドを使用します。

```
az acr list --resource-group pets-group --query“ [].
{acrLoginServer:loginServer}” --output table
```

このコマンドを実行した結果は次のとおりです。

```
AcrLoginServer
-------------------------
gnspetsregistry.azurecr.io
```

これで、ACR にレジストリがあることがわかるので、やっと Docker イメージにタグを付けることができます。この例で gnspetsregistry.azurecr.io という URL を使用すると、Docker イメージ pets-web:v1 に対するコマンドは次のようになります。

```
docker image tag pets-web:v1 gnspetsregistry.azurecr.io/pets-web:v1
```

これが完了すると、次のコマンドでイメージをプッシュします。

```
docker image push gnspetsregistry.azurecr.io/pets-web:v1
```

すべてが正常に動作すると、次のように表示されます。

```
The push refers to repository [gnspetsregistry.azurecr.io/pets-web]
9d5c9e1e5f97: Pushed
39f3a72e04a3: Pushed
3177c088200b: Pushed
5f896b8130b3: Pushed
287ef32bfa90: Pushed
ce291010afac: Pushed
73046094a9b8: Mounted from alpine
v1: digest: sha256:9a32931874f4fdf5... size: 1783
```

今度は Docker イメージ `pets-db:v1` に対して同じコマンドを実行します。

```
docker image tag pets-db:v1 gnspetsregistry.azurecr.io/pets-db:v1
docker image push gnspetsregistry.azurecr.io/pets-db:v1
```

サービス プリンシパルの構成

最終ステップとして、この章の前半で作成した AKS サービス プリンシパルに、コンテナ レジストリのイメージにアクセスしてイメージを引き出すために必要な権限を付与する必要があります。まず、次のコマンドを使用して、ACR リソース ID (`<acrId>`) を取得します。

```
az acr show --resource-group pets-group \
    --query“ id” \
    --output tsv \
    --name <registry-name>
```

この結果は次のように出力されます。

```
/subscriptions/186760ad-9152-4499-b317-xxxxxxxxxxxx/resourceGroups/
pets-group/providers/Microsoft.ContainerRegistry/registries/
gnsPetsRegistry
```

この情報があれば、`<appId>` によって識別されるサービス プリンシパルに、`<acrId>` によって識別されるコンテナ レジストリへのアクセスに必要な読み取り権限を割り当てることができます。

```
az role assignment create --assignee <appId> --scope <acrId>
--role Reader
```

ここで、<appId> は以前に保管したプリンシパルのアプリケーション ID で、<acrId> は ACR リソース ID です。この出力は次のようになります。

```
{
" canDelegate" : null,
" id" :" /subscriptions/.../roleAssignments/1b7c2a63-c4d3-41a9-a1bc-
bd9d65966f43" ,
" name" :" 1b7c2a63-c4d3-41a9-a1bc-bd9d65966f43" ,
" principalId" :" ab5fe519-3982-4aac-95e0-761c242aa61b" ,
" resourceGroup" :" pets-group" ,
" roleDefinitionId" :" /subscriptions/.../roleDefinitions/acdd72a7-3385-
48ef-bd42-f606fba81ae7" ,
" scope" :" /subscriptions/.../registries/gnspetsregistry" ,
" type" :" Microsoft.Authorization/roleAssignments"
}
```

アプリケーションを Kubernetes クラスターにデプロイする

Docker イメージが正常に ACR にプッシュされたら、AKS で Kubernetes クラスターにアプリケーションをデプロイする準備ができたことになります。このためには、マニフェスト ファイル pets.yaml を使用します。これはソース コードの一部で ch12 フォルダにあります。このファイルをエディターで開き、環境に合わせて内容を変更してください。Docker イメージのプレフィックスになっているリポジトリ URL を、各ユーザーに固有の URL に置き換えます。

次のスニペットでは、デフォルト値 gnsPetsRegistry が表示されていますが、それをご自分の <registry name> 値に置き換えてください。

```
containers:
- name: web
  image: gnsPetsRegistry.azurecr.io/pets-web:v1
```

変更を保存します。次に、kubectl を使用してマニフェストを適用します。

```
kubectl apply -f /src/pets.yaml
```

このコマンドを実行すると、次のように出力されます。

```
deployment.apps/db created
service/db created
deployment.apps/web created
service/web created
```

Pets アプリケーションのフロントエンドには、`LoadBalancer` タイプのサービスを作成します。このアプリケーションには、AKS によってパブリック IP アドレスが割り当てられます。これには少し時間がかかります。状況を確認するには次のコマンドを使用します。

```
kubectl get service web --watch
```

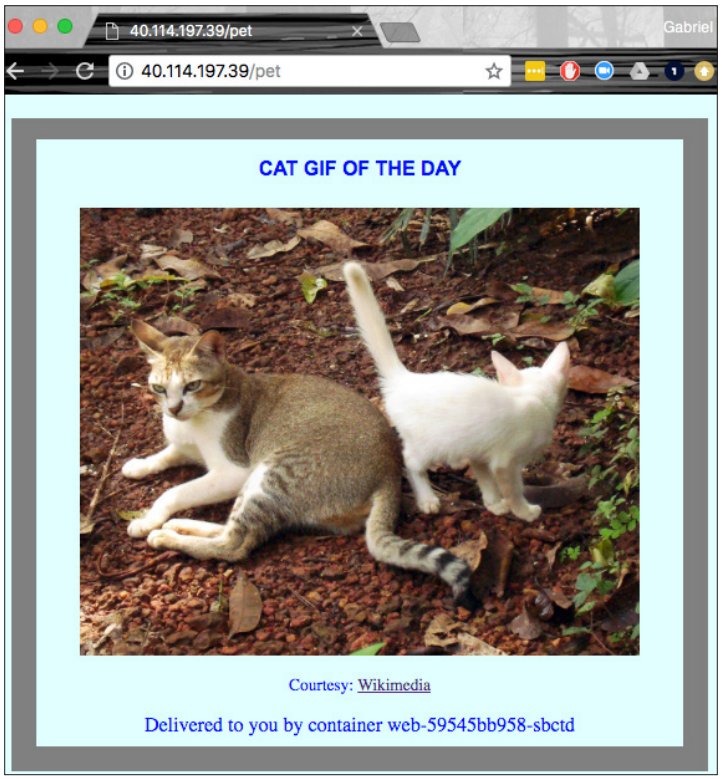
最初は次のように出力されます (`EXTERNAL-IP` カラムの `pending` という出力を参照)。

```
NAME      TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
web       LoadBalancer   10.0.49.36     <pending>      80:31035/TCP     41s
```

デプロイメントが完了したら、`<pending>` がパブリック IP アドレスに置き換えられ、Pets アプリケーションへのアクセスに使用できるようになります。

```
NAME    TYPE           CLUSTER-IP   EXTERNAL-IP     PORT(S)          AGE
web     LoadBalancer   10.0.49.36   40.114.197.39   80:31035/TCP     2m
```

ブラウザ ウィンドウで `http://<IP address>/pet` を開くと、かわいい猫の画像が表示されます。



AKS の Kubernetes で動作する Pets アプリケーション

これで、複雑なアプリケーションが AKS の Kubernetes クラスターに正常にデプロイされました。

Pets アプリケーションをスケーリングする

Pets アプリケーションのスケーリングには、次の 2 つの方法があります。

- 1 つは、アプリケーション インスタンスの数をスケーリングする方法です。
- もう 1 つは、クラスター内のワーカー ノードの数をスケーリングする方法です。

もちろん、2 番目のアプローチは、同時にアプリケーション インスタンスの数も増やして、追加のインスタンスが新しく追加されたクラスター ノードのリソースを活用する場合にのみ意味を持ちます。

アプリケーション インスタンス数のスケーリング

特定のアプリケーション サービスが実行されているインスタンスの数をスケーリングするのは簡単です。`kubectl` を使えばよいのです。Pets アプリケーションが大成功を収めて Web インスタンスの数を 3 つに増やす必要があるのに、データベースが依然として 1 つのインスタンスでしか着信トラフィックを処理できないとしましょう。まず、クラスター内で現在実行中のポッドを確認します。

```
bash-4.4# kubectl get pods
NAME                   READY   STATUS    RESTARTS  AGE
db-6746668f6c-wdscl    1/1     Running   0         2h
web-798745b679-8kh2j   1/1     Running   0         2h
```

ここでは、2 つのサービス (`web` と `db`) のそれぞれで 1 つのポッドが起動して実行中であることが確認できます。これまでのところ、どちらのポッドも再起動されていません。では、Web デプロイメントのインスタンスを 3 つに増やしてみます。

```
bash-4.4# kubectl scale --replicas=3 deployment/web
deployment.extensions/web scaled
```

スケーリング操作の結果は次のとおりです。

```
bash-4.4# kubectl get pods
NAME                  READY  STATUS   RESTARTS AGE
db-6746668f6c-wdscl   1/1    Running   0       2h
web-74dbc994bc-6f7qh  1/1    Running   0       14s
web-74dbc994bc-l99bh  1/1    Running   0       1m
web-74dbc994bc-rz8vs  1/1    Running   0       14s
```

Pets アプリケーションを実行しているブラウザで表示を数回更新して、コンテナ インスタンスの ID がその都度変わることを確認してください。これは Kubernetes ロード バランサーが動作して、呼び出しを別々の `Web` インスタンスに配分しているためです。

Web サービスのさまざまなインスタンスを取得する最善の方法は、ブラウザのシークレット ウィンドウを複数開くことです。

クラスター ノード数のスケーリング

これまでは、1 つのクラスター (ワーカー) ノードだけで作業してきました。Azure CLI を使用すると、ノードの数を増やしてアプリケーションの計算能力を強化できます。クラスターを 3 つのノードに拡張してみましょう。

```
az aks scale --resource-group=pets-group --name=pets-cluster
--node-count 3
```

これには数分かかりますが、終了したら、ターミナルに次のような応答が表示されます。

```
{
 " aadProfile" : null,
 " addonProfiles" : null,
 " agentPoolProfiles" : [
    {
     " count" : 3,
     " maxPods" : 110,
     " name" :" nodepool1" ,
" osDiskSizeGb" : null,
" osType" :" Linux" ,
" storageProfile" :" ManagedDisks" ,
" vmSize" :" Standard_DS1_v2" ,
" vnetSubnetId" : null
 }
 ],
" dnsPrefix" :" pets-clust-pets-group-186760" ,
" enableRbac" : true,
" fqdn" :" pets-clust-pets-group-186760-d706beb4.hcp.westeurope.
azmk8s.io" ,
" id" :" /subscriptions/186760ad-9152-4499-b317-xxxxxxxxxxxx/
resourcegroups/pets-group/providers/Microsoft.ContainerService/
managedClusters/pets-cluster" ,
" kubernetesVersion" :" 1.9.9" ,
" linuxProfile" : {
" adminUsername" :" azureuser" ,
" ssh" : {
```

```
" publicKeys" : [
 {
" keyData" :" ssh-rsa AAAAB3NzaC..."
 }
 ]
 }
 },
" location" :" westeurope" ,
" name" :" pets-cluster" ,
" networkProfile" : {
" dnsServiceIp" :" 10.0.0.10" ,
" dockerBridgeCidr" :" 172.17.0.1/16" ,
" networkPlugin" :" kubenet" ,
" networkPolicy" : null,
" podCidr" :" 10.244.0.0/16" ,
" serviceCidr" :" 10.0.0.0/16"
 },
" nodeResourceGroup" :" MC_pets-group_pets-cluster_westeurope" ,
" provisioningState" :" Succeeded" ,
" resourceGroup" :" pets-group" ,
" servicePrincipalProfile" : {
" clientId" :" a1a2bdbc-ba07-49bd-ae77-xxxxxxxxxxxx" ,
" secret" : null
 },
" tags" : null,
" type" :" Microsoft.ContainerService/ManagedClusters"
}
```

次のコマンドで、ワーカー ノードのプロビジョニングの進捗状況を確認できます。

```
bash-4.4# kubectl get nodes --watch
aks-nodepool1-54489083-0 Ready agent 1d v1.9.9
aks-nodepool1-54489083-2 NotReady agent 0s v1.9.9
...
aks-nodepool1-54489083-0 Ready agent 1d v1.9.9
aks-nodepool1-54489083-2 Ready agent 2m v1.9.9
aks-nodepool1-54489083-1 Ready agent 20s v1.9.9
```

もちろん、ワーカー ノードを追加しても、アプリケーションのサービス インスタンスが自動的に再配布されることはありません。これを実現するには、たとえば、Web サービスをもう一度スケーリングする必要があります。

```
bash-4.4# kubectl scale --replicas=5 deployment/web
deployment.extensions/web scaled
```

こうして広範囲に及ぶ出力を取得することで、ポッドがどのノードに到達したかを確認できます。

```
bash-4.4# kubectl get pods --output='wide'
NAME                  READY   STATUS    RESTARTS   AGE    IP
NODE ...
db-6746668f6c-wdscl   1/1     Running   0          3h     10.244.0.24
aks-nodepool1-54489083-0 ...
web-59545bb958-2v4zp  1/1     Running   0          2m     10.244.1.3
aks-nodepool1-54489083-2 ...
web-59545bb958-7mpfx  1/1     Running   0          31m    10.244.0.31
aks-nodepool1-54489083-0 ...
web-59545bb958-9mc6m  1/1     Running   0          2m     10.244.1.2
aks-nodepool1-54489083-2 ...
web-59545bb958-sbctd  1/1     Running   0          35m    10.244.0.29
aks-nodepool1-54489083-0 ...
web-59545bb958-tvthv  1/1     Running   0          31m    10.244.0.30
aks-nodepool1-54489083-0 ...
```

3 つのポッドが最初のノードにあり、2 つ (追加のポッド) が 3 番目のノードにデプロイされていることがわかります。

クラスターとアプリケーションを監視する

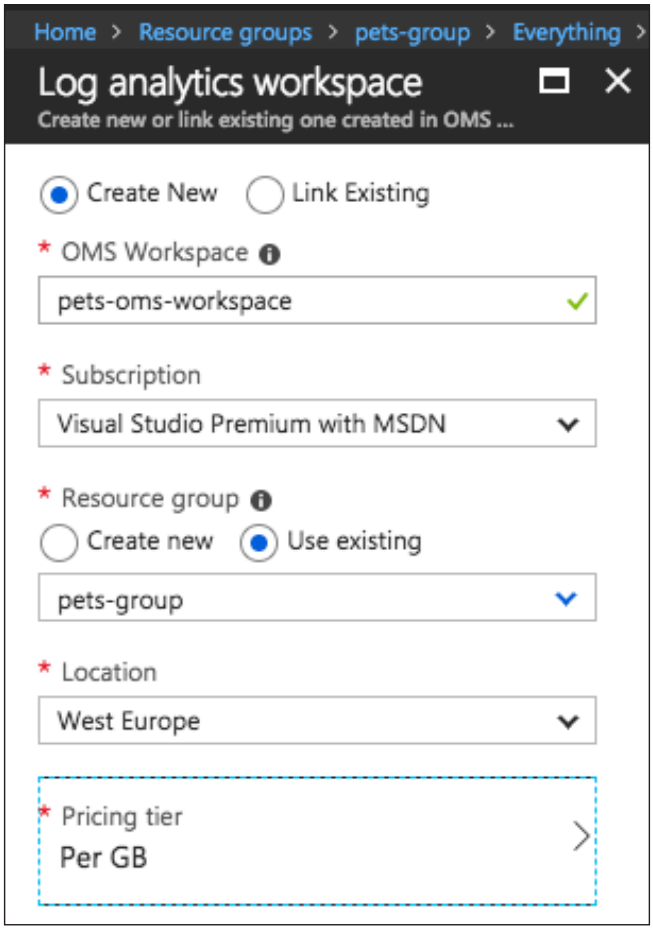
このセクションでは、Kubernetes クラスターの 3 つの異なる要素について説明します。内容は次のとおりです。

- コンテナの正常性を監視する
- Kubernetes マスターのログを表示する
- 各ワーカー ノードにインストールされた kublets のログを表示する

Log Analytics ワークスペースの作成

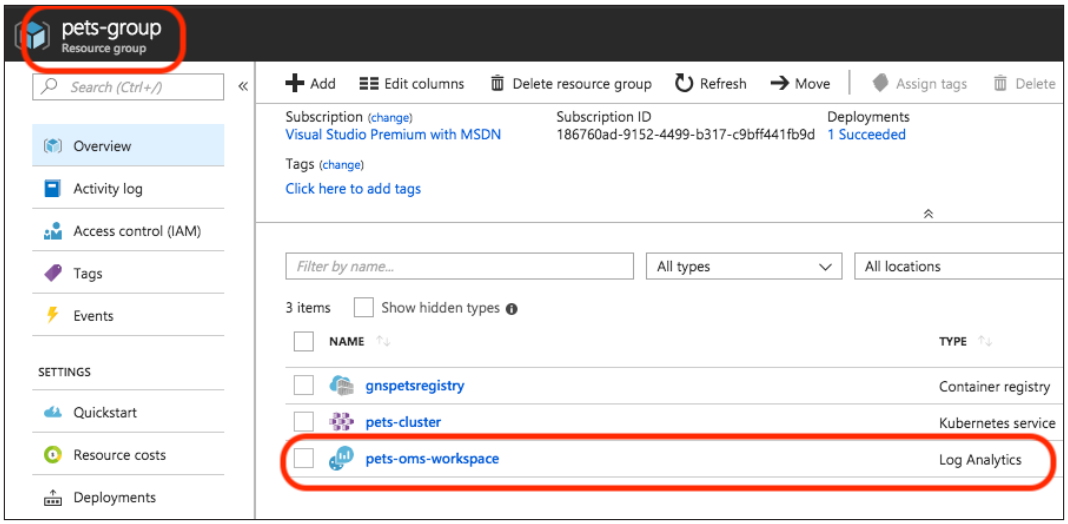
クラスターが生成する監視データとログ データを格納するには、Log Analytics ワークスペースが必要です。Azure ポータル GUI を使用して、pets-group リソース グループに以下の手順でワークスペースを作成します。

1. ポータルで **[+ リソースの作成]** (+Create a resource) オプションをクリックします。
2. **[Log Analytics]** オプションを選択し、次に **[作成]** (Create) を選択します。次のスクリーンショットを参考に、必要なデータを入力します。



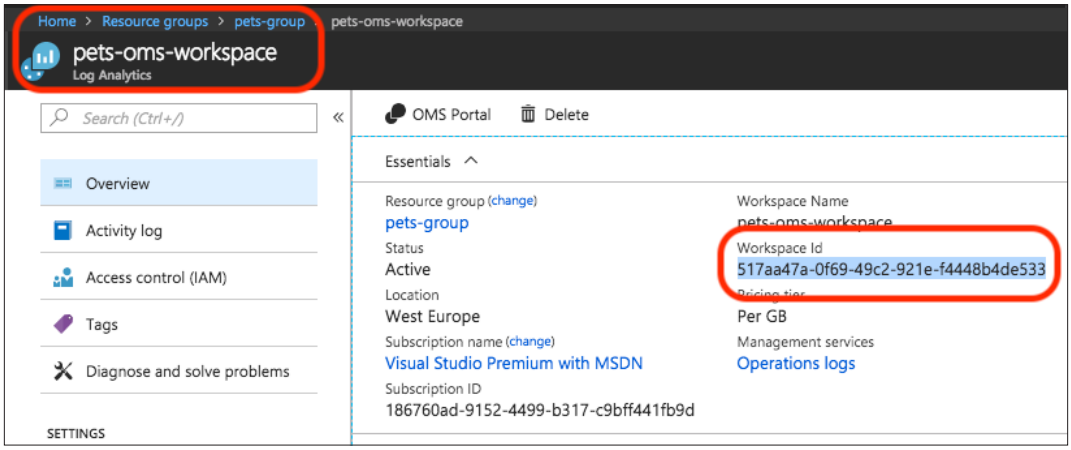
Log Analytics ワークスペースの作成

作成されたワークスペースは、`pets-group` リソース グループの概要に表示されます。



pets-group リソース グループの Log Analytics ワークスペース

3. `pets-oms-workspace` エントリをクリックすると、ワークスペースの詳細が表示されます。



Log Analytics ワークスペース pets-oms-workspace の詳細表示

今作成したワークスペースの `<workspace ID>` を書き留めておいてください。次のセクションで必要になります。

コンテナの正常性の監視

Azure CLI を使って `pets-cluster` の監視を有効にしましょう。これはクラスターを作成したときに実行できるのですが、後で実行することもできます。

クラスターの監視を有効にするには、やはり Azure CLI を使用します。使用するコマンドは次のとおりです。`<workspace ID>` を、先ほど作成したワークスペースの ID で必ず置き換えてください。

```
az aks enable-addons \
    -a monitoring \
    -g pets-group \
    -n pets-cluster \
    --workspace-resource-id <workspace ID>
```

このコマンドにはしばらく時間がかかるので、そのまま待っていてください。最終的には次のような応答が返されます (読みやすくするために短縮しています)。

```
 ...
“ properties” : {
  “ provisioningState” :“ Succeeded”
 },
 ...
```

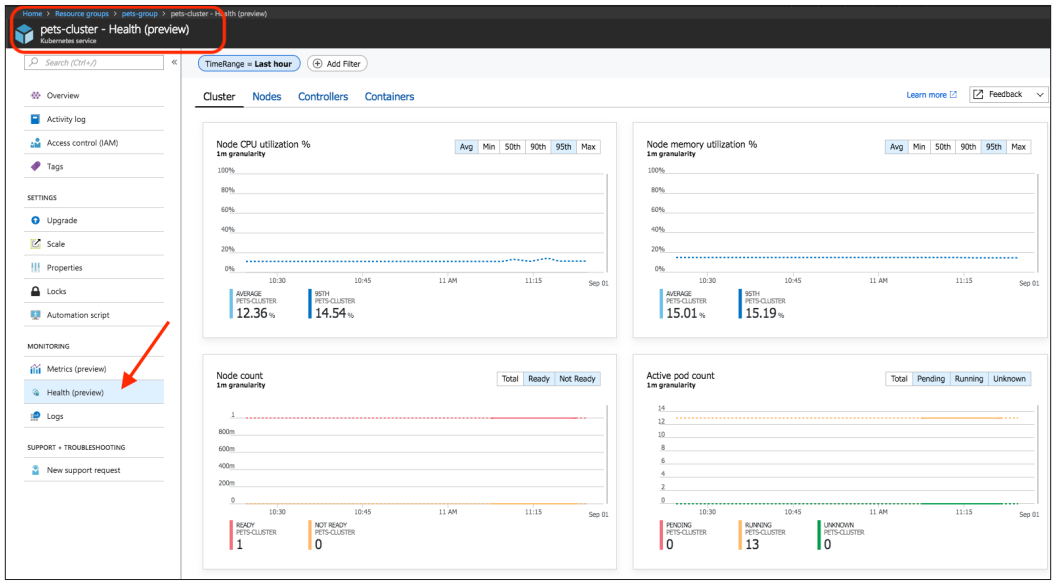
Azure ポータル GUI または Azure Resource Manager テンプレートを使用して監視を有効にすることもできます。

監視は **Log Analytics** エージェントがベースになっています。このエージェントは Kubernetes クラスターの各ノードで実行され、ネイティブの Kubernetes Metrics API が提供しているすべてのコントローラ、ノード、コンテナから、メモリおよびプロセッサのメトリックを収集します。エージェントに収集されたメトリックは、Log Analytics ワークスペースに転送され、格納されます。

クラスターの監視を有効にすると、Log Analytics エージェントが `kube-system` 名前空間に、ワーカー ノードごとに一度デプロイされていることが確認できます (3)。

```
bash-4.4# kubectl get ds omsagent --namespace=kube-system
NAME      DESIRED CURRENT READY UP-TO-DATE AVAILABLE
NODE SELECTOR                 AGE
omsagent   3       3       3     3         3    beta.kubernetes.io/
os=linux  1d
```

これで、Azure ポータルを開いてクラスターのメトリックを表示することができます。**[リソース グループ]** (Resource groups) | **[pets-group]** | **[pets-cluster]** と移動して、**[正常性 (プレビュー)]** (Health (preview)) オプションをクリックすると、次のスクリーンショットのような画面が表示されます。



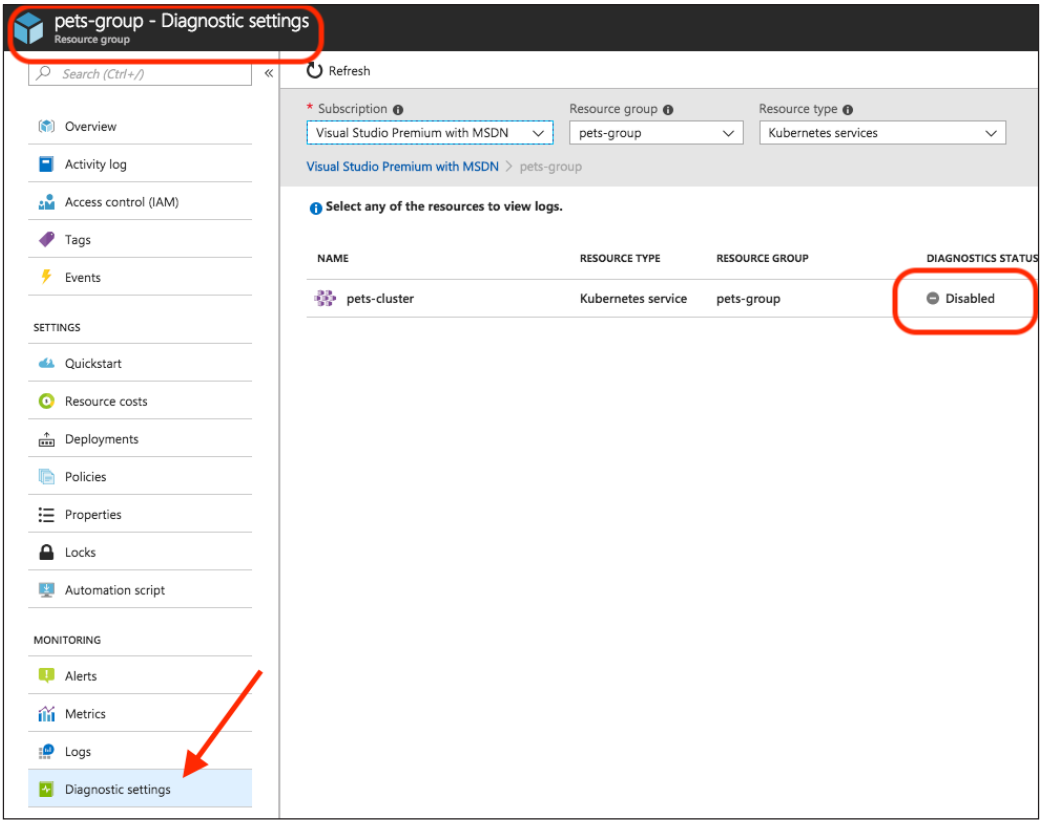
Microsoft Azure AKS の [正常性] (Health) ビュー

このビューでは、クラスター、ノード、コントローラ、コンテナごとに、CPU とメモリ使用率、およびノード数とアクティブなポッド数の集計メトリックを確認できます。

Kubernetes マスターのログの表示

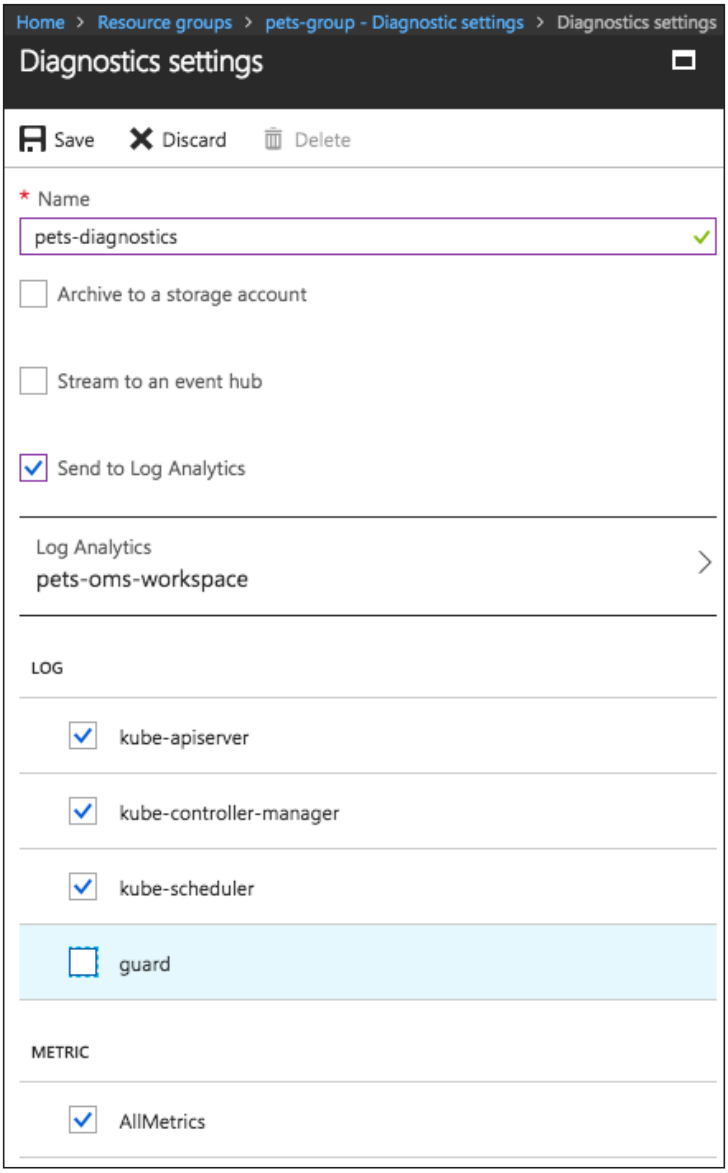
診断ログはデフォルトでは無効になっているので、まずログを有効にする必要があります。Log Analytics の `pets-oms-workspace` ワークスペースを、ログのターゲットとして使用できます。診断ログを有効にするために、今度は Azure ポータルを使用します。次の手順に従ってください。

1. `[pets-group]` リソース グループに移動して、左側の **[診断設定]** (Diagnostic settings) オプションを選択します。次のように表示されるはずです。



リソース グループ pets-group の診断ログ設定

2. `[pets-cluster]` エントリを選択して、**[診断を有効にする]** (Turn on diagnostics) リンクをクリックします。次のスクリーンショットを参考に、フォームに値を入力します。

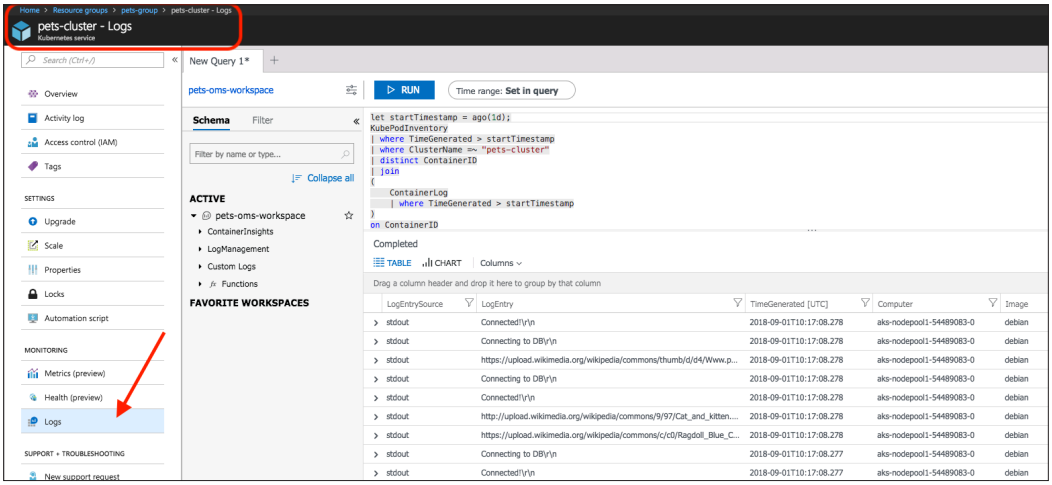


Kubernetes クラスターの診断ログの設定

3. **[Log Analytics に送信]** (Send to Log Analytics) チェックボックスが選択され、ログのターゲットとして `[pets-oms-workspace]` が選択されていることを確認してください。
4. **[保存]** (Save) ボタンをクリックして、設定を更新します。

次に進む前にシステムが関連情報を収集しますが、これには数分または数時間を要します。

ログを分析するには、`[pets-group]` リソース グループに移動して、`[pets-cluster]` エントリを選択します。次に左側の **[ログ]** (Logs) オプションを選択すると、次のスクリーンショットのような画面が表示されます。



Kubernetes クラスターとその上で実行されているコンテナによって生成されたログの分析

上記のスクリーンショットでは、ノード `aks-nodepool1-54489083-0` 上で Kubernetes を実行している `ssh-helper` コンテナによるログ エントリを生成したデフォルト クエリを確認できます。このコンテナについては次のセクションで紹介します。

これで、Azures Log Analytics が提供する豊富なクエリ言語を使用して、膨大な量のログ データをより詳細に把握できるようになりました。

kublet とコンテナのログの表示

場合によっては、Kubernetes クラスターで特定ノード上の kubelet のログを調べる必要があります。そのためには、その特定ノードへの SSH 接続を確立する必要があります。そうすれば、Linux ツール `journalctl` を使用してそのログにアクセスできます。

まず、kublet のログを調べるノード (または VM) を見つける必要があります。pets-cluster クラスター (westeurope リージョン内の pets-group リソース グループにあります) に含まれる VM をすべてリストアップしましょう。Azure では MC_<group name> _<cluster name>_<region name> というリソース グループが暗黙的に作成され、Kubernetes クラスターのすべてのリソース (VM を含む) はここに入れられます。したがって、本書の例ではグループ名は MC_pets-group_pets-cluster_westeurope になります。VM のリストを取得するコマンドは次のとおりです。

```
bash-4.4# az vm list --resource-group MC_pets-group_pets-cluster_
westeurope -o table

Name                       ResourceGroup
Location   Zones
-------------------------- ------------------------------------ -------
--- -------
aks-nodepool1-54489083-0 MC_pets-group_pets-cluster_westeurope
westeurope
```

これで、Azure CLI 経由でクラスターに接続する場合に使用するパブリック SSH キーを、この VM (名前は aks-nodepool1-54489083-0) に追加することができます (次のコマンドを使用します)。

```
bash-4.4# az vm user update \
 --resource-group MC_pets-group_pets-cluster_westeurope \
 --name aks-nodepool1-54489083-0 \
 --username azureuser \
 --ssh-key-value ~/.ssh/id_rsa.pub
```

今度は、この VM のアドレスを取得する必要があります。そのコマンドは次のとおりです。

```
bash-4.4# az vm list-ip-addresses --resource-group MC_pets-group_pets-
cluster_westeurope -o table

VirtualMachine             PrivateIPAddresses
-------------------------  --------------------
aks-nodepool1-54489083-0  10.240.0.4
```

上記の情報をすべて取得したうえで、この VM に対して SSH を使用する方法が必要になります。ローカル ワークステーションから直接実行するにはもう一手間必要ですが、簡単な方法は、Kubernetes クラスター内で ssh-helper というヘルパー コンテナをインタラクティブに実行することです。こうすれば、そのコンテナから VM に SSH を実行できます。このようなヘルパー コンテナを開始する kubectl コマンドは、次のとおりです。

```
bash-4.4# kubectl run -it --rm ssh-helper --image=debian
root@ssh-helper-86966767d-v2xqg:/#
```

このコンテナには SSH クライアントがインストールされていません。ここでインストールしましょう。この `helper` コンテナ内で、次のコマンドを実行します。

```
root@ssh-helper-86966767d-v2xqg:/# apt-get update && apt-get install
openssh-client -y
```

次のようにして、別のターミナルで Azure CLI コンテナに接続することもできます。

```
$ docker container exec azure-cli /bin/bash
```

次に、コンテナ内で次のコマンドを実行して、クラスターで実行中のポッドをすべて表示します。

```
bash-4.4# kubectl get pods
```

上記のコマンドを実行した結果を以下に示します。

```
bash-4.4# kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
db-6746668f6c-msv8m           1/1       Running   0          1d
ssh-helper-86966767d-v2xqg    1/1       Running   0          2m
web-74dbc994bc-5d2j6          1/1       Running   0          1d
web-74dbc994bc-ljml9          1/1       Running   0          1d
web-74dbc994bc-tj6p2          1/1       Running   0          1d
bash-4.4#
```

Kubernetes クラスターで実行中の SSH ヘルパー ポッド

次のステップは、プライベート SSH キーをポッドにコピーして、予想される場所に貼り付けることです。そのためには、次のコマンドを使用します。

```
bash-4.4# kubectl cp ~/.ssh/id_rsa ssh-helper-86966767d-v2xqg:/id_rsa
```

今度は `helper` コンテナの内部から、次のコマンドを使用してこの SSH キーへのアクセス権を変更する必要があります。

```
root@ssh-helper-86966767d-v2xqg:/# chmod 0600 id_rsa
```

これでやっと、ターゲット VM に SSH を実行する準備が整いました。

```
root@ssh-helper-86966767d-v2xqg:/# ssh -i id_rsa azureuser@10.240.0.4
```

画面には次のように表示されます。

```
The authenticity of host '10.240.0.4 (10.240.0.4)' can't be
established.
ECDSA key fingerprint is SHA256:pl03ZLFd0pkkPTtzDphSXCuNl0npBJO1JmUi
LI5aSzY.Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.240.0.4' (ECDSA) to the list of known
hosts.Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-1021-azure
x86_64)
```

```
 * Documentation: https://help.ubuntu.com * Management: https://
landscape.canonical.com
 * Support: https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

3 packages can be updated.
0 updates are security updates.


*** System restart required ***

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user“ root” ), use“ sudo
<command>” .
See“ man sudo_root” for details.

azureuser@aks-nodepool1-54489083-0:~$
```

成功です。これで、Kubernetes クラスターの目的のノードにリモートからアクセスできます。

ここまでで示したことは、ヘルパー コンテナまたはポッドを介してクラスターにアクセスすることで、これは、保護されたリモート Kubernetes クラスターをデバッグするための十分に確立された方法です。

これでやっと VM にアクセスできたので、SSH を使用して、ローカル kubelet のログにアクセスできます。これを実行するコマンドは次のとおりです。

```
azureuser@aks-nodepool1-54489083-0:~$ sudo journalctl -u kubelet -o cat
```

すると、次のような文字列が何行か表示されます (ここでは短縮して表示しています)。

```
Stopped Kubelet.
Starting Kubelet...
net.ipv4.tcp_retries2 = 8
Bridge table: nat
Bridge chain: PREROUTING, entries: 0, policy: ACCEPT
Bridge chain: OUTPUT, entries: 0, policy: ACCEPT
Bridge chain: POSTROUTING, entries: 0, policy: ACCEPT
Chain PREROUTING (policy ACCEPT)
...
```

```
I0831 08:30:52.872882 8787 server.go:182] Version: v1.9.9
I0831 08:30:52.873306 8787 feature_gate.go:226] feature gates: &{{}
map[]}
I0831 08:30:54.082665 8787 mount_linux.go:210] Detected OS with
systemd
W0831 08:30:54.083717 8787 cni.go:171] Unable to update cni config: No
networks found in /etc/cni/net.d
I0831 08:30:54.091357 8787 azure.go:249] azure: using client_
id+client_secret to retrieve access token
I0831 08:30:54.091777 8787 azure.go:382] Azure cloudprovider using
rate limit config: QPS=3, bucket=10
...
```

同様の方法で、このノードで実行されている任意のコンテナのログにもアクセスできるようになりました。Web フロントエンドで実行されているすべてのコンテナを一覧表示するには、次のコマンドを実行します。

```
azureuser@aks-nodepool1-54489083-0:~$ docker container ls | grep pets-
web

614b6d27dc13        gnspetsregistry.azurecr.io/pets-web@
sha256:43d3f3b3...
493341aff54a        gnspetsregistry.azurecr.io/pets-web@
sha256:43d3f3b3...
f5b730aa1449        gnspetsregistry.azurecr.io/pets-web@
sha256:43d3f3b3...
```

明らかに、このノードでは 3 つのインスタンスが実行されています。最初のインスタンスのログを分析してみましょう。

```
azureuser@aks-nodepool1-54489083-0:~$ docker container logs
614b6d27dc13

Listening at 0.0.0.0:80
Connecting to DB
Connected!
http://upload.wikimedia.org/wikipedia/commons/d/dc/Cats_Petunia_and_
Mimosa_2004.jpg
Connecting to DB
Connected!
https://upload.wikimedia.org/wikipedia/commons/9/9e/Green_eyes_kitten.
jpg
...
```

ただし、ここではクラスター ノードへのルート アクセス権が取得されるため、この手法には危険が伴う可能性があります。運用システム以外のシステムでのみ使用するか、関連情報にアクセスするもっと良い方法がない場合の緊急時にのみ使用してください。

ダウンタイムを発生させずにアプリケーションをアップグレードする

Pets アプリケーションをクラウド上で正常に実行できるようになったところで、今度は変更のリクエストが出てきています。ユーザーは、アプリケーションの背景色が特に気に入らないようです。背景色を変更し、次に、ダウンタイムを発生させることなく、その変更をアプリケーションに展開しましょう。

次の手順では、まずプロジェクトでコード変更を行い、対応するコンテナ イメージの新しいバージョンを作成し、それを ACR にプッシュしてそこからデプロイします。

1. `labs` フォルダの `ch12/web/src` フォルダで、`main.css` ファイルを見つけてエディターで開きます。
2. body 要素の `background` の色を好みの色 (`lightgreen` など) に変更します。
3. 変更を保存します。
4. Azure CLI コンテナ ビルドの内部から、新バージョン `v2` の `pets-web` コンテナにタグを付け、このコンテナをプッシュします。

```
docker image build -t pets-web:v2 /src/web
docker image tag pets-web:v2 gnspetsregistry.azurecr.io/pets-web:v2
docker push gnspetsregistry.azurecr.io/pets-web:v2
```

5. `Web` サービスの Docker イメージの新バージョンが ACR に入ったら、その `Web` サービス用のアップデート コマンドを発行できます。

```
kubectl set image deployment web web=gnspetsregistry.azurecr.io/pets-web:v2
```

6. (ローリング) アップデート中に、ポッドを監視することができます。

```
kubectl get pods --output=’wide’ --watch
```

`pets.yaml` マニフェストで、`maxSurge` と `maxUnavailable` の値をそれぞれ 1 と定義しました。これは、一度に更新されるポッドは 1 つだけで、いつでも少なくとも 4 つ (5 - 1 = 4) のポッドが使用可能であるため、アプリケーションは常に完全に機能するということです。

7. Pets アプリケーションが実行されているブラウザ ウィンドウを更新し、背景色が実際に新しい値に変更されていることを確認します。

新しいポッドは現在、Kubernetes クラスターの 3 つのワーカー ノードにうまく分散しています。

Kubernetes のアップグレード

最初の方で、クラスター ノードにインストールされている Kubernetes のバージョン (v1.9.9) がかなり古いことを指摘しました。ここで、Pets アプリケーションにダウンタイムを引き起こすことなく Kubernetes をアップグレードする方法を実演してみましょう。Kubernetes は段階的にしかアップグレードできないことに注意してください。つまり、一度にアップグレードできるのはマイナー バージョン 1 つだけです。次のコマンドで、アップグレードできるバージョンがわかります。

```
az aks get-upgrades --resource-group pets-group --name pets-cluster
--output table
```

このコマンドによって、次のような出力が生成されます。

```
Name    ResourceGroup   MasterVersion   NodePoolVersion   Upgrades
------- --------------- --------------- ----------------- ------------
------------------
default pets-group      1.9.9           1.9.9             1.9.10,
1.10.3, 1.10.5, 1.10.6
```

現在のバージョンが v1.9.9 なので、同じマイナー バージョン 9 の中でアップグレードするか、バージョン v1.10.x のいずれかにアップグレードすることになります。一度バージョン v1.10.x になれば、次は v1.11.x にアップグレードできます。

では、クラスターをバージョン v1.10.6 にアップグレードしてみましょう。

```
az aks upgrade --resource-group pets-group --name pets-cluster
--kubernetes-version 1.10.6
```

アップグレードは 1 ノードずつ行われ、クラスター上で実行されているアプリケーションはいつでも操作できます。アップグレードは全体で数分かかります。次のコマンドで、進捗状況を確認できます。

```
kubectl get nodes --watch
```

一度に 1 つのノードが解放され、無効化された後にアップグレードされ、再び使えるようになる様子がわかります。

完了したら、次のような出力が生成されます。

```
{
 “ aadProfile” : null,
 “ addonProfiles” : null,
 “ agentPoolProfiles” : [
    {
     “ count” : 3,
     “ maxPods” : 110,
     “ name” :“ nodepool1” ,
     “ osDiskSizeGb” : null,
     “ osType” :“ Linux” ,
```

```
        " storageProfile" :" ManagedDisks" ,
        " vmSize" :" Standard_DS1_v2" ,
        " vnetSubnetId" : null
       }
     ],
     " dnsPrefix" :" pets-clust-pets-group-186760" ,
     " enableRbac" : true,
     " fqdn" :" pets-clust-pets-group-186760-d706beb4.hcp.westeurope.
azmk8s.io" ,
     " id" :" /subscriptions/186760ad-9152-4499-b317-xxxxxxxxxxxx/
resourcegroups/pets-group/providers/Microsoft.ContainerService/
managedClusters/pets-cluster" ,
     " kubernetesVersion" :" 1.10.6" ,
     " linuxProfile" : {
       " adminUsername" :" azureuser" ,
       " ssh" : {
         " publicKeys" : [
           {
            " keyData" :" ssh-rsa ..."
           }
         ]
       }
     },
     " location" :" westeurope" ,
     " name" :" pets-cluster" ,
     " networkProfile" : {
       " dnsServiceIp" :" 10.0.0.10" ,
       " dockerBridgeCidr" :" 172.17.0.1/16" ,
       " networkPlugin" :" kubenet" ,
       " networkPolicy" : null,
       " podCidr" :" 10.244.0.0/16" ,
       " serviceCidr" :" 10.0.0.0/16"
     },
     " nodeResourceGroup" :" MC_pets-group_pets-cluster_westeurope" ,
     " provisioningState" :" Succeeded" ,
     " resourceGroup" :" pets-group" ,
     " servicePrincipalProfile" : {
       " clientId" :" a1a2bdbc-ba07-49bd-ae77-xxxxxxxxxxxx" ,
       " secret" : null
     },
     " tags" : null,
     " type" :" Microsoft.ContainerService/ManagedClusters"
   }
```

アップグレードを検証するには、次のコマンドを使用します。

```
az aks show --resource-group pets-group --name pets-cluster --output
table
```

すると、次のような結果が出力されます。

```
Name          Location   ResourceGroup   KubernetesVersion
ProvisioningState   Fqdn
------------ ---------- --------------- ------------------ ----------
--------- -------------
pets-cluster westeurope pets-group      1.10.6
Succeeded           pets-clust...
```

3 ノード クラスターが、Kubernetes `v1.9.9` から `v1.10.6` に、正常にアップグレードされました。このアップグレードにはマスター ノードが含まれます。このプロセス中、クラスターにデプロイされているアプリケーションは完全に機能していました。

AKS で実行中のアプリケーションをデバッグする

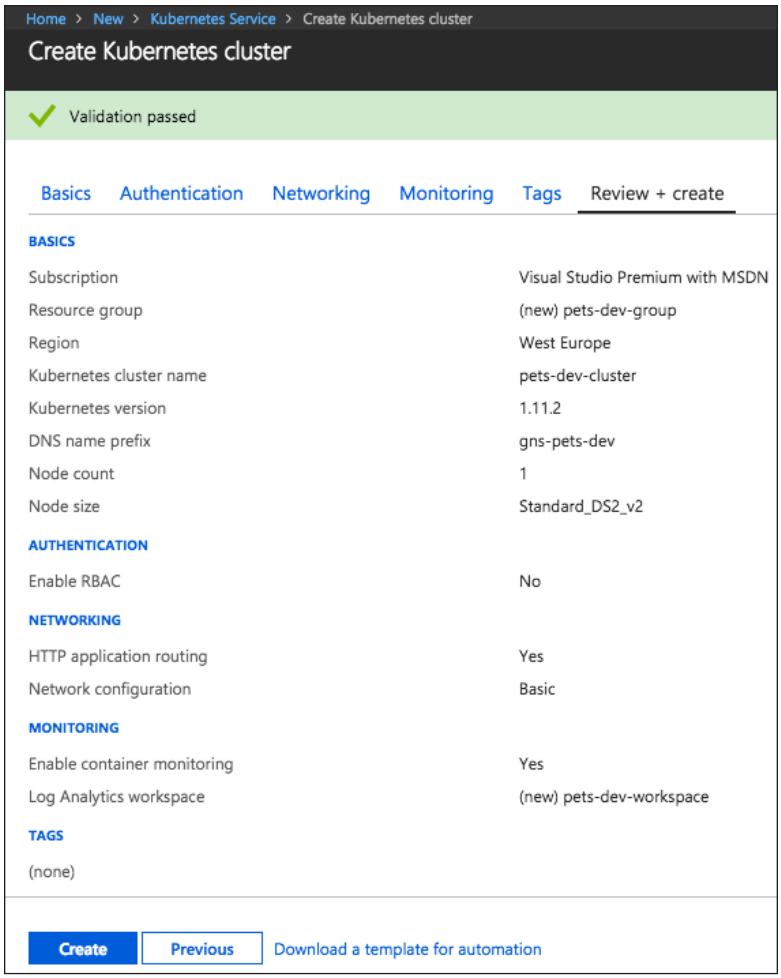
これまでは、AKS の Kubernetes でアプリケーションをデプロイして実行する方法を見てきました。これは、完成したアプリケーションで作業するオペレーション エンジニアには興味のある話題でしょう。ところが開発者は、直接クラウド内で、アプリケーションをインタラクティブに開発およびデバッグする方を好むことがあります。特に、そのアプリケーションが Kubernetes のコンテナで実行される多数の個別のサービスまたはコンポーネントで構成されている場合にその傾向があります。本書の執筆時点で、マイクロソフトは AKS でいわゆる Azure Dev Spaces のプレビューを提供しています。これにより、開発者はこの種のインタラクティブな開発とデバッグを正確に行うことができます。

開発用 Kubernetes クラスターの作成

このセクションでは、Azure の AKS で、開発目的で使用できる Kubernetes クラスターを作成します。このクラスターを使用して、クラスター上で実行されているアプリケーションをリモート デバッグする方法を示します。Azure ポータルを使用してクラスターをプロビジョニングします。

1. `https://portal.azure.com/` で Azure ポータルを開き、自分のアカウントにログインします。
2. **[+ リソースの作成]** (+ Create a resource) オプションを選択して、**[Kubernetes Service]** を選択します。
3. 開発目的で Kubernetes クラスターを作成するために必要な詳細を記入します。

次のスクリーンショットは、このセクションの作成時に著者が使用した構成サンプルです。



AKS での Kubernetes 開発クラスターの構成

具体的にいうと、新しいリソース グループ `pets-dev-group` を作成し、クラスター名に `pets-dev-cluster` という値を選択して、新しい Log Analytics ワークスペース `pets-dev-workspace` を作成しました。これは、クラスターとそのクラスター上で実行されているアプリケーションによって生成されるすべてのログおよび監視データをキャプチャするためです。HTTP アプリケーション ルーティングを有効にすることも重要です。

[作成] (Create) ボタンをクリックした後、プロビジョニングが完了するまでは数分かかります。その間に、ワークステーションに Azure CLI をインストールする作業に進むことができます。これについては次のセクションで説明します。

環境設定

このセクションでは、コマンド ラインから Azure を操作できるように、作業環境 (つまりコンピューター) を設定します。次の手順に従ってください。

1. Azure Dev Spaces で作業するには、Azure CLI をワークステーションにネイティブにインストールします。Mac では次のコマンドを使用します。

```
$ brew install azure-cli
```

2. 最新バージョンの CLI がインストールされていることを確認してください。返されるバージョンは `2.0.44` 以降である必要があります。

```
$ az --version
azure-cli (2.0.45)
...
```

3. Azure にログインします。

```
$ az login
```

   Kubernetes 開発クラスターのプロビジョニングが完了するまでは、先に進まないでください。

4. Kubernetes 開発クラスターで Azure Dev Space を設定するには、次のコマンドを使用します。

```
$ az aks use-dev-spaces -g pets-dev-group -n pets-dev-cluster

The installed extension 'dev-spaces-preview' is in preview.
Installing Dev Spaces (Preview) commands...
Installing Azure Dev Spaces (Preview) client components...

By continuing, you agree to the Microsoft Software License Terms
(https://aka.ms/azds-LicenseTerms) and Microsoft Privacy Statement
(https://aka.ms/privacystatement).Do you want to continue?(Y/n):

You may be prompted for your administrator password to authorize
the installation process.
Password:

[INFO] Downloading Azure Dev Spaces (Preview) Package...
[INFO] Downloading Bash completion script...

Successfully installed Azure Dev Spaces (Preview) to /usr/local/
bin/azds.

An Azure Dev Spaces Controller will be created that targets
resource 'pets-dev-cluster' in resource group 'pets-dev-group' .
Continue?(y/N): Y
```

```
Creating and selecting Azure Dev Spaces Controller 'pets-dev-
cluster' in resource group 'pets-dev-group' that targets resource
'pets-dev-cluster' in resource group 'pets-dev-group' ...

Select a dev space or Kubernetes namespace to use as a dev space.
[1] default
Type a number or a new name: pets

Dev space 'pets' does not exist and will be created.

Select a parent dev space or Kubernetes namespace to use as a
parent dev space.
[0] <none>
 [1] default
Type a number: 0

Creating and selecting dev space 'pets' ...2s

Managed Kubernetes cluster 'pets-dev-cluster' in resource group
'pets-dev-group' is ready for development in dev space 'pets' .Type
`azds prep` to prepare a source directory for use with Azure Dev
Spaces and `azds up` to run.
```

サービスのデプロイと実行

これで、AKS 上の Kubernetes クラスターで、最初のサービスを構築、デプロイ、および実行できる準備が整いました。

labs フォルダの ch12-dev-spaces/web サブフォルダに移動します。

```
$ cd ~/labs/ch12-dev-spaces/web
```

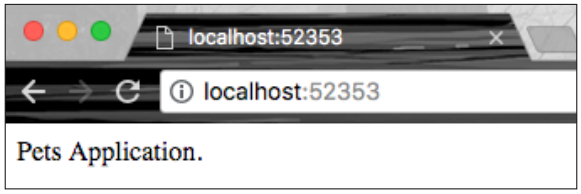
azds prep コマンドを実行します (前述の出力を参照)。これで、このコンポーネントの **Helm チャート**が作成されます。

```
$ azds prep --public

Preparing 'web' of type 'node.js' with files:
  /.dockerignore
  /azds.yaml
  /charts/web/.helmignore
  /charts/web/Chart.yaml
  /charts/web/templates/_helpers.tpl
  /charts/web/templates/deployment.yaml
  /charts/web/templates/ingress.yaml
  /charts/web/templates/NOTES.txt
  /charts/web/templates/secrets.yaml
  /charts/web/templates/service.yaml
  /charts/web/values.yaml
Type 'azds up' to run.
```

AKS で生成物を作成して実行するには、次のコマンドを使用するだけです。

```
$ azds up

Using dev space 'pets' with target 'pets-dev-cluster'
Synchronizing files...1s
Installing Helm chart...10s
Waiting for container image build...7s
Building container image...
Step 1/8 : FROM node:10.9-alpine
Step 2/8 : RUN mkdir /app
Step 3/8 : WORKDIR /app
Step 4/8 : COPY package.json /app/
Step 5/8 : RUN npm install
Step 6/8 : COPY ./src /app/src
Step 7/8 : EXPOSE 80
Step 8/8 : CMD node src/server.js
Built container image in 51s
Waiting for container...8s
(pending registration) Service 'web' port 'http' will be available at
http://web.2785a289211f45f6a8fa.westeurope.aksapp.io/
Service 'web' port 80 (TCP) is available at http://localhost:52353
press Ctrl+C to detach
web-6488b5585b-c9cg5: Listening at 0.0.0.0:80
```

この出力に示されているように、ブラウザ ウィンドウですぐに `http://localhost:52353` を開けば、AKS で実行されているサービスにアクセスすることができます。また数分後には、上記の出力で提供されているパブリック DNS (`http://web.2785a289211f45f6a8fa.westeurope.aksapp.io`) を使用してサービスにアクセスすることもできるはずです。次のスクリーンショットの画面が表示されます。



Azure Dev Spaces で動作する Pets アプリケーション

ターミナルで Ctrl + C キーを押せば、いつでもサービスを停止 (または切断) してコードを更新できます。完了後は `$ azds up` で新しいバージョンを再開するだけです。今やってみましょう。Ctrl + C キーを押してください。次に、`server.js` ファイル内のメッセージを「`My Pets Application`」に変更します。変更を保存し、次のコマンドを実行します。

```
$ azds up
```

アプリケーションの準備ができたらブラウザを更新します。すると、変更したメッセージが表示されます。

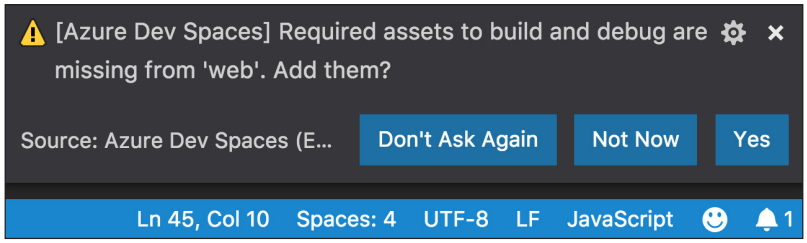
次の練習の準備として、Ctrl + C キーを押します。次のコマンドを実行して、Kubernetes クラスターからこのコンポーネントを停止して削除します。

```
$ azds down

'web' identifies Helm release 'pets-web-9c1bf6d2' which includes these
services:
  web
Are you sure you want to delete this release?(y/N): Y

Deleting Helm release 'pets-web-9c1bf6d2' ...19s
```

Visual Studio コードを使用した、サービスのリモート デバッグ

もちろんこれには少し時間がかかります。より速く処理できるようにするには、**Visual Studio Code** (**VS Code**) でリモート デバッグを行います。

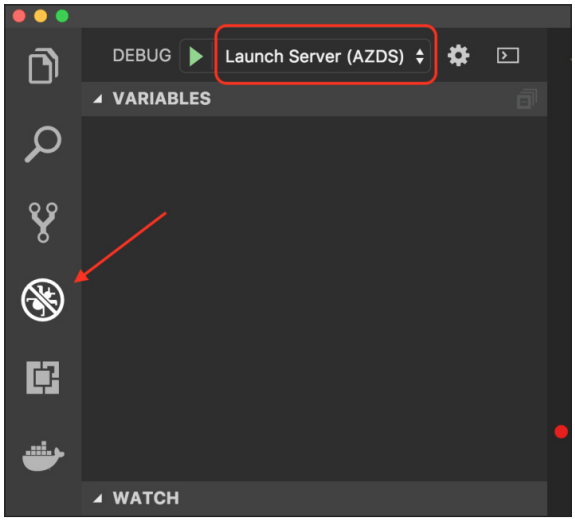
1. VS Code を開き、**Azure Dev Spaces** 拡張機能をダウンロードしてインストールします。その後、VS Code を再起動します。
2. `ch12-dev-spaces/web` フォルダから VS Code を開きます。Azure Dev Spaces 拡張機能で、`Web` コンポーネントのビルドとデバッグに必要な資産を作成するかどうかを尋ねられます。**[はい]** (Yes) オプションを選択します。



ビルドおよびデバッグ資産を作成する Visual Studio Code

すると、launch.JSON と tasks.JSON という 2 つのファイルが入っているプロジェクトに .vscode というフォルダが作成されます。これらのファイルは、Web サービスのリモート デバッグに使用されます。

3. VS Code では、F5 キーを押すと、デバッグモードの AKS でサービスがどのように構築され、実行されるかを観察できます。これには、**[サーバーの起動 (AZDS)]** (Launch Server (AZDS)) という起動タスクが使用されます。VS Code のデバッグ ペインでこのタスクを確認できます。



AKS のデバッグ モードで Web サービスを起動

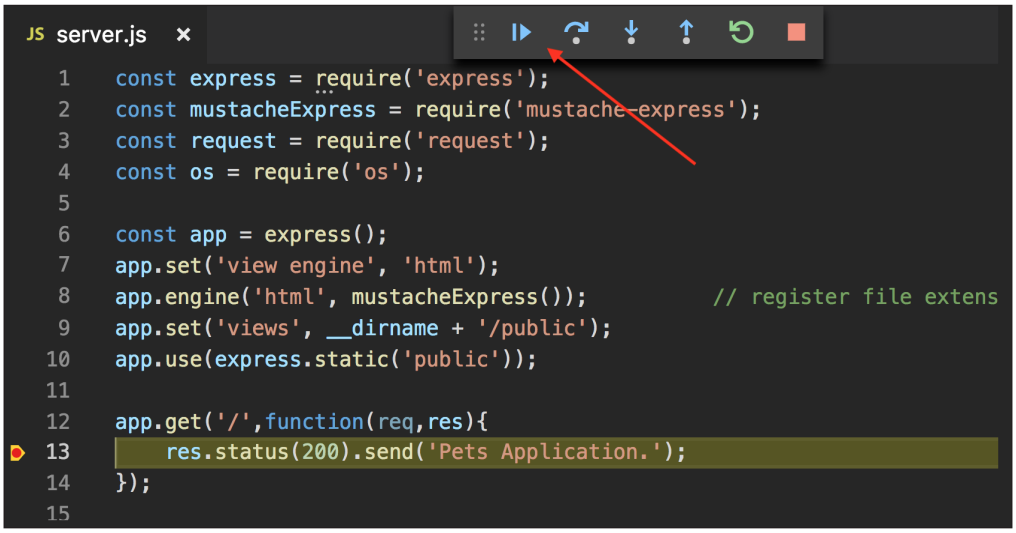
4. server.JSON ファイルの 13 行にブレークポイントを設定します。

```
11
12  app.get('/',function(req,res){
13      res.status(200).send('Pets Application.');
14  });
15
```

Web サービスのコードにブレークポイントを設定

ブラウザを http://localhost:52353 で今すぐ更新すると、コード実行はブレークポイントで停止します。

VS Code 内のデバッグ ツールバーを使用すれば、コードを 1 行ずつ段階的に実行することも、連続して実行することもできます。



段階的デバッグ

クラウドでのエディット コンティニュ スタイル開発の有効化

エディット コンティニュを体験できるようにすると、さらに利便性が増します。このために、第 2 の起動タスク **[接続 (AZDS)]** (Attach (AZDS)) があります。これは `nodemon` をベースにしたタスクです。コードをローカルで更新するたびに、AKS でリモートで実行されているコンテナにこれが反映されます。`nodemon` は変更を検出すると、自動的にコンテナ内のアプリケーションを再起動します。その結果、VS Code のコードの変更と更新、変更の保存が行われ、数秒後には AKS で新しいコードを使用およびデバッグできるようになります。

アプリケーションがまだ以前の練習問題から実行されている場合は、今すぐデバッグを停止してコンソールで Ctrl + C キーを押し、その後 `$ azds down` コマンドを実行してサービスの停止と削除を行います。

VS Code のデバッグ ビューで起動タスク **[サーバーへの接続 (AZDS)]** (Attach to Server (AZDS)) を選択し、F5 キーを押します。サービスがデプロイされると、前と同じようにブレークポイントを設定してコード内を移動できますが、これに加えて `server.js` のコードを変更することもできるようになります。

変更を保存すると、変更されたファイルは AKS と同期され、nodemon により、Kubernetes のコンテナ内部で実行されているアプリケーションが再起動されます。これを試すには、次のコード スニペットを server.js ファイルに追加します。

```
16  app.get('/pet',function(req,res){
17      res.render('app', {
18          url: 'https://bit.ly/2nngIvD',
19          hostname: os.hostname()
20      });
21  });
```

AKS で実行中のコンポーネントにコードを追加

ファイルを保存してしばらく待った後、URL http://localhost:52353/pet に移動します。かわいい猫の写真が見えるはずです。

クリーンアップする

不要なコストを避けるためには、Microsoft Azure で作成したすべてのリソースをクリーンアップ (削除) する必要があります。これはとても簡単です。すべてのリソースは pets-group リソース グループと pets-dev-group リソース グループにまとめられていますから、グループを削除すれば、そこに含まれているリソースもすべて削除されます。これは、Azure CLI で実行できます。

```
bash-4.4# az group delete --name pets-group
Are you sure you want to perform this operation?(y/n): y

bash-4.4# az group delete --name pets-dev-group
Are you sure you want to perform this operation?(y/n): y
```

プロセス全体が終了するまでに数分かかります。

Azure ポータルで、実際にすべてのリソースが削除されていることを再確認することもできます。

まとめ

この章では、完全にホストされた Kubernetes クラスターを、AKS を提供しているマイクロソフト クラウド上でプロビジョニングする方法を学びました。さらに、AKS 上のこのクラスターで実行されているアプリケーションをデプロイ、実行、監視、アップグレードし、インタラクティブにデバッグする方法についても学習しました。また、クラスター内で実行されているアプリケーションを中断することなく、AKS 上の Kubernetes のバージョンを更新する方法についても簡単に見てきました。

これで、本書による学習はひととおり終了しました。Docker の初心者から、複雑でミッションクリティカルなアプリケーションをコンテナ化して、クラウドで実行されている Kubernetes にデプロイするエキスパートへと成長する道のりのガイドとして、本書を選んでくださったことに感謝します。

質問

みなさんの知識を評価するために、以下の質問にお答えください。

1. 完全にホストされた Kubernetes クラスターを AKS にプロビジョニングするには 3 つのオプションがありますが、それは何ですか。
2. AKS を使用する際、ACR に Docker イメージをホストするのが理にかなっている理由を 2 つまたは 3 つ挙げてください。
3. コンテナ化されたアプリケーションを AKS で実行するために必要な主な手順を、3 つか 4 つの短い文で説明してください。
4. AKS で動作するアプリケーションのコンテナ ログ ファイルにアクセスする方法を教えてください。
5. AKS で Kubernetes クラスター内のノード数を拡張する方法を説明してください。

参考情報

以下の記事では、この章で説明したトピックに関連するさらに詳しい情報が入手できます。

- Azure Kubernetes Service (AKS) (`https://bit.ly/2JglX9d`)
- Upgrade an Azure Kubernetes Service (AKS) cluster (Azure Kubernetes Service (AKS) クラスターのアップグレード) (`https://bit.ly/2wCYA4P`)
- What is Azure Log Analytics? (Azure Monitor でログ データを分析する) (`https://bit.ly/2LN4Tbr`)
- Introducing Dev Spaces for AKS (AKS 用 Dev Spaces の概要) (`https://bit.ly/2NljFba`)

アセスメント

第 1 章 : コンテナとは何であり、なぜ使用すべきなのか

1. 正解は 4 番と 5 番です。
2. IT 担当者にとっての Docker コンテナは、運送業にとっての輸送コンテナと同様の意味を持ちます。これは、商品をどのようにパッケージ化するかという基準を規定するものです。この場合の商品は、アプリケーション開発者が作成します。サプライヤー (この場合は開発者) は、商品をコンテナに詰め、すべてが期待通りに収まるようにする責任があります。商品をコンテナに積み終わったら、出荷できます。標準的なコンテナであるため、荷主は大型トラックや列車、船舶などの輸送手段を標準化することが可能です。荷主は実際にコンテナに何が入っているかを気にかけたりはしません。また、ある輸送手段から別の輸送手段への積み降ろしプロセス (たとえば、列車から船舶) も高度に標準化可能です。そうすることで、輸送効率が大幅に向上します。これと同様に、IT 担当のオペレーション エンジニアは、開発者の作成したソフトウェア コンテナを使用して、コンテナの内容を気にかけることなく、それを本番システムに送り込み、標準化された方法で実行することが可能です。
3. コンテナは、次のような理由から、大変革をもたらすと思われます。
    - コンテナは自己完結型であるため、1 つのシステム上で実行すれば、コンテナを実行可能な場所ならどこでも動作します。
    - コンテナは、ハイブリッド環境だけでなく、オンプレミスでもクラウドでも動作します。オンプレミスからクラウドへのスムーズな移行が可能になるため、昨今の一般的な企業にとって重要です。
    - コンテナ イメージは、知識豊富な開発者によってビルドされ、パッケージ化されます。

- コンテナ イメージは不変で、これは優れたリリース管理をするうえで重要です。
- コンテナは、カプセル化 (Linux 名前空間と cgroup を使用)、シークレット、コンテンツ トラスト、イメージの脆弱性スキャンに基づいて、安全なソフトウェア サプライ チェーンを可能にします。

4. 次のような理由から、コンテナは、コンテナを実行可能な場所ならどこでも動作します。
   - コンテナは自己完結型のブラック ボックスです。アプリケーションだけでなく、ライブラリやフレームワーク、構成データ、証明書などのすべての依存関係もカプセル化します。
   - コンテナは、OCI などの広く受け入れられている標準に基づいています。
   - TODO: さらに理由を追加してください。
5. 間違いです。コンテナは、最新のアプリケーションや従来のアプリケーションをコンテナ化するのに役立ちます。後者を実行すれば、企業には膨大なメリットがあります。レガシー アプリケーションのメンテナンスでは、50% 以上のコスト削減が報告されています。また、このようなレガシー アプリケーションで新しいリリースまでに要する時間を最大で 90% 短縮できます。これらの数値は、実際のエンタープライズ カスタマーが公表しています。
6. 50% 以上。
7. コンテナは、Linux 名前空間 (ネットワークやプロセス、ユーザーなど) と **コントロール グループ** (**cgroup**) に基づいています。

第 2 章 : 作業環境の設定

1. `docker-machine` は以下を実行するために使用できます。
   - VirtualBox などの異なる環境で Docker ホストとして構成された VM を作成する
   - Docker ホストへの SSH
   - リモート Docker ホストからアクセスするためにローカル Docker CLI を設定する
   - 特定の環境内のホストをすべて一覧表示する
   - 既存のホストを削除または破棄する
2. 正しいです。Docker for Windows は Hyper-V で Linux VM を作成し、Linux コンテナを実行します。

3. コンテナは CI/CD ですべて自動化され最適に使用されます。コンテナ イメージをビルドし、イメージを出荷して、最後にこのイメージからコンテナを実行するまでの各ステップは、生産性を最大限に高めるために理想的にスクリプト化されています。これにより、反復可能で監査可能なプロセスが実現します。
4. Ubuntu 17.4 以降、CentOS 7.x、Alpine 3.x、Debian、Suse Linux、RedHat Linux など。
5. Windows 10 Professional または Enterprise Edition、Windows Server 2016。

第 3 章 : コンテナの操作

1. コンテナには、次の状態があります。
    - 作成済み
    - 実行中
    - 終了済み
2. 次のコマンドを使用すれば、現在ホスト上で何が実行されているかを確認できます。

```
$ docker container ls
```

3. 次のコマンドを使用して、すべてのコンテナの ID を一覧表示できます。

```
$ docker container ls -q
```

第 4 章 : コンテナ イメージの作成と管理

以下は、質問に対する回答例です。

1. Dockerfile:

```
FROM ubuntu:17.04
RUN apt-get update
RUN apt-get install -y ping
ENTRYPOINT ping
CMD 127.0.0.1
```

2. 結果を達成するには、次の手順を実行します。

```
$ docker container run -it --name sample \
    alpine:latest /bin/sh
/ # apk update && \
    apk add -y curl && \
```

```
    rm -rf /var/cache/apk/*
/ # exit
$ docker container commit sample my-alpine:1.0
$ docker container rm sample
```

3. ここでサンプルとして、C の `Hello World` を挙げます。
    1. このコンテンツを含む hello.c ファイルを作成します。

```
#include <stdio.h>
int main()
{
     printf("Hello, World!");
     return 0;
}
```

    2. このコンテンツを含む Dockerfile を作成します。

```
FROM alpine:3.5 AS build
RUN apk update && \
   apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello

FROM alpine:3.5
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

4. Docker イメージには、次のような特徴があります。
    - 不変である
    - 不変レイヤーで構成されている
    - 各レイヤーには、下のレイヤーに対して変化した部分 (差) のみが含まれる
    - イメージはファイルとフォルダの (大きな) tarball である
    - イメージはコンテナのテンプレートである
5. 3 番が正解です。最初に、ログインしていることを確認してから、イメージにタグを付け、最後にプッシュする必要があります。これはイメージであるため、(`docker container ...` ではなく) `docker image ...` (4 番に示した通り) を使用しています。

第 5 章 : データ ボリュームとシステム管理

ボリュームを調整する最も簡単な方法は、Docker for Mac または Docker for Windows を直接使用している場合に Docker Toolbox を使用するというもので、ボリュームは Docker for Mac/Win が透過的に使用する (やや隠された) Linux VM 内に格納します。

そのため、以下を推奨しています。

```
$ docker-machine create --driver virtualbox volume-test
$ docker-machine ssh volume-test
```

すると、`volume-test` という Linux VM に移動します。ここで次の演習を実行できます。

1. 名前付きの `volume` を作成するには、次のコマンドを実行します。

   ```
   $ docker volume create my-products
   ```

2. 次のコマンドを実行します。

   ```
   $ docker container run -it --rm \
       -v my-products:/data:ro \
       alpine /bin/sh
   ```

3. ボリュームの使用向けにホスト上のパスを取得するには、次のコマンドを実行します。

   ```
   $ docker volume inspect my-products | grep Mountpoint
   ```

   すると、このような結果になります (docker-machine と VirtualBox を使用している場合)。

   ```
   "Mountpoint": "/mnt/sda1/var/lib/docker/volumes/my-products/_
   data"
   ```

   ここで、次のコマンドを実行します。

   ```
   $ sudo su
   $ cd /mnt/sda1/var/lib/docker/volumes/my-products/_data
   $ echo "Hello world" > sample.txt
   $ exit
   ```

4. 次のコマンドを実行します。

   ```
   $ docker run -it --rm -v my-products:/data:ro alpine /bin/sh
   # / cd /data
   # / cat sample.txt
   ```

別のターミナルで、以下を実行します。

```
$ docker run -it --rm -v my-products:/app-data alpine /bin/sh
# / cd /app-data
# / echo "Hello other container" > hello.txt
# / exit
```

5. 次のようなコマンドを実行します。

```
$ docker container run -it --rm \
    -v $HOME/my-project:/app/data \
    alpine /bin/sh
```

6. 両方のコンテナを終了し、ホストに戻って次のコマンドを実行します。

```
$ docker volume prune
```

7. 次のコマンドを実行します。

```
$ docker system info | grep Version
```

次のような出力が表示されます。

```
Server Version: 17.09.1-ce
Kernel Version: 4.4.104-boot2docker
```

これまで `docker-machine` を使用して VirtualBox で Linux VM を作成および使用してきた場合は、完了後に必ずクリーンアップを実行してください。

```
$ docker-machine rm volume-test
```

第 6 章 : 分散アプリケーション アーキテクチャ

1. 多くの部品で構成されるシステムでは、遅かれ早かれ少なくとも 1 つの部品が故障します。このような状況でダウンタイムを回避するために実行するのが、各コンポーネントの複数のインスタンスです。そうすることで、インスタンスの 1 つに障害が発生しても、他のインスタンスによって継続して要求を処理することが可能となります。
2. 分散アプリケーション アーキテクチャには、多くの可動部品があります。サービス A がサービス B のインスタンスへのアクセスを必要とする場合、そのインスタンスがどこにあるかを検知することはできません。インスタンスはクラスターの任意のノード上に存在でき、オーケストレーション エンジンに応じて出入り可能です。そのため、ターゲット インスタンスは、たとえば IP アドレスとポートではなく、名前とポートによって識別されます。DNS サービスには、クラスター内で実行されているすべてのサービス インスタンスに関するすべての情報があり、そこにはサービス名を IP アドレスに解決する方法も含まれています。

3. サーキット ブレーカーは、単一の障害が発生したサービスによってトリガーされる分散アプリケーションの連鎖的な障害を回避するのに役立つメカニズムです。サーキット ブレーカーは、あるサービスから別のサービスへの要求を観察し、時間経過に伴う遅延と要求失敗またはタイムアウトの回数を測定します。特定のターゲット インスタンスが過度に多くの失敗を引き起こすと、その呼び出しはインターセプトされ、エラーコードが呼び出し元に返され、可能な場合はターゲットの復旧時間が即時に与えられます。これを受け、呼び出し元はサービスのグレードを低下させるべきか、あるいはそのターゲット サービスの別のインスタンスを試すべきかどうかを判断します。
4. モノリスは、高度に結合された単一のコード ベースで構成されるアプリケーションです。コードの変更が行われた場合、それがどんなに些細な変更であっても、アプリケーション全体をコンパイルし、パッケージ化して、デプロイし直す必要があります。モノリスは可動部分が非常に少ないため、デプロイと監視が簡単ですが、メンテナンスと拡張が困難です。分散アプリケーションは、多くの疎結合サービスで構成されています。各サービスは、独自の独立したソース コード ベースに由来します。個々のサービスは、独立したライフサイクルを持つことができ、それらは個々に開発、修正可能です。分散アプリケーションは、管理と監視がさらに困難です。
5. サービスの「ブルー」と呼ばれる現在実行中のバージョンが、同じサービスの「グリーン」と呼ばれる新しいリリースに置き換えられる場合、それはブルー グリーン デプロイメントと言われます。ブルー バージョンがまだ実行されている間、グリーン バージョンのサービスがシステムにインストールされ、準備が完了した後、トラフィックがすべてブルーではなくグリーンに向けられるよう、サービスへトラフィックを送信するルーターの設定を多少変更する必要があります。

第 7 章 : シングルホスト ネットワーキング

1. 3 つのコア要素は、サンドボックス、エンドポイント、ネットワークです。
2. 次のコマンドを実行します。

```
$ docker network create --driver bridge frontend
```

3. 次のコマンドを実行します。

```
$ docker container run -d --name n1 \
   --network frontend -p 8080:80 \
   nginx:alpine
$ docker container run -d --name n2 \
   --network frontend -p 8081:80 \
   nginx:alpine
```

両方の Nginx インスタンスが起動され実行されていることをテストします。

```
$ curl -4 localhost:8080
$ curl -4 localhost:8081
```

いずれの場合も、Nginx ウェルカム ページが表示されるはずです。

4. 接続されているすべてのコンテナの IP を取得するには、次のコマンドを実行します。

```
$ docker network inspect frontend | grep IPv4Address
```

次のように表示されます。

```
"IPv4Address": "172.18.0.2/16",
"IPv4Address": "172.18.0.3/16",
```

ネットワークで使用されるサブネットを取得するには、たとえば次を使用します。

```
$ docker network inspect frontend | grep subnet
```

以下に似た結果が得られます (前の例から取得)。

```
"Subnet": "172.18.0.0/16",
```

5. host ネットワークでは、ホストのネットワーク名前空間でコンテナを実行できます。
6. このネットワークは、デバッグ目的またはシステム レベルのツールを構築する場合にのみ使用します。host ネットワークは、本番環境のアプリケーション コンテナには絶対に使用しないでください。
7. none ネットワークは、基本的にコンテナがどのネットワークにも接続されていないことを示しています。これは、他のコンテナと通信する必要がなく、外部からアクセスする必要がないコンテナに使用します。
8. none ネットワークは、たとえば、ホストにマウントされたボリュームを通じてアクセス可能なファイルなど、ローカル リソースへのアクセスのみを必要とするコンテナ内で実行中のバッチ プロセスに使用できます。

第 8 章 : Docker Compose

1. 次のコードを使用して、デーモン モードでアプリケーションを実行できます。

```
$ docker-compose up -d
```

2. 実行中のサービスの詳細を表示するには、次のコマンドを実行します。

```
$ docker-compose ps
```

このコマンドを実行すると、以下が出力されます。

```
    Name              Command               State  Ports
    ------------------------------------------------------------------
----
    mycontent_nginx_1 nginx -g daemon off; Up     0.0.0.0:3000->80/
tcp
```

3. 次のコマンドを使用して、web サービスをスケール アップできます。

```
    $ docker-compose up --scale web=3
```

第 9 章 : オーケストレーター

以下は、この章の質問に対する回答例です。

1. オーケストレーション エンジンが必要な理由は、次のとおりです。
    - コンテナは一時的なものであり、自動化されたシステム (オーケストレーター) のみが効率的に処理できます。
    - 高可用性を実現するには、各コンテナの複数のインスタンスを実行する必要がありますが、それにより、管理するコンテナ数はあっという間に膨大になります。
    - 今日のインターネットの需要を満たすには、迅速にスケール アップとスケール ダウンを実行する必要があります。
    - VM とは対照的に、コンテナは、何か誤った動作をした際にペットのようにしつけを受けるのではなく、畜牛のように扱われます。つまり、万一誤った動作をすれば、キルされ (殺され) て、新しいインスタンスに置き換えられてしまいます。オーケストレーターは、正常でないコンテナをすぐに終了し、新しいインスタンスをスケジュールします。
2. コンテナ オーケストレーション エンジンは、次のような役割を果たします。
    - クラスター内のノードのセットを管理する
    - 十分な空きリソースがあるノードへのワークロードをスケジュールする
    - ノードとワークロードの正常性を監視する
    - アプリケーションとコンポーネントの現在の状態が目的の状態になるように調整する
    - サービス検出とルーティングを提供する
    - 負荷分散を要求する
    - シークレットのサポートを提供することにより、信頼できるデータを確保する

3. 以下は、普及度順に並べたオーケストレーターのリスト (不完全なリスト) です。
    - Google の Kubernetes (CNCF に寄贈)
    - Docker の SwarmKit ー **オペレーション サポート システム** (**OSS**)
    - Amazon の AWS ECS
    - マイクロソフトの Azure AKS
    - Apache の Mesos ー OSS
    - Rancher の Cattle
    - HashiCorp の Nomad

第 10 章 : Kubernetes を使用したコンテナ化アプリケーションのオーケストレーション

1. Kubernetes マスターは、クラスターを管理する役割を担います。オブジェクトを作成するためのすべての要求、ポッドのスケジューリング、`ReplicaSet` などは、マスターで実行されています。マスターは、本番環境または本番環境に近いクラスター内でアプリケーション ワークロードを実行しません。
2. 各ワーカー ノードには、kubelet とプロキシ、コンテナ ランタイムがあります。
3. 答えは「はい」です。スタンドアロン コンテナは Kubernetes クラスターでは実行できません。ポッドは、このようなクラスターのデプロイメントの原子単位です。
4. ポッド内で実行されているすべてのコンテナは、同じ Linux カーネル名前空間を共有します。そのため、これらのコンテナ内で実行されるすべてのプロセスは、ホスト上で直接実行されているプロセスまたはアプリケーションが `localhost` を通じて相互に通信できるのと同様の方法で、`localhost` を通じて相互に通信できます。
5. `pause` コンテナの唯一の役割は、ポッドで実行されるコンテナのポッドの名前空間を予約することです。
6. これは、ポッドのすべてのコンテナが同じ場所にあることになり、同じクラスター ノードで実行されるため、好ましいとはいえません。しかし、アプリケーションのさまざまなコンポーネント (つまり、`web`、`inventory`、`db`) には通常、スケーラビリティやリソース消費量に関して非常に異なる要件があります。`web` コンポーネントはトラフィックに応じてスケール アップ / スケール ダウンしなければならない可能性がある一方で、`db` コンポーネントには他のコンポーネントにはない特別なストレージ要件があります。すべてのコンポーネントを独自のポッドで実行することで、より柔軟にそれに対応できます。

7. クラスター内のポッドの複数のインスタンスを実行するだけでなく、ネットワークの分断またはクラスター ノードの障害によって個々のポッドがクラッシュまたは消滅した場合でも、実際に稼働しているポッドの数が期待どおりの数に常に一致するようにする必要があります。ReplicaSet は、あらゆるアプリケーション サービスに対してスケーラビリティと自己復旧を提供するメカニズムです。
8. サービスのダウンタイムを発生させることなく、Kubernetes クラスター内のアプリケーション サービスを更新するには、デプロイメント オブジェクトが必ず必要です。Deployment オブジェクトは、ReplicaSet にローリング アップデート機能とロールバック機能を追加します。
9. Kubernetes サービス オブジェクトは、アプリケーション サービスをサービス検出に関与させるために使用します。これらは、(通常、ReplicaSet またはデプロイメントによって管理される) ポッド セットに安定したエンドポイントを提供します。Kube サービスは、論理的なポッド セットとそのアクセス方法に関するポリシーを定義する抽象概念で、次の 4 種類があります。
    - **ClusterIP**: クラスター内からのみアクセス可能な IP アドレスにサービスを公開します。これは**仮想 IP** (**VIP**) です。
    - **NodePort**: すべてのクラスター ノードに 30,000 ~ 32,767 の範囲のポートを発行します。
    - **LoadBalancer**: AWS の ELB などのクラウド プロバイダーのロード バランサーを使用してアプリケーション サービスを外部に公開します。
    - **ExternalName**: データベースなどのクラスター外部サービスのプロキシを定義する必要がある場合に使用します。

第 11 章 : Kubernetes によるアプリケーションのデプロイ、更新、および保護

1. レジストリに 2 つのアプリケーション サービス (Web API と Mongo DB) の Docker イメージがあると想定した場合、以下を実行する必要があります。
    - `StatefulSet` を使用して Mongo DB のデプロイメントを定義する - このデプロイメントを `db-deployment` と呼ぶことにします。`StatefulSet` は、レプリカを 1 つ持つ必要があります (Mongo DB のレプリケーションはもう少し複雑で、本書の範囲外です)。
    - `db-deployment` の `ClusterIP` タイプの `db` という Kubernetes サービスを定義する。
    - Web API のデプロイメントを定義する - これを `web-deployment` と呼ぶことにします。このサービスを 3 つのインスタンスにスケーリングします。
    - `web-deployment` の `NodePort` タイプの `api` という Kubernetes サービスを定義する。

- シークレットを使用する場合は、`kubectl` を使用してシークレットを直接クラスター内で定義する。
- `kubectl` を使用してアプリケーションをデプロイする。

2. アプリケーションのレイヤー 7 ルーティングを実装する場合は、理想としては `IngressController` を使用します。`IngressController` は、Nginx のようなリバース プロキシで、Kubernetes Server API をリッスンしてリバース プロキシの設定を更新し、その変更が検出された場合に再起動するサイドカーを持っています。続いて、ルーティングを定義するクラスター内の Ingress リソースを定義する必要があります。たとえば、`https://example.com/pets` のようなコンテキストベースのルートから、`api/32001` などの `<a service name>/<port>` ペアへのルーティングを定義しなければなりません。Kubernetes がこの Ingress オブジェクトを作成または変更した瞬間に、`IngressController` のサイドカーがそれに応じてプロキシのルーティング設定を更新します。
3. これが**クラスター内部**のインベントリ サービスであると想定した場合
   - バージョン 1.0 をデプロイする場合は、`inventory-deployment-blue` というデプロイメントを定義し、ポッドに `color: blue` というラベルを付けます。
   - 前述の、`color: blue` を含むセレクターのあるデプロイメントに対し、`inventory` という `ClusterIP` タイプの Kubernetes サービスをデプロイします。
   - 新しいバージョンの支払サービスをデプロイする準備が完了したら、最初にバージョン 2.0 のサービスのデプロイメントを定義し、`inventory-deployment-green` という名前を付けます。`color: green` というラベルをポッドに追加します。
   - これで、「green」サービスのスモーク テストを実行できます。それで問題がなければ、`color: green` を含むセレクターなどのインベントリ サービスを更新できます。
4. 機密情報であるために、Kubernetes シークレットを通じてサービスに提供する必要がある情報には、パスワードや証明書、API キー ID、API キー秘密、トークンなどがあります。
5. シークレット値のソースには、ファイルまたは base-64 でエンコードされた値を指定できます。

第 12 章 : コンテナ化されたアプリケーションをクラウドで実行する

1. Azure ポータルのグラフィカル ユーザー インターフェイス、Azure CLI、Azure リソース テンプレートのいずれかを Terraform などのツールと組み合わせて使用すると、完全にホストされたクラスターを AKS でプロビジョニングできます。
2. Azure コンテナ レジストリを使用してイメージを保存することは、次のような理由で合理的です。
    1. イメージを本番システムにデプロイする際に待ち時間が長くなるのを回避するには、イメージから作成されたコンテナが実行される Kubernetes クラスターに近いコンテナ レジストリにイメージを保存しておくことが重要です。
    2. セキュリティ上の理由から、Docker Hub などの外部コンテナ レジストリからイメージをダウンロードする場合、Kubernetes クラスターのホストが Azure ネットワークの外部に到達しないようにした方がよいことがあります。ACR は、Kubernetes クラスターと同じ (内部) ネットワークに配置できます。
    3. コンテナ イメージのダウンロードに使用される帯域幅は、同じデータ センター内にある場合、外部データ リンクを使用する場合より安価になります。
    4. 単一のベンダーに集中し、複数のベンダーを扱いたくない場合。
3. AKS でコンテナ化アプリケーションを実行するには、以下を実行する必要があります。
    1. 完全に管理された Kubernetes クラスターを AKS で提供する
    2. コンテナ イメージをビルドして ACR にプッシュする
    3. `kubectl` を使用してアプリケーションをクラスターにデプロイする
4. コンテナ ログ、または任意のワーカー ノードの `kubectl` ログを表示する方法の 1 つとして、そのノードへの SSH の実行があります。その場合、そのホスト上で特別なコンテナを実行し、そこからホストへの SSH 接続を確立します。続いて、`journalctl` などのツールを使用してシステム ログを分析したり、ホスト上で通常の docker コマンドを実行してコンテナ ログを取得できます。
5. Azure CLI を使用すると、ワーカー ノード数を増減できます。それには、次のようなコマンドを実行します。

```
az aks scale --resource-group=<group-name> --name=<cluster-
name> --node-count <num-nodes>
```

参考になるその他の本

この本を読んだことがある人は、Packt の他の本にも興味があるかもしれません。

Docker on Windows

Elton Stoneman

ISBN: 978-1-78528-165-5

- Docker の重要な概念 (イメージ、コンテナ、レジストリ、およびスウォーム) を理解する
- Windows 10、Windows Server 2016、およびクラウドで Docker を実行する
- 複数の Docker コンテナに分散ソリューションをデプロイして監視する
- Docker Swarm で高可用性とフェールオーバーを備えたコンテナを実行する
- Docker プラットフォームを使用した高度なセキュリティを習得して、アプリの安全性を高める
- Docker で Jenkins を実行して、継続的デプロイメント パイプラインを構築する
- Visual Studio を使用して Docker コンテナで動作するアプリケーションをデバッグする
- 組織への Docker の導入を計画する

Docker for Serverless Applications

Chanwit Kaewkasi

ISBN: 978-1-78883-526-8

- サーバーレス アプリケーションと FaaS アプリケーションとは何かを学ぶ
- 3 つの主要なサーバーレス システムのアーキテクチャに精通する
- Docker テクノロジーがサーバーレス アプリケーションの開発にどのように役立つかを調査する
- FaaS インフラを作成して維持する
- オンプレミス FaaS インフラとして機能するように Docker インフラを設定する
- Docker コンテナを使用してサーバーレス アプリケーション用の関数を定義する

レビューを残す – あなたの考えを他の読者に知らせましょう

この本を購入したサイトにレビューを残すことによって、この本に対するあなたの考えを他者と共有してください。この本を Amazon で購入した場合は、この本の Amazon ページに率直なレビューを残してください。これは、他の潜在的な読者があなたの公平な意見を参考にして購入を決意するために重要であると同時に、マイクロソフトにとってはカスタマーが弊社の製品についてどう思っているかを理解する助けになります。また、弊社の執筆者は Packt と協力して作成したタイトルに対するフィードバックを参照することができます。フィードバックにかかる時間はほんの数分ですが、他の潜在カスタマー、弊社の執筆者、および Packt にとっては大きな価値があります。ありがとうございます！