

インプレスR&D [Next Publishing]



Cloud シリーズ

E-Book / Print Book

Next Publishing プレビュー



NextPublishing Sample

目次

はじめに	4
表記関係について	4
免責事項	4
底本について	4
第1章 Kubernetesは怖くない	5
1.1 Kubernetesとは	5
1.2 Docker Composeとは何が違うの？	5
1.3 Kubernetesを使うには公約を掲げる！？	6
1.4 リソース	7
1.5 まとめ	9
第2章 環境構築をしよう	10
2.1 想定環境	10
2.2 Dockerのインストール	10
2.3 kindのインストール	11
2.4 kubectlのインストール	12
2.5 Helmのインストール	13
第3章 Lesson1 Kubernetesを触ってみよう	14
3.1 Podを作成しよう	14
3.2 DeploymentでPodを制御しよう	15
3.3 Serviceで外部からアクセスできるようにしよう	19
3.4 まとめ	21
第4章 Lesson2 Kubernetes上にアプリケーションを作成しよう	22
4.1 自前のイメージをデプロイしよう	22
4.2 サービス無停止でアップデートを実施しよう	25
4.3 ConfigMapを使ってパラメーターを外出ししよう	28
4.4 機密情報はSecretで渡そう	37
4.5 RoleとRoleBindingで権限管理をしよう	43
4.6 オートスケーリングを使って高可用性を実現しよう	48
4.7 まとめ	55

第5章 Lesson3 Kubernetes上にElasticsearchクラスターを立てよう（ローカル編）	56
5.1 Elastic Operator とは	56
5.2 Elasticsearch を Kubernetes にデプロイしてみよう	57
5.3 インデックスをして検索しよう	61
5.4 Kibana も使えるようにしよう	63
5.5 ポートフォワーディングもマニフェストで管理しよう	69
5.6 Pod のログを収集しよう	75
5.7 Elasticsearch のメトリクスを監視しよう	79
5.8 カナリアリリースもお手のもの！？	87
5.9 まとめ	92
第6章 Lesson4 Kubernetes上にElasticsearchクラスターを立てよう（Azure編）	93
6.1 Microsoft Azure にログインしよう	93
6.2 リソースプロバイダーを登録しよう	98
6.3 AKS を使って Kubernetes クラスターを構築しよう	100
6.4 AKS クラスターに接続しよう	103
6.5 AKS クラスターに Elasticsearch をデプロイしよう	108
6.6 AKS 上でもカナリアリリースを実現しよう	113
6.7 データを永続化しよう	118
6.8 AKS クラスターを削除しよう	126
あとがき	129
謝辞	129

はじめに

この度は「Kubernetesで始める Elasticsearch 運用入門」をお手に取っていただき、ありがとうございます。

本書では、以下のような読者を想定しています。

- ・ Elasticsearchは知っているし、構築経験もある
- ・ Dockerはなんとなく知っている
- ・ Kubernetesはほぼわからない

タイトル通り、普段全文検索エンジンとして Elasticsearch を運用している方を対象としています。特に、VM上に Elasticsearch クラスターを苦労して立ち上げている人にこそ、読んでほしい一冊です。もし全文検索について詳しくなりたいという方は、「今日から始める AI 検索技術 Solr エンジニアのための最先端ガイド」という書籍で紹介していますので、こちらをご覧ください¹。

著者らも本書の執筆当初は、Kubernetesについては初心者でした。そんな私たちが、Kubernetes上で Elasticsearch クラスターを稼働させられるまでに学んだことをまとめています。本書を読めば、Kubernetesについてほとんど触ったことがない人であっても、Kubernetes上で Elasticsearch クラスターを構築できるようになります。本書内にはコード付きで解説しているのに加えて、GitHubにてサンプルコードも公開しています²。

Kubernetesは難しい、とつつきにくい印象があるかと思います。でも、使えるようになってくると、その苦労が報われる以上の恩恵が得られます。さあ、本書を通じて Kubernetes 街道への一歩を踏み出しましょう！

表記関係について

本書に記載されている会社名、製品名などは、一般に各社の登録商標または商標、商品名です。会社名、製品名については、本文中では®、™マークなどは表示していません。

免責事項

本書に記載された内容は、情報提供のみを目的としております。そのため、本書に記載された内容を用いた開発、制作、運用はご自身の責任と判断によって実施してください。利用の結果発生した事象に関しては、当方は一切の責任を負いかねます。

底本について

本書は、技術系同人誌即売会「技術書典 17」で頒布された「ゼロから学ぶ Kubernetes × Elasticsearch 運用」を底本としております³。

1.<https://nextpublishing.jp/book/18504.html>

2.<https://github.com/Sashimimochi/elastic-operator-tutorial>

3.<https://techbookfest.org/product/vvVL92XBaqw9v2hBgi4Rh8>

第1章 Kubernetesは怖くない

1.1 Kubernetesとは

Kubernetesと聞くと、複雑で難解なイメージを持つかもしれません。特に、初めて触れる人にとっては"コンテナオーケストレーション"や"クラスター"といった用語がより難解な印象を与えていく気がします。しかし恐れることなけれ、本書ではつい先日までみなさんと同じような知識レベルだった私が、転ばぬ先の杖となって必要事項を解説していきます。これを機に、Kubernetesと仲良くなれば、普段の開発や運用をぐっと楽にしてくれる心強いツールになります。

Kubernetesは一言でいうと、「コンテナの管理ツール」です。Dockerを触ったことのあるみなさんならイメージがつくと思いますが、各所に立ち上がっているコンテナを複数台のサーバーにまたがって管理できるようにしたのが、Kubernetesです。また、必要に応じてスケールアップしたり、トラブル発生時には自動でシステムの復旧すらやってくれます。

もしかしたら、大規模システムや複雑なアプリケーション用のツールと思われたかもしれません。しかし、シンプルなアプリケーションでも自動復旧などの恩恵によって、運用工数の削減につながるかもしれません。開発環境と本番環境をほぼ同じ設定で管理できるので、環境差異による運用の複雑さも減らせるでしょう。

Kubernetesには、本書で解説しきれないくらい多くの機能があります。ですが、そのすべてを必ずしも最初から知っておく必要はありません。最初の一歩として、「コンテナをよしなに管理するためのツールなんだ。便利だなあ」くらいの軽い認識にしておきましょう。そのうえで、少しずつ Kubernetesの世界を探索していくと、最初に感じた恐怖や難しさは薄れ、その便利さや柔軟性に感動することになるでしょう。

1.2 Docker Composeとは何が違うの？

Docker Composeも複数のコンテナを簡単に定義し、実行するためのツールです。データベースを用意して、そこからバックエンドサーバーを経由して、フロントエンドでWebページを作るといった開発をしたことがある方もいることでしょう。複数のコンテナを連携し、シンプルに管理したいというシチュエーションにおいては、Docker Composeは非常に便利なツールです。

Kubernetesも複数のコンテナを管理するという点ではDocker Composeと似ています。しかし、その規模や機能は大きく異なります。

1.2.1 スケーラビリティー

Docker Composeは、ローカル環境や小規模なプロジェクトにおいて有効です。一方、大規模なシステムや本番環境での安定的な稼働を求められる状況での利用には不安があります。たとえば、コンテナの数が増えてきたり、複数のサーバーにまたがってコンテナを配置する場合にはDocker

Composeでは管理しきれません。

Kubernetesは、このようなスケーラビリティーの問題に対応すべく設計されています。Kubernetesには、クラスターという形で複数台のサーバーをひとつのシステムとして機能させるための仕組みがあります。これによって、リソースの利用状況に応じて余裕があるサーバーに新しいコンテナを配置することが自動でできてしまします。それだけでなく、必要に応じてコンテナの数を増やしたり減らしたりといったチューニングまで勝手にやってくれます。普段皆さんがやっているような運用作業を Kubernetesにお任せできてしまうのです。

1.2.2 自動復旧

Docker Composeでは、コンテナがクラッシュしたり、異常な状態に陥った場合、手動で対応する必要があります。再起動をしたり、ログを確認したり、開発者や運用者が主体的に動いて復旧させる必要があります。

一方、Kubernetesでは、コンテナの状態は Kubernetesが見てくれます。もしコンテナがクラッシュしてしまったときは、Kubernetesが判断してコンテナの再起動を実施してくれます。

1.2.3 デプロイとロールアウト

Docker Composeでは、アプリケーションのバージョンアップや設定を変更する場合、一度すべてのコンテナを停止させる必要があります。つまり、アップデートのために一度サービスを止めないといけません。ユーザーにサービスを提供しているシステムでは、そう簡単にはサービスを止めることはできないでしょう。

しかし、Kubernetesでは、段階的にコンテナを古いものから新しいものに入れ替える仕組みがあります。これによって、サービスをほとんど停止させることなく、アップデートが可能になるのです。

1.2.4 インフラの抽象化

Docker Composeでは、基本的にローカルやシングルホスト環境での運用を前提とします。設定ファイルもシンプルで作成が容易です。反面、インフラ面についてはこれとは別に考える必要があります。コンテナに必要なCPUやメモリーが足りるかどうか、サーバーごとに見る必要があります。リソースが足りない場合は、複数台のサーバーにコンテナを配置することもあるかもしれません。リソースを互いに食い合わないように、どのサーバーでどのアプリケーションを動かすべきか、人間が設計する必要があります。

一方、Kubernetesは「Kubernetes」というひとつのインフラ上にアプリケーションを展開します。どのサーバーにどのアプリケーションを配置するかは、Kubernetesが決めてくれます。なので、開発者はアプリケーションの開発に専念できます。

1.3 Kubernetesを使うには公約を掲げる！？

Kubernetesの世界に足を踏み入れると、マニフェストという言葉を頻繁に耳にすることになるで

しょう。マニフェストは、Kubernetesの基本的な概念を理解するうえで、非常に重要なキーワードです。

1.3.1 マニフェストの役割

マニフェストは、簡単に言えばKubernetesの設定ファイルの総称です。サーバーのリソースやコンテナの数、各種設定をマニフェストという形で定義します。

Kubernetesでは、これらの設定をるべき理想の状態として「宣言」することで、Kubernetesはその宣言に沿っているかを確認します。もし沿っていなければ、るべき姿へと近づけようとKubernetesが動きます。

1.3.2 マニフェストの書き方

マニフェストはYAML形式で記述します。たとえば、以下のように記述します。

リスト 1.1: pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app
    image: httpd
    resources:
      limits:
        cpu: 500m
        memory: 512Mi
```

詳しい解説は第2章「環境構築をしよう」以降で説明しますが、使用するDockerイメージやリソースの定義をします。コードベースで管理できるので、Pull RequestのレビューやCI/CDなど普段の運用の中にも自然に組み込めます。

定義できる項目はたくさんありますが、必要な機能をすべて知っておく必要はありません。怖がらずに1歩ずつ覚えていきましょう。

1.4 リソース

マニフェストと並んで、**リソース**という概念もKubernetesの基礎において重要な概念です。一般的に「リソース」というとヒト・モノ・カネ、あるいはCPUやメモリーなどのマシンスペックを想像するかと思います。

Kubernetesにおける「リソース」は、Kubernetesのクラスター内で管理されるオブジェクトの

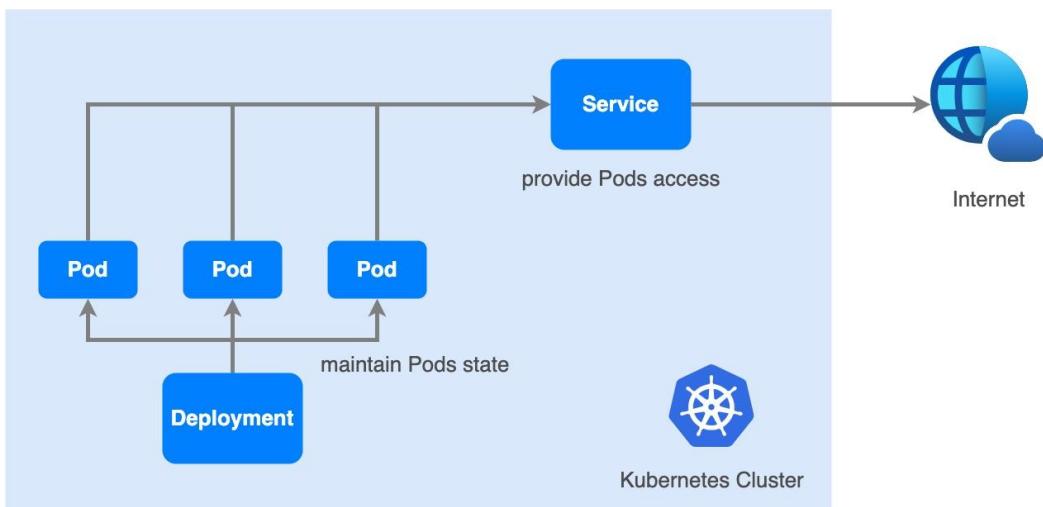
ことを指します。リソースはKubernetesに対して、「何をどのように動かすのか」という指示になります。たとえば、アプリケーションのコンテナやネットワーク設定、ストレージの構成などがリソースに当たります。リソースをマニフェストとして指示することで、Kubernetesはシステムが想定している状態にあるのか判断します。

Kubernetesには、様々な種類のリソースがあります。ここでは、代表的なものを3つだけ紹介します。

- Pod（ポッド）：ひとつ以上のコンテナをまとめたもの。Kubernetesの中で一番小さい基本的な実行単位。Kubernetesはコンテナは直接管理せず、Podの単位で管理する。同一Pod内のコンテナはネットワークやストレージを共有する。ひとつのマイクロサービスをひとつのPodとしてまとめることが多い。
- Deployment（デプロイメント）：Podの状態を管理するリソース。るべき姿を定義し、Kubernetesにその状態からずれていないか監視、管理する指示として機能する。
- Service（サービス）：Pod同士あるいは外部のアプリケーションからPodにアクセスするための入口を提供するリソース。Podが再構築されてIPアドレスが変わったとしても、Serviceが一貫した仮想IPやDNS名を提供し続ける限り、クライアントは同じアクセスポイントに対してリクエストが可能になる。

リソースには、互いに関係性があります。たとえば、DeploymentはPodを管理します。そのPodはServiceによって、外部からアクセスできるようになります。このように、リソース同士が連携することで、アプリケーションとして機能するようになります。

図1.1: 各リソースの関係



これ以外にもさまざまなリソースがありますが、登場した都度解説していきます。ひとまず、第3章「Lesson1 Kubernetesを触ってみよう」でも登場するこれら3つを念頭に置いておいてください。

1.5 まとめ

この章では、Kubernetesの基本的な概念について簡単に紹介しました。言葉や概念を読んだことで、少しは恐怖心が払拭できたでしょうか？

Kubernetesは単なる「コンテナ管理ツール」であり、私達の開発や運用を助けてくれるパートナーです。

Docker Composeで扱うシステムに比べて規模が大きかったり、可用性を求められるシステムでの真価を發揮します。

マニフェストという形でシステムのあるべき姿の指示書を作成することで、Kubernetesはその指示書通りにシステムが稼働しているか監視、管理を行います。

マニフェストはPodやDeployment、Serviceなどのリソースと呼ばれる単位で記述します。リソースが互いに連携することで、複雑な仕組みを実現できます。

次章からは具体的にマニフェストを書いていき、Kubernetes上でシステムを動かしていきます。まだぼんやりとした理解かもしれません、実際に触りながら理解を深めていきましょう！

第2章 環境構築をしよう

2.1 想定環境

本書で想定している環境は、Mac(Intel, Apple Silicon), Linux, Windows(WSL)です。サンプルコードは、基本的にはMacのものを記載しています。他環境では参考ドキュメントを示しますので、適宜読み替えてください。

また、第3章「Lesson1 Kubernetesを触ってみよう」～第5章「Lesson3 Kubernetes上にElasticsearchクラスターを立てよう（ローカル編）」ではローカルでの開発になります。第6章「Lesson4 Kubernetes上にElasticsearchクラスターを立てよう（Azure編）」では、Azureでの開発を行います。

2.2 Dockerのインストール

まずはDockerを使えるようにします。Dockerの環境構築については、公式ドキュメントで丁寧に解説されています¹。

WindowsとMacについては、専用のインストーラーを使ってインストールが可能です。Linuxマシンについては、aptなどの各種パッケージ管理ツールからインストールが可能です。

本書で想定しているDockerのバージョンは、以下のとおりです。

リスト2.1: Dockerのバージョン情報

```
$ docker version
Client:
Version:          27.0.3
API version:      1.46
Go version:       go1.21.11
Git commit:       7d4bcd8
Built:            Fri Jun 28 23:59:41 2024
OS/Arch:          darwin/amd64
Context:          desktop-linux

Server: Docker Desktop 4.32.0 (157355)
Engine:
Version:          27.0.3
API version:      1.46 (minimum version 1.24)
Go version:       go1.21.11
```

¹<https://docs.docker.jp/get-docker.html>

```
Git commit: 662f78c
Built: Sat Jun 29 00:02:50 2024
OS/Arch: linux/amd64
Experimental: false
containerd:
Version: 1.7.18
GitCommit: ae71819c4f5e67bb4d5ae76a6b735f29cc25774e
runc:
Version: 1.7.18
GitCommit: v1.1.13-0-g58aa920
docker-init:
Version: 0.19.0
GitCommit: de40ad0
```

2.3 kindのインストール

続いて、Kubernetesの環境構築を行います。今回は、kind(Kubernetes in Docker)を利用します。kindはローカルでKubernetesクラスターを簡単に構築するためのツールです。軽量なツールのため、ローカルでの開発環境としてKubernetesを扱うのに長けています。同じようなツールとしては、minikubeなども有名です²。

kindについても、インストール方法が公式ドキュメントで丁寧に解説されています³。たとえば、Homebrewが使える環境であれば、以下のコマンドでインストールが可能です。

リスト2.2: kindのインストールコマンド

```
$ brew install kind
```

バージョンについては、以下のコマンドで確認できます。

リスト2.3: kindのバージョン確認

```
$ kind version
kind v0.23.0 go1.22.3 darwin/amd64
```

kindが正常にインストールできていれば、

2.<https://minikube.sigs.k8s.io/docs/>

3.<https://kind.sigs.k8s.io/docs/user/quick-start/>

リスト 2.4: kind でクラスターを構築する

```
$ kind create cluster
```

というコマンドで、ローカルに Kubernetes のクラスターを立ち上げることができます。

デフォルトでは、kind-kind という名前でクラスターは立ち上がります。次のようにオプションを指定すれば、お好みの名前でクラスターを構築できます。

リスト 2.5: クラスター名を指定する

```
$ kind create cluster --name my-cluster
```

使用上の注意

kind はあくまでローカルでの開発用のツールです。そのため、本番環境での運用は推奨しません。

2.4 kubectl のインストール

kind で、Kubernetes のクラスターは構築できるようになりました。続いて、Kubernetes クラスターと通信するためのコマンドラインツール kubectl をインストールします。kubectl は Kubernetes クラスターを操作する標準的なツールです。こちらは本番環境でも使用されています。

kubectl についても、公式ドキュメントにインストール方法が載っています⁴。ちなみに、kind の公式ドキュメントについてもこちらのページから飛べます。

Homebrew が使えるのであれば、以下のコマンドでインストールができます。

リスト 2.6: kubectl のインストール

```
$ brew install kubectl
```

バージョンの確認は、以下のコマンドで行います。

⁴<https://kubernetes.io/docs/tasks/tools/>

リスト 2.7: kubectl のバージョン確認

```
$ kubectl version  
Client Version: v1.29.2  
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3  
Server Version: v1.30.0
```

2.5 Helm のインストール

最後に、Helm のインストールを行います。Helm は Kubernetes 用のパッケージマネージャーです。アプリケーションやサービスを Kubernetes にデプロイする際に非常に役立つツールです。Helm は第5章「Lesson3 Kubernetes 上に Elasticsearch クラスターを立てよう（ローカル編）」以降で使用します。

Homebrew が使える環境であれば、以下のコマンドでインストールできます。

リスト 2.8: helm のインストール

```
$ brew install helm
```

バージョンは、以下のコマンドで確認できます。

リスト 2.9: helm のバージョン確認

```
$ helm version  
version.BuildInfo{Version:"v3.16.1", GitCommit:"5a5449dc42be07001fd5771d56429132984ab3ab",  
GitTreeState:"dirty", GoVersion:"go1.23.1"}
```

Helm のインストールについても、詳細は公式ドキュメントを参照ください⁵。

Helm のバージョンに注意

Helm2 と Helm3 では仕様が異なっています。Helm2 用で作成された Helm チャートは、Helm3 では動作しない場合があります。

以上で、開発に必要な環境構築は完了です。

VS Code のおすすめ拡張機能

Visual Studio Code の拡張機能として、Kubernetes 用の拡張機能があると便利です⁶。たとえば、Kubernetes の設定ファイルの補完などを行ってくれます。

6.<https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools>

5.<https://helm.sh/ja/docs/intro/install/>

第3章 Lesson1 Kubernetesを触ってみよう

3.1 Podを作成しよう

まずは、kindを使ってKubernetesクラスターの起動をします。

リスト3.1: Kubernetesクラスターを起動する

```
$ kind create cluster
```

しばらく待つと、以下のようにクラスターが作成されたメッセージが出ます。

リスト3.2: クラスターの構築結果

```
Creating cluster "kind" ...
✓ Ensuring node image (kindest/node:v1.30.0)
✓ Preparing nodes
✓ Writing configuration
✓ Starting control-plane
✓ Installing CNI
✓ Installing StorageClass
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Thanks for using kind!
```

続いて、Podの定義ファイルを作成します。第1章「Kubernetesは怖くない」でお話した通り、PodはKubernetesの中で最も基本的で小さい単位のリソースです。ここでは、单一のコンテナを動かします。コンテナにはApacheを使用します。

以下のように、`pod.yaml`というファイルを作成します。

リスト3.3: pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    name: my-pod
```

```
spec:  
  containers:  
    - name: my-pod  
      image: httpd:latest  
      ports:  
        - containerPort: 80
```

ここでのポイントは次のとおりです。

- `apiVersion`: KubernetesのAPIバージョン。Podではv1を使用する
- `kind`: リソースの種類を指定する
- `metadata`: Podの名前やラベルなどを定義する
- `spec`: コンテナの詳細を定義する。ここでは、Apacheの最新イメージを使用し、80ポートを開放する

ところどころ、記法が`docker-compose.yaml`に似ていますね。

Podの定義ができたら、これをKubernetesクラスター上に適用しましょう。

リスト3.4: Pod定義を適用する

```
$ kubectl apply -f pod.yaml  
pod/my-pod created
```

デプロイができたら、Kubernetes上のPodを確認します。

リスト3.5: Podの確認

```
$ kubectl get pods  
NAME     READY   STATUS    RESTARTS   AGE  
my-pod   1/1     Running   0          65s
```

STATUSがRunningになっていれば、コンテナが正常に動作しています。これでPodの作成は成功です。

3.2 DeploymentでPodを制御しよう

Podをデプロイし、コンテナが稼働するようになりました。ただ、これだけならDockerやDocker Composeでも十分です。ですので、次はDeploymentの定義ファイルを作成し、Kubernetesの強力な機能を体感します。

第1章「Kubernetesは怖くない」でお話した通り、DeploymentはPodの状態を管理するリソースです。Deploymentによってあるべき姿を定義し、Kubernetesはその状態に合致するよう動きます。

さきほどのApacheのPodを複製して、3台のPodが立ち上がるようにしてみましょう。`deployment.yaml`というファイルを作成して、以下のように定義します。

リスト 3.6: deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-deployment
  template:
    metadata:
      labels:
        app: my-deployment
    spec:
      containers:
        - name: my-deployment
          image: httpd:latest
          ports:
            - containerPort: 80
```

ポイントは次のとおりです。

- kind: 今回は Deployment を指定する
 - spec.replicas: 立ち上げる Pod 数を指定する
 - template: Pod の定義を記述する。ここでは、pod.yaml と同じにしている
- Deployment が定義できたら、これを Kubernetes に対して適用しましょう。

リスト 3.7: Deployment を適用する

```
$ kubectl apply -f deployment.yaml
deployment.apps/my-deployment created
```

しばらく待つと、続々と Pod が立ち上がってきます。

リスト 3.8: Pod の起動を確認する

```
$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
my-deployment-69bff89df8-8mqld   1/1     Running   0          8s
my-deployment-69bff89df8-jnmjj   1/1     Running   0          8s
my-deployment-69bff89df8-rl8ln   1/1     Running   0          8s
```

my-pod	1/1	Running	0	2m3s
--------	-----	---------	---	------

先ほど立ち上げたPodとは別に、3つのPodが立ち上がっていることが確認できます。
また、Deploymentの様子も確認しましょう。

リスト3.9: Deploymentを確認する

```
$ kubectl get deployment
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
my-deployment   3/3     3           3          113s
```

AVAILABLEの列が`spec.replicas`で指定した3になっていれば、すべてのPodが正常に起動しています。

ここでKubernetesの代表的な機能である、Podの自動復旧を試してみましょう。なにか異常が発生したケースを想定して、Podをひとつ意図的に落としてみます。

リスト3.10: Podをひとつ削除する

```
$ kubectl delete pod my-deployment-69bff89df8-8mqld
pod "my-deployment-69bff89df8-8mqld" deleted
```

指定するPod名は、上で確認したDeploymentによって作成したPodの中から、適当にひとつ選んでください。その後、再びPodの様子を確認すると、新しいPodが自動で立ち上がっていることがわかります。

リスト3.11: Podの再確認

```
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
my-deployment-69bff89df8-gn2cv   1/1     Running   0          42s
my-deployment-69bff89df8-jnmjj   1/1     Running   0          69s
my-deployment-69bff89df8-rl8ln   1/1     Running   0          69s
my-pod        1/1     Running   0          3m4s
```

AGEが若い一番上のPodがKubernetesによって、自動で起動されたものです。手動作業なしに自動で普及できるのは驚きですよね！この機能がKubernetesの魅力のひとつと言っても、過言ではありません。

また、Deploymentのレプリカ数を変更して再度適用すれば、Pod数を増減させることもできます。試しに、`replicas`を4にしてみましょう。

リスト3.12: deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: my-deployment
spec:
  replicas: 4
  selector:
    matchLabels:
      app: my-deployment
  template:
    metadata:
      labels:
        app: my-deployment
    spec:
      containers:
        - name: my-deployment
          image: httpd:latest
          ports:
            - containerPort: 80
```

これを Kubernetes に適用すると、4つめの Pod が立ち上がります。

リスト 3.13: Deployment を修正して再適用する

```
$ kubectl apply -f deployment.yaml
deployment.apps/myapp configured
$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
my-deployment-69bff89df8-gn2cv   1/1     Running   0          97s
my-deployment-69bff89df8-jnmjj   1/1     Running   0          2m4s
my-deployment-69bff89df8-rl8ln   1/1     Running   0          2m4s
my-deployment-69bff89df8-wtjz7   1/1     Running   0          17s
my-pod                       1/1     Running   0          3m59s
```

逆に不要になったリソースについても、定義ファイルを使って削除できます。

リスト 3.14: リソースを削除する

```
$ kubectl delete -f pod.yaml -f deployment.yaml
pod "my-pod" deleted
deployment.apps "my-deployment" deleted
```

```
$ kubectl get pods  
No resources found in default namespace.
```

Docker や Docker Compose と同じで削除しないといつまでも動き続けるので、不要になったらリソースを削除しましょう。

3.3 Serviceで外部からアクセスできるようにしよう

Kubernetes で Pod を起動しても、そのままではその Pod にほかの Pod や外部のアプリケーションからはアクセスできません。そこで、Service というリソースを定義します。Service を用意することで、Kubernetes クラスター内の Pod や外部のアプリケーションから対象の Pod にアクセスできるようになります。

`service.yaml` というファイルを作成し、以下のような定義を記述します。

リスト 3.15: service.yaml

```
apiVersion: v1  
kind: Service  
metadata:  
  labels:  
    app: my-service  
    name: my-service  
spec:  
  selector:  
    app: my-deployment  
  ports:  
  - protocol: TCP  
    port: 80  
    targetPort: 80  
  type: ClusterIP
```

ポイントは次のとおりです。

- `kind: Service` を指定する
- `spec.selector:` トラフィックをルーティングする Pod の `metadata.labels` を指定する
- `ports:` Service が受け付けるポート番号と Pod 内部のポート番号を指定する
- `type:` アクセス可能なタイプを指定する。ここではクラスター内のみからアクセス可能にしている

Deployment に関しては、先に作成したものを引き続き利用します。

定義ができたら、Service を Kubernetes にデプロイしましょう。Deployment を削除した場合は、Deployment についてもデプロイしておきます。

リスト 3.16: Service を適用する

```
$ kubectl apply -f service.yaml  
service/my-service created
```

正常にデプロイできたかは、以下のコマンドで確認できます。

リスト 3.17: Service を確認する

```
$ kubectl get service  
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE  
kubernetes  ClusterIP  10.96.0.1    <none>        443/TCP     21m  
my-service  ClusterIP  10.96.147.4  <none>        80/TCP      97s
```

これで、Service がクラスター内で Apache の Pod にアクセスするためのエンドポイントを提供できていることが確認できます。

無事デプロイができたら、Service を使って名前解決ができるか確認しましょう。今回は、デプロイした Pod の中に入って確認します。Pod 名を確認して、Docker や Docker Compose のときと似たコマンドで Pod の中に入ります。

リスト 3.18: Pod の中に入る

```
$ kubectl get pods  
NAME                  READY   STATUS    RESTARTS   AGE  
my-deployment-8fd5489bb-472cp  1/1     Running   0          2m46s  
my-deployment-8fd5489bb-64l4m  1/1     Running   0          2m46s  
my-deployment-8fd5489bb-h8kkq  1/1     Running   0          2m46s  
my-pod                 1/1     Running   0          4m18s  
$ kubectl exec -it my-deployment-8fd5489bb-472cp -- /bin/bash
```

Pod の中に入ったら、`my-service` という名前で名前解決ができるか確認します。

リスト 3.19: Service 経由で HTTP リクエストを送る

```
root@my-deployment-8fd5489bb-472cp:/usr/local/apache2# apt get update
root@my-deployment-8fd5489bb-472cp:/usr/local/apache2# apt-get install -y curl
root@my-deployment-8fd5489bb-472cp:/usr/local/apache2# curl my-service
<html><body><h1>It works!</h1></body></html>
```

Apache のデフォルトの HTML ページが返ってくれば、Service が正しく機能していることが確認できます。

確認ができたら、exit で Pod から抜けます。また、不要になったリソースは削除しておきましょう。

3.4 まとめ

本章では、Kubernetes の基本的なリソースとして Pod、Deployment、Service の 3つを使ってみました。Pod の立ち上げ、Pod の自動復旧、Pod へのアクセスなど、基本的な定義や操作方法は体感できたかと思います。次章では、より実践的な機能について学んでいきます。

第4章 Lesson2 Kubernetes上にアプリケーションを作成しよう

4.1 自前のイメージをデプロイしよう

Docker Hub にあるイメージは、マニフェストの適用時に自動でローカルに pull されます¹。では、ローカルでビルドしたイメージを使ったコンテナは、どのようにデプロイすればよいのでしょうか。kind を使って Kubernetes クラスターを作成している場合は、kind 上にビルドしたイメージを読み込むことで使えるようになります。

まずはともかく、自前のイメージを作りましょう。Python で実装した簡単な Web API をデプロイすることを考えます。app.py を以下のように定義します。

リスト 4.1: app.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, Kubernetes!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

http://localhost:5000/ にアクセスした際に "Hello, Kubernetes!" というメッセージを返すだけのシンプルな API です。

これをコンテナイメージとしてビルドします。Dockerfile を以下のように書きます。

リスト 4.2: Dockerfile

```
# ベースイメージとしてPythonを使用
FROM python:3.11-slim

# Flaskをインストール
RUN pip install Flask

# アプリケーションのソースコードをコピー
```

¹<https://hub.docker.com/>

```
COPY app.py /app/app.py

# 作業ディレクトリーを設定
WORKDIR /app

# コンテナ起動時に実行するコマンド
CMD ["python", "app.py"]
```

このDockerfileを使って、イメージをビルドします。

リスト4.3: イメージをビルドする

```
$ docker build -t python-web-app:1.0 .
```

イメージがビルドできたら、kindに読み込みます。kindのクラスターを立ち上げていない場合は、事前に立ち上げておきましょう。

リスト4.4: イメージをkindに読み込ませる

```
$ kind load docker-image python-web-app:1.0
```

これでkindでpython-web-appが使えるようになりました。

続いて、マニフェストを作成してKubernetes上にデプロイしましょう。以下のように、`python-deployment.yaml`を作成します。

リスト4.5: python-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python-web-app
  template:
    metadata:
      labels:
        app: python-web-app
    spec:
      containers:
        - name: python-web-app
          image: python-web-app:1.0
```

```
  ports:
    - containerPort: 5000
```

これだけでは外部からアクセスできないので、`service.yaml`も作成します。

リスト4.6: `service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: python-web-app
spec:
  selector:
    app: python-web-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

マニフェストが書けたら、Kubernetesにデプロイしましょう。

リスト4.7: マニフェストを適用する

```
$ kubectl apply -f python-deployment.yaml -f service.yaml
```

例によって、正常にデプロイできたか確認してみましょう。

リスト4.8: リソースを確認する

```
$ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
python-web-app-77cccd5468f-gmfhk   1/1     Running   0          2m36s
$ kubectl get service python-web-app
NAME         TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
python-web-app   ClusterIP   10.96.7.162   <none>        80/TCP      3m6s
```

問題なさそうですね。あとは、ホストマシンからアクセスできるようにポートフォワーディングをします。

リスト4.9: ホストの5000番ポートをコンテナの80番ポートに転送する

```
$ kubectl port-forward service/python-web-app 5000:80
```

ポートフォワーディングができたら、ブラウザまたはcurlで `http://localhost:5000` に接続してみましょう。これで、"Hello, Kubernetes!"というレスポンスが得られればバッチリです！

リスト 4.10: Pod にアクセスしてみる

```
$ curl http://localhost:5000/  
Hello, Kubernetes!
```

図 4.1: Python の Web API にアクセスできた！

Hello, Kubernetes!

4.2 サービス無停止でアップデートを実施しよう

自前のイメージについても、デプロイができましたね。ただ、アプリケーションは一度作ったら「はい、おしまい」ではないですよね。機能追加やリファクタリングなどで改修を繰り返すものです。

Docker Composeの場合、新しい機能や変更を反映するには一度コンテナを停止・削除して、再構築する必要があります。つまり、更新している間はサービスを停止する時間が生まれてしまします。これは可能な限り避けたいところです。

Kubernetesには、そんなニーズに応えるべく、**ローリングアップデート**という機能があります。ローリングアップデートを使えば、サービスのダウンタイムを最小限に抑えながら、既存のアプリケーションを更新することができます。

さきほどのPythonのアプリケーションの改修とデプロイを通じて、ローリングアップデートを体感してみましょう。アップデート内容として、レスポンスマッセージを変更します。メッセージを"Hello, Kubernetes!"から"Hello, Kubernetes! Updated version!"に変更します。

リスト 4.11: app.py

```
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route('/')
```

```
def hello():
    return "Hello, Kubernetes! Updated version!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

コードを改修したら、再度 Docker イメージをビルドします。以下のコマンドを使って、新しいバージョンのイメージを作成します。イメージタグを 1.1 として、変更が反映された新しいバージョンを作成しています。

リスト 4.12: 修正したコードでイメージを再ビルト & ロード

```
# 新しいタグでイメージをビルト
$ docker build -t python-web-app:1.1 .
# kindに読み込ませる
$ kind load docker-image python-web-app:1.1
```

このイメージにローリングアップデートをするのですが、その定義は Deploymentで行います。Deploymentは、Podのライフサイクル管理を自動化し、ローリングアップデートなどの機能を提供します。

リスト 4.13: python-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python-web-app
  template:
    metadata:
      labels:
        app: python-web-app
    spec:
      containers:
        - name: python-web-app
          image: python-web-app:1.1 # イメージタグを更新する
      ports:
        - containerPort: 5000
```

このDeploymentのマニフェストをKubernetesに適用すると、ローリングアップデートが自動的に実行されます。

リスト4.14: イメージタグを更新したdeploymentを適用

```
$ kubectl apply -f python-deployment.yaml
```

python-deployment.yamlに従って、Kubernetesは既存のPodを新しいバージョンに徐々に置き換えます。新しいバージョンのPodがひとつ立ち上がると、古いPodがひとつ削除されます。これを繰り返して、すべてのPodを新しいバージョンに置き換えます。そのため、アプリケーションのダウンタイムがほとんど発生しません。

ローリングアップデートの進行状況は、以下のコマンドで確認できます。

リスト4.15: ローリングアップデートの様子を眺める

```
$ kubectl rollout status deployment/python-web-app
Waiting for deployment "python-web-app" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "python-web-app" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "python-web-app" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "python-web-app" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "python-web-app" rollout to finish: 1 old replicas are pending termination...
deployment "python-web-app" successfully rolled out
```

このコマンドを実行すると、Deploymentの状態をリアルタイムで確認できます。Podが順々に入れ替えられている様子が見て取れます。すべてのPodが正常に更新されると、「deployment "python-web-app" successfully rolled out」というメッセージが表示されます。

新しいバージョンがデプロイされたことを確認するために、Serviceのエンドポイントに再度アクセスします。<http://localhost:5000>にアクセスして、"Hello, Kubernetes! Updated version!"が返ってくれば、ローリングアップデートが正常に完了できています。

リスト 4.16: アップデートしたアプリケーションにアクセスする

```
$ curl http://localhost:5000/  
Hello, Kubernetes! Updated version!
```

図 4.2: アップデート後の Web API のレスポンス

Hello, Kubernetes! Updated version!

4.3 ConfigMapを使ってパラメーターを外出ししよう

Kubernetesを使ってアプリケーションをデプロイする際、環境ごとにパラメーターを変更したくなることがあると思います。たとえば、開発環境と本番環境で接続するエンドポイントを変えたいなどです。そこで登場するのが、ConfigMapです。

ConfigMapは、Kubernetesでアプリケーションの設定情報を管理するためのリソースです。ConfigMapを使えば、環境ごとに異なる設定を簡単に切り替えることができます。ConfigMapは次のような特徴があります。

- ・アプリケーションの設定ファイルや環境変数を外部から指定できる
- ・開発環境や本番環境など、異なる環境ごとに設定を切り替えられる
- ・コンテナの再ビルトを行わずに設定を変更できる

ConfigMapの活用例として、PostgreSQLとPythonのアプリケーションを接続させてみようと思います。まずは、app.pyを次のように直します。

リスト 4.17: app.py

```
import os  
import psycopg2  
from flask import Flask, jsonify  
  
app = Flask(__name__)
```

```

def get_db_connection():
    db_user = os.getenv("POSTGRES_USER")
    db_password = os.getenv("POSTGRES_PASSWORD")
    db_name = os.getenv("POSTGRES_DB")
    db_host = "postgres"

    try:
        conn = psycopg2.connect(
            host=db_host,
            database=db_name,
            user=db_user,
            password=db_password
        )
        return conn
    except Exception as e:
        print(f"Error connecting to the database: {e}")
        return None

@app.route('/')
def get_current_time():
    ENV = os.getenv("ENV")
    conn = get_db_connection()
    if conn is None:
        return "Database connection failed", 500

    cur = conn.cursor()
    try:
        cur.execute("SELECT NOW()")
        current_time = cur.fetchone()[0]
        return jsonify({
            'current_time': str(current_time),
            'env': ENV
        })
    except Exception as e:
        return str(e), 500
    finally:
        cur.close()
        conn.close()

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)

```

PostgreSQLにアクセスして、現在時刻を取得して表示するというシンプルなものです。接続情報は環境変数から取得しています。この環境変数の定義をConfigMapを使って行います。

`db-config.yaml`は、以下のように記述します。環境変数から読み込みたい値について、ConfigMapで設定しています。

リスト 4.18: db-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  POSTGRES_DATABASE: "sampledb"
  POSTGRES_USER: "user"
  POSTGRES_PASSWORD: "password"
  POSTGRES_ROOT_PASSWORD: "rootpassword"
  ENV: "dev"
```

ConfigMapを実際にPodで使用するには、Podのマニフェスト内でそのConfigMapを参照します。`python-deployment.yaml`に、ConfigMapの設定を追加します。また、Pythonのアプリケーションのイメージについても変更しておきます。

リスト 4.19: python-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python-web-app
  template:
    metadata:
      labels:
        app: python-web-app
  spec:
    containers:
      - name: python-web-app
        image: python-web-app:1.2 # イメージタグを更新する
        envFrom: # ConfigMapを参照
          - configMapRef:
```

```
name: db-config
ports:
- containerPort: 5000
```

ここまでできたら、一度イメージをビルドしておきましょう。インストールパッケージが増えてきたので、`requirements.txt`を作成しておきます。

リスト 4.20: requirements.txt

```
Flask==3.0.3
psycopg2-binary==2.9.9
```

Dockerfile も requirements.txt からインストールするように書き直します。

リスト 4.21: Dockerfile

```
# ベースイメージとしてPythonを使用
FROM python:3.11-slim

# Flaskをインストール
COPY requirements.txt .
RUN pip install -r requirements.txt

# アプリケーションのソースコードをコピー
COPY app.py /app/app.py

# 作業ディレクトリーを設定
WORKDIR /app

# コンテナ起動時に実行するコマンド
CMD ["python", "app.py"]
```

これをビルドします。

リスト 4.22: イメージの再ビルドとロード

```
$ docker build -t python-web-app:1.2 .
$ kind load docker-image python-web-app:1.2
```

続いて、PostgreSQL の Pod を用意します。`postgresql-deployment.yaml` を新規に作成して、以下のように定義します。

リスト 4.23: postgresql-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  selector:
    matchLabels:
      app: postgres
  replicas: 1
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:14.13
          envFrom:
            - configMapRef:
                name: db-config
          ports:
            - containerPort: 5432
```

こちらでも、db-configを使えるように設定しておきます。また、PostgreSQLのPodにアクセスできるよう、`postgres-service.yaml`も用意します。

リスト 4.24: postgresql-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  ports:
    - port: 5432
  selector:
    app: postgres
```

これで必要なマニフェストはすべて書けました。あとはそれぞれをデプロイしましょう。

リスト 4.25: マニフェストを適用する

```
$ kubectl apply -f db-config.yaml  
$ kubectl apply -f postgres-deployment.yaml -f postgres-service.yaml  
$ kubectl apply -f python-deployment.yaml
```

しばらく待つと、それぞれPodが立ち上がってきます。

リスト 4.26: Podの起動確認する

```
$ kubectl get pods  
NAME                      READY   STATUS    RESTARTS   AGE  
postgres-db7bcf545-ck4fv   1/1     Running   0          2m6s  
python-web-app-7c5c768674-7xjsf 1/1     Running   0          2m6s  
python-web-app-7c5c768674-8v7dv 1/1     Running   0          2m6s  
python-web-app-7c5c768674-cqfxd 1/1     Running   0          2m6s
```

PostgreSQLはPodが立ち上がってからも、プロセスが完全に起動し切るまで少し待つ必要があります。

リスト 4.27: Podのログを見て起動を待つ

```
$ kubectl logs -f postgres-db7bcf545-ck4fv  
2024-10-11 11:35:21.907 UTC [1] LOG:  listening on Unix socket  
"/var/run/postgresql/.s.PGSQL.5432"  
2024-10-11 11:35:21.913 UTC [66] LOG:  database system was shut down at  
2024-10-11 11:35:21 UTC  
2024-10-11 11:35:21.920 UTC [1] LOG:  database system is ready to accept  
connections
```

PostgreSQLサーバーが完全に立ち上がったら、ポートフォワードをして <http://localhost:5000> にアクセスしてみましょう。無事現在時刻が表示されていれば、バッチリです。

リスト 4.28: アプリケーションにアクセスする

```
$ curl http://localhost:5000/  
{"current_time": "2024-10-11 11:40:38.414641+00:00", "env": "dev"}
```

図 4.3: PostgreSQL と接続させた Web API

```
1 // 20241011203955  
2 // http://localhost:5000/  
3  
4 {  
5     "current_time": "2024-10-11 11:39:54.979956+00:00",  
6     "env": "dev"  
7 }
```



デプロイしたConfigMapの内容は、`kubectl`コマンドで確認することができます。このコマンドを実行すると、ConfigMapの名前やその他の基本情報が一覧で表示されます。

リスト 4.29: ConfigMap のリソースを確認する

```
$ kubectl get configmaps  
NAME          DATA   AGE  
db-config      4      74s  
kube-root-ca.crt 1      7m55s
```

特定のConfigMapの詳細な内容を確認したい場合は、以下のコマンドを実行します。

リスト 4.30: ConfigMap リソースの詳細を確認する

```
$ kubectl describe configmaps db-config  
Name:         db-config  
Namespace:    default  
Labels:       <none>  
Annotations:  <none>  
  
Data  
=====  
POSTGRES_PASSWORD:  
----
```

```
password
POSTGRES_ROOT_PASSWORD:
-----
rootpassword
POSTGRES_USER:
-----
user
ENV:
-----
dev
POSTGRES_DATABASE:
-----
sampledb

BinaryData
=====

Events: <none>
$ kubectl get configmap db-config -o yaml
apiVersion: v1
data:
  ENV: dev
  POSTGRES_DATABASE: sampledb
  POSTGRES_PASSWORD: password
  POSTGRES_ROOT_PASSWORD: rootpassword
  POSTGRES_USER: user
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"ENV":"dev","POSTGRES_DATABASE":"sampledb","POSTGRES_PASSWORD":"password","POSTGRES_ROOT_PASSWORD":"rootpassword","POSTGRES_USER":"user"}}
    creationTimestamp: "2024-10-11T11:35:05Z"
  name: db-config
  namespace: default
  resourceVersion: "515"
  uid: ae5bedbc-2c2a-48d0-9e0a-b06011e59176
```

ConfigMap を変更すると、イメージの再ビルトを行うことなく値の変更ができます。たとえば、`db-config.yaml` の `ENV` を `dev` から `pro` に変更します。

リスト 4.31: db-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  POSTGRES_DATABASE: "sampledb"
  POSTGRES_USER: "user"
  POSTGRES_PASSWORD: "password"
  POSTGRES_ROOT_PASSWORD: "rootpassword"
  ENV: "pro" # devからproに変更
```

これを適用します。

リスト 4.32: ConfigMap を更新して適用する

```
$ kubectl apply -f db-config.yaml
```

リソースを確認すると、反映されていますね。

リスト 4.33: 更新した ConfigMap リソースを確認する

```
$ kubectl get configmap db-config -o yaml
apiVersion: v1
data:
  ENV: pro
  POSTGRES_DATABASE: sampledb
  POSTGRES_PASSWORD: password
  POSTGRES_ROOT_PASSWORD: rootpassword
  POSTGRES_USER: user
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"ENV":"pro","POSTGRES_DATABASE":"sampledb","POSTGRES_PASSWORD":"pas
creationTimestamp: "2024-10-11T11:35:05Z"
name: db-config
namespace: default
resourceVersion: "1326"
uid: ae5bedbc-2c2a-48d0-9e0a-b06011e59176
```

Podにも反映させるためには、Podの再デプロイが必要です。サービスを止めずにPodの再デプロイをしたいので、ここでもローリングアップデートが活躍します。マニフェストを変更せずにロー

リングアップデートをさせる場合は、次のようにします。

リスト 4.34: Pod のローリングアップデートをする

```
$ kubectl rollout restart deployment python-web-app
```

これにより、Deployment内の全Podが順次再起動され、新しいConfigMapの内容が反映されます。kubectl rollout restartコマンドは、Deploymentの設定に変更がない場合でもPodを再起動します。このため、ConfigMapの変更を反映させるために便利です。

http://localhost:5000 にアクセスして、画面にも反映されたか見てみましょう。

リスト 4.35: アプリケーションにアクセスする

```
$ curl http://localhost:5000/
{"current_time": "2024-10-11 12:05:16.557876+00:00", "env": "pro"}
```

図 4.4: ENV パラメータを変更

```
1 // 20241011210500
2 // http://localhost:5000/
3
4 ▼ {
5     "current_time": "2024-10-11
6         12:04:59.507212+00:00",
7     "env": "pro"
8 }
```



もし、ローリングアップデート中にトラブルが発生した場合は、以下のコマンドで以前の状態にロールバックができます。

リスト 4.36: デプロイをロールバックする

```
$ kubectl rollout undo deployment python-web-app
```

4.4 機密情報は Secret で渡そう

ConfigMap を使うことで、イメージを再ビルドすることなくパラメーターを変更することができます。しかし、パスワードを平文で管理するのは、セキュリティー的にいただけないです。

そこでKubernetesでは、機密性の高い情報を安全に管理するためにSecretというリソースが提供されています。ConfigMapと似た仕組みですが、機密性の高い情報を扱う場合に使用されます。

たとえば、以下のようなシーンでSecretが活躍します。

- ・データベース接続情報: アプリケーションがデータベースに接続するためのパスワードやユーザー名
- ・APIキーやトークン: 外部サービスと連携するために必要なAPIキーやトークン
- ・TLS証明書: HTTPS通信を行う際に必要なTLS証明書と秘密鍵

これらの情報は、セキュリティー上の観点から環境変数や設定ファイルに直接埋め込むべきではありません。ConfigMapは一般的な設定情報を扱い、パスワードのような機密情報はSecretで安全に利用するべきです。このセクションでは、PostgreSQLの接続パスワードをConfigMapからSecretに移行する手順を紹介します。

以下のように、db-secret.yamlでPostgreSQL接続用のパスワードを含むSecretを定義します。SecretにはBase64エンコードした値を保存します。

リスト4.37: db-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  POSTGRES_PASSWORD: cGFzc3dvcmQ= # "password"をBase64エンコードしたもの
  POSTGRES_ROOT_PASSWORD: cm9vdHBhc3N3b3Jk # "rootpassword"をBase64エンコードしたもの
```

Base64でのエンコードは、以下のコマンドでできます。

リスト4.38: Base64でパスワードをエンコードする

```
$ echo -n "password" | base64
cGFzc3dvcmQ=
```

これにより、エンコードされたパスワード(cGFzc3dvcmQ=)が取得できます。この値をSecretのdataフィールドに記述します。

--decodeオプションを付ければデコードができます。

リスト4.39: Base64でデコードする

```
$ echo "cGFzc3dvcmQ=" | base64 --decode
password
```

マニフェストファイルを作成したら、以下のコマンドでSecretをクラスターに適用します。

リスト 4.40: Secret を適用する

```
$ kubectl apply -f db-secret.yaml
```

正常に適用できたかは、以下のコマンドで確認できます。

リスト 4.41: Secret を確認する

```
$ kubectl get secrets -o yaml
apiVersion: v1
items:
- apiVersion: v1
  data:
    POSTGRES_PASSWORD: cGFzc3dvcmQ=
    POSTGRES_ROOT_PASSWORD: cm9vdHBhc3N3b3Jk
  kind: Secret
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"v1","data":{"POSTGRES_PASSWORD":"cGFzc3dvcmQ=","POSTGRES_ROOT_PASSWORD":"cm9vdHBhc3N3b3Jk"}}
    creationTimestamp: "2024-10-11T12:47:36Z"
    name: db-secret
    namespace: default
    resourceVersion: "499"
    uid: 7d6a0137-b46a-4848-a02d-2cc3266ac76d
  type: Opaque
kind: List
metadata:
  resourceVersion: ""
```

デコードをすれば、平文が確認できます。

リスト 4.42: Secret で定義したパスワードをデコードする

```
$ kubectl get secrets db-secret -o json | jq -r '.data."POSTGRES_PASSWORD"' |
base64 --decode
password
```

これを Pod から利用できるようにします。postgresql-deployment.yaml を以下のように書き換えます。

リスト 4.43: postgresql-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  selector:
    matchLabels:
      app: postgres
  replicas: 1
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:14.13
          env:
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secret # ここでSecretを参照
                  key: POSTGRES_PASSWORD
            - name: POSTGRES_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secret # ここでSecretを参照
                  key: POSTGRES_ROOT_PASSWORD
              envFrom:
                - configMapRef:
                    name: db-config
          ports:
            - containerPort: 5432
```

これで環境変数にdb-secretから取得した値を書き込みます。同様にpython-deployment.yamlも、Secretの値を取得できるように修正します。

リスト 4.44: python-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```

name: python-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python-web-app
  template:
    metadata:
      labels:
        app: python-web-app
    spec:
      containers:
        - name: python-web-app
          image: python-web-app:1.2
          env:
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secret # ここでSecretを参照
                  key: POSTGRES_PASSWORD
            envFrom:
              - configMapRef:
                  name: db-config
      ports:
        - containerPort: 5000

```

ConfigMap でパスワードの情報は不要になったので、db-config.yamlからは削除しておきます。

リスト 4.45: db-config.yaml

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  POSTGRES_DATABASE: "sampledb"
  POSTGRES_USER: "user"
  ENV: "dev"

```

各種修正ができたら、それぞれ適用します。

リスト 4.46: マニフェストを適用する

```
$ kubectl apply -f postgresql-deployment.yaml -f python-deployment.yaml -f db-config.yaml
```

ポートフォワードして `http://localhost:5000` にアクセスすると、問題なく PostgreSQL に接続できているかと思います。

リスト 4.47: アプリケーションにアクセスする

```
$ curl http://localhost:5000/
{"current_time":"2024-10-11 12:50:09.570288+00:00","env":"dev"}
```

ConfigMap を確認すると、パスワードはなくなっています。

リスト 4.48: ConfigMap を確認する

```
$ kubectl get configmap db-config -o yaml
apiVersion: v1
data:
  ENV: dev
  POSTGRES_DATABASE: sampledb
  POSTGRES_USER: user
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"ENV":"dev","POSTGRES_DATABASE":"sampledb","POSTGRES_USER":"user"}},
  creationTimestamp: "2024-10-11T12:47:36Z"
  name: db-config
  namespace: default
  resourceVersion: "498"
  uid: 9915c24d-15d2-4407-8d18-75e95d16e341
```

これで、機密情報を安全に Pod に渡すことができました。

体感いただけたように、Secret は Kubernetes クラスター内で安全に機密情報を管理するために有用な機能です。しかし、より適切に運用するためには、いくつか注意点があります。

- ・暗号化の使用: Secret には暗号化して保存していても、etcd（Kubernetes クラスター内のデータストア）には平文で保存されている。EncryptionConfiguration リソースを使って、etcdへの保存内容も暗号化を有効にすることで、より強力に Secret を保護できる。
- ・RBAC (Role-Based Access Control) の利用: Secret に対するアクセスは、適切な Role と RoleBinding を設定して管理するのが望ましい。特定のユーザーやサービスアカウントにのみ Secret へのアクセス権限を与え、不必要的ユーザーが Secret にアクセスできないようにすることが重要である。

- Secretの監視とローテーション: 機密情報が漏洩するリスクを最小限に抑えるために、Secretは定期的に更新することが推奨される。また、Secretの変更が適切に適用されているかを確認するための監視も重要である。

これらの点に注意して、Secretを適切に活用しましょう。うまく活用できれば、アプリケーションのセキュリティが向上し、より安全で安定した運用が可能となります。

4.5 RoleとRoleBindingで権限管理をしよう

Secretリソースを使うことで、機密性の高い情報を暗号化して使用することができました。ただ、単にSecretを使うだけでは、十分に安全とは言い切れません。そこで、KubernetesのRole-Based Access Control (RBAC)機能を使用しましょう。RBACを使うことで、特定のユーザーやサービスアカウントに対して、リソースの読み取りや書き込みといった操作権限を細かく制御できます。RBACでは、以下のような操作の権限制御ができます。

- Secretの作成 (create)
- Secretの取得 (get)
- Secretのリスト表示 (list)
- Secretの削除 (delete)

まずは、アクセス権限を定義するRoleというリソースを作成します。以下は、db-configというSecretに対して「読み取り」権限を持つRoleの定義です。secret-reader-role.yamlというファイルを作成して、以下のように定義します。

リスト 4.49: secret-reader-role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: secret-reader
  namespace: default
rules:
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["db-secret"] # このSecretにのみアクセスを制限
  verbs: ["get"]
```

このRoleは、名前空間defaultにあるdb-configという名前のSecretに対して、get(読み取り)権限を与えます。他のSecretやリソースには、アクセスできません。

Roleは、特定の名前空間内でのアクセス制御を定義します。クラスター全体ではなく、名前空間単位でのリソースへのアクセス権限を管理する点に注意です。

マニフェストが書けたら適用しましょう。

リスト 4.50: Role を適用する

```
$ kubectl apply -f secret-reader-role.yaml
```

確認すると、正常にRoleが作成されていますね。

リスト 4.51: Role リソースを確認する

```
$ kubectl get role
NAME          CREATED AT
secret-reader  2024-10-11T12:58:03Z
$ kubectl get role -o yaml
apiVersion: v1
items:
- apiVersion: rbac.authorization.k8s.io/v1
  kind: Role
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"rbac.authorization.k8s.io/v1","kind":"Role","metadata":{"annotations":{},"name":"secret-reader","namespace":"default","resourceVersion":"1390","uid":"faecc734-9a66-49f9-8230-c4f7af311521"}}
        creationTimestamp: "2024-10-11T12:58:03Z"
    name: secret-reader
    namespace: default
    resourceVersion: "1390"
    uid: faecc734-9a66-49f9-8230-c4f7af311521
  rules:
  - apiGroups:
    - ""
      resourceNames:
      - db-config
      resources:
      - secrets
      verbs:
      - get
  kind: List
  metadata:
    resourceVersion: ""
```

これでRoleの作成はできました。しかし、Roleを定義しただけでは権限制御は有効になりません。特定のユーザーやサービスアカウントに、そのRoleをバインド（紐づけ）する必要があります。これには、RoleBindingというリソースを使用します。

以下の例では、secret-readerというRoleをmy-app-saというサービスアカウントにバインドしています。secret-reader-binding.yamlを作成して、以下のように定義します。

リスト 4.52: secret-reader-binding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-secret-reader
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-app-sa # ここでサービスアカウントを指定
  namespace: default
roleRef:
  kind: Role
  name: secret-reader # 先ほど作成したRoleを指定
  apiGroup: rbac.authorization.k8s.io
```

サービスアカウント secret-reader-sa.yaml も作成しましょう。

リスト 4.53: secret-reader-sa.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-app-sa
  namespace: default
```

RoleBinding と ServiceAccount が定義できたら、適用しましょう。

リスト 4.54: マニフェストを適用する

```
$ kubectl apply -f secret-reader-binding.yaml -f secret-reader-sa.yaml
```

この設定により、my-app-sa というサービスアカウントを使って動作する Pod は、Secret の読み取りが可能になります。

適切な権限制御ができているかは、以下のコマンドで確認できます。

リスト 4.55: 権限を確認する

```
# 読み取り権限があるか
$ kubectl auth can-i get secret/db-secret --as=system:serviceaccount:default:my-app-sa
yes
# 更新権限があるか
$ kubectl auth can-i patch secret/db-secret --as=system:serviceaccount:default:my-app-sa
no
```

これを見ると、my-app-saにはdb-secretの読み取り権限はある（yes）が、更新権限はない（no）ことがわかります。

Podにもサービスアカウントを紐づけましょう。

リスト 4.56: python-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python-web-app
  template:
    metadata:
      labels:
        app: python-web-app
    spec:
      serviceAccountName: my-app-sa # サービスアカウントを指定
      containers:
        - name: python-web-app
          image: python-web-app:1.2
          env:
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secret # ここでSecretを参照
                  key: POSTGRES_PASSWORD
              envFrom:
                - configMapRef:
                    name: db-config
          ports:
            - containerPort: 5000
```

マニフェストを更新したら、忘れずに適用しておきましょう。

リスト 4.57: 更新した Deployment を適用する

```
$ kubectl apply -f python-deployment.yaml
```

これで実際に適当な Pod に入って確認すると、読み取り以外の操作ができないことがわかります。

リスト 4.58: Podのアクセス権限を確認する

```
$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
postgres-bc78df6c4-drnfw   1/1     Running   0          40m
python-web-app-69bf9889b8-4l5mp   1/1     Running   0          6m39s
python-web-app-69bf9889b8-fxjgg   1/1     Running   0          6m35s
python-web-app-69bf9889b8-vkncz   1/1     Running   0          6m37s

# kubectlをインストール
$ kubectl exec -it python-web-app-69bf9889b8-4l5mp -- /bin/bash
root@python-web-app-69bf9889b8-4l5mp:/app# apt-get update && apt-get install -y
apt-transport-https ca-certificates curl gnupg
root@python-web-app-69bf9889b8-4l5mp:/app# curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.3
| gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
root@python-web-app-69bf9889b8-4l5mp:/app# chmod 644 /etc/apt/keyrings/kubernetes-apt-keyring.gpg
root@python-web-app-69bf9889b8-4l5mp:/app# echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.31/deb/' | tee /etc/apt/sources.list.d/kubernetes.list
root@python-web-app-69bf9889b8-4l5mp:/app# chmod 644 /etc/apt/sources.list.d/kubernetes.list
root@python-web-app-69bf9889b8-4l5mp:/app# apt-get update
root@python-web-app-69bf9889b8-4l5mp:/app# apt-get install -y kubectl
# getはできる
root@python-web-app-69bf9889b8-4l5mp:/app# kubectl get secret db-secret
NAME      TYPE      DATA   AGE
db-secret  Opaque    2      7m36s
# patchはできない
root@python-web-app-69bf9889b8-4l5mp:/app# kubectl patch secret db-secret -p
'{"data":{"POSTGRES_PASSWORD":"newpassword"}}'
Error from server (Forbidden): secrets "db-secret" is forbidden: User
"system:serviceaccount:default:my-app-sa" cannot patch resource "secrets" in
API group "" in the namespace "default"
```

このようにRBACを活用することで、特定のユーザーやサービスアカウントにのみSecretへの必要最小限のアクセス権限を付与できました。今回はSecretを対象としましたが、ConfigMapなど他のリソースにも同様に権限制御ができます。

RoleやRoleBindingは、特定の名前空間内の権限制御でした。クラスター全体に対して、Secretや他のリソースへのアクセスを制御する場合は、ClusterRoleとClusterRoleBindingを使用します。これらを使えば、全名前空間に対してアクセス権限を付与できます。

RBACを使って、よりセキュアにKubernetesクラスターを運用しましょう。

4.6 オートスケーリングを使って高可用性を実現しよう

この章の締めとして、オートスケーリングを扱います。サービスを運用しているとメディアで取り上げられたり、イベントによって普段は想定もしないようなリクエストが来ることがあります。一般的なシステムでは、あらかじめサーバーの台数やスペックは決まっており、その範囲内でリクエストをさばきます。さばききれない量のリクエストが来た場合は、最悪システムがダウンしてしまうことでしょう。なので、あらかじめピーク予想を立てておき、その負荷に耐えられるようにサーバースペックやシステム構成を設計しておきます。

しかし、常にピーク時に近いリクエスト数が来ているわけではないと思います。あらかじめピーク時に耐えうるサーバースペックを確保しておくわけですが、普段はオーバースペックとなっています。サーバー費用もタダではないので、安全のために普段は過剰な投資をしていることになります。かといって、サービスの利用状況を見て人力で都度リソース調整を行うのは現実的ではありません。

システムダウンがもっとも避けるべき事態とするならば、過剰投資は妥当な投資と言えるでしょう。しかし欲を言えば、そのときに必要な分だけのコストを払えばいい状態だと理想的ですよね。それを実現可能にするのが、Kubernetesのオートスケーリング機能です。

オートスケーリングは、アプリケーションの負荷に応じて自動的にPodの数を増減させる非常に便利な機能です。本書では、Pythonのアプリケーションに負荷をかけることで、オートスケーリングが発動する様子を体感してみましょう。

Kubernetesでは、Horizontal Pod Autoscaler (HPA) というリソースを使うことで、オートスケールが実現できます。autoscaler.yamlを作成して、以下のように定義します。

リスト 4.59: autoscaler.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: python-web-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: python-web-app # ここでスケール対象のDeploymentを指定
  minReplicas: 2        # 最小Pod数
  maxReplicas: 10       # 最大Pod数
  metrics:
  - type: Resource
    resource:
      name: cpu        # CPU使用率に基づいてスケール
    target:
      type: Utilization
      averageUtilization: 50 # 50%のCPU使用率を目標にスケーリング
```

この設定では、Deployment全体でCPU使用率が50%を超えたときに、新しいPodが追加されるようになっています。Podの最小数は2、最大数は10としています。

マニフェストが書けたら適用しましょう。

リスト 4.60: HPA を適用する

```
$ kubectl apply -f autoscaler.yaml
```

これでHPAのリソースが作成できました。

リスト 4.61: HPA リソースを確認する

```
$ kubectl get hpa
NAME           REFERENCE          TARGETS          MINPODS   MAXPODS
REPLICAS     AGE
python-web-app-hpa   Deployment/python-web-app   cpu: <unknown>/50%   2
10            3                 1s
```

ただ、これだけだと不十分です。見ての通り、TARGETSの部分が<unknown>状態になっています。リソースの負荷状況がKubernetes（kube-api-server）から見える状態を整える必要があります。

そのためのツールが**Metrics Server**です²。Metrics Serverはコンテナを監視しているkubeletのSummary APIから統計情報を収集して、kube-api-serverからアクセスできるようにするツールです。

公式ドキュメントのインストール手順に従うと、公式が用意しているMetrics Serverのマニフェストを適用するだけでインストールが完了すると書かれています。

リスト 4.62: MetricsServer を適用する

```
$ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download
```

一般的にはこれでよいのです。ですが、kindを使ってKubernetesクラスターを立てている場合は、オプションが足らず、これではMetrics ServerのPodは起動しません。

そこで、マニフェストをカスタマイズして使用します。まずは、さきほどのマニフェストをローカルにダウンロードします。

リスト 4.63: MetricsServer のマニフェストファイルをダウンロードする

```
$ wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components
-0 metrics-server.yaml
```

そして、Deploymentの部分を探し、以下のように編集します。

²<https://github.com/kubernetes-sigs/metrics-server>

リスト 4.64: metrics-server.yaml

```
spec:  
  containers:  
    - args:  
        - --cert-dir=/tmp  
        - --secure-port=4443  
        - --kubelet-preferred-address-types=InternalIP,Hostname,ExternalIP  
        - --kubelet-insecure-tls # この行を追加  
    image: k8s.gcr.io/metrics-server/metrics-server:v0.6.1  
    name: metrics-server
```

編集ができたら、適用しましょう。

リスト 4.65: MetricsServer のマニフェストを適用する

```
$ kubectl apply -f metrics-server.yaml
```

これで、Metrics Server の Pod が正常に起動するようになります。

リスト 4.66: MetricsServer のリソースを確認する

```
$ kubectl get pod -n kube-system | grep metrics  
NAME                               READY   STATUS    RESTARTS   AGE  
metrics-server-b79d5c976-dmphx     1/1     Running   0          2m
```

Metrics Server を導入すると、`top` コマンドを使ってクラスター全体のリソース状況も確認できるようになります。

リスト 4.67: Top コマンドでクラスターのリソースを確認する

```
$ kubectl top nodes  
NAME           CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%  
kind-control-plane   461m       11%    2406Mi         30%
```

このあたりの設定変更が必要な件は、次のブログ記事で解説されていました³。

最後にもうひと手間加えます。CPU ベースの HPA を機能させるためには、対象の Pod に CPU リクエスト（下限）とリミット（上限）を追加する必要があります。Pod のリソース要求が設定されていないと、HPA はどのタイミングでオートスケーリングをすればいいのか、判別できません。

といっても、Deployment を以下のように修正するだけです。

3.<https://mackyson.hatenablog.com/entry/2022/05/13/020506>

リスト 4.68: python-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python-web-app
  template:
    metadata:
      labels:
        app: python-web-app
    spec:
      serviceAccountName: my-app-sa
      containers:
        - name: python-web-app
          image: python-web-app:1.2
          resources:
            requests:
              cpu: "50m"  # CPUリクエストを追加
              memory: "64Mi"
            limits:
              cpu: "100m"  # CPUリミットを追加
              memory: "128Mi"
          env:
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secret
                  key: POSTGRES_PASSWORD
            envFrom:
              - configMapRef:
                  name: db-config
          ports:
            - containerPort: 5000
```

これも忘れずに適用しておきましょう。

リスト 4.69: Deployment を適用する

```
$ kubectl apply -f python-deployment.yaml
```

しばらくすると、TARGETS に CPU の使用率が表示されるようになります。

リスト 4.70: HPA リソースを確認する

```
$ kubectl get hpa
```

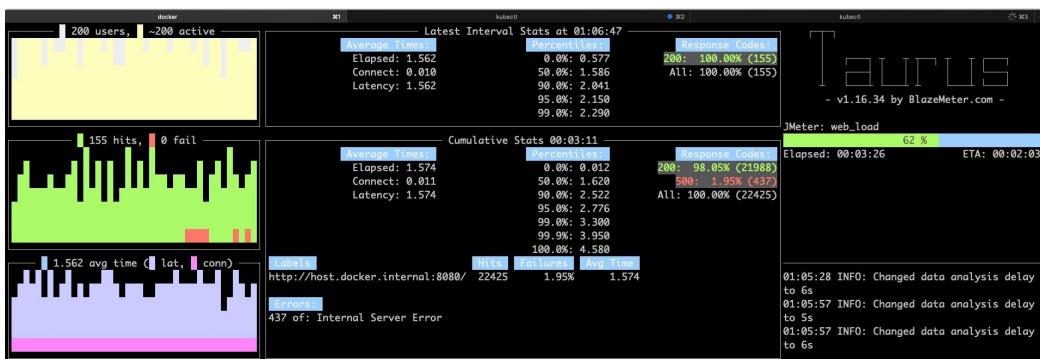
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS
REPLICAS AGE				
python-web-app-hpa	Deployment/python-web-app	cpu: 0%/50%	2	10
2	4m			

これで、正常に HPA が設定されました。いよいよアプリケーションに負荷をかけ、自動的に Pod がスケールする様子を観察してみましょう。

そのためにも、負荷試験ツールを用意します。今回は Taurus を使用します⁴。 YAML 形式で、簡単にテストシナリオを記述できるのが特徴です。

- ・シンプルな設定: 複雑なスクリプトを書かなくても、簡単な YAML ファイルを使ってテストシナリオが作成できる
- ・他ツールとの連携: JMeter や Gatling など、他の負荷テストツールとも統合可能
- ・レポート生成: テスト結果をわかりやすくレポートとして生成してくれるので、テスト後の分析がしやすい

図 4.5: Taurus 実行中の画面



今回は、`http://localhost:5000` に対して、200 並列でリクエストを5分間かけ続けるシナリオを作成します。より正確には、Taurus の Docker コンテナから localhost にアクセスするために、`host.docker.internal` という専用の DNS を指定しています。

4. <https://gettaurus.org/>

リスト 4.71: test.yaml

```
execution:
  - concurrency: 50
    hold-for: 5m
    ramp-up: 30s
    scenario: web_load

scenarios:
  web_load:
    requests:
      - http://host.docker.internal:5000/
```

これを `test.yaml` として、シナリオファイルを保存しておきます。テストシナリオが書けたら、負荷試験を実施しましょう。あらかじめ、`http://localhost:5000/` にアクセスできる状態にしておき、以下を実行します。これにより、テストシナリオに従って負荷試験が実施されます。

リスト 4.72: Taurus でテストシナリオを実行する

```
$ docker run --rm -it -v `pwd`:/bzt-configs blazemeter/taurus:latest test.yaml
```

負荷によって CPU 使用率が上昇し、50% を超えると HPA が発動します。あとは使用率に応じて、自動的に Pod をスケールアウトします。

オートスケーリングが正常に動作しているかどうかは、以下のコマンドで確認できます。このコマンドを実行すると、HPA の現在の状態や、現在の Pod 数、CPU 使用率などが確認できます。

リスト 4.73: HPA リソースを確認する

```
$ kubectl get hpa -w
NAME          REFERENCE          TARGETS          MINPODS   MAXPODS
REPLICAS     AGE
python-web-app-hpa Deployment/python-web-app  cpu: 2%/50%   2          10
3            78s
python-web-app-hpa Deployment/python-web-app  cpu: 18%/50%   2          10
3            3m16s
python-web-app-hpa Deployment/python-web-app  cpu: 67%/50%   2          10
3            3m46s
python-web-app-hpa Deployment/python-web-app  cpu: 71%/50%   2          10
5            4m1s
python-web-app-hpa Deployment/python-web-app  cpu: 48%/50%   2          10
5            4m16s
python-web-app-hpa Deployment/python-web-app  cpu: 2%/50%   2          10
5            13m
python-web-app-hpa Deployment/python-web-app  cpu: 2%/50%   2          10
```

3	13m				
python-web-app-hpa		Deployment/python-web-app	cpu: 2%/50%	2	10
2	14m				

TARGETSのCPU使用率が50%を超えると、REPLICASが2から3や5に変化しています。そして負荷試験が終了し、使用率が収まると2に戻っています。

Podが実際にスケールしている様子を別視点からも見てみましょう。

リスト 4.74: Pod リソースを確認する

NAME	READY	STATUS	RESTARTS	AGE
postgres-bc78df6c4-8472q	1/1	Running	0	55s
python-web-app-6d9cc557cf-7dn9x	1/1	Running	0	55s
python-web-app-6d9cc557cf-fvndq	1/1	Running	0	55s
python-web-app-6d9cc557cf-pghpg	1/1	Running	0	55s
python-web-app-6d9cc557cf-qbddg	0/1	Pending	0	0s
python-web-app-6d9cc557cf-fmsfd	0/1	Pending	0	0s
python-web-app-6d9cc557cf-qbddg	0/1	ContainerCreating	0	0s
python-web-app-6d9cc557cf-fmsfd	0/1	ContainerCreating	0	0s
python-web-app-6d9cc557cf-qbddg	1/1	Running	0	3s
python-web-app-6d9cc557cf-fmsfd	1/1	Running	0	4s
python-web-app-6d9cc557cf-7dn9x	1/1	Terminating	0	13m
python-web-app-6d9cc557cf-pghpg	1/1	Terminating	0	13m
python-web-app-6d9cc557cf-fvndq	1/1	Terminating	0	13m
python-web-app-6d9cc557cf-pghpg	0/1	Terminating	0	14m
python-web-app-6d9cc557cf-7dn9x	0/1	Terminating	0	14m
python-web-app-6d9cc557cf-fvndq	0/1	Terminating	0	14m

最初は3台だったPodが、途中で5台まで増えています。そして、最終的に余剰のPodが削除され2台に減っています。

これらがすべて人の手を介することなく実現できています。すごいですよね！

Kubernetes の Horizontal Pod Autoscaler (HPA)、体感できましたでしょうか。これで Kubernetes クラスター全体のサーバーリソースの範囲内で、柔軟にスケールアウトができるようになります。持てる資材を最大限活用できるよう、HPA を使いこなしてください。

負荷試験の結果確認

Taurus は負荷試験後に詳細なレポートを提供します。レポートには、各リクエストの応答時間や成功率、エラーレートなどが記載されています。

リスト 4.75:

Percentile, %	Resp. Time, s
---------------	---------------

```
+-----+-----+
| 0.0 | 0.034 |
| 50.0 | 1.795 |
| 90.0 | 2.694 |
| 95.0 | 2.998 |
| 99.0 | 3.6 |
| 99.9 | 4.094 |
| 100.0 | 4.88 |
+-----+-----+
14:09:38 INFO: Request label stats:
+-----+-----+-----+-----+
| label | status | succ | avg_rt | error |
+-----+-----+-----+-----+
| http://host.docker.internal:5000/ | OK | 100.00% | 1.822 || |
+-----+-----+-----+-----+
```

4.7 まとめ

本章では、Kubernetesのちょっと高度な使い方に踏み込んで解説しました。各種リソースの使い所やローリングアップデート、オートスケーリングなどKubernetesの恩恵も感じられたかと思います。

さて、次章はいよいよ本書の本題となる Elasticsearch を Kubernetes 上に構築しようというお話をします。これまでの章で解説した内容を駆使して、複雑な Elasticsearch クラスターを簡単に構築してしまいましょう。

第5章 Lesson3 Kubernetes上にElasticsearchクラスターを立てよう（ローカル編）

Elasticsearchは強力で、可用性の高い検索エンジンとして多くの企業で採用されています。 Elasticsearchのクラスターといえば、従来はVM上での構築や運用が一般的でした。一方で、 Elasticsearchの高可用性を活かしたクラスターを組むのは難しいとも言われています。

Amazon OpenSearch Serviceというマネージドサービスを使うことで、運用コストを減らせる可能性はあります。特にServerlessを使えば、オートスケーリング機能によってかなり管理が楽になる可能性があります¹。しかし、最小構成であっても\$数百～\$数千/月かかるので、選択が難しいこともあります²。

また、Kubernetesの普及に伴って、Elasticsearchをコンテナ化してKubernetes上で効率的に管理・運用したいというニーズが増えてきました。そこで本章では、いよいよ ElasticsearchクラスターをKubernetes上で動かしてみようと思います。

5.1 Elastic Operatorとは

Elastic Operatorは、Kubernetes上でElasticsearchクラスターを管理・運用するために作られたツールです。KubernetesのOperatorパターンを採用していて、Elasticsearchのデプロイやスケーリング、バックアップなどの管理作業を自動化できます。

Elastic OperatorはHelm Chartと呼ばれる形式で、Elastic社から公開されています³⁴。Elastic License 2.0なので、商用利用も可能です。ちなみに、2019年から厳密にはOSSではなくなってしまったElasticsearchとKibanaですが、2024年8月30日にAGPLライセンスが追加付与され、AGPLライセンス下においては再びOSSとしての提供が再開されましたね⁵。

Helmはパッケージマネージャーの名称です。これまでの章で学んできたように、Kubernetes上にアプリケーションをデプロイする際にはリソースごとにそれぞれのマニフェストファイルを作成してきました。しかし、体感した通りシステムを構築するたびに、1からこれらのファイルを作成するのは骨が折れます。そこで、複数のKubernetesリソースをひとつの「チャート」という形式でまとめて、管理できるようにしたのがHelm Chartです。

Helm Chartには、Elastic OperatorやElasticsearchを簡単にデプロイするためのテンプレートが含まれており、これらを利用してKubernetes上にElasticsearchをすばやく構築できます。

- ・迅速なデプロイ: Elasticsearchの構成があらかじめパッケージングされているため、複雑な設定

1.<https://aws.amazon.com/jp/about-aws/whats-new/2023/01/amazon-opensearch-serverless-available/>

2.<https://dev.classmethod.jp/articles/amazon-opensearch-serverless-price-down/>

3.<https://www.elastic.co/jp/elastic-cloud-kubernetes>

4.<https://github.com/elastic/cloud-on-k8s>

5.<https://www.elastic.co/jp/blog/elasticsearch-is-open-source-again>

を1から記述する必要がありません。

- ・バージョン管理: Helm Chartはバージョン管理ができるため、簡単に以前のバージョンにロールバックできます。
- ・再利用性: Helm Chartはチーム内で簡単に共有でき、同じ設定を複数の環境にデプロイできます。

5.2 ElasticsearchをKubernetesにデプロイしてみよう

それではさっそくElastic Operatorを使っていきます。kindでKubernetesクラスターを立ち上げておいてください。基本的な手順は公式ドキュメントに準拠します⁶⁷。

さて、Elastic OperatorのHelm Chartを使うには、まずHelmリポジトリの登録が必要です。yumやaptなどで、パッケージのリポジトリと同じようなものです。

リスト5.1: Elastic OperatorのHelmリポジトリを追加する

```
helm repo add elastic https://helm.elastic.co  
helm repo update
```

登録ができたら、Elastic OperatorをKubernetes上にデプロイします。

まずはOperatorのPodを立ち上げよう

いきなり Elasticsearch をデプロイするのではなく、Operatorを先にデプロイしておく必要があります。

リスト5.2: Elastic OperatorのPodを立ち上げる

```
$ helm install elastic-operator elastic/eck-operator --version 2.14.0 \  
--namespace elastic-system \  
--create-namespace
```

これによって、elastic-systemという名前空間にElastic Operatorがインストールされます。

バージョンはお好きなものを選んでください。本書では、執筆時点で最新版の2.14.0を指定しています。

正常にインストールできたか、確認してみましょう。今回はnamespaceがdefaultではなく、elastic-systemです。なので、kubectlコマンドにもnamespaceの指定が必要になります。

6.<https://www.elastic.co/guide/en/cloud-on-k8s/current/k8s-deploy-elasticsearch.html>

7.<https://github.com/elastic/cloud-on-k8s/tree/main/deploy>

リスト 5.3: 名前空間を指定して Pod リソースを確認する

```
$ kubectl get pods -n elastic-system
NAME             READY   STATUS    RESTARTS   AGE
elastic-operator-0   1/1     Running   0          38s
```

上のように、`elastic-operator`がRunning状態になっていればOKです。

Operatorが無事インストールできたら、いよいよ Elasticsearch を起動しましょう。Elastic Operator をインストールしたこと、「`Elasticsearch`」という専用リソースが使えるようになります。これを使ってマニフェストを書きます。

リスト 5.4: `elasticsearch.yaml`

```
apiVersion:.elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: quickstart
  namespace: elastic-system
spec:
  version: 8.15.2
  nodeSets:
  - name: default
    count: 3
    config:
      node.store.allow_mmap: false
  podTemplate:
    spec:
      containers:
      - name: elasticsearch
        resources:
          requests:
            memory: 512Mi
            cpu: 500m
          limits:
            memory: 1Gi
            cpu: 1
```

このマニフェストは、`count: 3`とあるように、3ノード構成の Elasticsearch クラスターをデプロイするための設定になっています。`version`で Elasticsearch のバージョンを指定しています。

では、このマニフェストを適用して、Elasticsearch クラスターをデプロイしましょう。

リスト 5.5: Elasticsearch のマニフェストを適用する

```
$ kubectl apply -f elasticsearch.yaml
elasticsearch.elasticsearch.k8s.elastic.co/quickstart created
```

たったこれだけで、Elasticsearch のクラスターが自動的に作成されます。

リスト 5.6: リソースを確認する

```
$ kubectl get pods -n elastic-system
NAME             READY   STATUS    RESTARTS   AGE
elastic-operator-0   1/1     Running   0          23m
quickstart-es-default-0   1/1     Running   0          20m
quickstart-es-default-1   1/1     Running   0          20m
quickstart-es-default-2   1/1     Running   0          20m

$ kubectl get svc -n elastic-system
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
elastic-operator-webhook   ClusterIP   10.96.180.156   <none>        443/TCP
23m
quickstart-es-default      ClusterIP   None           <none>        9200/TCP
20m
quickstart-es-http         ClusterIP   10.96.128.143   <none>        9200/TCP
20m
quickstart-es-internal-http   ClusterIP   10.96.176.238   <none>        9200/TCP
20m
quickstart-es-transport     ClusterIP   None           <none>        9300/TCP
20m

$ kubectl get cm -n elastic-system
NAME       DATA   AGE
elastic-licensing   15     23m
elastic-operator     1      23m
elastic-operator-uuid   1      23m
kube-root-ca.crt     1      23m
quickstart-es-scripts   6      20m
quickstart-es-unicast-hosts   1      20m

$ kubectl get sts -n elastic-system
NAME             READY   AGE
elastic-operator   1/1     23m
quickstart-es-default   3/3     20m

$ kubectl get elasticsearch -n elastic-system
NAME      HEALTH   NODES   VERSION   PHASE   AGE
quickstart   green    3       8.15.2    Ready    20m
```

たったこれだけで、Elasticsearchのクラスターが用意できました。すごくなっていますか！？
Elasticsearchが立ち上がったら、ポートフォワードをして Elasticsearchにアクセスしてみましょう。

リスト 5.7: Elasticsearchのエンドポイントをポートフォワードする

```
$ kubectl port-forward svc/quickstart-es-http 9200:9200 -n elastic-system &
```

アクセスにはパスワードが必要なので、Secretから取得しておきます。

リスト 5.8: パスワードを取得する

```
$ export NAMESPACE=elastic-system
$ export PASSWORD=$(kubectl get secret quickstart-es-elastic-user -n ${NAMESPACE} -o go-template='{{.data.elastic | base64decode}}')
$ echo $PASSWORD
W885MbA3et3849pW7mLi4EnR
```

ポートフォワードができたら、適当なcurlやブラウザーで https://localhost:9200 にアクセスしてみましょう。アクセスにはユーザー名とパスワードが求められるので、さきほど取得したものを使用します。

リスト 5.9: Elasticsearchにアクセスする

```
$ curl -u "elastic:$PASSWORD" -k "https://localhost:9200"
{
  "name" : "quickstart-es-default-0",
  "cluster_name" : "quickstart",
  "cluster_uuid" : "ZSFNkcKnRRaVm-6Z4mrtoA",
  "version" : {
    "number" : "8.15.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "98adf7bf6bb69b66ab95b761c9e5aadb0bb059a3",
    "build_date" : "2024-09-19T10:06:03.564235954Z",
    "build_snapshot" : false,
    "lucene_version" : "9.11.1",
    "minimum_wire_compatibility_version" : "7.17.0",
    "minimum_index_compatibility_version" : "7.0.0"
  },
  "tagline" : "You Know, for Search"
}
$ curl -u "elastic:$PASSWORD" -k "https://localhost:9200/_cluster/health?pretty"
{
```

```
"cluster_name" : "quickstart",
"status" : "green",
"timed_out" : false,
"number_of_nodes" : 3,
"number_of_data_nodes" : 3,
"active_primary_shards" : 0,
"active_shards" : 0,
"relocating_shards" : 0,
"initializing_shards" : 0,
"unassigned_shards" : 0,
"delayed_unassigned_shards" : 0,
"number_of_pending_tasks" : 0,
"number_of_in_flight_fetch" : 0,
"task_max_waiting_in_queue_millis" : 0,
"active_shards_percent_as_number" : 100.0
}
```

図 5.1: Elasticsearch のトップ画面にアクセスできた！

```
1 // 20241013103503
2 // https://localhost:9200/
3
4 {
5   "name": "quickstart-es-default-0",
6   "cluster_name": "quickstart",
7   "cluster_uuid": "ZSFNkcKnRRaVm-6Z4mrtoA",
8   "version": {
9     "number": "8.15.2",
10    "build_flavor": "default",
11    "build_type": "docker",
12    "build_hash": "98ad77bf6bb69b66ab95b761c9e5aadb0bb059a3",
13    "build_date": "2024-09-19T10:06:03.564235954Z",
14    "build_snapshot": false,
15    "lucene_version": "9.11.1",
16    "minimum_wire_compatibility_version": "7.17.0",
17    "minimum_index_compatibility_version": "7.0.0"
18  },
19  "tagline": "You Know, for Search"
20 }
```



ステータスを見ると、オールグリーンになっているはずです。ここまでできればバッチリです！

5.3 インデックスをして検索しよう

Elasticsearch クラスターのセットアップが完了しました。次は実際にデータをインデックスし、検索ができるることを確認しましょう。

適当なインデックスを用意します。

リスト5.10: インデックス用のデータを用意する

```
{ "index": { "_index": "products", "_id": 1 } }
{ "product_id": 1, "name": "Elasticsearch", "category": "Search Engine", "price": 0, "in_stock": true, "release_date": "2010-02-08", "description": "A distributed, RESTful search and analytics engine." }
{ "index": { "_index": "products", "_id": 2 } }
{ "product_id": 2, "name": "Kibana", "category": "Data Visualization", "price": 0, "in_stock": true, "release_date": "2013-09-10", "description": "A data visualization and exploration tool built on top of Elasticsearch." }
{ "index": { "_index": "products", "_id": 3 } }
{ "product_id": 3, "name": "Logstash", "category": "Log Management", "price": 0, "in_stock": true, "release_date": "2013-06-14", "description": "A server-side data processing pipeline that ingests, transforms, and sends data to Elasticsearch." }
{ "index": { "_index": "products", "_id": 4 } }
{ "product_id": 4, "name": "Beats", "category": "Data Shippers", "price": 0, "in_stock": true, "release_date": "2015-11-16", "description": "A platform for lightweight shippers that send data from edge machines to Elasticsearch." }
```

これをさきほど作った Elasticsearch に投入します。

リスト5.11: Elasticsearchにデータを入れる

```
$ curl -u "elastic:$PASSWORD" -X POST -k "https://localhost:9200/_bulk?pretty" -H 'Content-Type: application/json' --data-binary @data.json
```

投入できたら、検索をしてみましょう。

リスト5.12: データを検索をする

```
$ curl -u "elastic:$PASSWORD" -X GET -k "https://localhost:9200/products/_search?q=name:Elasticsearch"
{
  "took" : 376,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    }
  }
}
```

```

},
"max_score" : 1.2809337,
"hits" : [
{
  "_index" : "products",
  "_id" : "1",
  "_score" : 1.2809337,
  "_source" : {
    "product_id" : 1,
    "name" : "Elasticsearch",
    "category" : "Search Engine",
    "price" : 0,
    "in_stock" : true,
    "release_date" : "2010-02-08",
    "description" : "A distributed, RESTful search and analytics engine."
  }
}
]
}
}

```

問題なく検索できていますね。

5.4 Kibanaも使えるようにしよう

続いて、Kibana を使えるようにしましょう。本書の読者にとっては馴染に説法でしょうが、Kibana は Elasticsearch のデータ解析・可視化に長けた Elastic Stack のひとつです⁸。Elasticsearch をデータベースにたとえるなら、Kibana はフロントエンドの UI に相当します。主に時系列データの取り扱いが得意ですが、Elasticsearch 内のデータの検索や表示、時間軸分析、より高度なデータ分析も可能なツールです。Elasticsearch は、Solr で言うところの Solr Admin のような管理画面を持ちません。そのため、Kibana が実質管理画面のような役割を果たします。

Elastic Operator を使った Kibana の構築についてですが、基本的には公式ドキュメントのとおりにマニフェストを書けば、簡単に構築できます⁹¹⁰。

本書では、`kibana.yaml` を作成して以下のように記述します。デフォルトだとそれなりにサーバーリソースを消費するので、`limits` 指定しています。

8.<https://www.elastic.co/jp/kibana>

9.<https://www.elastic.co/guide/en/cloud-on-k8s/current/k8s-deploy-kibana.html>

10.<https://www.elastic.co/guide/en/cloud-on-k8s/current/k8s-kibana-advanced-configuration.html>

リスト 5.13: kibana.yaml

```
apiVersion: kibana.k8s.elastic.co/v1
kind: Kibana
metadata:
  name: my-kibana
  namespace: elastic-system
spec:
  version: 8.15.3
  count: 1
  elasticsearchRef:
    name: quickstart
  podTemplate:
    spec:
      containers:
        - name: kibana
          resources:
            limits:
              memory: 1Gi
              cpu: 1
            requests:
              memory: 250Mi
              cpu: 500m
          env:
            - name: NODE_OPTIONS
              value: "--max-old-space-size=2048"
```

マニフェストが書けたら適用しましょう。

リスト 5.14: Kibana リソースを適用する

```
$ kubectl apply -f kibana.yaml
```

しばらく待つと、Pod が立ち上がってきます。

リスト 5.15: Kibana のリソースを確認する

```
$ kubectl get pods -n elastic-system
NAME                      READY   STATUS    RESTARTS   AGE
elastic-operator-0         1/1     Running   0          28m
my-kibana-kb-5bf5c689d9-4w4sr 1/1     Running   0          27m
quickstart-es-default-0   1/1     Running   0          27m
quickstart-es-default-1   1/1     Running   0          27m
quickstart-es-default-2   1/1     Running   0          27m
```

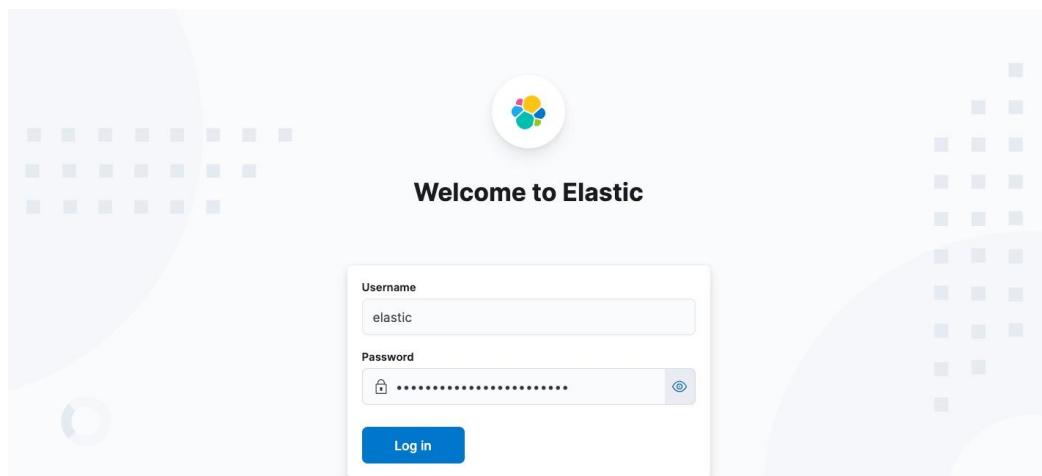
適用できたらポートフォワーディングをして、Kibanaの画面にアクセスできるようにしましょう。

リスト 5.16: Kibana のエンドポイントをポートフォワードする

```
$ kubectl port-forward svc/my-kibana-kb-http 5601:5601 -n elastic-system &
```

あとは適当なブラウザーで <https://localhost:5601> にアクセスすると、Kibana のログイン画面が表示されます。

図 5.2: Kibana のログイン画面



ユーザー名とパスワードを入力してログインしましょう。Elasticsearch にアクセスしたときと同じくユーザー名は `elastic` で、パスワードは以下のコマンドで取得できます。

リスト 5.17: パスワードを取得する

```
$ export NAMESPACE=elastic-system
$ export PASSWORD=$(kubectl get secret quickstart-es-elastic-user -n ${NAMESPACE}
-o go-template='{{.data.elastic | base64decode}}')
$ echo $PASSWORD
W885MbA3et3849pW7mLi4EnR
```

無事ログインできたら、Kibanaのトップ画面が表示されます。

図 5.3: Kibana のトップ画面

The screenshot shows the Kibana home page. At the top, there's a banner with the text "Help us improve the Elastic Stack" and "Usage collection is enabled." Below this, a "Dismiss" button is visible. The main heading "Welcome home" is centered. Below it are four large cards:

- Search**: Create search experiences with a refined set of APIs and tools.
- Observability**: Consolidate your logs, metrics, application traces, and system availability with purpose-built UIs.
- Security**: Prevent, collect, detect, and respond to threats for unified protection across your infrastructure.
- Analytics**: Explore, visualize, and analyze your data using a powerful suite of analytical tools and applications.

Below these cards, there's a section titled "Get started by adding integrations". It includes a "Add integrations" button, a "Try sample data" link, and a "Upload a file" link. To the right of this section is a "Try managed Elastic" card with a "Move to Elastic Cloud" button. Further down, there's a "Management" section with four sub-options: "Manage permissions", "Monitor the stack", "Back up and restore", and "Manage index lifecycles". At the bottom left, there's a "Display a different page on log in" link. At the very bottom, there are navigation links for "Dev Tools" and "Stack Management".

Kibana の使い方ですが、たとえば左のハンバーガーメニューから Dev Tools に移動します。

図5.4: Dev Toolsに移動する

The screenshot shows the Kibana interface with the 'Management' section selected in the sidebar. Under 'Management', 'Dev Tools' is highlighted. On the right, there are three cards: 'Monitor the stack' (Track the real-time health and performance of your deployment), 'Back up and restore' (Save snapshots to a backup repository, and restore to recover index and cluster state.), and 'Manage index lifecycles' (Define lifecycle policies to automatically perform operations as an index ages.). At the bottom left, there is a blue button labeled '+ Add integrations'.

ここで、Consoleタブの左窓にcurlで検索したときと同様に検索クエリを書いてあげます。

リスト5.18: Kibanaの画面からデータを検索する

```
GET /products/_search/q=name:Elasticsearch&pretty
```

その行の実行ボタン（右矢印）を押すと、右側に検索結果が表示されます。つまり、UIから Elasticsearch の検索結果を確かめることができます。

図5.5: KibanaのDev Tools

The screenshot shows the Kibana Dev Tools interface with the 'Console' tab selected. In the left pane, a code editor contains the following Elasticsearch search query:

```
1 # Welcome to the Dev Tools Console!
2 #
3 # You can use Console to explore the Elasticsearch API. See the Elasticsearch API
4 # reference to learn more:
5 # https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html
6 # Here are a few examples to get you started.
7
8
9 # Create an index
10 PUT /my-index
11
12
13 # Add a document to my-index
14 POST /my-index/_doc
15 {
16   "id": "park-rocky-mountain",
17   "title": "Rocky Mountain",
18   "description": "Bisected north to south by the Continental Divide, this
portion of the Rockies has ecosystems varying from over 150 riparian lakes
to montane and subalpine forests to treeless alpine tundra."
19 }
20
21
22 # Perform a search in my-index
23 GET /products/_search?q=name:Elasticsearch&pretty
```

The right pane displays the search results:

```
200 - OK 211 ms
{
  "took": 211,
  "timed_out": false,
  "_shards": [
    {
      "total": 1,
      "successful": 1,
      "skipped": 0,
      "failed": 0
    }
  ],
  "hits": [
    {
      "total": {
        "value": 1
      },
      "relation": "eq"
    },
    {
      "max_score": 1.2039728,
      "hits": [
        {
          "_index": "products",
          "_id": "1",
          "_score": 1.2039728,
          "_source": {
            "product": {
              "name": "Elasticsearch",
              "category": "Search Engine",
              "price": 0,
              "in_stock": true,
              "release_date": "2010-02-08",
              "description": "A distributed, RESTful search and analytics engine."
            }
          }
        }
      ]
    }
  ]
}
```

そのほかにもインデックスの作成やログ分析など、Kibanaには便利な機能がたくさんあります。賢く使って、Elasticsearchを最大限活用しましょう。

5.5 ポートフォワーディングもマニフェストで管理しよう

ここまでで、基本的な Elasticsearch の機能が使えるようになりました。しかし、不便な点がいくつか残っています。

そのひとつが、Elasticsearch に Kubernetes 外からアクセスする際に、ポートフォワーディングが必要な点です。毎度ポートフォワーディングのコマンドを打って、Elasticsearch にアクセスできるようにするのは手間です。そこで、マニフェストを書いて、ポートフォワーディングの設定もファイルで管理しましょう。

Kubernetes で Service 外部からのアクセスを管理する際は、**Ingress** というリソースを使うことが一般的です。Ingress は、Kubernetes 外部からのアクセスを管理する API オブジェクトです¹¹。たとえば、負荷分散や SSL 終端、複数サービスをひとつのドメインで管理する名前ベースの仮想ホスティングなどといった機能を提供します。Ingress のリソースは単体では機能せず、Ingress コントローラーと組わせて使うことで、アクセス制御が実現できます。

手始めに、Ingress コントローラーを Kubernetes 上にインストールします。そのためには、Kubernetes クラスターの設定を変更する必要があります。`kind delete cluster` でクラスターを削除したら、`cluster.yaml` を作成します。

リスト 5.19: cluster.yaml

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
kubeadmConfigPatches:
- |
  kind: InitConfiguration
  nodeRegistration:
    kubeletExtraArgs:
      node-labels: "ingress-ready=true"
      authorization-mode: "AlwaysAllow"
```

`extraPortMappings` で、ホストマシンのポート 80 と 443 を kind クラスター内のコンテナの同じポートにマッピングしています。これによって、Ingress コントローラーが HTTP および HTTPS リ

11. <https://kubernetes.io/ja/docs/concepts/services-networking/ingress/>

クエストを受け取れるようになります。

`node-labels` で、クラスター作成時に `ingress-ready=true` のラベルがノードに付与されます。これにより、Ingress コントローラーがスケジュールされるためのノードラベル要件を満たします。これがないと、Ingress コントローラーの Pod が正常に立ち上がりません。

また、`authorization-mode: "AlwaysAllow"` によって、すべてのリクエストを許可するように設定しています。これは開発環境用の簡易的な設定です。便利な反面、本番環境ではセキュリティ上の観点から慎重に検討する必要があります。

Ingress コントローラーのセーフティロック

Ingress コントローラーは、Kubernetes クラスター外部からのトラフィックを適切なサービスに転送する重要な役割があります。そのため、セキュリティーや性能が十分担保された特定のノードでのみ実行されるようにセーフティがかかっています。

そこで、このノードでは Ingress コントローラーを実行していいと、Kubernetes 側に伝える必要があります。`ingress-ready=true` は、このノードは Ingress コントローラーを実行するに適したノードであることを示すラベルです。このラベルが付いているノードでのみ、Ingress コントローラーは実行されます。

これらは Pod スケジュールやノードセレクターという仕組みで実現されています。気になった方はぜひ調べてみてください。

Kubernetes クラスター用のマニフェストが書けたら、これを使ってクラスターを構築します。

リスト 5.20: マニフェストを使ってクラスターを構築する

```
$ kind create cluster --config cluster.yaml
```

クラスターができたら、Ingress コントローラーをインストールします。今回は Nginx 製のコントローラーを使用します。他のコントローラーを使ってみたい方は、公式ドキュメントを参考にお好みのものを選択してください¹²。

マニフェストは公式が用意してくれているものがあるので、それをそのまま使います。

リスト 5.21: Ingress コントローラーのマニフェストを適用する

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/pr
```

コントローラーが Running になっていたら、正常にインストールできています。

リスト 5.22: Ingress コントローラーの Pod を確認する

```
$ kubectl get pods -n ingress-nginx
NAME                               READY   STATUS    RESTARTS   AGE
ingress-nginx-admission-create-lfl9g   0/1     Completed   0          13m
ingress-nginx-admission-patch-mxqft    0/1     Completed   0          13m
ingress-nginx-controller-8fb8cdb7c-n4fl2  1/1     Running    0          13m
```

12. <https://kubernetes.io/ja/docs/concepts/services-networking/ingress-controllers/>

もしPendingのまま進行しない場合は、以下のコマンドを使ってPodの様子を見てみましょう。

リスト5.23: Podの詳細を確認する

```
$ kubectl describe pod ingress-nginx-controller-8fb8cdb7c-d7rpp -n ingress-nginx
Events:
  Type      Reason     Age           From            Message
  ----      -----     --            --              -----
  Warning   FailedScheduling 4m53s (x4 over 20m) default-scheduler 0/1 nodes
are available: 1 node(s) didn't match Pod's node affinity/selector. preemption:
0/1 nodes are available: 1 Preemption is not helpful for scheduling.
```

このようにFailedSchedulingになっている場合は、上述のKubernetesクラスターの設定(cluster.yaml)が不十分な可能性があります。設定ファイルを見直して、クラスターを再作成してください。

無事、Ingressコントローラーがインストールできたら、Ingressのリソースとしてingress.yamlを作成します。

リスト5.24: ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: elastic-ingress
  namespace: elastic-system
spec:
  ingressClassName: nginx
  rules:
    - host: kibana.local
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: my-kibana-kb-http
                port:
                  number: 5601
    - host: elastic.local
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
```

```
service:  
  name: quickstart-es-http  
  port:  
    number: 9200
```

Kibana と Elasticsearch のそれぞれに対して、`http://kibana.local` と `http://elastic.local` で Kubernetes 内部の `http://my-kibana-kb-http:5601` と `http://quickstart-es-http:9200` にポートフォワーディングを行うよう設定しています。

Kubernetes 側の設定とは別に、ローカルマシンの方にも設定が必要です。`kibana.local` および `elastic.local` でアクセスできるよう、ホストファイルにエントリを追加します。ホストマシン上の `/etc/hosts` ファイルを開いて編集しましょう。

リスト 5.25: `/etc/hosts`

```
$ sudo vim /etc/hosts
```

ファイルを開いたら、末尾に以下を追加します。

リスト 5.26: `/etc/hosts`

```
127.0.0.1 kibana.local  
127.0.0.1 elastic.local
```

以下のようになっていれば、大丈夫です。

リスト 5.27: `/etc/hosts`

```
$ tail -n 2 /etc/hosts  
127.0.0.1      kibana.local  
127.0.0.1      elastic.local
```

ホストファイルの設定が終わったら、最後にもうひと仕上げです。Elasticsearch と Kibana を HTTP で通信できるようにしておきます。HTTPS のままでよいのですが、その場合は TLS 証明書が必要になります。その手間を省くために TLS を無効化します。もし、TLS 有効化のままポートフォワーディングをしたい場合は、以下のブログ記事が参考になります¹³。

具体的には、`elasticsearch.yaml` を以下のように書き換えます。

リスト 5.28: + の部分を追加する

```
apiVersion: elasticsearch.k8s.elastic.co/v1  
kind: Elasticsearch  
metadata:  
  name: quickstart
```

13.<https://qiita.com/ipppeei/items/3a9337259bea726a2936>

```
namespace: elastic-system
spec:
  version: 8.15.2
+ http:
+   tls:
+     selfSignedCertificate:
+       disabled: true
nodeSets:
- name: default
  count: 3
  config:
    node.store.allow mmap: false
podTemplate:
  spec:
    containers:
- name: elasticsearch
      resources:
        requests:
          memory: 512Mi
          cpu: 500m
        limits:
          memory: 1Gi
          cpu: 1
```

kibana.yamlについても同様に、TSL無効化の設定を入れておきます。

リスト5.29: +の部分を追加する

```
apiVersion: kibana.k8s.elastic.co/v1
kind: Kibana
metadata:
  name: my-kibana
  namespace: elastic-system
spec:
  version: 8.15.3
+ http:
+   tls:
+     selfSignedCertificate:
+       disabled: true
  count: 1
  elasticsearchRef:
    name: quickstart
```

```
podTemplate:  
  spec:  
    containers:  
      - name: kibana  
        resources:  
          limits:  
            memory: 1Gi  
            cpu: 1  
          requests:  
            memory: 250Mi  
            cpu: 500m  
        env:  
          - name: NODE_OPTIONS  
            value: "--max-old-space-size=2048"
```

最後に、それぞれのマニフェストをデプロイしましょう。

リスト 5.30: マニフェストを適用する

```
$ kubectl apply -f ingress.yaml -f elasticsearch.yaml -f kibana.yaml
```

Ingressのリソースが正しく作成されていれば、以下のようになります。

リスト 5.31: Ingress リソースを確認する

```
$ kubectl get ingress -n elastic-system  
NAME           CLASS      HOSTS           ADDRESS      PORTS      AGE  
elastic-ingress   nginx     kibana.local,elasticsearch.local  localhost   80          8m44s
```

これで、`http://elastic.local` にアクセスすれば Elasticsearch に繋がります。同様に `http://kibana.local` にアクセスすれば Kibana に繋がります。

より詳しい転送の様子は、以下のコマンドで確認できます。

リスト 5.32: Ingress リソースの詳細を確認する

```
$ kubectl describe ingress elastic-ingress -n elastic-system  
Name:           elastic-ingress  
Labels:         <none>  
Namespace:      elastic-system  
Address:        localhost  
Ingress Class:  nginx  
Default backend: <default>  
Rules:  
  Host          Path  Backends  
  74 | 第5章 Lesson3 Kubernetes 上に Elasticsearch クラスターを立てよう（ローカル編）
```

```

-----
kibana.local
    /   my-kibana-kb-http:5601 (10.244.0.10:5601)
elastic.local
    /   quickstart-es-http:9200 (10.244.0.11:9200)
Annotations: <none>
Events:
  Type   Reason  Age           From            Message
  ----  -----  --  -----
  Normal Sync   8m58s (x2 over 9m29s)  nginx-ingress-controller  Scheduled for
sync

```

これで、毎回ポートフォワーディングを設定する手間を省けるようになりました。

5.6 Podのログを収集しよう

システムの運用をしていると、エラー発生時などにログを確認したくなることは当然あると思います。各Podのログは `kubectl logs` コマンドで確認できます。

リスト 5.33: Podのログを確認する

```
$ kubectl logs quickstart-es-default-0 -n elastic-system | head -n 5
Defaulted container "elasticsearch" out of: elasticsearch,
elasticsearch-internal-init-filesystem (init), elasticsearch-internal-suspend (init)
Skipping security auto configuration because the configuration file
[/usr/share/elasticsearch/config/elasticsearch.yml] is missing or is not a
regular file
Oct 31, 2024 2:21:04 PM sun.util.locale.provider.LocaleProviderAdapter <clinit>
WARNING: COMPAT locale provider will be removed in a future release
{@timestamp:"2024-10-31T14:21:12.119Z", "log.level": "INFO",
"message":"Using native vector library; to disable start with
-Dorg.elasticsearch.nativeaccess.enableVectorLibrary=false", "ecs.version":
"1.2.0","service.name":"ES_ECS","event.dataset":"elasticsearch.server","process.thread.name"
```

Podが残存しているうちは、これで事足りるかもしれません。しかし、Podは障害発生時には壊して作り直せばよいという使い捨ての思想で作られています。そのため、Podが再構築されると、Pod内のデータもろとも初期化されてしまいます。つまり、一番ログを見たいのが障害発生時なのに、障害発生時にはログは初期化されて見えなくなってしまいます。

このような事態に対処すべく、Docker や Docker Compose を使う際は、Volume オプションを使ってホスト側にディスクのマウント・永続化をしているかと思います。Kubernetes でも同様に、データの永続化をすることでログの消失を防げます。

方法は色々あるのですが、今回は Fluentd を使ってログの収集と永続化を行います。Kubernetes

の場合、ひとつのクラスター上でさまざまなPodが稼働しています。Fluentdを使えば、複数のPodから出力されるログを1箇所に集約できるため、ログ管理が簡単になります。

では、さっそくFluentdのPodを用意しましょう。ここでは、簡単のためにHelm Chartを使って構築します。詳しくは公式ドキュメントを参照ください¹⁴。

まずは、Solr Operatorと同様にリポジトリの登録を行います。

リスト 5.34: Helm リポジトリを追加し、更新する

```
# Helmリポジトリを追加し、更新する
$ helm repo add fluent https://fluent.github.io/helm-charts
$ helm repo update
```

リポジトリの登録ができたら、`fluent-values.yaml`を作成してパラメーターをカスタマイズします。

リスト 5.35: fluent-values.yaml

```
# Fluentdのボリューム設定
volumes:
- name: podlog
  hostPath:
    path: /var/log/pods

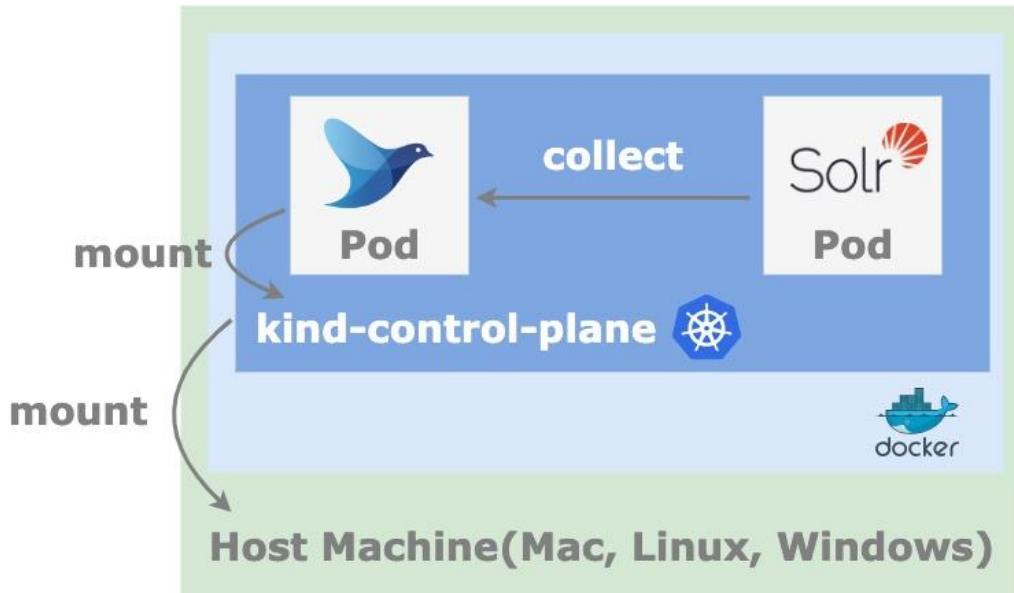
volumeMounts:
- name: podlog
  mountPath: /var/log/pods
```

`volumes`では、FluentdのPodから参照したいホスト側のパスを定義します。反対に`volumeMounts`では、ホスト側から参照したいFluentdのPodのパスを定義します。ここでいう「ホスト」というのは、`kind`によって作られたKubernetesクラスターのコンテナのことです。

`kind`を使ってKubernetesクラスターを作っているのでややこしいことになっていますが、以下のような構造になっています。MacやLinuxなどのホストマシンの上にKubernetesクラスター(`kind-control-plane`)コンテナが稼働していて、その上にPodが立っています。ですので、FluentdのPodから見た「ホスト」はKubernetesクラスターのコンテナなのです。上記の設定では、FluentdのPodのデータをKubernetesクラスターに対して永続化します。

14.<https://github.com/fluent/helm-charts/tree/main/charts/fluentd>

図5.6: コンテナ構造



そして、KubernetesクラスターのデータをMacやLinuxなどの真の意味でのホスト側に永続化するためにはcluster.yamlを書き換えます。

リスト5.36: cluster.yaml

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
  extraMounts:
  - hostPath: /tmp/logs/pods
    containerPath: /var/log/pods
kubeadmConfigPatches:
- |
  kind: InitConfiguration
  nodeRegistration:
    kubeletExtraArgs:
      node-labels: "ingress-ready=true"
```

```
authorization-mode: "AlwaysAllow"
```

`extraMounts` で Kubernetes クラスター上の `containerPath` をホストマシン上の `hostPath` にマウントします。このあたりの `kind` の仕組みは、次のブログ記事にわかりやすくまとまっています¹⁵。

ホスト側にマウントするディレクトリーがない場合は作成しましょう。ディレクトリーには、書き込み権限を付与するのを忘れずに。

リスト 5.37: ログをマウントする用のディレクトリーを作成する

```
$ mkdir -p /tmp/logs/pods  
$ chmod -R 755 /tmp/logs/pods
```

ここまで準備できたら、Fluentd をインストールします。

リスト 5.38: 変数定義ファイルを使って Fluentd の Pod を立ち上げる

```
$ helm install fluent fluent/fluentd -f fluent-values.yaml
```

しばらく待つと、Fluentd の Pod が起動します。

リスト 5.39: Fluentd の Pod を確認する

```
$ kubectl get pods  
NAME          READY   STATUS    RESTARTS   AGE  
fluentd-mqjk9  1/1     Running   0          13m
```

Fluentd の Pod が動き始めると、Pod のログが収集されます。ホストでマウントしたディレクトリーからも参照できるようになります。これで、`kubectl logs` で見たログがホストマシン上に永続化できました。

リスト 5.40: Fluentd で収集したログを Mac 上から確認する

```
$ head -n 4 /tmp/logs/pods/elasticsearch-quickstart-es-default-0_1ead71cc-d841-4bcc-8e03-854b20e950c  
2024-10-31T14:40:24.936432529Z stdout F Skipping security auto configuration  
because the configuration file [/usr/share/elasticsearch/config/elasticsearch.yml]  
is missing or is not a regular file  
2024-10-31T14:40:34.995173872Z stderr F Oct 31, 2024 2:40:34 PM  
sun.util.locale.provider.LocaleProviderAdapter <clinit>  
2024-10-31T14:40:34.995577883Z stderr F WARNING: COMPAT locale provider will be  
removed in a future release  
2024-10-31T14:40:38.215574904Z stdout F {"@timestamp":"2024-10-31T14:40:38.052Z",  
"log.level": "INFO", "message": "Using native vector library; to disable start  
with -Dorg.elasticsearch.nativeaccess.enableVectorLibrary=false", "ecs.version":
```

15.https://qiita.com/Hiroyuki_OSAKI/items/2395e6bbb98856df12f3

```
"1.2.0","service.name":"ES_ECS","event.dataset":"elasticsearch.server","process.thread.name"
```

Kubernetes クラスターを経由してホストマシンにマウントしているため、Kubernetes クラスター内からも同じログを参照できます。

リスト 5.41: Kubernetes クラスターコンテナからもログが参照できる

```
$ docker exec -it kind-control-plane ls -R /var/log/pods | grep quickstart
elastic-system_quickstart-es-default-0_1ead71cc-d841-4bcc-8e03-854b20e95daa
/var/log/pods/elastic-system_quickstart-es-default-0_1ead71cc-d841-4bcc-8e03-854b20e95daa:
/var/log/pods/elastic-system_quickstart-es-default-0_1ead71cc-d841-4bcc-8e03-854b20e95daa/el
/var/log/pods/elastic-system_quickstart-es-default-0_1ead71cc-d841-4bcc-8e03-854b20e95daa/el
/var/log/pods/elastic-system_quickstart-es-default-0_1ead71cc-d841-4bcc-8e03-854b20e95daa/el
```

kind のコマンドでログを出力する

その瞬間の Pod のログを単発で取得するだけであれば、以下のコマンドでホストマシン上に書き出せます。

リスト 5.42: kind のログをホストマシンに書き出す

```
$ kind export logs
```

Exporting logs for cluster "kind" to:

```
/private/var/folders/kz/yrw32ls57hnf4c1p2l0bxm_c0000gn/T/1456733268
```

あくまでスナップショットになりますが、Fluentdなどの設定は一切不要です。

これで、障害発生時の調査やクエリログの分析ができますね。

5.7 Elasticsearch のメトリクスを監視しよう

Elasticsearch を安定的に運用をするには、ログだけでは不十分で、サーバーのパフォーマンスや運用状況を監視したくなります。そのようなメトリクス監視に便利なツールとして、Elasticsearch Exporter なるものがあります。Elasticsearch Exporter は、ヒープの使用率やGCの様子、インデックスサイズなど、Elasticsearch の内部状態を計測・発信してくれます¹⁶。また、Prometheus や Grafana などの監視ツールと組み合わせることで、中長期的にデータの蓄積と可視化、アラート通知が可能になります。

Elastic Operator を使っている場合は、Elasticsearch クラスターと同様に Elasticsearch Exporter についてもマニフェストで定義可能です。基本的には、公式ドキュメントに従えば構築できます¹⁷。

まずは、メトリクスの受取先になる Prometheus や Grafana を用意します。これらも Helm Chart が用意されているので、カスタマイズ不要なら即座に適用、構築ができます。namespace は、Elasticsearch クラスターと合わせる必要はありません。

16.<https://www.elastic.co/guide/en/elasticsearch/reference/current/es-monitoring-exporters.html>

17.<https://github.com/prometheus-community/helm-charts/blob/main/charts/prometheus-elasticsearch-exporter/README.md>

リスト 5.43: 監視基盤を Helm Chart を使って構築する

```
# リポジトリの登録と更新
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
$ helm repo add stable https://charts.helm.sh/stable
$ helm repo update
# Prometheusなどのインストール
$ helm upgrade --install mon prometheus-community/kube-prometheus-stack \
  --namespace monitoring \
  --create-namespace \
  --set kubeStateMetrics.enabled=true \
  --set nodeExporter.enabled=true
```

しばらく待つと、Prometheus や Grafana の Pod が起動します。

待っている間に、Elasticsearch Exporter を用意しましょう。Helm Chart を使えば、1 行で構築が完了します。

ただ、デフォルトの設定だと正常に連携できないので、ちょっとカスタマイズをします。ベースはデフォルトの values.yaml を使います¹⁸。特に、以下の箇所を修正します。

リスト 5.44: elastic-exporter-values.yaml

```
envFromSecret: "quickstart-es-elasticsearch" # Elasticsearch のパスワードが格納された
Secret 名を指定

es:
  uri: http://elastic:$elastic@quickstart-es-http:9200 # Elasticsearch の Service
を指定する

serviceMonitor: # Prometheus との連携設定
  enabled: true # 有効化する
  apiVersion: "monitoring.coreos.com/v1"
  namespace: "monitoring"
  labels:
    release: mon
  interval: 30s
  jobLabel: ""
  scrapeTimeout: 30s
  scheme: http
  relabelings: []
  targetLabels: []
  metricRelabelings: []
```

18. <https://github.com/prometheus-community/helm-charts/blob/main/charts/prometheus-elasticsearch-exporter/values.yaml>

```
sampleLimit: 0
```

これを適用します。

リスト 5.45: 変数定義ファイルを使って Elasticsearch Exporter の Pod を立ち上げる

```
$ helm install elastic-exporter prometheus-community/prometheus-elasticsearch-exporter  
-f elastic-exporter-values.yaml -n elastic-system
```

Elasticsearch Exporter が正常にデプロイされると、以下のコマンドで Pod が起動していることを確認できます。

リスト 5.46: Elasticsearch Exporter の Pod を確認する

NAME	READY	STATUS
RESTARTS AGE		
elastic-exporter-prometheus-elasticsearch-exporter-6d9bfb69ccml 0 16s	1/1	Running
elastic-operator-0 0 6m21s	1/1	Running
my-kibana-kb-7646d595b4-nlgtw 0 6m3s	1/1	Running
quickstart-es-default-0 0 6m4s	1/1	Running

Elasticsearch Exporter の出力結果を直接確認したい場合は、以下のようにポートフォワードを設定してください。

リスト 5.47: Elasticsearch Exporter のエンドポイントをポートフォワードする

```
$ kubectl port-forward svc/elastic-exporter-prometheus-elasticsearch-exporter  
9108:9108 -n elastic-system
```

<http://localhost:9108/metrics> にアクセスすると、以下のようなメトリクス情報が見られます。

リスト 5.48: Elasticsearch Exporter で収集したメトリクス情報

```
# HELP elasticsearch_cluster_health_status Whether all primary and replica shards  
are allocated.  
# TYPE elasticsearch_cluster_health_status gauge  
elasticsearch_cluster_health_status{cluster="quickstart",color="green"} 1  
elasticsearch_cluster_health_status{cluster="quickstart",color="red"} 0  
elasticsearch_cluster_health_status{cluster="quickstart",color="yellow"} 0  
...
```

最後に、Prometheus や Grafana などの監視ツールを使って、Elasticsearch のメトリクスを可視化してみましょう。Prometheus で収集・蓄積したデータは、Grafana を使ってダッシュボード化できます。Helm Chart を利用している場合は、デフォルトでいくつかのダッシュボードが用意されています。その他にも、Grafana の公式が用意しているテンプレートや 3rd Party 製のテンプレートをインポートしたり、独自のカスタムダッシュボードを作成することもできます。

そういうしている間に、Prometheus や Grafana の Pod が立ち上がっていると思うので、確認しましょう。

リスト 5.49: 監視基盤の Pod を確認する

\$ kubectl get pods -n monitoring		READY	STATUS
NAME	RESTARTS AGE		
alertmanager-mon-kube-prometheus-stack-alertmanager-0	2/2 12m	Running	0
mon-grafana-5645b6d587-swnrb	3/3 12m	Running	0
mon-kube-prometheus-stack-operator-6d9b9f6f66-x247f	1/1 12m	Running	0
mon-kube-state-metrics-586b684c9b-zbwvv	1/1 12m	Running	0
mon-prometheus-node-exporter-ptdf9	1/1 12m	Running	0
prometheus-mon-kube-prometheus-stack-prometheus-0	2/2 12m	Running	0

起動していたら、ブラウザからアクセスできるようにポートフォワードをしましょう。 Elasticsearch と同様にアドホックではなく、Ingress を使って設定してもよいです。

リスト 5.50: Grafana のエンドポイントをフォーとフォワードする

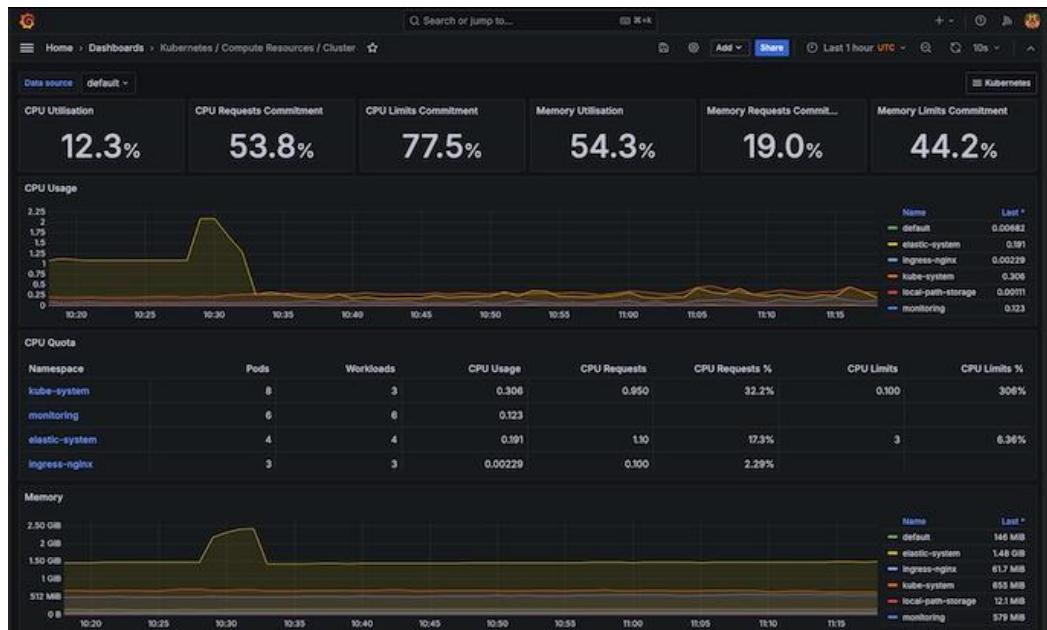
```
$ kubectl port-forward svc/mon-grafana 3000:80 -n monitoring &
```

ポートフォワードができたら、お好みのブラウザーで `http://localhost:3000` にアクセスしましょう。Grafana のログイン画面にアクセスできます。デフォルトのログイン情報は以下の通りです。ログイン後、必要に応じて変更してください。

- ・ユーザー名:admin
- ・パスワード:prom-operator

デフォルトで用意されているダッシュボードを見ると、Kubernetes クラスターのリソース状況が一目でわかります。便利ですね。

図 5.7: Cluster ダッシュボードのようす



Elasticsearch のメトリクスについても同様に、ダッシュボードを使って可視化してみましょう。自分で作ってもよいですが、Elasticsearch 公式が推奨しているダッシュボードテンプレートがあるので紹介しておきます。

Elasticsearch Exporter Quickstart and Dashboard というダッシュボードで、Elasticsearch クラスターの概要や Jetty、Java VM、OS メトリクスなどが詳細に分析できるテンプレートです¹⁹。

導入も簡単で、Grafana の Web UI で、Home > Dashboards > New > import の順に選択します。

19. <https://grafana.com/grafana/dashboards/14191-elasticsearch-overview/>

図5.8: ダッシュボードの新規作成

The screenshot shows the Grafana interface for managing dashboards. At the top right, there is a blue button labeled "New" with a dropdown arrow. A context menu is open, showing options: "New dashboard", "New folder", and "Import". Below the menu, there is a search bar and a filter section with "Filter by tag" and "Starred" checkboxes. A "Tags" section shows three tags: "alertmanager-mixin" (purple), "coredns" (blue), and "dns" (green). The main area lists three dashboards:

Name	Tags
Alertmanager / Overview	alertmanager-mixin
CoreDNS	coredns dns

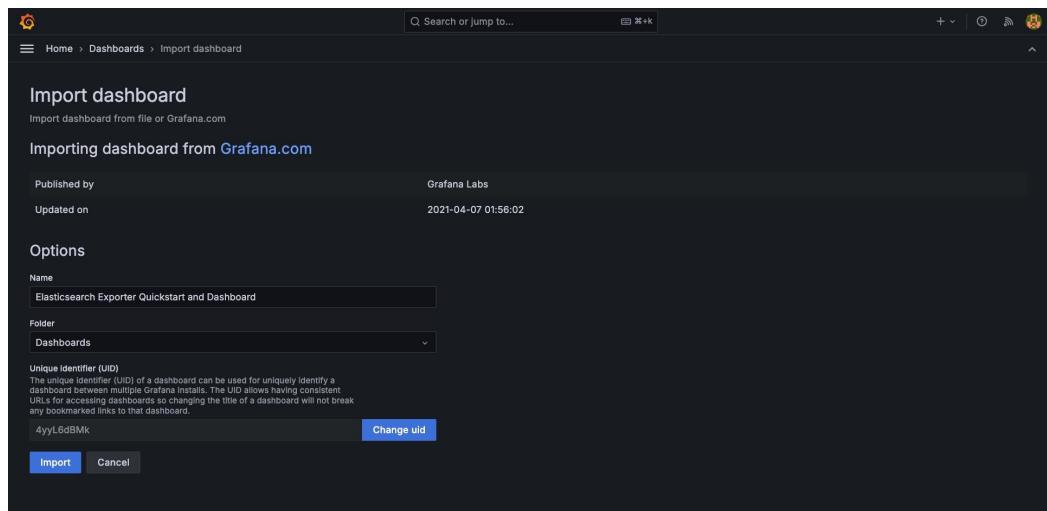
すると、JSONファイルをアップロードするか公開されているテンプレートの番号を入力するかで、テンプレートの読み込みができます。ダッシュボード番号である14191を入力してimportします。

図5.9: ダッシュボードのインポート

The screenshot shows the "Import dashboard" page. At the top, there is a search bar and a "Load" button. Below it, there is a section for uploading a JSON file, with a placeholder text: "Upload dashboard JSON file" and "Drag and drop here or click to browse Accepted file types: .json, .txt". Further down, there is a text input field containing the number "14191" and a "Load" button. At the bottom, there are "Load" and "Cancel" buttons.

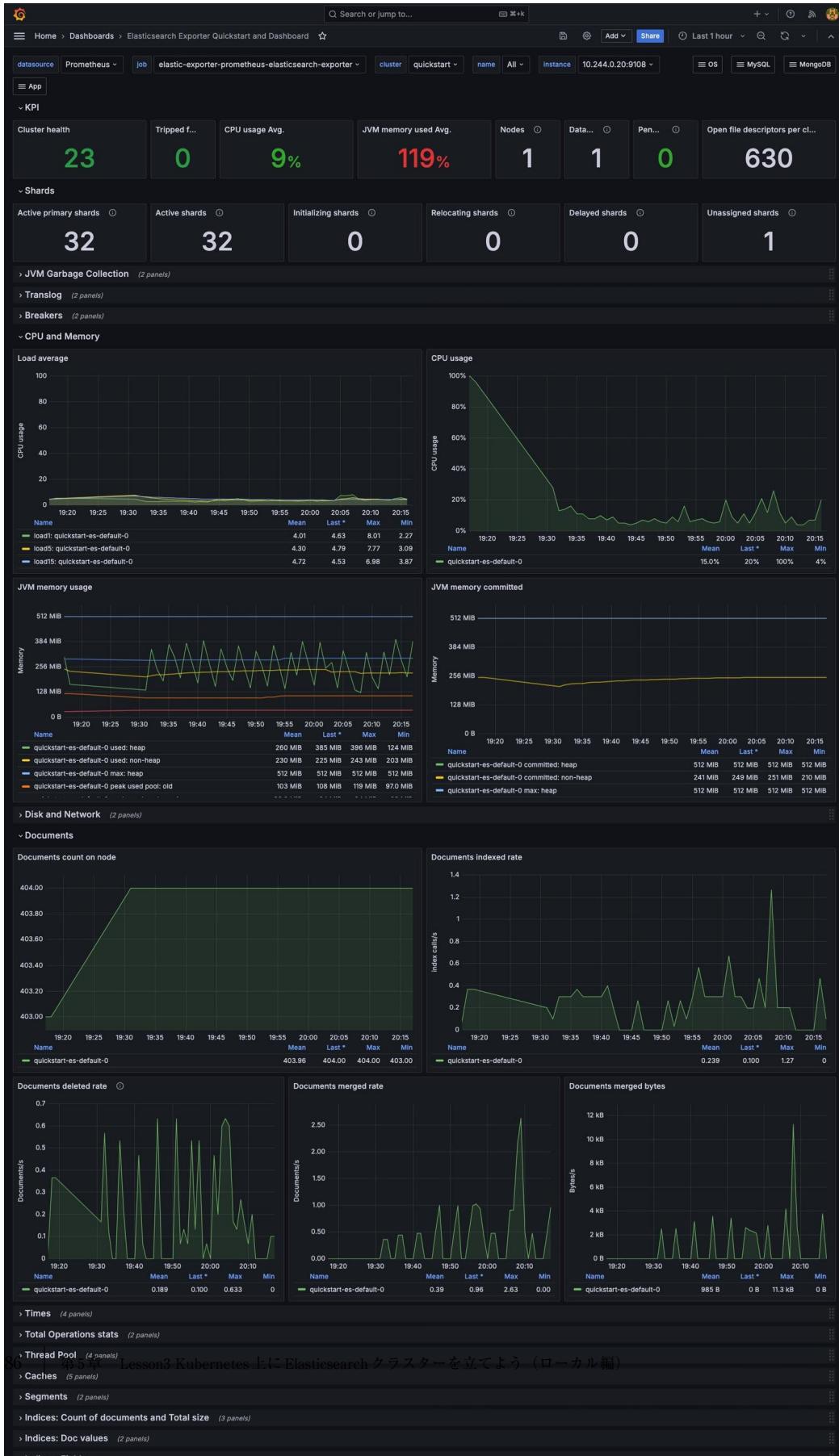
無事Loadができたら、importボタンを押せば完了です。

図 5.10: Options の設定



正常にimportができれば、ダッシュボードに遷移して、以下のような形でメトリクスが可視化されます。

図5.11: Elasticsearchのダッシュボードのようす



Lesson3 Kubernetes上にElasticsearchクラスターを立てる（ローカル編）

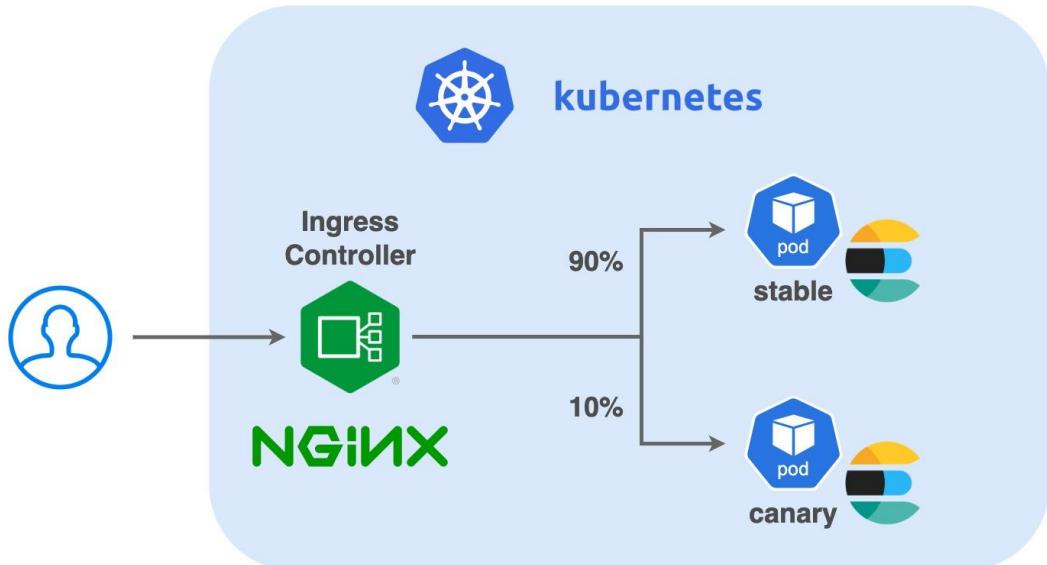
きっとパフォーマンスチューニングや障害発生時の原因調査に役立つことでしょう。

5.8 カナリアリリースもお手のもの！？

Kubernetesを使えば、カナリアリリースのようなことも簡単に実現できてしまいます。カナリアリリースとは、新機能や機能改善を行った際に一度に全ユーザーに展開するのではなく、一部のユーザーにだけお試しで出して、問題ないことを確認してから全展開をするという段階的なリリース手法です。炭鉱開発でカナリアに危険察知をさせていたことにちなんで、この名がつけられています。

Kubernetesにおいてカナリアリリースの実現方法はいくつかありますが、本書ではなるべく Kubernetesネイティブの機能を使って実現する手順を紹介します。システムのイメージ図は、以下になります。

図5.12: Ingressを使ったカナリアリリースのイメージ



図にあるとおり、すでにリリース済みのstable版Elasticsearchクラスターと、並行して動作するcanary版のElasticsearchクラスターを用意します。その後、Nginx Ingress Controllerのカナリア機能を使って、特定割合のトラフィックをcanary版に振り分けます。これにより、canary版のElasticsearchクラスターの動作を確認しながら、問題がなければstable版からcanary版に切り替えることができます。

まずはこれまで説明したとおり、kind上にElasticsearchクラスターとIngressコントローラーがデプロイされていることを前提とします。その上で、canary版のElasticsearchクラスターを用意します。基本的には、stable版と同じマニフェスト定義になります。`metadata.name`と`nodeSets.name`を変更します。

リスト 5.51: elasticsearch-canary.yaml

```
apiVersion:.elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: quickstart-canary
  namespace: elastic-system
spec:
  version: 8.15.2
  http:
    tls:
      selfSignedCertificate:
        disabled: true
  nodeSets:
    - name: default-canary
      count: 1
      config:
        node.store.allow mmap: false
  podTemplate:
    spec:
      containers:
        - name: elasticsearch
          resources:
            requests:
              memory: 512Mi
              cpu: 500m
            limits:
              memory: 1Gi
              cpu: 1
```

作成できたら適用しましょう。

リスト 5.52: カナリア版 Elasticsearch クラスターを適用する

```
$ kubectl apply -f elasticsearch-canary.yaml
$ kubectl get pods -n elastic-system
NAME                                     READY   STATUS
RESTARTS      AGE
elastic-exporter-prometheus-elasticsearch-exporter-769b6c5bhddb   1/1     Running
0           1h
elastic-operator-0                         1/1     Running
0           1h
my-kibana-kb-7f554f585c-mwlfs           1/1     Running
```

0	1h		
quickstart-canary-es-default-canary-0		1/1	Running
0	1h		
quickstart-es-default-0		1/1	Running
0	1h		

これで、quickstart-canary という別名の Elasticsearch クラスターが立ち上がります。

stable 版と canary 版のふたつの Elasticsearch クラスターが立ち上がったら、Ingress を使ってリクエストの振り分け設定を行います。今回は、stable 版と canary 版の両 Service を同じひとつのホスト名で公開し、うち 10% を canary 版へ振り分けます。既存の `ingress.yaml` とは別に、以下のような Ingress リソースを作成します。

リスト 5.53: ingress-canary.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: elastic-ingress-canary
  namespace: elastic-system
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10"
spec:
  ingressClassName: nginx
  rules:
    - host: elastic.local
      http:
        paths:
          - path: "/"
            pathType: Prefix
            backend:
              service:
                name: quickstart-canary-es-http
                port:
                  number: 9200
```

`annotations` 部分で canary の設定を行っています。`nginx.ingress.kubernetes.io/canary` で canary 版の Ingress を有効化し、`nginx.ingress.kubernetes.io/canary-weight` で振り分ける割合を指定します。

マニフェストが書けたら、これも適用しましょう。

リスト 5.54: canary 版用の Ingress 定義を適用する

```
$ kubectl apply -f ingress-canary.yaml
$ kubectl get ingress -n elastic-system
NAME          CLASS   HOSTS           ADDRESS      PORTS
AGE
elastic-ingress  nginx  kibana.local,elastic.local  localhost  80
1h
elastic-ingress-canary  nginx  elastic.local        localhost  80
1h
```

これで 90% が stable 版 (quickstart)、10% が canary 版に (quickstart-canary) にルーティングされるようになりました。

基本的にはこれで設定は完了なのですが、もうひとつだけ設定変更をしておきます。このままだと、stable 版と canary 版で Elasticsearch のパスワードが異なっています。アクセス先は自動的に振り分けられるので、事前にどちらの Pod にアクセスするかわかりません。そのため、適切なパスワードを指定できません。そこで、stable 版と canary 版でパスワードを合わせます。

リスト 5.55: canary 版のパスワードを stable 版のパスワードに再設定する

```
$ kubectl create secret generic quickstart-canary-es-elastic-user
-n "elastic-system" --from-literal=elastic=`kubectl get secret
quickstart-es-elastic-user -n "elastic-system" -o go-template='{{.data.elastic
| base64decode}}'` --dry-run=client -o yaml | kubectl apply -f -
```

以下のような警告文が出ますが、これで stable 版と canary 版でパスワードが同じになりました。

リスト 5.56: 警告文

```
Warning: resource secrets/quickstart-canary-es-elastic-user is missing the
kubernetes.io/last-applied-configuration annotation which is required
by kubectl apply. kubectl apply should only be used on resources created
declaratively by either kubectl create --save-config or kubectl apply. The
missing annotation will be patched automatically.
secret/quickstart-canary-es-elastic-user configured
```

パスワード上書き以外での解決案

上記の一手間は、Elasticsearch 側で自動で用意されたパスワードを使用するために起こるものです。ですので、パスワードを無効化してしまえば、stable 版や canary 版でパスワードの違いを考えなくてよくなります。厳密にいうと、セキュリティ機能をオフにはできないため、匿名ユーザーを作成し「認証なしアクセス」を実現します。

リスト 5.57: elasticsearch.yaml

```
nodeSets:
- name: default
```

```
count: 1
config:
node.store.allow_mmap: false
+ xpack.security.authc:
+ anonymous:
+ username: anonymous
+ roles: superuser
+ authz_exception: false
代わりに、自前で作成した Secret や外部SSOでの認証を通じて Elasticsearch にアクセスできるようにすれば、セキュリティ性は高められます。
```

実際に振り分けができているか確認してみましょう。Healthcheck用のエンドポイントにアクセスすると、Elasticsearchのバージョン情報が取れます。何回かアクセスしてみると、stable版とcanary版の両方にアクセスできていることが確認できます。

リスト 5.58: stable版とcanary版にアクセスが分散できている

```
curl -u "elastic:`kubectl get secret quickstart-es-elastic-user -n
"elastic-system" -o go-template='{{.data.elastic | base64decode}}'`" -k
"http://elastic.local/_cluster/health?pretty"
{
  "cluster_name" : "quickstart",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 0,
  "active_shards" : 0,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "delayed_unassigned_shards" : 0,
  "number_of_pending_tasks" : 87,
  "number_of_in_flight_fetch" : 0,
  "task_max_waiting_in_queue_millis" : 22907,
  "active_shards_percent_as_number" : 100.0
}
curl -u "elastic:`kubectl get secret quickstart-es-elastic-user -n
"elastic-system" -o go-template='{{.data.elastic | base64decode}}'`" -k
"http://elastic.local/_cluster/health?pretty"
{
  "cluster_name" : "quickstart-canary",
  "status" : "green",
  "timed_out" : false,
```

```
"number_of_nodes" : 1,  
"number_of_data_nodes" : 1,  
"active_primary_shards" : 0,  
"active_shards" : 0,  
"relocating_shards" : 0,  
"initializing_shards" : 0,  
"unassigned_shards" : 0,  
"delayed_unassigned_shards" : 0,  
"number_of_pending_tasks" : 0,  
"number_of_in_flight_fetch" : 0,  
"task_max_waiting_in_queue_millis" : 0,  
"active_shards_percent_as_number" : 100.0  
}
```

以上で、追加のツールなしでも、Kubernetesの基本リソースのみでカナリアリリースを実現できます。他にも、ヘッダーやクッキーベースでルーティングをさせることができます²⁰²¹。つまり、A/Bテストのような場面でも有効です。IngressのCanary機能を使いこなせれば、柔軟で安全なリリースが実現できます。

5.9 まとめ

いかがでしたか？Elastic Operatorを使うことで、これまで大変だったElasticsearchクラスターの構築があつという間にできました。しかも構成管理がマニフェストでできてしまうので、IaCも実現できています。ローリングアップデートやオートスケーリングを組み合わせれば、より高可用なシステムになります。これがわざわざKubernetesを使ってまで構築する、大きな魅力です。

次章ではクラウド環境を使って、Kubernetes上にElasticsearchを構築するより実践的な方法を学びます。サーバーレスでElasticsearchクラスターが構築できるすごさをとくと体感あれ！

20.<https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/annotations/#canary>

21.<https://qiita.com/s-shirayama/items/afc6a3afa0eb41b804d3>

第6章 Lesson4 Kubernetes上にElasticsearchクラスターを立てよう（Azure編）

これまでの章では、ローカル環境に構築したKubernetes上で Elasticsearch を動かしていました。本章では Azure Kubernetes Service (AKS) を使って、 Elasticsearch クラスターを構築してみましょう。

AKSは、Microsoft Azure上でKubernetesベースのコンテナ管理を扱えるマネージドサービスです¹。マネージドサービスなので、Kubernetesを動かすための仮想マシンを用意する必要はありません。作成したコンテナを AKS 上に展開するだけで、簡単にシステムが構築できてしまいます。

また、 Azure Container Registry (ACR) にコンテナを登録しておけば、 AKS の制御下で自動でデプロイも可能です。もちろん、他の Azure サービスとの連携相性もよいです。監視やスケーリング、 クラスター作成などの面倒な運用は Azure にお任せして、エンジニアは開発に専念できるようになります。

Amazon Web Services (AWS) や Google Cloud などにも同じような Kubernetes 用のマネージドサービスがあります。 AKS とこれらの他のクラウドサービスとの大きな違いは、開発・検証用の無料のクラスターが構築できる点です²³。 今回はこの Free プランを使って、 Elasticsearch クラスターをクラウド上に構築してみましょう！

業務で使用する際の注意

本章で説明するのは、あくまで個人が実験用途で使用するものです。セキュリティ要件などが十分でない点があります。実際のビジネスやプロダクト目的で使用する場合は、十分に注意が必要です。

6.1 Microsoft Azureにログインしよう

まずは、何はともあれアカウントを作成します。「Azure アカウント作成」で検索するなどして、 Azure のログイン画面に行きましょう⁴。

1.<https://azure.microsoft.com/ja-jp/products/kubernetes-service>

2.<https://learn.microsoft.com/ja-jp/azure/aks/free-standard-pricing-tiers>

3.<https://techblog.asia-quest.jp/202404/introduction-to-containers-part-2-containers-in-the-cloud>

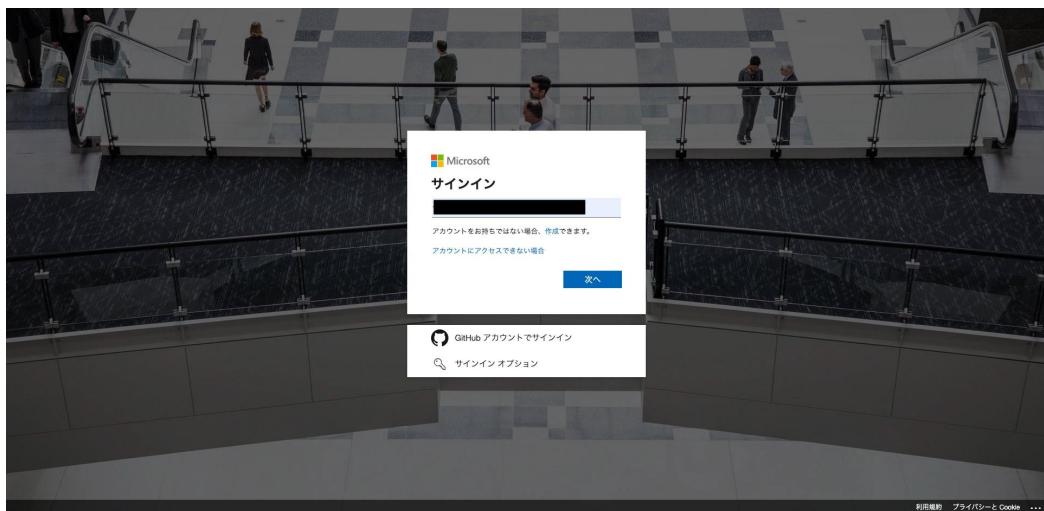
4.<https://azure.microsoft.com/ja-jp/pricing/purchase-options/azure-account?icid=azurefreeaccount>

図6.1: Azureのトップ画面から無料アカウントを作成する



Microsoft アカウントまたは GitHub のアカウントがすでにあれば、それを使って Azure のアカウントが作成できます。どちらもなければ、新規に作成してください。

図6.2: MS アカウントか GitHub アカウントを使ってサインイン



サインインできたら、プロフィール情報を入力します。画面の案内に従って、氏名、メールアドレス、電話番号、住所を入力していきます。

図6.3: プロフィールの入力

プロフィール

国/地域

日本

請求先住所と一致する場所を選択してください。この選択を後で変更することはできません。お客様の国が一覧にない場合、お客様のリージョンではプランを使用できません。[詳細情報](#)

名

ミドル ネーム (省略可能)

姓

電子メール アドレス

電話

例: 090 XXXX XXXX

SMS認証ができたら、クレジットカード情報を入力します。画面の案内にあるように、カード情報を登録したからと言って直ちに高額請求が来るわけではないので、安心してください。

図6.4: カード情報の入力

プロフィール ▼

カードによる本人確認 ▲

クレジットカードまたはデビットカードを指定してください。
このカードで一時的な信用照会が行われますが、**従量課金制の価格に移行しない限り、請求されることはありません。**プリペイドカードは受け入れられません。

次のカードを使用できます:

名義

カード所有者は必須フィールドです。

カード番号

有効期限

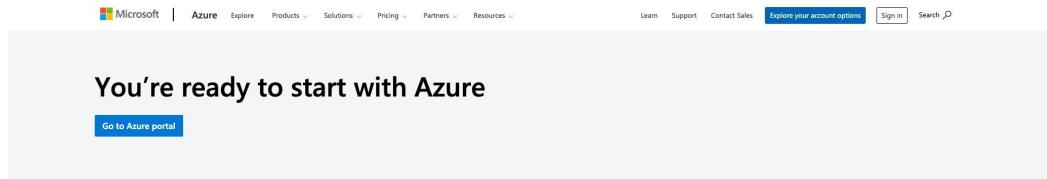
月 ▼ 年 ▼

セキュリティコード

CVVとは何ですか?

一通りの情報の入力が済むと、アカウントの認証と設定が行われます。そのまましばらく待つと、準備完了の画面に遷移します。

図 6.5: サインアップの完了



Join a Q&A session with an Azure technical expert in a live webinar format. Participate in discussions with peers and get your questions answered. Schedule a session anytime you have questions or are looking for inspiration.

Watch the [on-demand demo](#) series to learn how to:

- Navigate & work in the Azure portal.
- Architect your solutions & organize resources.
- Estimate & manage costs.
- Deploy common services, including virtual machines, web apps, and SQL databases.

The demo series is available in English, Chinese (Traditional), German, Korean, Portuguese (Brazilian), Spanish, and Turkish. Captions are available in many other languages.

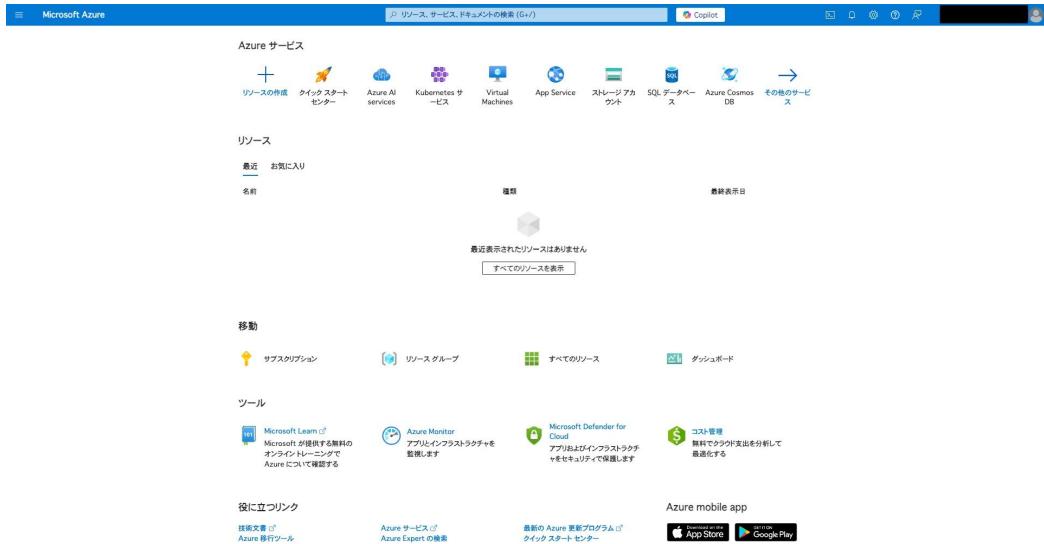
This screenshot shows the main Microsoft Azure landing page. It features a large search bar at the top with the placeholder "Get the Azure mobile app". Below the search bar is a navigation menu with sections like "Explore Azure", "Products and pricing", "Solutions and support", "Partners", "Resources", and "Cloud computing". Each section contains several links to specific Azure services or documentation. At the bottom of the page, there's a footer with links for "English (United States)", "Your Privacy Choices", "Consumer Health Privacy", "Sitemap", "Contact Microsoft", "Privacy", "Terms of use", "Trademarks", "Safety & eco", "Recycling", "About our ads", "© Microsoft 2014", and a "Chat with sales" button.

必要に応じて右上のボタンなどからサインインし直せば、Azure ポータルに移動できます。

図 6.6: ウエルカムページ



図6.7: Azureのポータル画面



これでアカウント作成が完了です。

期間限定で無料クレジットがもらえる

新規作成の場合は、30日間有効な\$200の無料クレジットももらえます。

6.2 リソースプロバイダーを登録しよう

アカウントを作成しただけでは、VMなどのリソースは作成できません。リソースプロバイダーをサブスクリプションに登録することで、初めてリソースを作成できるようになります。

AKSのようなクラスターを作成する際には、Azureリソースマネージャーと呼ばれる機能が裏で動いていて、リソースマネージャーがAPIのようなものを使ってVMの構築などを自動で行います。このリソースをAPIで制御できるようにする機能が、リソースプロバイダーです。たとえば、VMのリソースを提供するMicrosoft.Computeというリソースプロバイダーがあります。

一方、サブスクリプションはAzureの独自概念で、課金・管理・スケールの単位です。Azure上のVMなどのリソースはひとつのサブスクリプションに紐づいていて、サブスクリプション単位で課金やリソース上限が決まります。

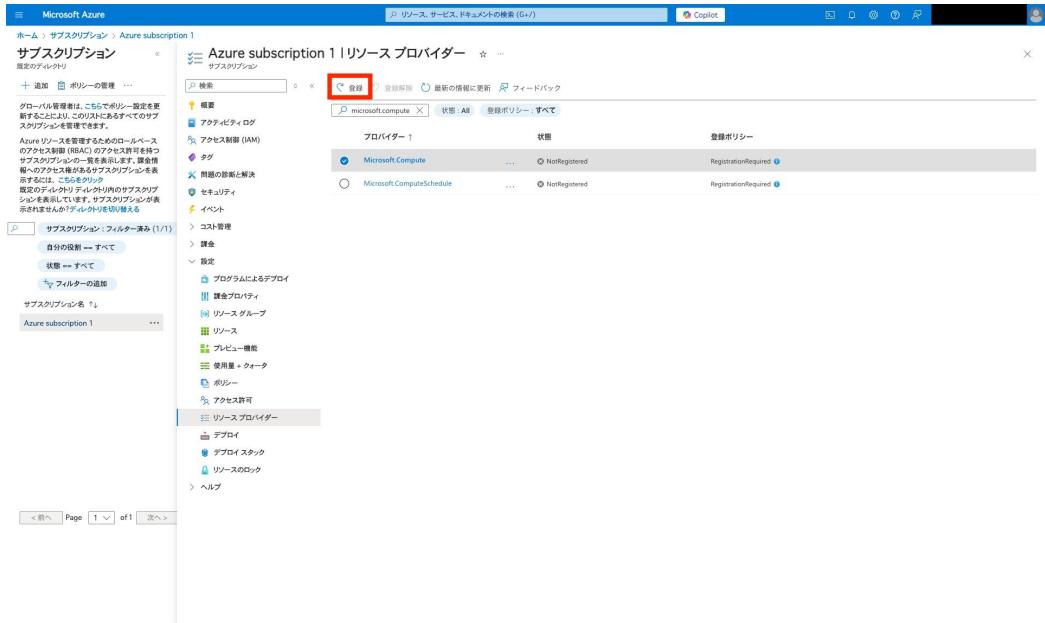
勝手に課金がされることがないようにか、デフォルトだとサブスクリプションにはリソースプロバイダーが紐づいていません。そのため、VMが払い出せず、AKSクラスターを構築しようとしても失敗してしまいます。

リソースプロバイダーとサブスクリプションの紐づけは、Azure Portalから設定できます。Azure Portal上部の検索窓にサブスクリプションと入力するなどして、サブスクリプションの画面に移動

します。

デフォルトで「Azure subscription 1」というサブスクリプションがあるので、これを選択します。その中でリソースプロバイダーを選択します。出てきた画面の検索窓から「microsoft.compute」と入力すると、「Microsoft.Compute」のリソースプロバイダーが絞り込みます。これを選択して「登録」ボタンを押します。

図6.8: リソースプロバイダーをサブスクリプションに登録する



数分待つと、チェックマークがついて状態が「registered」になります。

図6.9: リソースプロバイダーの紐づけが完了

The screenshot shows the Azure portal interface with the search bar set to 'リソース、サービス、ドキュメントの検索 (G+J)'. The main content area displays 'Azure subscription 1 | リソース プロバイダー' (Azure subscription 1 | Resource Provider). On the right, there is a '通知' (Notification) pane with a message about the provider being registered. The provider list shows 'Microsoft.Compute' with a status of 'registered' and 'Microsoft.ComputeSchedule' with a status of 'Not Registered'. The left sidebar shows the navigation menu, and the bottom has standard Azure navigation buttons.

これで、リソースプロバイダーとサブスクリプションの紐づけは完了です。

6.3 AKSを使ってKubernetesクラスターを構築しよう

リソースプロバイダーの登録ができたら、次はAKSクラスターを作成しましょう。作成方法にはCLIやTerraformを使った方法もありますが、今回はAzure PortalからGUIベースで作成します。公式ドキュメントにチュートリアルがあるので、これを参考に作成していきます⁵。

まずは、ポータル上部の検索窓から「AKS」と入力して、AKSのページに移動します。移動したら「作成」ボタンを押して、クラスターの設定画面に移動します。

5.<https://learn.microsoft.com/ja-jp/azure/aks/learn/quick-kubernetes-deploy-portal?tabs=azure-cli>

図 6.10: AKS のサービス画面

その後はチュートリアルに従って、必要事項を入力していきます。

図 6.11: AKS の構築設定画面

リソースグループは、新規作成から適当な名前をつけます。今回は「aks-es-test」としました。クラスターのプリセット構成は、必ずデフォルトの「Dev/Test」を選択してください。このプリセットだけが、無料で使える AKS のタイプになります。

あとは適当な Kubernetes クラスター名を決めて、デフォルトのまま「次へ」を押します。

ノードプールのタブに遷移するので、「ノードサイズ」の「サイズを選択」を押します。デフォルトで設定されているマシンタイプも、チュートリアルに書かれているマシンタイプも選択できないマシンタイプになっているので、使用可能なものに変更します。ここでは、チュートリアルにも書

かれているDシリーズの後継機である「Standard D2plds v5」を選択しました。このあたりはお好みで選択してください。

図6.12: ノードサイズの選択

ノードプール名: agentpool

モード: ブラウザ

OS SKU: Ubuntu Linux

可用性ゾーン: なし

Azure Spot インスタンスを有効にする: なし

ノードサイズ: Standard D2plds v5
2 vcpu 数, 4 GB のメモリ
サイズの選択

スケーリング方法: 自動スケーリング - 推奨
このオプションを選択すると、現在実行中のワークロードに合わせてクラスターが自動的に新しいサイズ変更されるため、お勧めです。

最小ノード数: 2

最大ノード数: 5
AKS クラスターで許可される最大ノード数は、ノードプールあたり 1000 個で、このクラスター内では、すべてのノードプール全体で 5000 ノードです。

オプション設定

ノードあたりの最大ボルト数: 110
15 - 250

ノードごとのパブリック IP を有効にする: なし

ラベル

ラベルはキーバリュペアであり、ノードなどの Kubernetes リソースに固有の情報を分類したり、追加したりするために使用できます。ノードプールのラベルは、そのノードプールの各ノードに適用されます。[詳細情報](#)。

キー: ラベル

値: ラベル

タイム

タイムは、どのノードでどのボルトをスケジュールできるかを決定するため、tolerations との組み合いで使用されるタブルです。ボルトをノードスケジュールするには、そのノードに適用されているすべてのタイムを評価する必要があります。ノードプールのタイムは、そのノードプールの各ノードに適用されます。[詳細情報](#)。

キー: 時間
値: 結果

更新 キャンセル

それ以外はチュートリアルに従って、すべてデフォルトの設定で「確認と作成」ボタンを押します。設定した内容でクラスターが構成できるか検証が走るので、しばらく待ちます。問題がなければ設定内容の確認画面が出るので、そのまま「作成」ボタンを押します。

再びしばらく待つとデプロイが完了し、AKSのクラスターが作成されます。

図 6.13: AKS クラスターのデプロイ完了

「リソースに移動」を選択すると、リソースの詳細ページに飛べます。

6.4 AKS クラスターに接続しよう

AKSのクラスターが作成できたら、さっそく接続してみましょう。接続には、Azure CLIを使用するのが便利です。

先ほど移動したリソースの詳細画面上部の「接続」を押すと、右ペインにクラスターへの接続手順が表示されます。Azure CLIのタブに移動して、画面の案内に従って操作します。

図 6.14: AKS リソースの詳細画面

図6.15: 接続方法の案内

myAKSCluster へ接続する

Cloud Shell Azure CLI 実行コマンド

コマンド ライン ツールを使用してクラスターに接続し、Kubernetes 用コマンド ライン ツール kubectl を使用してクラスターと直接対話します。kubectl は、既定で Azure Cloud Shell 内で使用でき、ローカルにインストールすることもできます。

前提条件

- 1 Azure CLI をインストールする
- 2 kubectl をインストールする

クラスター コンテキストを設定する

- 1 ターミナルを開く
- 2 次のコマンドを実行します

Azure アカウントにログインする

```
az login
```

クラスター サブスクリプションを設定する

```
az account set --subscription [REDACTED]
```

クラスター資格情報をダウンロードする

```
az aks get-credentials --resource-group aks-es-test --name myAKSCluster --overwri...
```

サンプル コマンド

上のコマンドを実行してクラスターに接続すると、任意の kubectl コマンドを実行できます。こちらでは、お試しいただける便利なコマンドの例をいくつかご紹介します。

すべての名前空間内のすべてのデプロイを
一覧表示する

```
kubectl get deployments --all-namespaces=true
```

まずは、ローカル環境に Azure CLI をインストールします。インストール手順は、公式ドキュメントに従ってください⁶。

macOSであれば、Homebrew で簡単にインストールできます。

リスト 6.1: Azure CLI のインストール

```
# azure cliのインストール
$ brew install azure-cli
# バージョン確認
$ az version
{
  "azure-cli": "2.65.0",
  "azure-cli-core": "2.65.0",
  "azure-cli-telemetry": "1.1.0",
  "extensions": {}
}
```

Azure CLI がインストールできたら、CLI を使って Azure にログインします。ブラウザが自動的に開き、Microsoft アカウントでのサインインを求められます。サインインに成功すると、Azure CLI 上にサブスクリプション情報が表示されます。

リスト 6.2: Azure にログインする

```
$ az login
Retrieving tenants and subscriptions for the selection...

[Tenant and subscription selection]

No      Subscription name      Subscription ID          Tenant
-----  -----
[1] *   Azure subscription 1  xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx  既定のディレクトリー

The default is marked with an *; the default tenant is '既定のディレクトリー' and
subscription is 'Azure subscription 1' (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx).

Select a subscription and tenant (Type a number or Enter for no changes):
```

今回はデフォルトのサブスクリプションを使うので、何も入力せずにそのまま Enter を押します。ログインに成功したら、クラスターの資格情報をダウンロードします。これによって、kubectl コマンドが Azure 上の Kubernetes クラスターと紐づきます。

6.<https://learn.microsoft.com/ja-jp/cli/azure/install-azure-cli?view=azure-cli-latest>

リスト 6.3: kubectl を AKS と紐づける

```
$ az aks get-credentials --resource-group aks-es-test --name myAKSCluster  
--overwrite-existing  
Merged "myAKSCluster" as current context in ~/.kube/config
```

もし以下のようなエラーメッセージが出たら、既存のkubectlの設定が悪さをしている可能性があります。

リスト 6.4: ローカル (kind) の設定を使って AKS の資格情報を取得しようとしている

```
No such key 'clusters' in existing config, please confirm whether it is a valid  
config file. May back up this config file, delete it and retry the command.
```

そのため、AKS用にkubectlのコンテキストを切り替える必要があります。

リスト 6.5: kubectl の context を aks 用に切り替える

```
# 現在のコンテキストを確認する  
$ kubectl config get-contexts  
CURRENT      NAME          CLUSTER        AUTHINFO           NAMESPACE  
*           kind-kind     kind-kind     kind-kind  
           myAKSCluster   myAKSCluster  clusterUser_aks-es-test_myAKSCluster  
  
# コンテキストをAKS用に切り替える  
$ kubectl config use-context myAKSCluster  
Switched to context "myAKSCluster".  
  
# 現在のコンテキストを再確認する  
$ kubectl config get-contexts  
CURRENT      NAME          CLUSTER        AUTHINFO           NAMESPACE  
           kind-kind     kind-kind     kind-kind  
*           myAKSCluster   myAKSCluster  clusterUser_aks-es-test_myAKSCluster
```

AKS用のコンテキストに切り替えたら、再度資格情報を取得してみましょう。なお、コンテキストの詳細は、以下のコマンドで確認できます。

リスト 6.6: kubectl のコンテキスト詳細を確認する

```
$ kubectl config view  
apiVersion: v1  
clusters:  
- cluster:  
    certificate-authority-data: DATA+OMITTED  
    server: https://127.0.0.1:49433  
    name: kind-kind  
- cluster:
```

```
certificate-authority-data: DATA+OMITTED
server: https://myakscluster-dns-xxx.hcp.eastus.azmk8s.io:443
name: myAKSCluster
contexts:
- context:
    cluster: kind-kind
    user: kind-kind
    name: kind-kind
- context:
    cluster: myAKSCluster
    user: clusterUser_aks-es-test_myAKSCluster
    name: myAKSCluster
current-context: kind-kind
kind: Config
preferences: {}
users:
- name: clusterUser_aks-es-test_myAKSCluster
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
- name: kind-kind
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
```

それでもうまくいかない場合は、既存のkubectlの設定を退避させてから、再度資格情報の取得をしてみてください。

リスト 6.7: kubectlの設定をバックアップする

```
$ mv ~/.kube/config ~/.kube/config.bak
$ az aks get-credentials --resource-group aks-es-test --name myAKSCluster
--overwrite-existing
Merged "myAKSCluster" as current context in ~/.kube/config
```

うまく資格情報が取得できたら、これ以後はkubectlコマンドの対象クラスターがAKS上のクラスターに切り変わります。試しに、作成されているリソースを見てみましょう。稼働状態のノードがふたつあることが確認できます。

リスト 6.8: AKS上のノードを確認する

```
$ kubectl get nodes
NAME                               STATUS  ROLES   AGE    VERSION
aks-agentpool-20878302-vmss000000  Ready   <none>  70m   v1.29.9
aks-agentpool-20878302-vmss000001  Ready   <none>  70m   v1.29.9
```

6.5 AKSクラスターにElasticsearchをデプロイしよう

ここまで来たら、第5章「Lesson3 Kubernetes上にElasticsearchクラスターを立てよう（ローカル編）」でローカルでやっていたことと同じです。まずは、リポジトリを登録して更新しましょう。

リスト 6.9: Helm リポジトリの登録と更新

```
$ helm repo add elastic https://helm.elastic.co
$ helm repo update
```

次は、Elastic Operatorのインストールでしたね。

リスト 6.10: Elastic Operatorのインストール

```
$ helm install elastic-operator elastic/eck-operator --version 2.14.0 \
--namespace elastic-system \
--create-namespace
```

無事インストールできていれば、OperatorのPodが起動しています。

リスト 6.11: Elastic Operatorの起動確認

```
$ kubectl get pods -n elastic-system
NAME        READY  STATUS    RESTARTS  AGE
elastic-operator-0  1/1    Running   0          24s
```

、elasticsearch.yamlを作成します。

リスト 6.12: elasticsearch.yaml

```
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: quickstart
  namespace: elastic-system
spec:
  version: 8.15.2
  nodeSets:
```

```
- name: default
  count: 3
  config:
    node.store.allow_mmap: false
podTemplate:
  spec:
    containers:
      - name: elasticsearch
        resources:
          requests:
            memory: 512Mi
            cpu: 500m
          limits:
            memory: 1Gi
            cpu: 1
```

これを AKS クラスターに適用して、Elasticsearch のクラスターも構築してみましょう。

リスト 6.13: Elasticsearch のマニフェストを適用する

```
$ kubectl apply -f elasticsearch.yaml
```

しばらく待つと、クラスターが起動します。とっても簡単ですね！

リスト6.14: Elasticsearch Podsの起動確認

```
$ kubectl get pods -n elastic-system
NAME                  READY   STATUS    RESTARTS   AGE
elastic-operator-0    1/1     Running   0          5m46s
quickstart-es-default-0 1/1     Running   0          95s
quickstart-es-default-1 1/1     Running   0          95s
quickstart-es-default-2 1/1     Running   0          95s
```

ポータル上からも、Kubernetesリソースの様子が確認できます。

図6.16: ポータル上からもリソースが確認できる

The screenshot shows the Microsoft Azure portal interface. In the left sidebar, under 'Kubernetes' services, 'myAKSCluster' is selected. Under 'Services & Ingress', the 'Ingress' section is highlighted. A table lists the following ingress resources:

Name	NameSpace	Status	Type	ClusterIP	External IP	Port	Last Seen
kubernetes	default	OK	ClusterIP	10.0.0.1		443/TCP	1時間
kube-dns	kube-system	OK	ClusterIP	10.0.0.10		53/UDPS/53/TCP	1時間
metrics-server	kube-system	OK	ClusterIP	10.0.134.36		443/TCP	1時間
azure-wi-webhook-webhook-service	kube-system	OK	ClusterIP	10.0.220.240		443/TCP	1時間
elastic-operator-webhook	elastic-system	OK	ClusterIP	10.0.148.101		443/TCP	8分
quickstart-es-http	elastic-system	OK	ClusterIP	10.0.826		9200/TCP	4分
quickstart-es-internal-Http	elastic-system	OK	ClusterIP	10.0.101.231		9200/TCP	4分
quickstart-es-transport	elastic-system	OK	ClusterIP	None		9300/TCP	4分
quickstart-es-default	elastic-system	OK	ClusterIP	None		9200/TCP	4分

ローカルと同様にポートフォワードをすれば、手元のマシンからも https://localhost:9200 で接続できるようになります。

リスト6.15: AKS上のElasticsearch Podにポートフォワードする

```
$ kubectl port-forward svc/quickstart-es-http 9200 -n elastic-system
```

パスワードは、以下のコマンドで取得できるんでしたね。

リスト6.16: Elasticsearchのパスワードを取得する

```
$ export NAMESPACE=elastic-system
$ export PASSWORD=$(kubectl get secret quickstart-es-elastic-user -n ${NAMESPACE} -o go-template='{{.data.elastic | base64decode}}')
$ echo $PASSWORD
```

ユーザー名に「elastic」でパスワードに取得したパスワードを入力すれば、AKS上のElasticsearchにアクセスできます。

図 6.17: AKS クラスター上の Elasticsearch に接続できた

```
1 // 20241102151902
2 // https://localhost:9200/
3 {
4   "name": "quickstart-es-default-1",
5   "cluster_name": "quickstart",
6   "cluster_uuid": "xxKSCAd5Rb-kIcMwDY5tSQ",
7   "version": {
8     "number": "8.15.2",
9     "build_flavor": "default",
10    "build_type": "docker",
11    "build_hash": "98addf7fb6bb69b66ab95b761c9e5aadb0bb059a3",
12    "build_date": "2024-09-19T10:06:03.564Z",
13    "build_snapshot": false,
14    "lucene_version": "9.11.1",
15    "minimum_wire_compatibility_version": "7.17.0",
16    "minimum_index_compatibility_version": "7.0.0"
17  },
18 },
19 "tagline": "You Know, for Search"
20 }
```



ただ、毎度ポートフォワードするのは大変なので外部IPを付与して、ポートフォワードなしで接続できるようにしましょう。

`elastic-service.yaml`を作成して、以下のように記述します。

リスト 6.17: elastic-service.yaml

```
$ cat elastic-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-external
  namespace: elastic-system # Elastic Operatorがインストールされたnamespace
spec:
  type: LoadBalancer
  selector:
    elasticsearch.k8s.elastic.co/cluster-name: quickstart
  ports:
    - port: 9200
      targetPort: 9200
      protocol: TCP
```

`cluster-name`は、以下のコマンドで確認できます。

リスト 6.18: cluster-name を確認する

```
$ kubectl get elasticsearch -n elastic-system
NAME      HEALTH   NODES  VERSION  PHASE  AGE
quickstart  green     3      8.15.2   Ready   17m
```

LoadBalancer のリソースが定義できたら、適用してみましょう。

リスト 6.19: LoadBalancer のマニフェストを適用する

```
$ kubectl apply -f elastic-service.yaml
```

正常に適用できていたら、EXTERNAL-IP が設定されたリソースが表示されるようになります。

リスト 6.20: サービスリソースを確認する

```
$ kubectl get svc -n elastic-system
NAME           TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)        AGE
elastic-operator-webhook   ClusterIP   10.0.148.101 <none>
443/TCP       23m
elasticsearch-external   LoadBalancer 10.0.21.242  xx.xxx.xxx.xxx
9200:30842/TCP 15s
quickstart-es-default   ClusterIP   None         <none>
9200/TCP      19m
quickstart-es-http      ClusterIP   10.0.8.26   <none>
9200/TCP      19m
quickstart-es-internal-http ClusterIP 10.0.101.231 <none>
9200/TCP      19m
quickstart-es-transport  ClusterIP   None         <none>
9300/TCP      19m
```

表示された外部 IP を使って <https://xx.xxx.xxx.xxx:9200> にアクセスします。これで、ポートフォワードしたときと同様に、Elasticsearch に接続できるようになります。

図 6.18: 外部 IP を使って Elasticsearch にアクセスできた

```
1 // 20241102153012
2 // https://[REDACTED]:9200/
3
4 +
5   "name": "quicksstart-es-default-0",
6   "cluster_name": "quicksstart",
7   "cluster_uid": "xxK5CAo5Rb-kIcwD9YtsQ",
8 +
9   "version": {
10     "number": "8.15.2",
11     "build_flavor": "default",
12     "build_type": "docker",
13     "build_hash": "98adff7bf6bb69b66ab95b761c9e5a0db0bb059a3",
14     "build_date": "2024-09-19T10:06:03.564235954Z",
15     "build_snapshot": false,
16     "lucene_version": "9.11.1",
17     "minimum_wire_compatibility_version": "7.17.0",
18     "minimum_index_compatibility_version": "7.0.0"
19   },
20   "tagline": "You Know, for Search"
```

誰でもアクセスできる状態にある！？

このままだと IP アドレスさえ知っていれば、誰でもアクセスできてしまう脆弱な状態です。実際にサービスで使用する場合は、Private Link や VPN を設定するなどして、セキュアな状態にしたうえでご利用ください。

6.6 AKS上でもカナリアリリースを実現しよう

ローカルと同様に、AKS 上に Elasticsearch クラスターでも、カナリアリリースを試してみましょう。まずは、canary 用の Elasticsearch クラスターを作成します。`elasticsearch-canary.yaml`を作成して、以下のように記述します。内容は第5章の内容と同じです。

リスト 6.21: `elasticsearch-canary.yaml`

```
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: quickstart-canary
  namespace: elastic-system
spec:
  version: 8.15.2
  http:
    tls:
      selfSignedCertificate:
        disabled: true
  nodeSets:
```

```

- name: default-canary
  count: 1
  config:
    node.store.allow_mmap: false
  podTemplate:
    spec:
      containers:
        - name: elasticsearch
          resources:
            requests:
              memory: 512Mi
              cpu: 500m
            limits:
              memory: 1Gi
              cpu: 1

```

これを AKS クラスターに適用して、canary 版の Elasticsearch のクラスターを構築してみましょう。

リスト 6.22: Elasticsearch のカナリアリリース用マニフェストを適用する

```

$ kubectl apply -f manifest/elasticsearch-canary.yaml
elasticsearch.elasticsearch.k8s.elastic.co/quickstart-canary created
$ kubectl get pods -n elastic-system -w
NAME                           READY   STATUS    RESTARTS   AGE
elastic-operator-0             1/1     Running   0          14m
quickstart-es-default-0       1/1     Running   0          5ms
quickstart-canary-es-default-canary-0 1/1     Running   0          1m

```

次に Ingress Controller を使って、リクエストの振り分けをするんでしたね。Azure の場合は、マネージドで利用できる Nginx Ingress があるので、これを利用します⁷⁸。

公式ドキュメントに従って、アプリケーションルーティングアドオンを有効化します。アドオンを有効化することで、AKS 上に Ingress Controller がデプロイされます。アドオンの有効化は Azure CLI を使って行います。クラスター名やリソースグループは、AKS のリソース詳細画面から確認できます。

リスト 6.23: アプリケーションルーティングアドオンを有効化する

```
$ az aks approuting enable --resource-group aks-es-test --name myAKSCluster
```

無事有効化が成功すると、以下のように Ingress Controller がデプロイされます。

7.<https://learn.microsoft.com/ja-jp/azure/aks/app-routing>

8.<https://techblog.ap-com.co.jp/entry/2024/01/18/120000>

リスト 6.24: Ingress Controller のリソース確認

```
$ kubectl get all -n app-routing-system
NAME                      READY   STATUS    RESTARTS   AGE
pod/nginx-7f6784b4b5-gkkvj 1/1     Running   0          24m
pod/nginx-7f6784b4b5-k4wkw 1/1     Running   0          23m

NAME              TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
service/nginx     LoadBalancer   10.0.156.189   xxx.xxx.xxx.xx
80:31718/TCP,443:32099/TCP   24m
service/nginx-metrics ClusterIP     10.0.30.72    <none>        10254/TCP
24m

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx  2/2       2            2           24m

NAME          DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-7f6784b4b5  2         2         2        24m

NAME          REFERENCE   TARGETS
MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/nginx   Deployment/nginx   cpu: 0%/70%   2
100        2          24m

$ kubectl get ingressclass
NAME          CONTROLLER   PARAMETERS
AGE
nginx          k8s.io/ingress-nginx
<none>        19m
webapprouting.kubernetes.azure.com  webapprouting.kubernetes.azure.com/nginx
<none>        24m
```

これで、Ingress リソースがデプロイ可能になりました。ローカルのときと同様に、Ingress のマニフェストファイルを作成します。

リスト 6.25: ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: elastic-ingress
  namespace: elastic-system
spec:
```

```
ingressClassName: webapprouting.kubernetes.azure.com
rules:
- http:
  paths:
  - path: /
    pathType: Prefix
  backend:
    service:
      name: quickstart-es-http
      port:
        number: 9200
```

リスト 6.26: ingress-canary.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: elastic-ingress-canary
  namespace: elastic-system
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10"
spec:
  ingressClassName: webapprouting.kubernetes.azure.com
  rules:
  - http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: quickstart-canary-es-http
          port:
            number: 9200
```

spec.ingressClassName に webapprouting.kubernetes.azure.com を指定するのがポイントです。

これをデプロイします。

リスト 6.27: Ingress のマニフェストを適用する

```
$ kubectl apply -f ingress.yaml
ingress.networking.k8s.io/elastic-ingress configured
```

```

$ kubectl apply -f ingress-canary.yaml
ingress.networking.k8s.io/elastic-ingress-canary created
$ kubectl get ingress -n elastic-system
NAME          CLASS
PORTS   AGE
elastic-ingress  webapprouting.kubernetes.azure.com  *      xxx.xxx.xxx.xx
80       10m

```

これでADDRESSに表示されたIPアドレスにアクセスすると、Elasticsearchのクラスターに指定の割合で振り分けてアクセスできるようになります。狙い通り振り分けがうまくいっているか、確認しましょう。

リスト6.28: カナリアリリースの確認

```

# パスワードをstable版とcanary版で合わせておく
$ kubectl create secret generic quickstart-canary-es-elastic-user
-n "elastic-system" --from-literal=elastic=`kubectl get secret
quickstart-es-elastic-user -n "elastic-system" -o go-template='{{.data.elastic
| base64decode}}'` --dry-run=client -o yaml | kubectl apply -f -
secret/quickstart-canary-es-elastic-user configured
$ curl -u "elastic:${PASSWORD}" -k "http://{YOUR-EXTERNAL-IP}/_cluster/health?pretty"
{
  "cluster_name" : "quickstart",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 0,
  "active_shards" : 0,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "delayed_unassigned_shards" : 0,
  "number_of_pending_tasks" : 0,
  "number_of_in_flight_fetch" : 0,
  "task_max_waiting_in_queue_millis" : 0,
  "active_shards_percent_as_number" : 100.0
}
$ curl -u "elastic:${PASSWORD}" -k "http://{YOUR-EXTERNAL-IP}/_cluster/health?pretty"
{
  "cluster_name" : "quickstart-canary",
  "status" : "green",

```

```

"timed_out" : false,
"number_of_nodes" : 1,
"number_of_data_nodes" : 1,
"active_primary_shards" : 0,
"active_shards" : 0,
"relocating_shards" : 0,
"initializing_shards" : 0,
"unassigned_shards" : 0,
"delayed_unassigned_shards" : 0,
"number_of_pending_tasks" : 0,
"number_of_in_flight_fetch" : 0,
"task_max_waiting_in_queue_millis" : 0,
"active_shards_percent_as_number" : 100.0
}

```

以上で、AKSでもカナリアリリースができました。お気付きかもしませんが、Ingressリソースをデプロイしたため、ポートフォワードなしでアクセスできるようになっています。また、80ポートでアクセスするようになっています。

ここではIPアドレス直打ちでしたが、Azure DNSと組み合わせることで、独自ドメインで名前解決できるようになります⁹。

6.7 データを永続化しよう

現状だと、ElasticsearchのデータはPod内のストレージに保存されています。そのため、Podが削除されると、データも失われてしまいます。

外部のDBにデータを保存していれば、再インデックスによってデータの復元が可能でしょう。しかし、Elasticsearchをプライマリストレージとして使っている場合は致命的です。特にログデータなどは、Elasticsearchをプライマリストレージとして使うことが多いかと思います。そこで、Azureのストレージサービスを使って、Elasticsearchのデータを永続化してみましょう。

永続化先として、以下の3つを比較しました。パフォーマンス面や費用面、Elasticsearchとの相性(事例)を考慮すると、Azure Diskが最も適しているようです。構成によりますが、Azure Diskは基本的にはサイズ課金のみで、読み書きには課金されません。なので、頻繁に読み書きがあるケースでは向いているようです。

- ・ Azure Disk (ディスクストレージ)
- ・ Azure Files (共有ファイルストレージ)
- ・ Azure Blob Storage (オブジェクトストレージ)

Elasticsearch公式的にも、永続化先にはローカルSSDを推奨しているようです¹⁰。ただし、Azure

9.<https://learn.microsoft.com/ja-jp/azure/dns/dns-overview>

10.<https://www.elastic.co/docs/deploy-manage/deploy/cloud-on-k8s/storage-recommendations>

Blob Storageに関してはブレーキングが標準サポートされているので、スナップショットのバックアップ先としては向いているかもしれません¹¹。

図 6.19: Azure のストレージサービスの比較

	Azure Files	Azure Blob Storage
費用 (GB単価+アクセス課金)	<ul style="list-style-type: none"> - 主にサイズ課金。アクセス課金なし (スナップショット等除く)。 - 例: Standard HDD 約0.01 USD/GB・月 (地域差あり)。 	<ul style="list-style-type: none"> - サイズ課金 + IOPS/スルーパート課金 (Premium) または PAYG (Standard)、アクセス課金はストレージアカウントのトラフィック課金。 - 例: Standard SSD/HDD 約0.02 USD/GB・月 操作数課金あり。
パフォーマンス (IOPS、スループット、レイテンシ)	<ul style="list-style-type: none"> - 超高速 Ultra SSD (最大400,000 IOPS、10,000 MB/s 以上)、Premium SSD (数万IOPS、数百MB/s)、Standard SSD/HDD (千数千IOPS、十数MB/s)など選択可能。 - レイテンシでブロックストレージ接続。VM に直接タッチされるため安定して速度を出せる。 	<ul style="list-style-type: none"> - 共有型ファイルストレージで、レイテンシはネットワーク越しの SMB/NFSアクセスとなり、Disk 性能や大きさ。 - Premium ファイル (SSD) ではファイル共有単位で最大約 10 GB/s (Standard HDDでは約120 MB/s) のスループットが見込める。 - 1ファイルあたりは Premium で最大1,024 MB/s、Standard で60 MB/s。
可用性/耐久性/SLA	<ul style="list-style-type: none"> - ゾーン冗長化 (ZRS) が可能で、ゾーン内3重コピー (LRS) で 99.999999999% 程度のデータ耐久性。 - 初期性 (GLR) で 99.99%、ゾーン冗長で 99.99%。 - マネージドドライブの管理下で自動的に複数コピー実行。自動RA-GRS でリージョン横断冗長化も可能。 	<ul style="list-style-type: none"> - RAID5/RA-GRS/リージョン間複制から選択ができる。 - RA-GRS で 99.999999999% (12.9%) 級の耐久性。 - ファイル共有自身は「ファイル共有スナップショット」(最大200世代) でリバーバー可能。可能性はアカウント冗長設定に依存する。 - SLA 例: Premium ファイル (ゾーン冗長) で 99.9% 保証。
バックアップ (スナップショット) のしやすさ	<ul style="list-style-type: none"> - Azure Disk のスナップショット機能でブロック単位に即時取得可能。 - Kubernetes の CSI VolumeSnapshot と組み合わせれば自動化できること。 - スナップショットを Azure Blob (スナップショットレジストリ) に転送すれば、Elasticsearch のスナップショットリストアを実装できる。 	<ul style="list-style-type: none"> - Azure Files のタイプ別スナップショット機能でファイル全体を保存する場合にのみ可能。 - CKS/RGS/RGRS により複数コピーができる。 - RA-GRS で 99.999999999% (12.9%) 級の耐久性。 - ファイル共有自身は「ファイル共有スナップショット」(最大200世代) でリバーバー可能。可能性はアカウント冗長設定に依存する。 - SLA 例: RAID5 で 99.9% 保証。 - Blob Storage だとオブジェクトのページングやスナップショットが選択可能。 - Elasticsearch のスナップショットリポジトリとしては「Azure Blob リポジトリ」が標準サポートされている。 - しかし、Blob 自体をデータディレクトリとして使う運用は一般的ではない。 - Blob Storage はデータディレクトリとしてファイルシステムに保存するため、Blob (オブジェクトストア) を直連データボリュームとして使う事例はない。 - ECK 例: https://github.com/elastic/csi-driver-aws/blob/main/docs/using-with-aks.md#using-with-aks
Elasticsearch/ECK との相性・サポート	<ul style="list-style-type: none"> - Kubernetes での一部内外ブロックボリューム (CSI) マネージドディスクとしての運用が可能。 - ECK が公式トクマントで特定の要件がない限りマネージドディスク利用が推奨されている。 - シングルAZ構成ではノードに付き 1 ディスクをタッチする形で Pod を配置し、シャドードレプリケーションで耐障害性を確保する。 	<ul style="list-style-type: none"> - Elasticsearch はデータディレクトリとしてファイルシステムに保存するため、Blob (オブジェクトストア) を直連データボリュームとして使う事例はない。 - ECK 例: https://github.com/elastic/csi-driver-aws/blob/main/docs/using-with-aks.md#using-with-aks - ただし、Elasticsearch のスナップショットバックアップ先 (リポジトリ) としては Azure Blob を公式にサポートしている。

Azure Storage の種類や選び方については、以下のブログ記事が参考になりました¹²。

それでは早速、Azure Disk を使って、Elasticsearch のインデックスデータを永続化させてみましょう。まずは、AKS で各種ストレージと連携するための、Container Storage Interface (CSI) ドライバーを有効化します¹³。

リスト 6.29: Azure Disk の CSI ドライバーを有効化する

```
az aks update --name myAKScluster --resource-group aks-es-test
--enable-disk-driver
```

これで、AKS と Azure Disk が連携できるようになりました。同様に、Azure Files や Blob Storage ドライバーやスナップショットコントローラーを有効化することもできます。

ドライバーを有効化できたら、StorageClass のリソースを作成します。

リスト 6.30: storageclass-ssd.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: managed-standard-ssd
provisioner: disk.csi.azure.com
parameters:
  skuname: StandardSSD_LRS
  kind: Managed          # オプション: マネージドディスクの種類
  cachingMode: None       # オプション: キャッシュ無効化
reclaimPolicy: Delete
```

11. <https://www.elastic.co/docs/reference/search-connectors/es-connectors-azure-blob>

12. https://www.climb.co.jp/blog_vmware/azure-7810

13. <https://learn.microsoft.com/ja-jp/azure/aks/csi-storage-drivers>

```
allowVolumeExpansion: true  
volumeBindingMode: WaitForFirstConsumer
```

これに Elasticsearch を繋げます。わかりやすいように一度、既存の Elasticsearch の Pod は削除しておきましょう。

リスト 6.31: Elasticsearch の Pod を削除する

```
$ kubectl delete -f elasticsearch.yaml  
$ kubectl delete -f elasticsearch-canary.yaml
```

その後、以下のようにマニフェストを書き直してデプロイします。

リスト 6.32: elasticsearch.yaml

```
apiVersion: elasticsearch.k8s.elastic.co/v1  
kind: Elasticsearch  
metadata:  
  name: quickstart  
  namespace: elastic-system  
spec:  
  version: 8.15.2  
  volumeClaimDeletePolicy: DeleteOnScaledownOnly  
  http:  
    tls:  
      selfSignedCertificate:  
        disabled: true  
  nodeSets:  
  - name: default  
    count: 1  
    config:  
      node.store.allow_mmap: false  
  podTemplate:  
    spec:  
      containers:  
      - name: elasticsearch  
        resources:  
          requests:  
            memory: 512Mi  
            cpu: 500m  
          limits:  
            memory: 1Gi  
            cpu: 1
```

```

volumeClaimTemplates:
- metadata:
    name: elasticsearch-data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Gi
  storageClassName: managed-standard-ssd

```

`volumeClaimDeletePolicy` の部分で、先ほど作成した StorageClass と接続します。

また、`volumeClaimDeletePolicy` で永続化ボリュームポリシーが勝手に削除されないようにしています。

Elasticsearch Operator で Elasticsearch クラスターを作成した場合、`volumeClaimDeletePolicy` がデフォルトで、`DeleteOnScaledownAndClusterDeletion` になっています。つまり、Elasticsearch クラスター削除時に PVC が自動削除されてしまいます。これだと、永続化したディスクに Pod が接続できません。そこで、`DeleteOnScaledownOnly` などのようにして、クラスターが削除時されても PVC を保持するようにしています。

マニフェストが変更できたら、Elasticsearch をデプロイしましょう。永続化ボリュームが作成され、Azure Disk に繋がっていることが確認できます。

リスト 6.33: Elasticsearch のマニフェストを適用する

```

$ kubectl apply -f manifest/elasticsearch.yaml
elasticsearch.elasticsearch.k8s.elastic.co/quickstart created
$ kubectl get pv -n elastic-system
NAME                                     CAPACITY   ACCESS MODES  RECLAIM
POLICY     STATUS     CLAIM
VOLUMEATTRIBUTESCLASS     REASON     AGE
pvc-9ea2e9bd-8687-46d9-ac43-6fd55d9e506a   10Gi       RWO          Delete
Bound      elastic-system/elasticsearch-data-quickstart-es-default-0
managed-standard-ssd   <unset>           166s
$ kubectl get pvc -n elastic-system
NAME                                     STATUS     VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS          VOLUMEATTRIBUTESCLASS     AGE
elasticsearch-data-quickstart-es-default-0  Bound      pvc-9ea2e9bd-8687-46d9-ac43-6fd55d9e506a
10Gi       RWO          managed-standard-ssd  <unset>           116s

```

ちなみに Azure Disk と繋げないデフォルトの場合、以下のような永続化ボリュームが作成されます。

リスト 6.34: Elasticsearch のマニフェストを適用する

```
$ kubectl get pv -n elastic-system
NAME                                     CAPACITY   ACCESS MODES  RECLAIM
POLICY      STATUS     CLAIM                                         STORAGECLASS
VOLUMEATTRIBUTESCLASS  REASON    AGE
pvc-fd4e5f27-360f-42b0-821c-b9dc8b8fca35  1Gi        RWO          Delete
Bound      elastic-system/elasticsearch-data-quickstart-canary-es-default-canary-0
default      <unset>           5m
$ kubectl get pvc -n elastic-system
NAME                                     STATUS      VOLUME
CAPACITY   ACCESS MODES  STORAGECLASS
elasticsearch-data-quickstart-canary-es-default-canary-0  Bound
pvc-fd4e5f27-360f-42b0-821c-b9dc8b8fca35  1Gi        RWO          default
<unset>           5m
```

この状態で、Elasticsearch にデータを投入してみましょう。

リスト 6.35: Elasticsearch にインデックスを作成する

```
$ curl -u "elastic:${PASSWORD}" -X PUT -k "${YOUR-EXTERNAL-IP}/products?pretty"-k
"${ELA"
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "products"
}
$ curl -u "elastic:${PASSWORD}" -X POST -k "${YOUR-EXTERNAL-IP}/_bulk?pretty" -H
'Content-Type: application/json' --data-binary @data.json
{
  "errors" : false,
  "took" : 200,
  "items" : [
    {
      "index" : {
        "_index" : "products",
        "_id" : "1",
        "_version" : 1,
        "result" : "created",
        "_shards" : {
          "total" : 2,
          "successful" : 1,
          "failed" : 0
        }
      }
    }
  ]
}
```

```
        },
        "_seq_no" : 0,
        "_primary_term" : 1,
        "status" : 201
    }
},
{
    "index" : {
        "_index" : "products",
        "_id" : "2",
        "_version" : 1,
        "result" : "created",
        "_shards" : {
            "total" : 2,
            "successful" : 1,
            "failed" : 0
        },
        "_seq_no" : 1,
        "_primary_term" : 1,
        "status" : 201
    }
},
{
    "index" : {
        "_index" : "products",
        "_id" : "3",
        "_version" : 1,
        "result" : "created",
        "_shards" : {
            "total" : 2,
            "successful" : 1,
            "failed" : 0
        },
        "_seq_no" : 2,
        "_primary_term" : 1,
        "status" : 201
    }
},
{
    "index" : {
        "_index" : "products",
```

```
"_id" : "4",
"_version" : 1,
"_result" : "created",
"_shards" : {
  "total" : 2,
  "successful" : 1,
  "failed" : 0
},
"_seq_no" : 3,
"_primary_term" : 1,
"status" : 201
}
}
]
}
}

$ curl -u "elastic:${PASSWORD}" -X GET -k "${YOUR-EXTERNAL-IP}/products/_search?q=name:Elasticsearch"
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 1.2039728,
    "hits" : [
      {
        "_index" : "products",
        "_id" : "1",
        "_score" : 1.2039728,
        "_source" : {
          "product_id" : 1,
          "name" : "Elasticsearch",
          "category" : "Search Engine",
          "price" : 0,
        }
      }
    ]
  }
}
```

```

        "in_stock" : true,
        "release_date" : "2010-02-08",
        "description" : "A distributed, RESTful search and analytics engine."
    }
}
]
}
}

```

データが入ったことが確認できたので、ElasticsearchのPodを削除します。

リスト6.36: ElasticsearchのPodを削除する

```
$ kubectl delete -f manifest/elasticsearch.yaml
elasticsearch.elasticsearch.k8s.elastic.co "quickstart" deleted
```

従来だと、Podの削除のタイミングでインデックスデータは削除されてしまっていました。しかし、永続化ボリュームと繋げたことで、データが削除されずに残っています。

リスト6.37: 永続化ボリュームが残っていることを確認する

```
$ kubectl get pv -n elastic-system
NAME                                     CAPACITY   ACCESS MODES  RECLAIM
POLICY      STATUS      CLAIM
VOLUMEATTRIBUTESCLASS      REASON      AGE
pvc-9ea2e9bd-8687-46d9-ac43-6fd55d9e506a   10Gi       RWO          Delete
Bound      elastic-system/elasticsearch-data-quickstart-es-default-0
managed-standard-ssd <unset>                  3m11s
$ kubectl get pvc -n elastic-system
NAME           STATUS      VOLUME
CAPACITY   ACCESS MODES  STORAGECLASS      VOLUMEATTRIBUTESCLASS      AGE
elasticsearch-data-quickstart-es-default-0   Bound      pvc-9ea2e9bd-8687-46d9-ac43-6fd55d9e506a
10Gi        RWO          managed-standard-ssd <unset>                  3m11s
```

再びElasticsearchをデプロイしてみましょう。

リスト6.38: Elasticsearchのマニフェストを適用する

```
$ kubectl apply -f manifest/elasticsearch.yaml
elasticsearch.elasticsearch.k8s.elastic.co/quickstart created
$ curl -u "elastic:${PASSWORD}" -X GET -k "${YOUR-EXTERNAL-IP}/products/_search?q=name:Elast
{
  "took" : 3,
  "timed_out" : false,
```

```
_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
},
"hits" : {
    "total" : {
        "value" : 1,
        "relation" : "eq"
    },
    "max_score" : 1.2039728,
    "hits" : [
        {
            "_index" : "products",
            "_id" : "1",
            "_score" : 1.2039728,
            "_source" : {
                "product_id" : 1,
                "name" : "Elasticsearch",
                "category" : "Search Engine",
                "price" : 0,
                "in_stock" : true,
                "release_date" : "2010-02-08",
                "description" : "A distributed, RESTful search and analytics engine."
            }
        }
    ]
}
```

インデックスを再作成することなく、即座に検索ができました。内容も Pod削除前と同じです。以上のように、Azure Disk を使って、Elasticsearch のデータを永続化することができました。

6.8 AKS クラスターを削除しよう

最後に、後始末としてクラスターを削除しておきましょう。リソース詳細画面上の「削除」ボタンからクラスターを削除できます。

図 6.20: AKS クラスターの削除

myAKSCluster

概要

リソース グループ : aks-es-test

電源の状態 : Running

クラスター操作の状態 : 成功

サブスクリプション : Azure subscription 1

場所 : East US

タグ (編集)

削除

図 6.21: AKS クラスターの削除確認



NextPublishing Sample

あとがき

本書を最後までお読みいただき、ありがとうございます。

かねてより書きたかった、Elasticsearch x Kubernetesの本がようやく形になりました。Solr関連の本を出すたびに「なんで Elasticsearch はないんですか?」と聞かれ続け、ついに形になりました。個人的には、Solr コミュニティーももっと盛り上がってほしいなと思う一方、世間的なニーズを鑑みて本書の執筆に至った次第です。

当初は、Solr も Elasticsearch も同じ Lucene ベースだから苦労しないだろうと、高を括って執筆を始めました。実際は、なまじ Solr の知識がある分、用語や微妙な設計思想の違いに苦戦しました。振り返ってみると苦労と学びの多い、貴重な経験ができました。

みなさんにとって、本書はいかがでしたでしょうか? 本書を読む前は Kubernetes 初心者だった読者の方は、多少なりとも使える気になれましたでしょうか? そこまでいかなくても、Kubernetes のすごさは感じていただけたでしょうか?

私たちも本書を執筆開始した時点では、Kubernetes のことはおっかなびっくり触っている状態でした。今でもいわゆる「完全に理解した」レベルですが、思わず勉強してでも業務で使ってみたくなるほど魅力を感じるツールでした。

本書の執筆途中に開催された Rails World の基調講演では、「全文検索エンジンはユーザーにとってはすばらしい機能を提供する一方で、開発者や運用者にとっては拷問のような存在だ」と語られていたのが印象的でした¹。これは個人的には誇張された表現だと思いますが、それを差し引いても複雑なシステムの構築・運用が求められていることは間違いないと思います。特に進んで検索エンジニアを目指す人は少なく、限られたメンバーで回していくのは大変だというのが、著者らの実感です。そんな厳しい状況での助けになるかもしれないという光明を感じていただけただけでも、本書を執筆した甲斐があったなと思います。

さて、次回は何について書きましょうかね? またまた検索系の何かの話か、最近ハマっている旅行関連のテックブックもいいかもしれませんね。

もしご意見、ご感想、書評いただけたら、とても嬉しいです。SNS や GitHub、メールなどなど、なんでも歓迎です²³⁴。それでは、またいつかどこかでお会いしましょう。

謝辞

本書の執筆にあたって、多くの方にご協力いただきました。お声がけくださったインプレス NextPublishing の山城敬さんにはいつも助けていただいております。

また、本書の底本となった同人版の執筆にあたっても、色々な方にご尽力いただきました。そして何より、本書を手に取ってくださった読者のみなさんのおかげで、毎度刊行ができます。

1.https://www.publickey1.jp/blog/24/rails_8rails_81webelasticsearchactive_recordddhh.html

2.<https://github.com/Sashimimochi>

3.<https://x.com/Sashimimochi343>

4.<https://sashimimochi.netlify.app/>

本当にありがとうございます。

著者紹介

○○ ○○ (○○ ○○)

○

○○ ○○ (○○ ○○)

○

◎本書スタッフ

アートディレクター/装丁：岡田章志+GY

編集協力：■■■■

ディレクター：栗原 翔

（表紙イラスト）

■■■■

技術の泉シリーズ・刊行によせて

技術者の知見のアウトプットである技術同人誌は、急速に認知度を高めています。インプレス NextPublishingは国内最大級の即売会「技術書典」(<https://techbookfest.org/>)で発表された技術同人誌を底本とした商業書籍を2016年より刊行し、これらを中心とした『技術書典シリーズ』を開催してきました。2019年4月、より幅広い技術同人誌を対象とし、最新の知見を発信するため『技術の泉シリーズ』へリニューアルしました。今後は「技術書典」をはじめとした各種即売会や、勉強会・LT会などで発表された技術同人誌を底本とした商業書籍を刊行し、技術同人誌の普及と発展に貢献することを目指します。エンジニアの“知の結晶”である技術同人誌の世界に、より多くの方が触れていただくきっかけになれば幸いです。

インプレス NextPublishing
技術の泉シリーズ 編集長 山城 敬

●お断り

掲載したURLは202■年■月1日現在のものです。サイトの都合で変更されることがあります。また、電子版ではURLにハイパーリンクを設定していますが、端末やビューアー、リンク先のファイルタイプによっては表示されないことがあります。あらかじめご了承ください。

●本書の内容についてのお問い合わせ先

株式会社インプレス

インプレス NextPublishing メール窓口

np-info@impress.co.jp

お問い合わせの際は、書名、ISBN、お名前、お電話番号、メールアドレスに加えて、「該当するページ」と「具体的なご質問内容」「お使いの動作環境」を必ず明記ください。なお、本書の範囲を超えるご質問にはお答えできないのでご了承ください。

電話やFAXでのご質問には対応しておりません。また、封書でのお問い合わせは回答までに日数をいただく場合があります。あらかじめご了承ください。

奥付サンプル

201X年X月X日 初版発行Ver.1.0 (PDF版)

著 者 × × × ×
編集人 × × × ×
発行人 × × × ×
発 行 株式会社インプレスR&D

〒101-0051
東京都千代田区神田神保町一丁目105番地
<http://nextpublishing.jp/>

●本書は著作権法上の保護を受けています。本書の一部あるいは全部について株式会社インプレスR&Dから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

©201X, All rights reserved.

ISBN978-4-XXXX-XXXX-X



Next Publishing®

●インプレス Next Publishingは、株式会社インプレスR&Dが開発したデジタルファースト型の出版モデルを承継し、幅広い出版企画を電子書籍+オンデマンドによりスピーディで持続可能な形で実現しています。<https://nextpublishing.jp/>

NextPublishing Sample