OpenShift Container Platform 4.5

Serverless

OpenShift Serverless のインストール、使用法、およびリリースノート

Last Updated: 2023-05-22

OpenShift Container Platform 4.5 Serverless

OpenShift Serverless のインストール、使用法、およびリリースノート

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

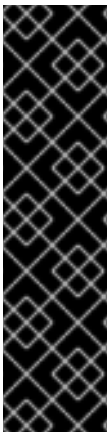
概要

本書では、OpenShift Container Platform で OpenShift Serverless を使用する方法について説明します。

目次

第1章 OPENSHIFT SERVERLESS リリースノート

OpenShift Serverless 機能の概要については、OpenShift Serverless の使用開始 を参照してください。



重要

Knative Eventing はテクノロジープレビュー機能としてのみご利用いただけます。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、https://access.redhat.com/ja/support/offerings/techpreview/ を参照してください。

1.1. RED HAT OPENSHIFT SERVERLESS 1.10.1 のリリースノート

OpenShift Serverless の本リリースでは、CVE (Common Vulnerabilities and Exposures) およびバグ修正に対応しています。

1.1.1. 修正された問題

- 本リリースは、Universal Base Image (UBI) が 1.10.0 の **ubi8-minimal-container-8.2-349** から **ubi8-minimal-container-8.3-230** にアップグレードされました。

1.2. RED HAT OPENSHIFT SERVERLESS 1.10.0 のリリースノート

1.2.1. 新機能

- OpenShift Serverless は Knative Operator 0.16.0 を使用するようになりました。
- OpenShift Serverless は Knative Serving 0.16.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.16.0 を使用しています。
- OpenShift Serverless は Kourier 0.16.0 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.16.1 を使用するようになりました。
- これまでブローカー作成の namespace にラベルを付けるために使用されていた **knative-eventing-injection=enabled** アノテーションが非推奨になりました。新しいアノテーションは **eventing.knative.dev/injection=enabled** です。詳細は、**ブローカーおよびトリガーを使用したイベント配信ワークフロー** についてのドキュメントを参照してください。
- マルチコンテナーのサポートがテクノロジープレビュー機能として Knative で利用可能になりました。**config-features** 設定マップでマルチコンテナーのサポートを有効にすることができます。詳細は、Knative ドキュメント を参照してください。

1.2.2. 修正された問題

- 以前のリリースでは、Knative Serving には **queue-proxy** 用に固定された最小の CPU 要求 25m

が含まれていました。クラスターにこの値と競合する値が設定されていた場合、たとえば **defaultRequest** の最小 CPU 要求として **25m** を超える値が設定されていた場合、Knative サービスはデプロイに失敗しました。この問題は 1.10.0 で修正されています。

1.3. RED HAT OPENSHIFT SERVERLESS 1.9.0 のリリースノート

1.3.1. 新機能

- OpenShift Serverless は Knative Operator 0.15.2 を使用するようになりました。Knative Serving および Knative Eventing Operator は共通の Operator に統合されるようになりました。
- OpenShift Serverless は Knative Serving 0.15.2 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.15.2 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.15.2 を使用するようになりました。
- OpenShift Serverless は Kourier 0.15.0 を使用するようになりました。
- OpenShift Serverless は、サイドカーの有効化や JSON Web Token (JWT) 認証などの一部の統合 Red Hat OpenShift Service Mesh 機能をサポートするようになりました。サポートされる機能は、**ネットワーク**ガイドに記載されています。

1.3.2. 既知の問題

- **KnativeEventing** カスタムリソース (CR) を削除した後に、 **v0.15.0-upgrade-xr55x** および **storage-version-migration-eventing-99c7q** Pod はクラスター上に残り、 **Completed** ステータスを表示します。**KnativeEventing** CR がインストールされた namespace を削除して、これらの Pod を完全に削除できます。

1.4. RED HAT OPENSHIFT SERVERLESS 1.8.0 のリリースノート

1.4.1. 新機能

- OpenShift Serverless は Knative Serving 0.14.1 を使用するようになりました。
- OpenShift Serverless は Knative Serving Operator 0.14.0 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.14.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.14.2 を使用するようになりました。
- OpenShift Serverless は Knative Eventing Operator 0.14.0 を使用するようになりました。
- OpenShift Serverless は Kourier 0.14.1 を使用するようになりました。

1.4.2. 既知の問題

- Knative Serving には、 **queue-proxy** について固定された最小 **25m** の CPU 要求が設定されます。クラスターにこの値と競合する値が設定されている場合、たとえば **defaultRequest** の最小 CPU 要求として **25m** を超える値が設定される場合、Knative サービスはデプロイに失敗します。回避策として、Knative サービスに対して **resourcePercentage** アノテーションを個別に設定できます。

resourcePercentage の設定例

```
spec:
  template:
    metadata:
      annotations:
        queue.sidecar.serving.knative.dev/resourcePercentage: "10" (1)
```

(1) **queue.sidecar.serving.knative.dev/resourcePercentage** は、**queue-proxy** に使用されるユーザーコンテナーリソースの割合です。これには 0.1 - 100 の範囲を設定できます。

- OpenShift Container Platform 4.5 以降のバージョンでは、トラフィック分配機能を使用して Knative サービスをデプロイすると、Web コンソールの **Developer** パースペクティブに、一般的なサービスアドレスの無効な URL が表示されます。
  正しい URL が YAML リソースと CLI コマンドの出力に表示されます。

- OpenShift Serverless で ping ソースを使用している場合、他のすべての Knative Eventing コンポーネントをアンインストールし、削除した後に、**pingsource-jobrunner Deployment** リソースは削除されません。

- シンクに接続されているシンクバインディングを削除する前にシンクを削除すると、**SinkBinding** オブジェクトの削除がハングする可能性があります。
  回避策として、**SinkBinding** オブジェクトを編集し、ハングの原因となるファイナライザーを削除できます。

```
finalizers:
 - sinkbindings.sources.knative.dev
```

- シンクバインディングの動作が OpenShift Serverless 1.8.0 で変更され、これにより後方互換性が失われます。
  シンクバインディングを使用するには、クラスター管理者は、**bindings.knative.dev/include:"true"** で **SinkBinding** オブジェクトに設定された namespace にラベルを付ける必要があります。

  SinkBinding オブジェクトで設定されるリソースにも、**bindings.knative.dev/include:"true"** でラベル付けする必要がありますが、このタスクは すべての OpenShift Serverless ユーザーが実行できます。

  1. クラスター管理者は、以下のコマンドを入力して namespace にラベルを付けることができます。

     ```
     $ oc label namespace <namespace> bindings.knative.dev/include=true
     ```

  2. ユーザーは、**bindings.knative.dev/include=true** ラベルをリソースに手動で追加する必要があります。
     たとえば、このラベルを **CronJob** オブジェクトに追加するには、以下の行をジョブリソースの YAML 定義に追加します。

     ```
     jobTemplate:
       metadata:
         labels:
           app: heartbeat-cron
           bindings.knative.dev/include: "true"
     ```

1.5. RED HAT OPENSHIFT SERVERLESS 1.7.2 のリリースノート

OpenShift Serverless の本リリースでは、CVE (Common Vulnerabilities and Exposures) およびバグ修正に対応しています。

1.5.1. 修正された問題

- 以前のバージョンの OpenShift Serverless では、**KnativeServing** カスタムリソースは、Kourier がデプロイしない場合でも **Ready** のステータスを表示します。この問題は OpenShift Serverless 1.7.2 で修正されました。

1.6. RED HAT OPENSHIFT SERVERLESS 1.7.1 のリリースノート

1.6.1. 新機能

- OpenShift Serverless は Knative Serving 0.13.3 を使用するようになりました。
- OpenShift Serverless は Knative Serving Operator 0.13.3 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.13.2 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.13.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing Operator 0.13.3 を使用するようになりました。

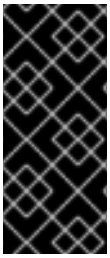
1.6.2. 修正された問題

- OpenShift Serverless 1.7.0 では、ルートは不要になった場合に継続的に調整されました。この問題は OpenShift Serverless 1.7.1 で修正されました。

1.7. RED HAT OPENSHIFT SERVERLESS 1.7.0 のリリースノート

1.7.1. 新機能

- OpenShift Serverless 1.7.0 は、OpenShift Container Platform 4.3 以降のバージョンで一般に利用可能 (GA) になりました。以前のバージョンでは、OpenShift Serverless はテクノロジープレビューでした。
- OpenShift Serverless は Knative Serving 0.13.2 を使用するようになりました。
- OpenShift Serverless は Knative Serving Operator 0.13.2 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.13.2 を使用するようになりました。
- Knative **kn** CLI ダウンロードが、非接続またはネットワークが制限されたインストールをサポートするようになりました。
- Knative **kn** CLI ライブラリーが Red Hat によって署名されるようになりました。
- Knative Eventing が OpenShift Serverless でテクノロジープレビューとして利用可能になりました。OpenShift Serverless は Knative Eventing 0.13.2 を使用するようになりました。



重要

最新の Serverless リリースにアップグレードする前に、事前にインストールしている場合には、コミュニティー Knative Eventing Operator を削除する必要があります。Knative Eventing Operator をインストールすると、OpenShift Serverless 1.7.0 に含まれる Knative Eventing の最新のテクノロジープレビューバージョンをインストールできなくなります。

- 高可用性 (HA) は、**autoscaler-hpa**、 **controller**、**activator**、 **kourier-control**、および **kourier-gateway** コンポーネントに対してデフォルトで有効にされます。
以前のバージョンの OpenShift Serverless をインストールしている場合、**KnativeServing** カスタムリソース (CR) が更新されると、デプロイメントはデフォルトで **KnativeServing.spec.high-availability.replicas = 2** が指定される HA 設定になります。

  高可用性コンポーネントの設定の手順を実行して、これらのコンポーネントの HA を無効にすることができます。

- OpenShift Serverless は、OpenShift Container Platform のクラスター全体のプロキシーで **trustedCA** 設定をサポートし、OpenShift Container Platform のプロキシー設定と完全に互換性があります。

- OpenShift Serverless は、OpenShift Container Platform ルートに登録されているワイルドカード証明書を使用して HTTPS をサポートするようになりました。Knative Serving の http および https の詳細は、**サーバーレスアプリケーションのデプロイメントの確認**を参照してください。

1.7.2. 修正された問題

- 以前のバージョンでは、API グループを指定せずに **KnativeServing** CR を要求すると (コマンド **oc get knativeserving -n knative-serving** を使用するなど)、エラーが発生することがありました。この問題は OpenShift Serverless 1.7.0 で修正されました。

- 以前のバージョンでは、Knative Serving コントローラーは、サービス CA 証明書のローテーションにより新規サービス CA 証明書が生成されても通知されませんでした。サービス CA 証明書のローテーション後に作成された新規リビジョンはエラーを出して失敗しました。

  > Revision "foo-1" failed with message: Unable to fetch image "image-registry.openshift-image-registry.svc:5000/eap/eap-app": failed to resolve image to digest: failed to fetch image information: Get https://image-registry.openshift-image-registry.svc:5000/v2/: x509: certificate signed by unknown authority.

  OpenShift Serverless Operator は、新規サービス CA 証明書が生成されるたびに Knative Serving コントローラーを再起動するようになりました。これにより、コントローラーは常に現在のサービス CA 証明書を使用するように設定されます。詳細は、OpenShift Container Platform ドキュメントの **認証**の**サービス提供証明書のシークレットによるサービストラフィックのセキュリティー保護**を参照してください。

1.7.3. 既知の問題

- OpenShift Serverless 1.6.0 から 1.7.0 にアップグレードする場合、HTTPS のサポートではルートの形式を変更する必要があります。OpenShift Serverless 1.6.0 で作成された Knative サービスは、古い形式の URL で到達できなくなりました。OpenShift Serverless のアップグレード後に、各サービスの新規 URL を取得する必要があります。詳細は、**OpenShift Serverless のアップグレード**についてのドキュメントを参照してください。

- Azure クラスターで Knative Eventing を使用している場合、 **imc-dispatcher** Pod が起動しない可能性があります。これは、Pod のデフォルト **resources** 設定によって生じます。回避策として、**resources** 設定を削除できます。

- クラスターに 1000 の Knative サービスがあり、Knative Serving の再インストールまたはアップグレードを実行する場合、 **KnativeServing** CR の状態が Ready になった後に最初の新規サービスを作成すると遅延が生じます。
  3scale-kourier-control コントローラーは、新規サービスの作成を処理する前に以前の Knative サービスをすべて調整します。これにより、新規サービスは状態が **Ready** に更新されるまで **IngressNotConfigured** または **Unknown** の状態になり、約 800 秒の時間がかかります。

1.8. 追加リソース

OpenShift Serverless はオープンソースの Knative プロジェクトに基づいています。

- 最新の Knative Serving リリースについての詳細は、Knative Serving リリースページ を参照してください。

- 最新の Knative Serving Operator のリリースについての詳細は、Knative Serving Operator リリースページ を参照してください。

- 最新の Knative CLI リリースの詳細は、Knative client リリースページ を参照してください。

- 最新の Knative Eventing リリースの詳細は、Knative Eventing リリースページ を参照してください。

第2章 OPENSHIFT SERVERLESS のサポート

2.1. サポート

本書で説明されている手順で問題が発生した場合は、Red Hat カスタマーポータル (http://access.redhat.com) にアクセスしてください。カスタマーポータルでは、次のことができます。

- Red Hat 製品に関する技術サポート記事の Red Hat ナレッジベースの検索またはブラウズ。
- Red Hat グローバルサポートサービス (GSS) へのサポートケースの送信
- その他の製品ドキュメントへのアクセス

本書の改善が提案されている場合やエラーが見つかった場合は、**Documentation** コンポーネントの **Product** に対して、http://bugzilla.redhat.com から Bugzilla レポートを送信してください。コンテンツを簡単に見つけられるよう、セクション番号、ガイド名、OpenShift Serverless のバージョンなどの詳細情報を記載してください。

2.2. サポート用の診断情報の収集

サポートケースを作成する際、ご使用のクラスターについてのデバッグ情報を Red Hat サポートに提供していただくと Red Hat のサポートに役立ちます。

must-gather ツールを使用すると、OpenShift Serverless に関連するデータを含む、 OpenShift Container Platform クラスターについての診断情報を収集できます。

迅速なサポートを得るには、OpenShift Container Platform と OpenShift Serverless の両方の診断情報を提供してください。

2.2.1. must-gather ツールについて

oc adm must-gather CLI コマンドは、以下のような問題のデバッグに必要となる可能性のあるクラスターからの情報を収集します。

- リソース定義
- 監査ログ
- サービスログ

--image 引数を指定してコマンドを実行する際にイメージを指定できます。イメージを指定する際、ツールはその機能または製品に関連するデータを収集します。

oc adm must-gather を実行すると、新しい Pod がクラスターに作成されます。データは Pod で収集され、**must-gather.local** で始まる新規ディレクトリーに保存されます。このディレクトリーは、現行の作業ディレクトリーに作成されます。

2.2.2. OpenShift Serverless データの収集について

oc adm must-gather CLI コマンドを使用してクラスターについての情報を収集できます。これには、OpenShift Serverless に関連する機能およびオブジェクトが含まれます。**must-gather** を使用して OpenShift Serverless データを収集するには、インストールされたバージョンの OpenShift Serverless イメージおよびイメージタグを指定する必要があります。

手順

- **oc adm must-gather** コマンドを使用してデータを収集します。

```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-rhel8:<image_version_tag>
```

コマンドの例

```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-rhel8:1.10.0
```

第3章 OPENSHIFT SERVERLESS の使用開始

OpenShift Serverless は、開発者のインフラストラクチャーのセットアップまたはバックエンド開発に対する要件を軽減することにより、開発から実稼働までのコードの提供プロセスを単純化します。

3.1. OPENSHIFT SERVERLESS の仕組み

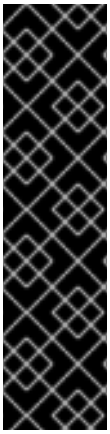
OpenShift Serverless 上の開発者は、使い慣れた言語およびフレームワークと共に、提供される Kubernetes ネイティブの API を使用してアプリケーションおよびコンテナーのワークロードをデプロイできます。

OpenShift Container Platform 上の OpenShift Serverless を使用することにより、ステートレスのサーバーレスワークロードのすべてを、自動化された操作によって単一のマルチクラウドコンテナープラットフォームで実行することができます。開発者は、それぞれのマイクロサービス、レガシーおよびサーバーレスアプリケーションをホストするために単一プラットフォームを使用することができます。

OpenShift Serverless はオープンソースの Knative プロジェクトをベースとし、エンタープライズレベルのサーバーレスプラットフォームを有効にすることで、ハイブリッドおよびマルチクラウド環境における移植性と一貫性をもたらします。

3.2. サポートされる設定

OpenShift Serverless(最新バージョンおよび以前のバージョン) のサポートされる機能、設定、および統合のセットは、 サポートされる設定 についてのページで確認できます。



重要

Knative Eventing はテクノロジープレビュー機能としてのみご利用いただけます。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、https://access.redhat.com/ja/support/offerings/techpreview/ を参照してください。

3.3. 次のステップ

- OpenShift Serverless Operator を OpenShift Container Platform クラスターにインストールして開始します。
- OpenShift Serverless リリースノート を確認します。

第4章 OPENSHIFT SERVERLESS のインストール

4.1. OPENSHIFT SERVERLESS のインストール

以下では、クラスター管理者を対象に、OpenShift Serverless Operator の OpenShift Container Platform クラスターへのインストールについて説明します。

注記

OpenShift Serverless は、ネットワークが制限された環境でのインストールに対してサポートされません。詳細は、ネットワークが制限された環境での Operator Lifecycle Manager の使用 を参照してください。

重要

最新の Serverless リリースにアップグレードする前に、事前にインストールしている場合には、コミュニティー Knative Eventing Operator を削除する必要があります。Knative Eventing Operator をインストールすると、OpenShift Serverless Operator を使用して Knative Eventing の最新のバージョンをインストールできなくなります。

4.1.1. OpenShift Serverless インストールのクラスターサイズ要件の定義

OpenShift Serverless をインストールし、使用するには、OpenShift Container Platform クラスターのサイズを適切に設定する必要があります。OpenShift Serverless の最小要件は、10 CPU および 40GB メモリーを持つクラスターです。OpenShift Serverless を実行するための合計サイズ要件は、デプロイされたアプリケーションによって異なります。デフォルトで、各 Pod は約 400m の CPU を要求し、推奨値のベースはこの値になります。指定されるサイズ要件において、アプリケーションはレプリカを最大 10 つにスケールアップできます。アプリケーションの実際の CPU 要求を減らすと、レプリカ数が増える可能性があります。

注記

指定される要件は、OpenShift Container Platform クラスターのワーカーマシンのプールにのみ関連します。マスターノードは一般的なスケジューリングには使用されず、要件から省略されます。

注記

以下の制限は、すべての OpenShift Serverless デプロイメントに適用されます。

- Knative サービスの最大数: 1000
- Knative リビジョンの最大数: 1000

4.1.2. 高度なユースケースの追加要件

OpenShift Container Platform でのロギングまたはメータリングなどの高度なユースケースの場合は、追加のリソースをデプロイする必要があります。このようなユースケースで推奨される要件は 24 vCPU および 96GB メモリーです。

クラスターで高可用性 (HA) を有効にしている場合、これには Knative Serving コントロールプレーンの各レプリカについて 0.5 - 1.5 コアおよび 200MB - 2GB のメモリーが必要です。HA は、デフォルトで一部の Knative Serving コンポーネントについて有効にされます。OpenShift Serverless での高可用性レ

プリカの設定 についてのドキュメントに従って HA を無効にできます。

4.1.3. マシンセットを使用したクラスターのスケーリング

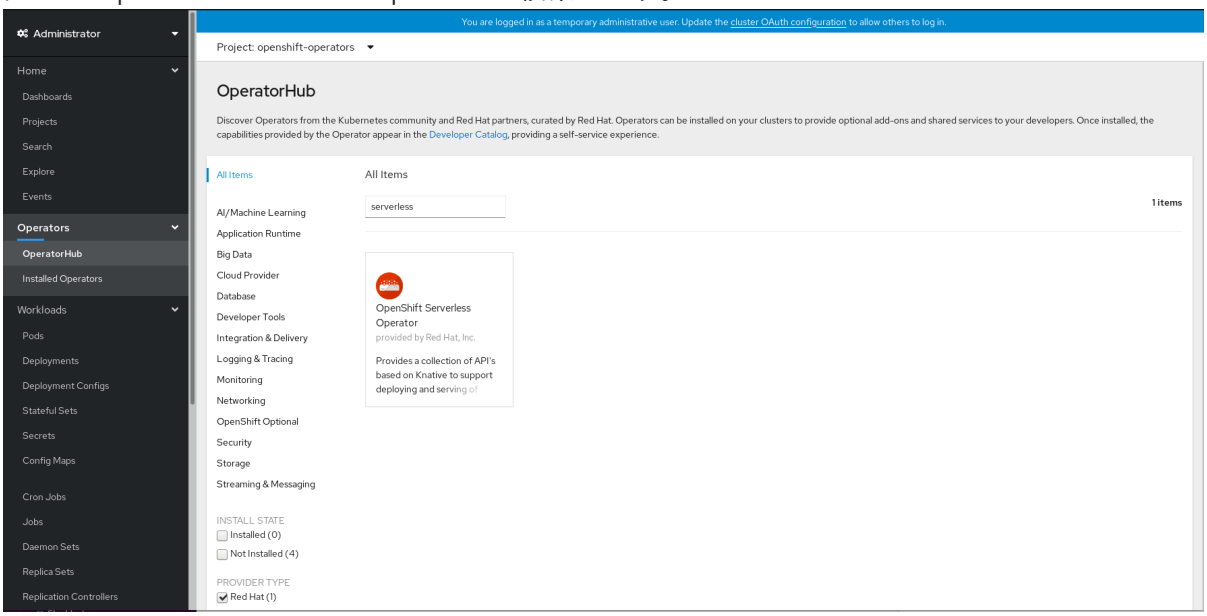
OpenShift Container Platform **MachineSet** API を使用して、クラスターを必要なサイズに手動でスケールアップすることができます。最小要件は、通常 2 つのマシンを追加することによってデフォルトのマシンセットのいずれかをスケールアップする必要があることを意味します。マシンセットの手動によるスケーリング を参照してください。

4.1.4. OpenShift Serverless Operator のインストール

この手順では、OpenShift Container Platform Web コンソールを使用して、OperatorHub から OpenShift Serverless Operator をインストールし、これにサブスクライブする方法を説明します。

手順

1. OpenShift Container Platform Web コンソールで、**Operators → OperatorHub** ページに移動します。
2. スクロールするか、またはこれらのキーワード **Serverless** を **Filter by keyword ボックス** に入力して OpenShift Serverless Operator を検索します。



3. Operator についての情報を確認してから、**Install** をクリックします。
4. **Install Operator** ページで以下を行います。
   a. **Installation Mode** は **All namespaces on the cluster (default)**になります。このモードは、デフォルトの **openshift-operators** namespace で Operator をインストールし、クラスターのすべての namespace を監視し、Operator をこれらの namespace に対して利用可能にします。
   b. **Installed Namespace** は **openshift-operators** になります。
   c. **Update Channel** として **4.5** チャネルを選択します。**4.5** チャネルは、OpenShift Serverless Operator の最新の安定したリリースのインストールを可能にします。
   d. **Automatic** または **Manual** 承認ストラテジーを選択します。

5. **Install** をクリックし、Operator をこの OpenShift Container Platform クラスターの選択した namespace で利用可能にします。

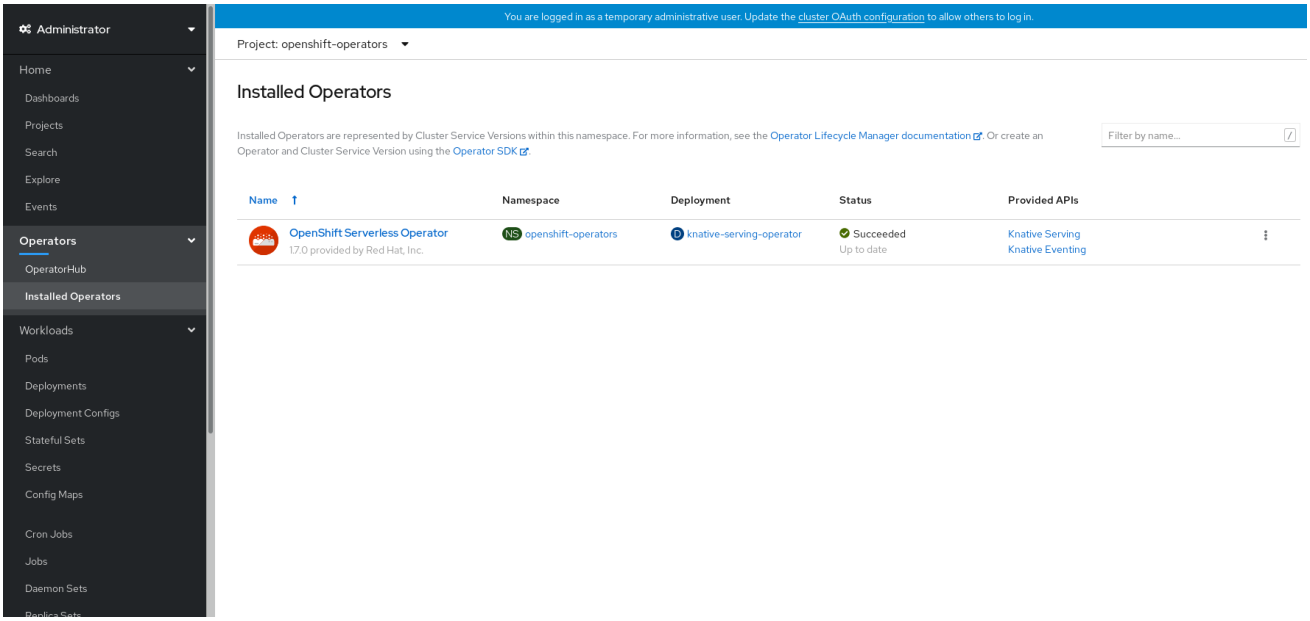
6. **Catalog → Operator Management** ページから、OpenShift Serverless Operator サブスクリプションのインストールおよびアップグレードの進捗をモニターできます。

   a. **手動** の承認ストラテジーを選択している場合、サブスクリプションのアップグレードステータスは、その Install Plan を確認し、承認するまで **Upgrading** のままになります。**Install Plan** ページでの承認後に、サブスクリプションのアップグレードステータスは **Up to date** に移行します。

   b. **自動** の承認ストラテジーを選択している場合、アップグレードステータスは、介入なしに **Up to date** に解決するはずです。

検証

サブスクリプションのアップグレードステータスが **Up to date** に移行したら、**Catalog → Installed Operators** を選択して OpenShift Serverless Operator が表示され、その **Status** が最終的に関連する namespace で **InstallSucceeded** に解決することを確認します。



上記通りにならない場合:

1. **Catalog → Operator Management** ページに切り替え、**Operator Subscriptions** および **Install Plans** タブで **Status** の下の失敗またはエラーの有無を確認します。

2. さらにトラブルシューティングの必要な問題を報告している Pod のログについては、**Workloads → Pods** ページの **openshift-operators** プロジェクトの Pod のログで確認できます。

追加リソース

- 詳細は、OpenShift Container Platform ドキュメントの **Operator のクラスターへの追加** について参照してください。

4.1.5. 次のステップ

- OpenShift Serverless Operator がインストールされた後に、Knative Serving コンポーネントをインストールできます。Knative Serving のインストール についてのドキュメントを参照してください。
- OpenShift Serverless Operator がインストールされた後に、Knative Eventing コンポーネントをインストールできます。Knative Eventing のインストール についてのドキュメントを参照してください。

4.2. KNATIVE SERVING のインストール

OpenShift Serverless Operator のインストール後に、本書で説明されている手順に従って Knative Serving をインストールできます。

本書では、デフォルト設定を使用した Knative Serving のインストールについて説明します。ただし、KnativeServing カスタムリソース定義でより高度な設定を行うことができます。

KnativeServing カスタムリソース定義の設定オプションについての詳細は、高度なインストール設定オプション を参照してください。

4.2.1. knative-serving namespace の作成

knative-serving namespace を作成する際に、**knative-serving** プロジェクトも作成されます。

重要

Knative Serving をインストールする前に、この手順を完了する必要があります。

Knative Serving のインストール時に作成された **KnativeServing** オブジェクトが **knative-serving** namespace で作成されていない場合、これは無視されます。

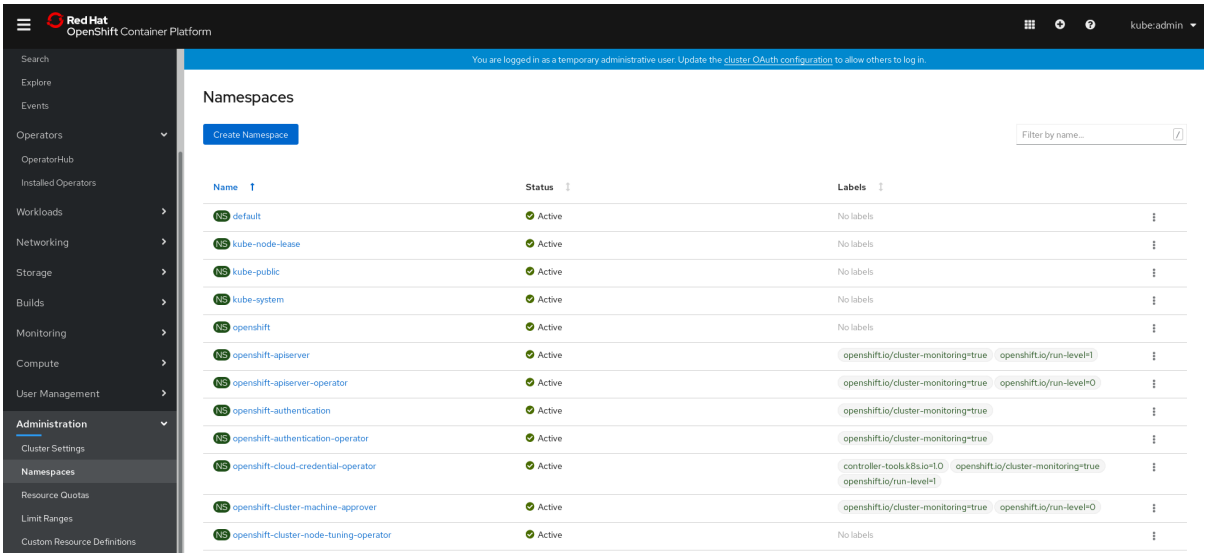
前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウント。
- OpenShift Serverless Operator がインストールされていること。

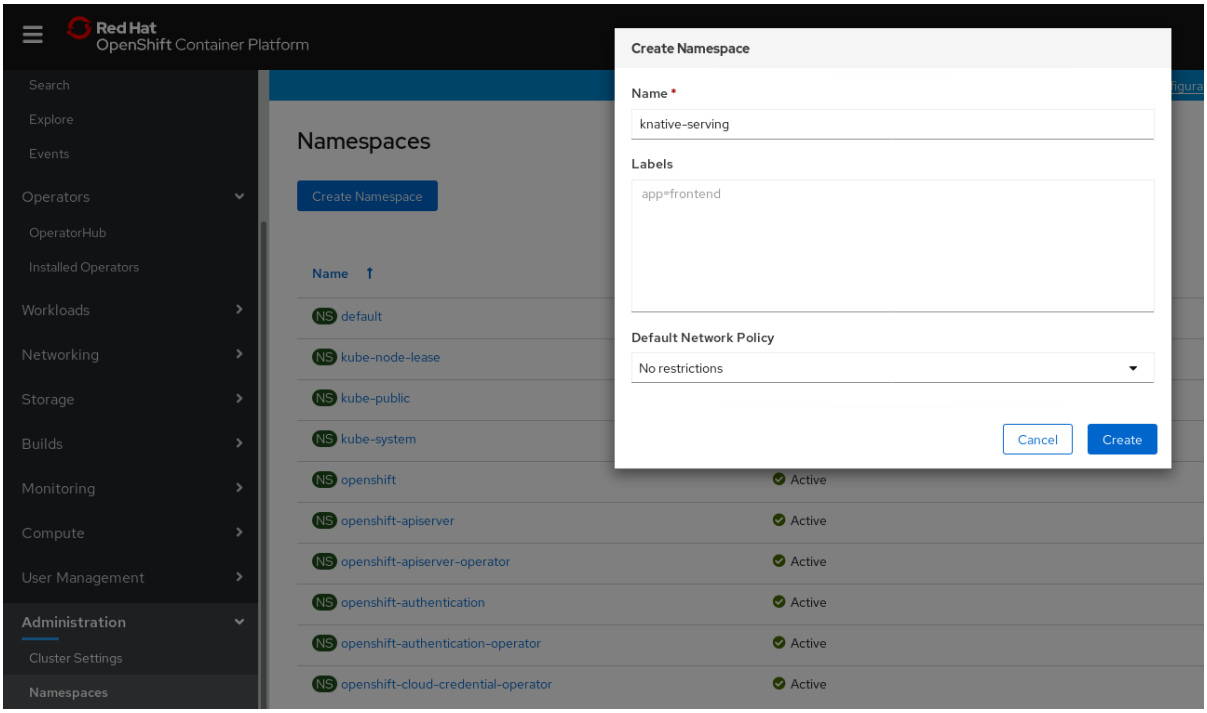
4.2.1.1. Web コンソールを使用した knative-serving namespace の作成

手順

1. OpenShift Container Platform Web コンソールで、**Administration → Namespaces** に移動します。



2. プロジェクトの **Name** として **knative-serving** を入力します。他のフィールドはオプションです。



3. **Create** をクリックします。

4.2.1.2. CLI を使用した knative-serving namespace の作成

手順

1. 以下を入力して **knative-serving** namespace を作成します。

```
$ oc create namespace knative-serving
```

4.2.2. 前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウント。
- OpenShift Serverless Operator がインストールされていること。

- **knative-serving** namespace が作成されていること。

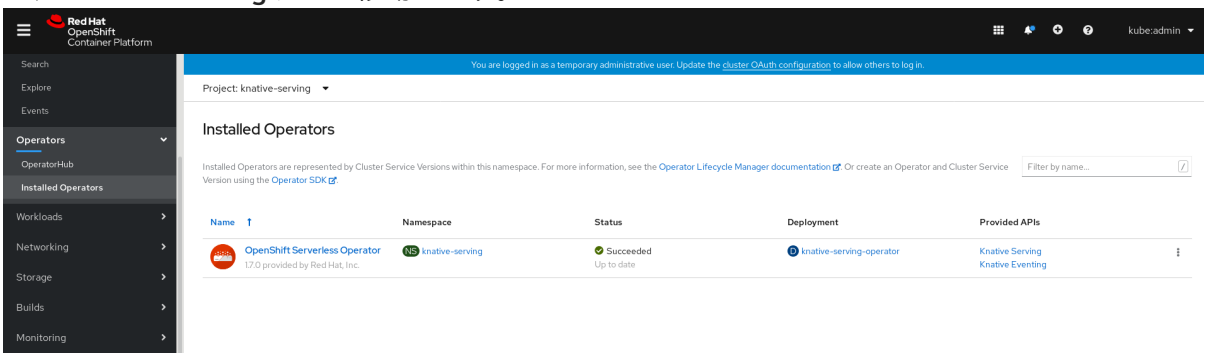
4.2.3. Web コンソールを使用した Knative Serving のインストール

手順

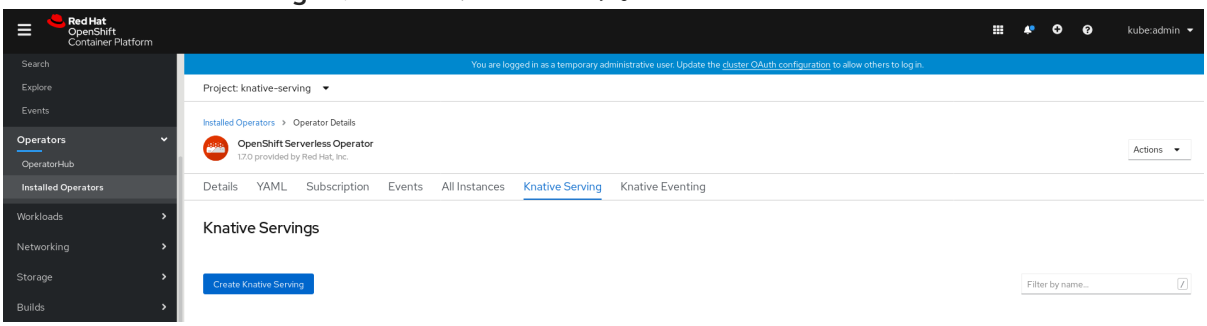
1. OpenShift Container Platform Web コンソールの **Administrator** パースペクティブで、**Operators → Installed Operators** に移動します。

2. ページ上部の **Project** ドロップダウンメニューが **Project: knative-serving** に設定されていることを確認します。

3. OpenShift Serverless Operator の **Provided API** 一覧で **Knative Serving** をクリックし、**Knative Serving** タブに移動します。



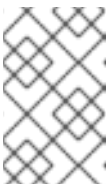
4. **Create Knative Serving**ボタンをクリックします。



5. **Create Knative Serving**ページで、**Create** をクリックしてデフォルト設定を使用し、Knative Serving をインストールできます。
   また、Knative Serving インストールの設定を変更するには、提供されるフォームを使用するか、または YAML を編集して **KnativeServing** オブジェクトを編集します。

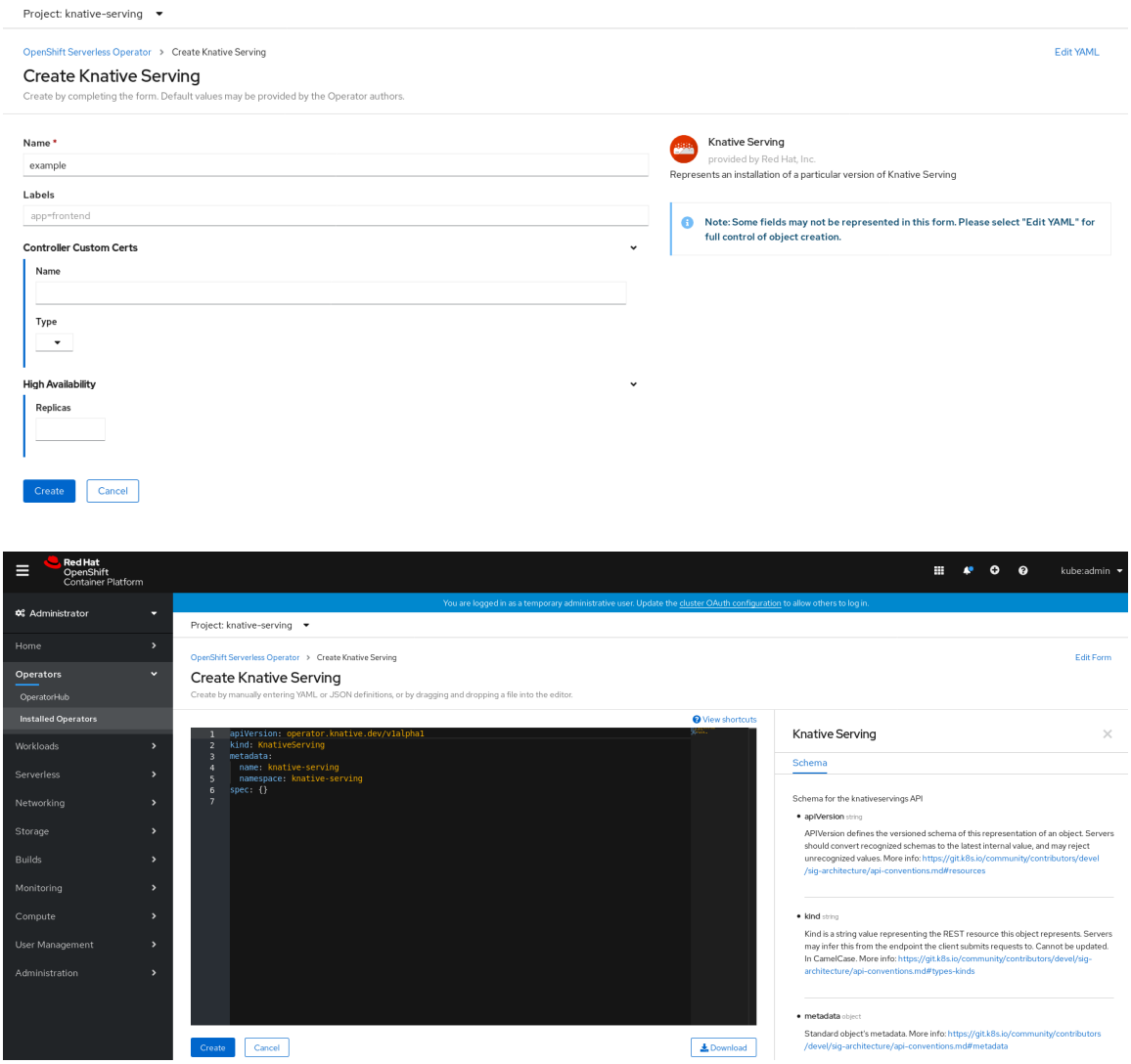
   - **KnativeServing** オブジェクト作成を完全に制御する必要がない単純な設定には、このフォームの使用が推奨されます。

   - **KnativeServing** オブジェクトの作成を完全に制御する必要のあるより複雑な設定には、YAML の編集が推奨されます。YAML にアクセスするには、**Create Knative Serving**ページの右上にある **edit YAML** リンクをクリックします。
     フォームを完了するか、または YAML の変更が完了したら、**Create** をクリックします。

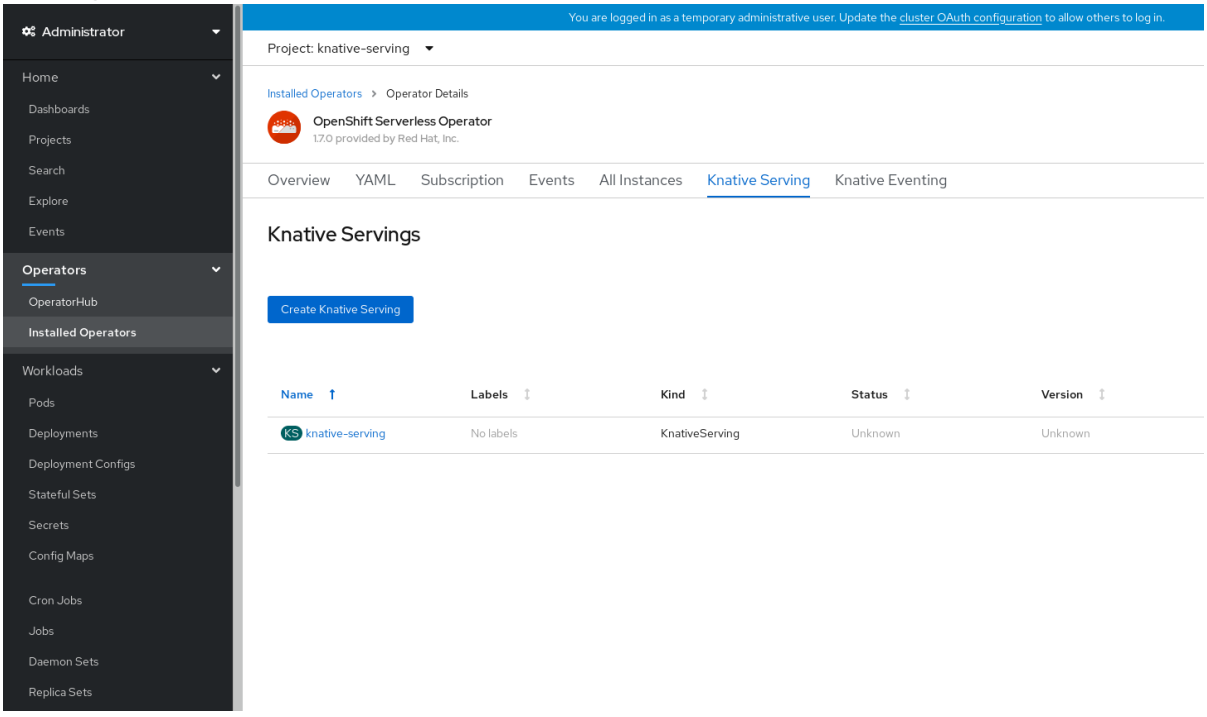


注記

KnativeServing カスタムリソース定義の設定オプションについての詳細は、**高度なインストール設定オプション** についてのドキュメントを参照してください。



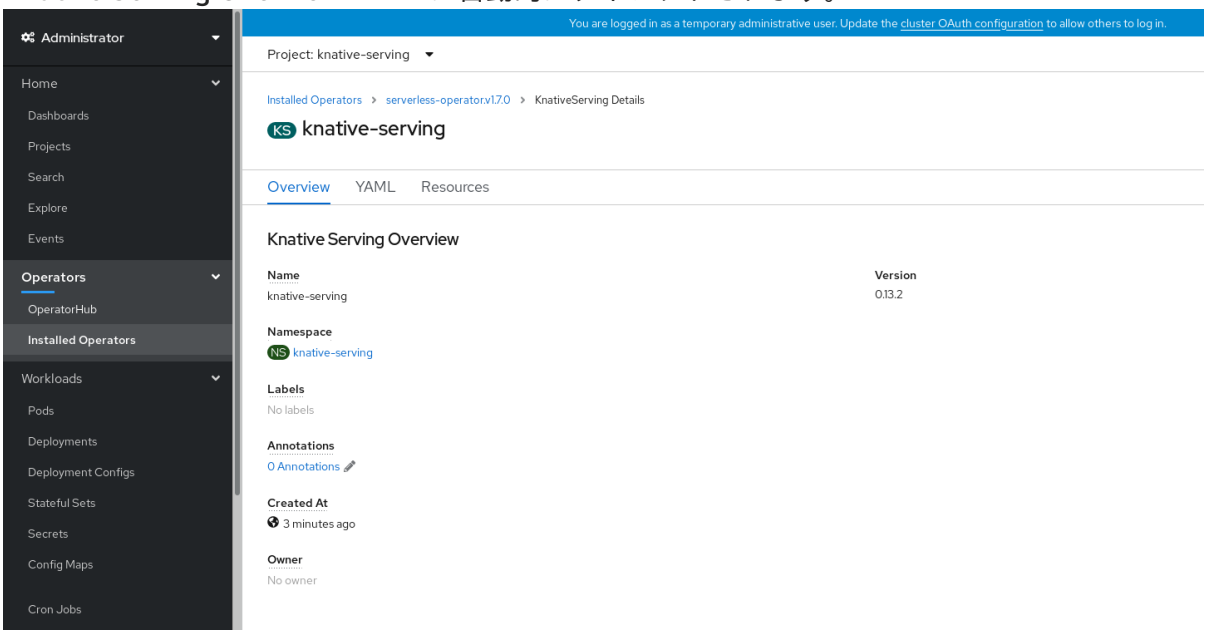
6. Knative Serving のインストール後に、**KnativeServing** オブジェクトが作成され、**Knative Serving** タブに自動的にダイレクトされます。



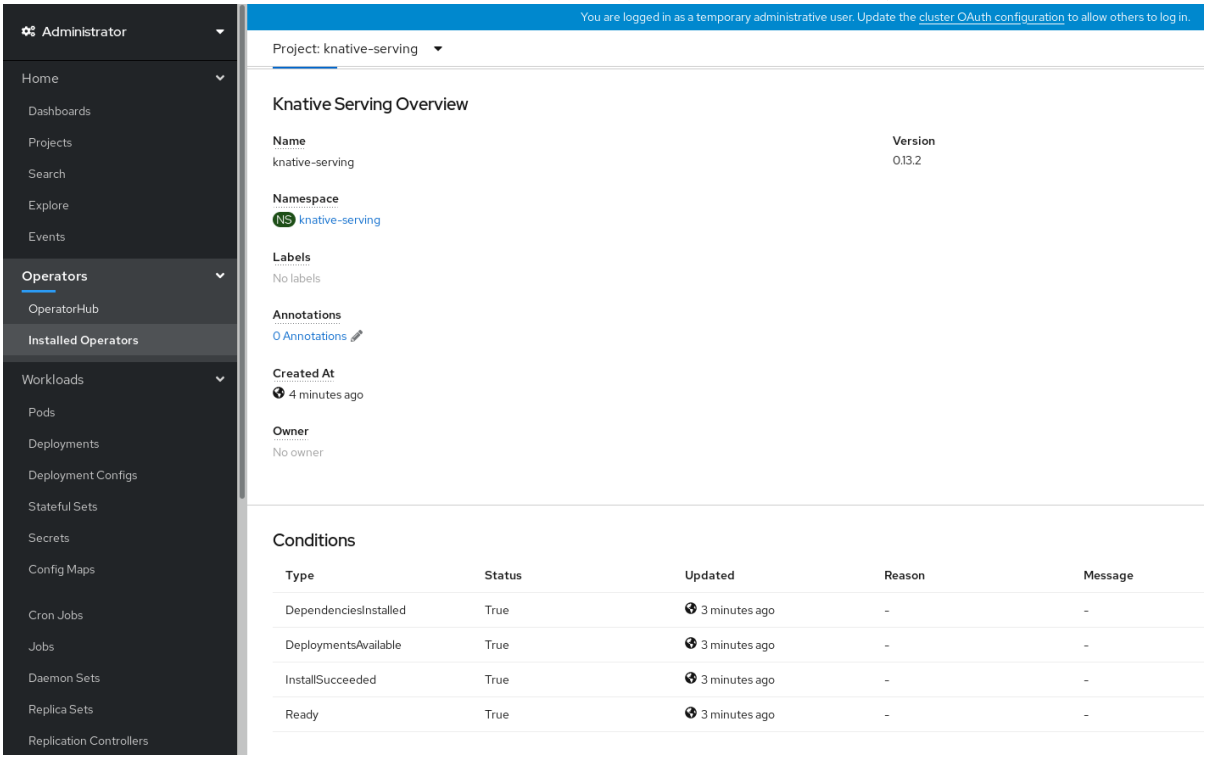
リソースの一覧に **knative-serving** が表示されます。

検証

1. Knative Serving タブの **knative-serving** をクリックします。
2. **Knative Serving Overview** ページに自動的にダイレクトされます。



3. スクロールダウンして、**Conditions** の一覧を確認します。
4. ステータスが **True** の条件の一覧が表示されます (例のイメージを参照)。



注記

Knative Serving リソースが作成されるまでに数分の時間がかかる場合があります。**Resources** タブでステータスを確認できます。

5. 条件のステータスが **Unknown** または **False** である場合は、しばらく待ってから、リソースが作成されたことを再度確認します。

4.2.4. YAML を使用した Knative Serving のインストール

手順

1. **serving.yaml** という名前のファイルを作成します。
2. 以下のサンプル YAML を **serving.yaml** にコピーします。

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
```

3. **serving.yaml** ファイルを適用します。

```
$ oc apply -f serving.yaml
```

検証

1. インストールが完了したことを確認するには、以下のコマンドを実行します。

```
$ oc get knativeserving.operator.knative.dev/knative-serving -n knative-serving --
template='{{range .status.conditions}}{{printf "%s=%s\n" .type .status}}{{end}}'
```

出力は以下のようになります。

```
DependenciesInstalled=True
DeploymentsAvailable=True
InstallSucceeded=True
Ready=True
```

注記

Knative Serving リソースが作成されるまでに数分の時間がかかる場合があります。

2. 条件のステータスが **Unknown** または **False** である場合は、しばらく待ってから、リソースが作成されたことを再度確認します。
3. 以下を入力して Knative Serving リソースが作成されていることを確認します。

```
$ oc get pods -n knative-serving
```

出力は以下のようになります。

```
NAME                                READY   STATUS    RESTARTS   AGE
activator-5c596cf8d6-5l86c          1/1     Running   0          9m37s
activator-5c596cf8d6-gkn5k          1/1     Running   0          9m22s
autoscaler-5854f586f6-gj597         1/1     Running   0          9m36s
autoscaler-hpa-78665569b8-qmlmn     1/1     Running   0          9m26s
autoscaler-hpa-78665569b8-tqwvw     1/1     Running   0          9m26s
```

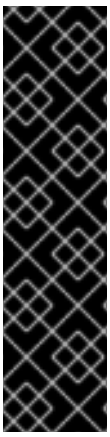
```
controller-7fd5655f49-9gxz5        1/1     Running   0          9m32s
controller-7fd5655f49-pncv5        1/1     Running   0          9m14s
kn-cli-downloads-8c65d4cbf-mt4t7   1/1     Running   0          9m42s
webhook-5c7d878c7c-n267j           1/1     Running   0          9m35s
```

4.2.5. 次のステップ

- OpenShift Serverless のクラウドイベント機能については、Knative Eventing コンポーネントをインストールできます。Knative Eventing のインストール についてのドキュメントを参照してください。
- Knative CLI をインストールして、Knative Serving で **kn** コマンドを使用します。例: **kn service** コマンドKnative CLI (**kn**)のインストール についてのドキュメントを参照してください。

4.3. KNATIVE EVENTING のインストール

OpenShift Serverless Operator のインストール後に、本書で説明されている手順に従って Knative Eventing をインストールできます。



重要

Knative Eventing はテクノロジープレビュー機能としてのみご利用いただけます。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、https://access.redhat.com/ja/support/offerings/techpreview/ を参照してください。

本書では、デフォルト設定を使用した Knative Eventing のインストールについて説明します。

4.3.1. knative-eventing namespace の作成

knative-eventing namespace の作成時に、 **knative-eventing** プロジェクトも作成されます。

重要

Knative Serving をインストールする前に、この手順を実行する必要があります。

Knative Eventing のインストール時に作成された **KnativeEventing** オブジェクトが **knative-eventing** namespace で作成されていない場合、これは無視されます。

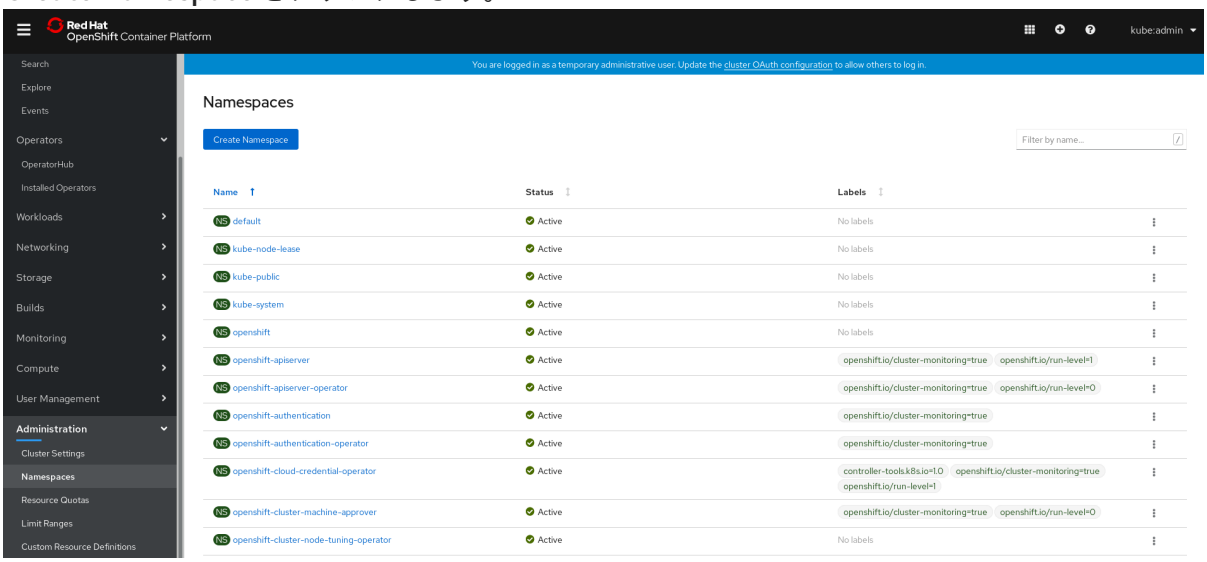
前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウント。
- OpenShift Serverless Operator がインストールされていること。

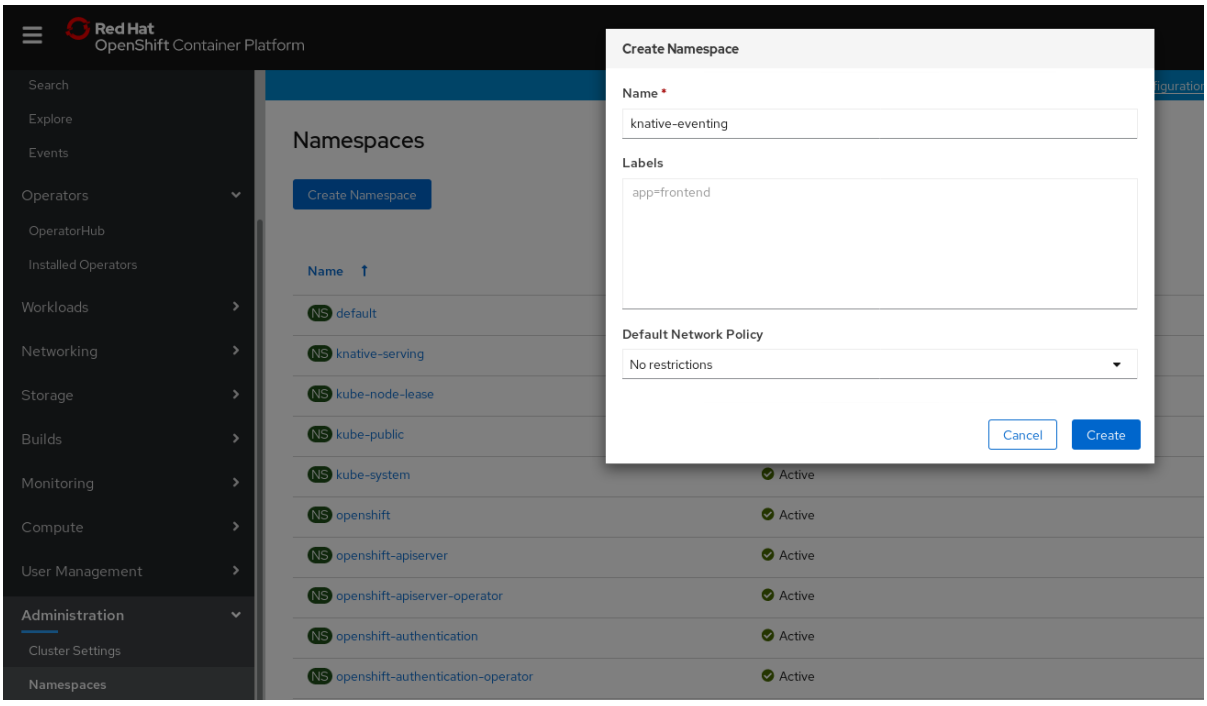
4.3.1.1. Web コンソールを使用した knative-eventing namespace の作成

手順

1. OpenShift Container Platform Web コンソールで、**Administration → Namespaces** に移動します。
2. **Create Namespace** をクリックします。



3. プロジェクトの **Name** として **knative-eventing** を入力します。他のフィールドはオプションです。



4. **Create** をクリックします。

4.3.1.2. CLI を使用した knative-eventing namespace の作成

手順

1. 以下を入力して **knative-eventing** namespace を作成します。

```
$ oc create namespace knative-eventing
```

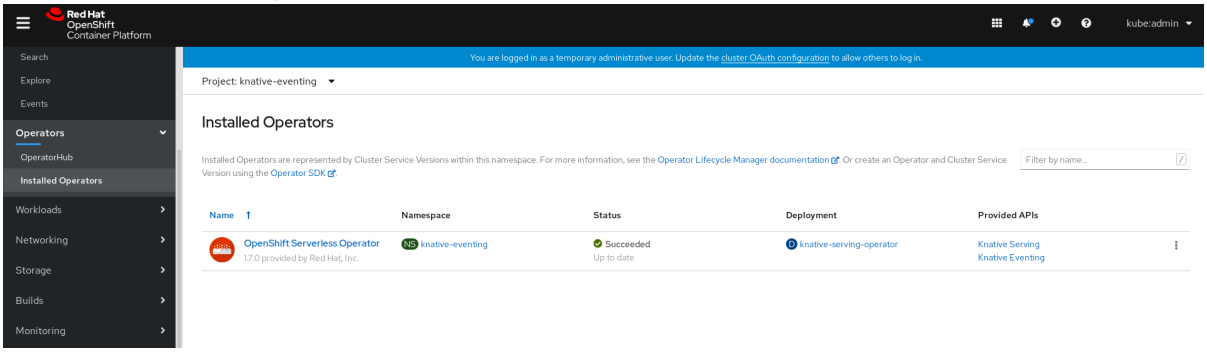
4.3.2. 前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウント。
- OpenShift Serverless Operator がインストールされていること。
- **knative-eventing** namespace が作成されていること。

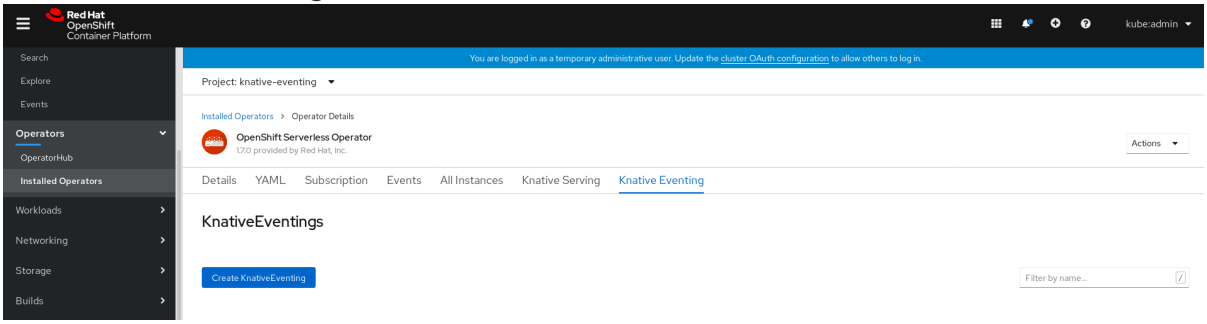
4.3.3. Web コンソールを使用した Knative Eventing のインストール

手順

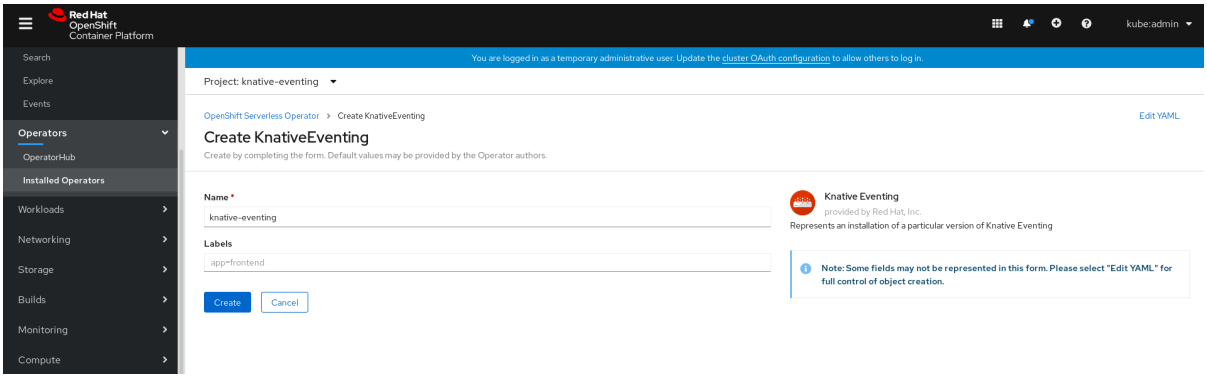
1. OpenShift Container Platform Web コンソールの **Administrator** パースペクティブで、**Operators → Installed Operators** に移動します。
2. ページ上部の **Project** ドロップダウンメニューが **Project: knative-eventing** に設定されていることを確認します。
3. OpenShift Serverless Operator の **Provided API** 一覧で **Knative Eventing** をクリックし、**Knative Eventing** タブに移動します。



4. **Create Knative Eventing**ボタンをクリックします。

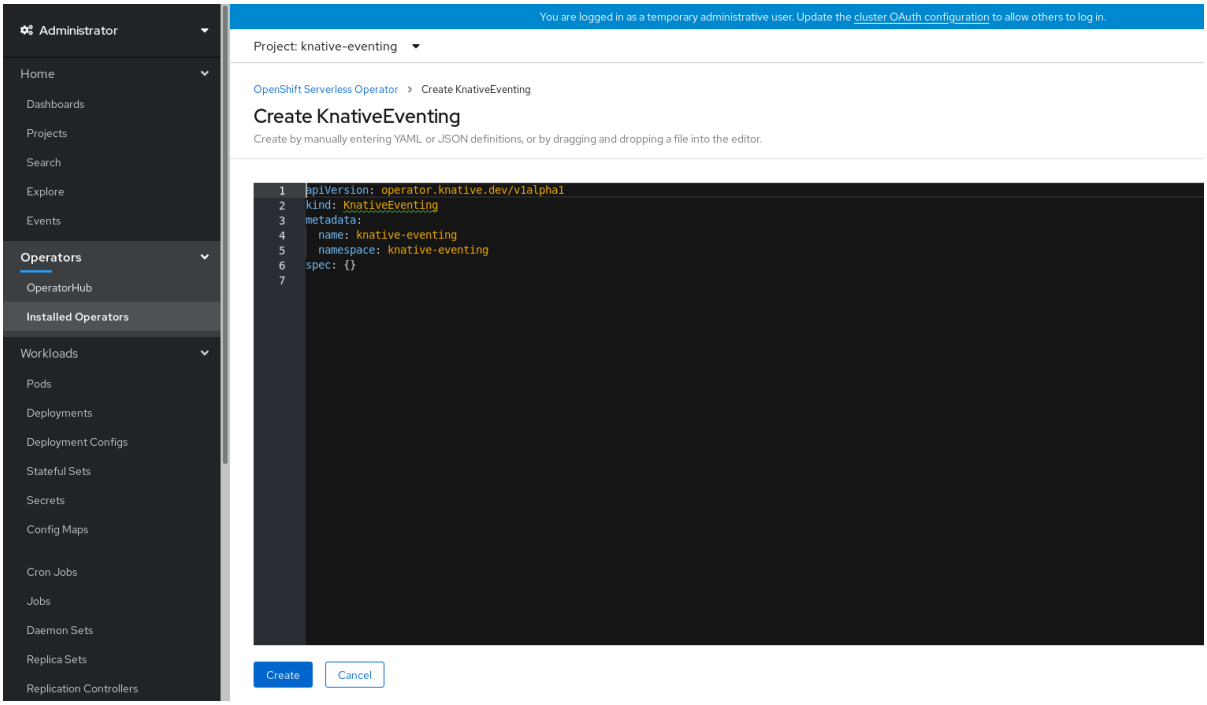


5. **Create Knative Eventing**ページでは、提供されるデフォルトのフォームを使用するか、または YAML を編集して **KnativeEventing** オブジェクトを設定できます。
   - **KnativeEventing** オブジェクト作成を完全に制御する必要がない単純な設定には、このフォームの使用が推奨されます。
     オプション。フォームを使用して **KnativeEventing** オブジェクトを設定する場合は、Knative Eventing デプロイメントに対して実装する必要のある変更を加えます。
6. **Create** をクリックします。

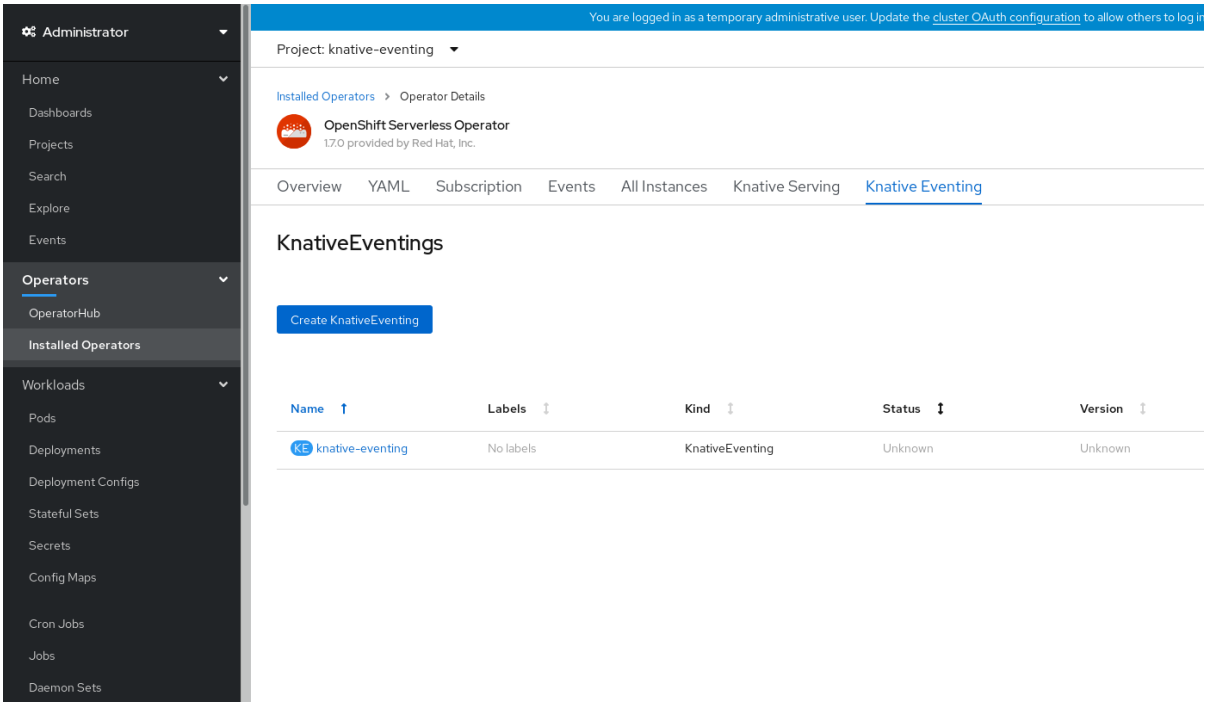


- **KnativeEventing** オブジェクトの作成を完全に制御する必要のあるより複雑な設定には、YAML の編集が推奨されます。YAML にアクセスするには、**Create Knative Eventing**ページの右上にある **edit YAML** リンクをクリックします。
  オプション。YAML を編集して **KnativeEventing** オブジェクトを設定する場合は、Knative Eventing デプロイメントについて実装する必要のある変更を YAML に加えます。

7. **Create** をクリックします。



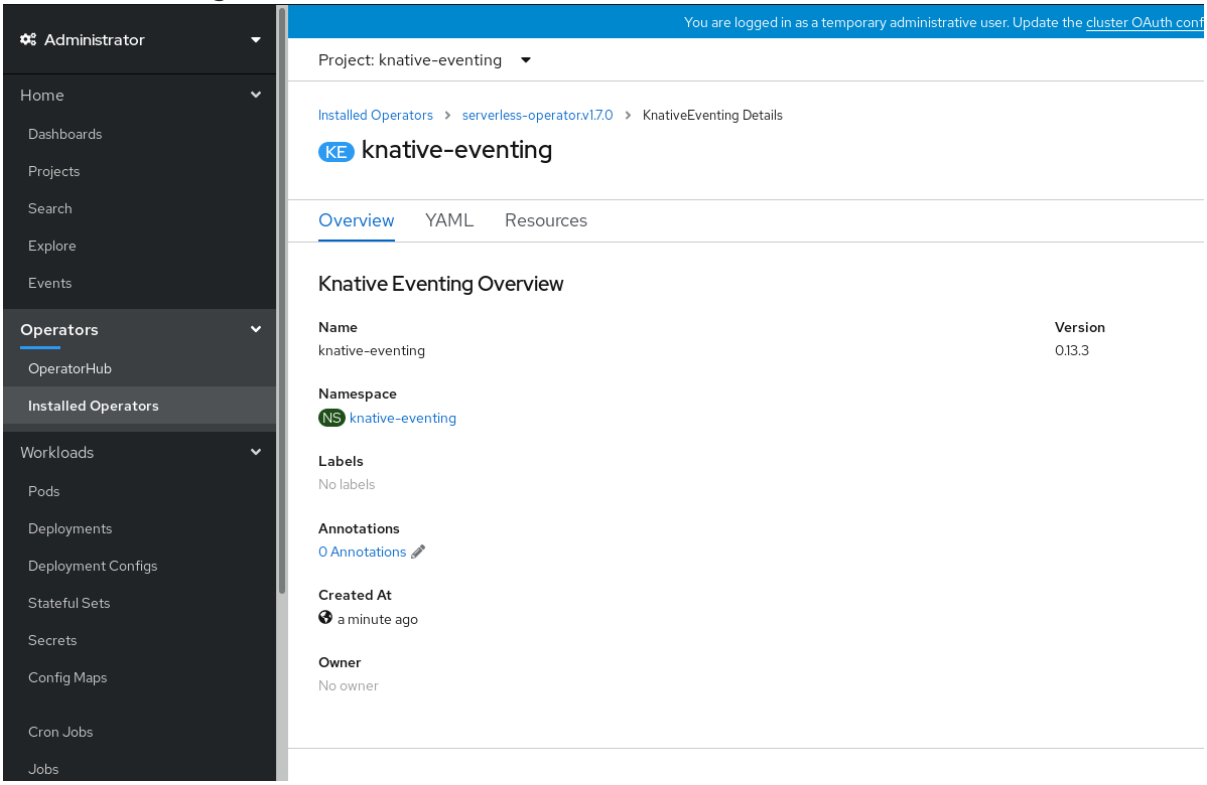
8. Knative Eventing のインストール後に、**KnativeEventing** オブジェクトが作成され、**Knative Eventing** タブに自動的にダイレクトされます。



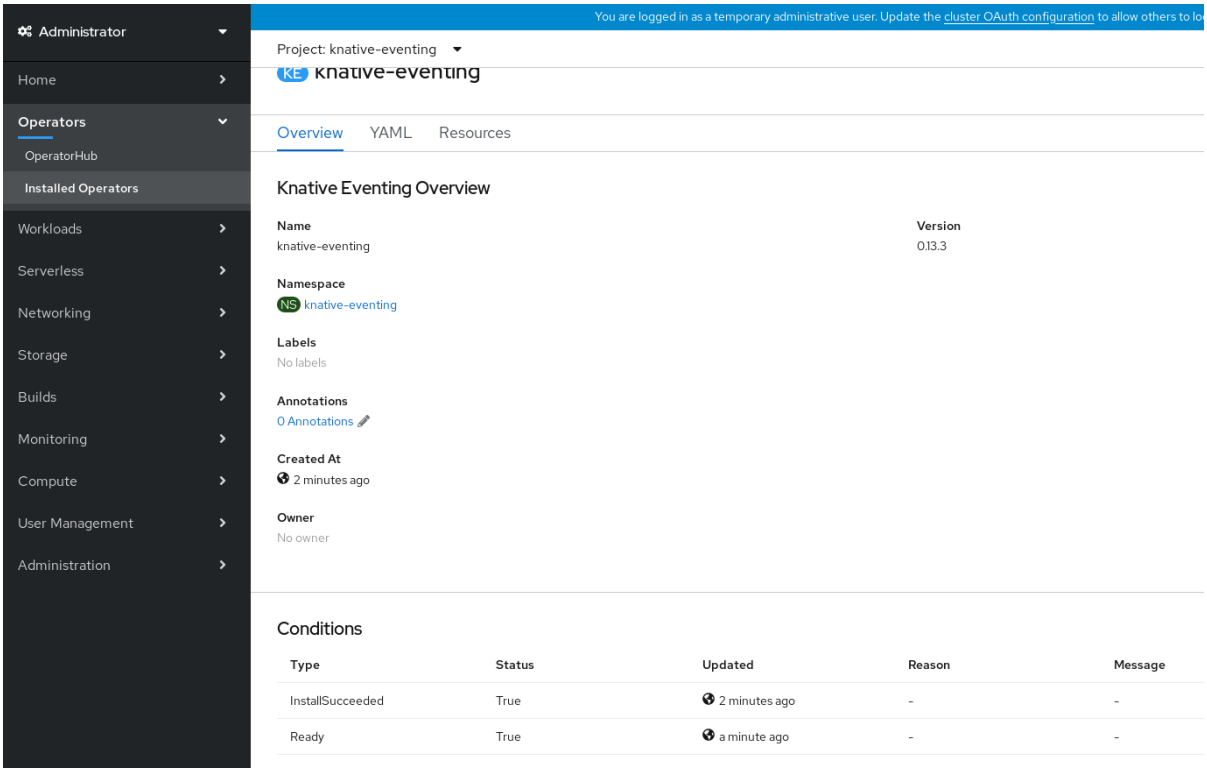
リソースの一覧に **knative-eventing** が表示されます。

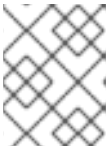
検証

1. Knative Eventing タブの **knative-eventing** をクリックします。
2. **Knative Eventing Overview** ページに自動的にダイレクトされます。



3. スクロールダウンして、**Conditions** の一覧を確認します。
4. ステータスが **True** の条件の一覧が表示されます (例のイメージを参照)。





注記

Knative Eventing リソースが作成されるまでに数秒の時間がかかる場合があります。**Resources** タブでステータスを確認できます。

5. 条件のステータスが **Unknown** または **False** である場合は、しばらく待ってから、リソースが作成されたことを再度確認します。

4.3.4. YAML を使用した Knative Eventing のインストール

手順

1. **eventing.yaml** という名前のファイルを作成します。
2. 以下のサンプル YAML を **eventing.yaml** にコピーします。

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
    name: knative-eventing
    namespace: knative-eventing
```

3. オプション。Knative Eventing デプロイメントについて実装する必要のある変更を YAML に加えます。
4. 以下を入力して **eventing.yaml** ファイルを適用します。

```
$ oc apply -f eventing.yaml
```

検証

1. インストールが完了したことを確認するには、以下を入力します。

```
$ oc get knativeeventing.operator.knative.dev/knative-eventing \
  -n knative-eventing \
  --template='{{range .status.conditions}}{{printf "%s=%s\n" .type .status}}{{end}}'
```

出力は以下のようになります。

```
InstallSucceeded=True
Ready=True
```

注記

Knative Eventing リソースが作成されるまでに数秒の時間がかかる場合があります。

2. 条件のステータスが **Unknown** または **False** である場合は、しばらく待ってから、リソースが作成されたことを再度確認します。

3. 以下のコマンドを実行して Knative Eventing リソースが作成されていることを確認します。

```
$ oc get pods -n knative-eventing
```

出力は以下のようになります。

```
NAME                                  READY   STATUS    RESTARTS   AGE
broker-controller-58765d9d49-g9zp6    1/1     Running   0          7m21s
eventing-controller-65fdd66b54-jw7bh  1/1     Running   0          7m31s
eventing-webhook-57fd74b5bd-kvhlz     1/1     Running   0          7m31s
imc-controller-5b75d458fc-ptvm2       1/1     Running   0          7m19s
imc-dispatcher-64f6d5fccb-kkc4c       1/1     Running   0          7m18s
```

4.3.5. 次のステップ

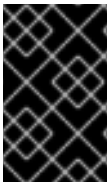
- Knative CLI をインストールして、Knative Eventing で **kn** コマンドを使用します。例: **kn source** コマンドKnative CLI (**kn**)のインストール についてのドキュメントを参照してください。

4.4. 高度なインストール設定オプション

以下では、OpenShift Serverless コンポーネントの高度なインストール設定オプションについてのクラスター管理者向けの情報を提供します。

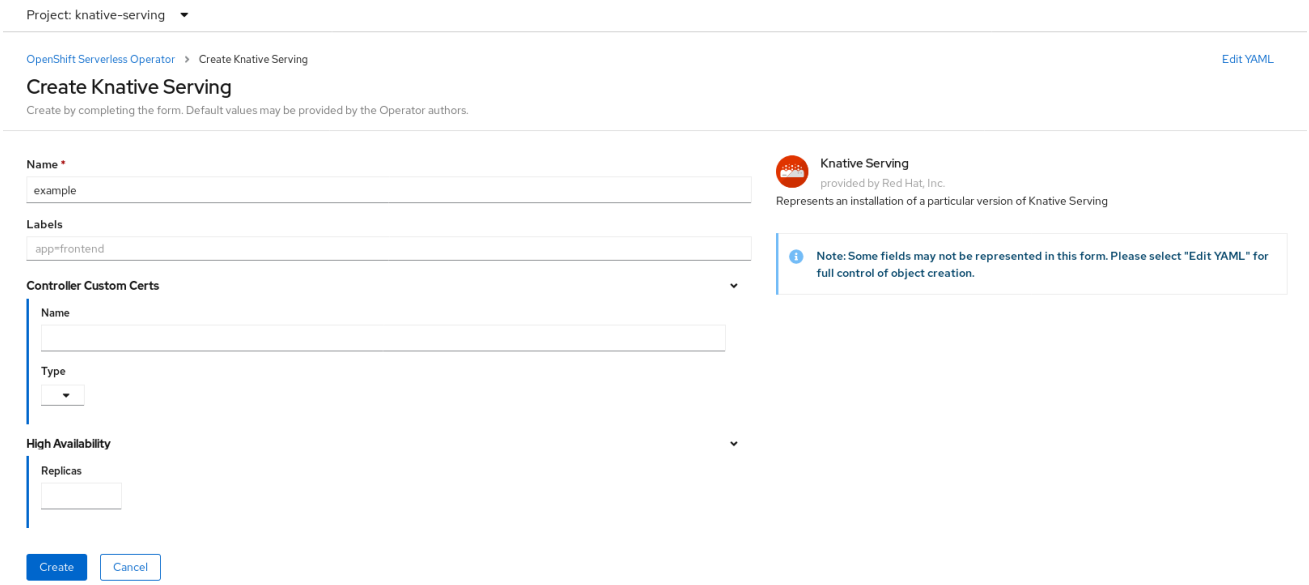
4.4.1. Knative Serving でサポートされるインストール設定オプション

以下では、Knative Serving の高度なインストール設定オプションについてのクラスター管理者向けの情報を提供します。



重要

config フィールドに含まれる YAML は変更しないでください。このフィールドの設定値の一部は OpenShift Serverless Operator によって挿入され、これらを変更すると、デプロイメントはサポートされなくなります。



4.4.1.1. コントローラーのカスタム証明書

レジストリーが自己署名証明書を使用する場合、設定マップまたはシークレットを作成して、tag-to-digest の解決策を有効にする必要があります。次に、OpenShift Serverless Operator は Knative Serving コントローラーをレジストリーにアクセスできるように自動的に設定します。

tag-to-digest の解決策を有効にするには、Knative Serving コントローラーがコンテナーレジストリーにアクセスする必要があります。

重要

設定マップまたはシークレットは Knative Serving カスタムリソース定義 (CRD) と同じ namespace になければなりません。

次の例では、以下を実行するために OpenShift Serverless Operator をトリガーします。

1. コントローラーに証明書を含むボリュームを作成してマウントします。
2. 必要な環境変数を適切に設定します。

サンプル YAML

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  controller-custom-certs:
    name: certs
    type: ConfigMap
```

以下の例では、**knative-serving** namespace の **certs** という名前の設定マップの証明書を使用します。

サポートされるタイプは **ConfigMap** および **Secret** です。

コントローラーカスタム証明書が指定されていない場合、デフォルトは **config-service-ca** 設定マップに設定されます。

デフォルト YAML の例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  controller-custom-certs:
    name: config-service-ca
    type: ConfigMap
```

4.4.1.2. 高可用性

レプリカの数が指定されていない場合、高可用性 (HA) はデフォルトでコントローラーあたり **2** つのレプリカに設定されます。

これを **1** に設定すると HA を無効にするか、またはより高い整数を設定してレプリカを追加できます。

サンプル YAML

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  high-availability:
    replicas: 2
```

4.4.2. 追加リソース

- 高可用性の設定に関する詳細は、OpenShift Serverless での高可用性 を参照してください。

4.5. OPENSHIFT SERVERLESS のアップグレード

以前のバージョンの OpenShift Serverless をインストールしている場合には、本ガイドの手順に従って最新バージョンにアップグレードしてください。

重要

最新の Serverless リリースにアップグレードする前に、事前にインストールしている場合はコミュニティー Knative Eventing Operator を削除する必要があります。Knative Eventing Operator をインストールすると、Knative Eventing の最新のテクノロジープレビューバージョンをインストールできなくなります。

4.5.1. サブスクリプションチャネルのアップグレード

前提条件

- 以前のバージョンの OpenShift Serverless Operator をインストールし、インストールプロセス時に自動更新を選択している。

注記

手動更新を選択した場合は、本書で説明するようにチャネルの更新後に追加の手順を実行する必要があります。Subscription のアップグレードステータスは、その Install Plan を確認し、承認するまで **Upgrading** のままになります。Install Plan についての詳細は、OpenShift Container Platform Operator のドキュメントを参照してください。

- OpenShift Container Platform Web コンソールにログインしている。

手順

1. OpenShift Container Platform Web コンソールで **openshift-operators** namespace を選択します。
2. **Operators → Installed Operators** ページに移動します。
3. **OpenShift Serverless Operator** を選択します。
4. **Subscription → Channel** をクリックします。
5. **Change Subscription Update Channel** ウィンドウで **4.5** を選択し、**Save** をクリックします。
6. すべての Pod が **knative-serving** namespace でアップグレードされ、**KnativeServing** カスタムリソース (CR) が最新の Knative Serving バージョンを報告するまで待機します。

検証

アップグレードが成功したことを確認するには、**knative-serving** namespace の Pod のステータスと **KnativeServing** CR のバージョンを確認します。

1. Pod のステータスを確認します。

```
$ oc get knativeserving.operator.knative.dev knative-serving -n knative-serving -
o=jsonpath='{.status.conditions[?(@.type=="Ready")].status}'
```

上記のコマンドは **True** のステータスを返すはずです。

2. **KnativeServing** CR のバージョンを確認します。

```
$ oc get knativeserving.operator.knative.dev knative-serving -n knative-serving -
o=jsonpath='{.status.version}'
```

直前のコマンドは、Knative Serving の最新バージョンを返すはずです。OpenShift Serverless Operator リリースノートで最新バージョンを確認できます。

4.6. OPENSHIFT SERVERLESS の削除

本書では、OpenShift Serverless Operator および他の OpenShift Serverless コンポーネントを削除する方法を説明します。

注記

OpenShift Serverless Operator を削除する前に、Knative Serving および Knative Eventing を削除する必要があります。

4.6.1. Knative Serving のアンインストール

Knative Serving をアンインストールするには、そのカスタムリソースを削除してから **knative-serving** namespace を削除する必要があります。

手順

1. Knative Serving を削除するには、以下のコマンドを実行します。

```
$ oc delete knativeservings.operator.knative.dev knative-serving -n knative-serving
```

2. コマンドが実行され、すべての Pod が **knative-serving** namespace から削除された後に、以下のコマンドを使用して namespace を削除します。

```
$ oc delete namespace knative-serving
```

4.6.2. Knative Eventing のアンインストール

Knative Eventing をアンインストールするには、そのカスタムリソースを削除してから **knative-eventing** namespace を削除する必要があります。

手順

1. Knative Eventing を削除するには、以下のコマンドを実行します。

```
$ oc delete knativeeventings.operator.knative.dev knative-eventing -n knative-eventing
```

2. コマンドが実行され、すべての Pod が **knative-eventing** namespace から削除された後に、以下のコマンドを入力して namespace を削除します。

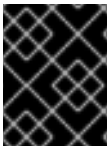
```
$ oc delete namespace knative-eventing
```

4.6.3. OpenShift Serverless Operator の削除

Operator のクラスターからの削除方法 についての OpenShift Container Platform の説明に従って、OpenShift Serverless Operator をホストクラスターから削除できます。

4.6.4. OpenShift Serverless CRD の削除

OpenShift Serverless のアンインストール後に、Operator および API カスタムリソース定義 (CRD) はクラスター上に残ります。以下の手順を使用して、残りの CRD を削除できます。



重要

Operator および API CRD を削除すると、Knative サービスを含む、それらを使用して定義されたすべてのリソースも削除されます。

前提条件

- Knative Serving および Knative Eventing をアンインストールし、OpenShift Serverless Operator を削除していること。

手順

- OpenShift Serverless CRD を削除します。

```
$ oc get crd -oname | grep 'knative.dev' | xargs oc delete
```

4.7. KNATIVE CLI (KN) のインストール

注記

kn には、独自のログインメカニズムは含まれません。クラスターにログインするには、**oc** CLI をインストールし、**oc** ログインを使用する必要があります。

oc CLI のインストールオプションは、お使いのオペレーティングシステムによって異なります。

ご使用のオペレーティングシステム用に **oc** CLI をインストールする方法および **oc** でのログイン方法についての詳細は、CLI の使用開始 についてのドキュメントを参照してください。

4.7.1. OpenShift Container Platform Web コンソールを使用した kn CLI のインストール

OpenShift Serverless Operator がインストールされると、OpenShift Container Platform Web コンソールの **Command Line Tools** ページから Linux、macOS および Windows の **kn** CLI をダウンロードするためのリンクが表示されます。

Command Line Tools ページには、Web コンソールの右上の  アイコンをクリックして、ドロップダウンメニューの **Command Line Tools** を選択します。

手順

1. **Command Line Tools** ページから **kn** CLI をダウンロードします。
2. アーカイブを展開します。

```
$ tar -xf <file>
```

3. **kn** バイナリーをパスにあるディレクトリーに移動します。
4. PATH を確認するには、以下を実行します。

```
$ echo $PATH
```

注記

RHEL または Fedora を使用しない場合は、**libc** がライブラリーパスのディレクトリーにインストールされていることを確認してください。**libc** が利用できない場合は、CLI コマンドの実行時に以下のエラーが表示される場合があります。

```
$ kn: No such file or directory
```

4.7.2. RPM を使用した Linux 用の kn CLI のインストール

Red Hat Enterprise Linux (RHEL) の場合、Red Hat アカウントに有効な OpenShift Container Platform サブスクリプションがある場合は、**kn** を RPM としてインストールできます。

手順

- 以下のコマンドを使用して、**kn** をインストールします。

```
# subscription-manager register
# subscription-manager refresh
# subscription-manager attach --pool=<pool_id> 1
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-x86_64-rpms"
# yum install openshift-serverless-clients
```

1 有効な OpenShift Container Platform サブスクリプションのプール ID

4.7.3. Linux の kn CLI のインストール

Linux ディストリビューションの場合、CLI を **tar.gz** アーカイブとして直接ダウンロードできます。

手順

1. CLI をダウンロードします。
2. アーカイブを展開します。

```
$ tar -xf <file>
```

3. **kn** バイナリーをパスにあるディレクトリーに移動します。
4. PATH を確認するには、以下を実行します。

```
$ echo $PATH
```

注記

RHEL または Fedora を使用しない場合は、**libc** がライブラリーパスのディレクトリーにインストールされていることを確認してください。**libc** が利用できない場合は、CLI コマンドの実行時に以下のエラーが表示される場合があります。

```
$ kn: No such file or directory
```

4.7.4. macOS の kn CLI のインストール

macOS の **kn** は、**tar.gz** アーカイブとして提供されます。

手順

1. CLI をダウンロードします。
2. アーカイブを展開および解凍します。
3. **kn** バイナリーをパスにあるディレクトリーに移動します。
4. パスを確認するには、ターミナルウィンドウを開き、以下を実行します。

```
$ echo $PATH
```

4.7.5. Windows の kn CLI のインストール

Windows の CLI は zip アーカイブとして提供されます。

手順

1. CLI をダウンロードします。
2. ZIP プログラムでアーカイブを解凍します。
3. **kn** バイナリーをパスにあるディレクトリーに移動します。
4. パスを確認するには、コマンドプロンプトを開いて以下のコマンドを実行します。

```
C:\> path
```

第5章 アーキテクチャー

5.1. KNATIVE SERVING アーキテクチャー

OpenShift Container Platform 上の Knative Serving により、開発者は サーバーレスアーキテクチャー を使用して、クラウドネイティブアプリケーション を作成できます。Serverless は、アプリケーション開発者がサーバーのプロビジョニングやアプリケーションのスケーリングを管理する必要がないクラウドコンピューティングのモデルです。これらのルーチンタスクはプラットフォームによって抽象化されるため、開発者は従来のモデルの場合よりも速くコードを実稼働にプッシュできます。

Knative Serving は、OpenShift Container Platform クラスター上のサーバーレスワークロードの動作を定義し、制御する Kubernetes カスタムリソース定義 (CRD) としてオブジェクトのセットを提供し、クラウドネイティブアプリケーションのデプロイおよび管理をサポートします。CRD についての詳細は、カスタムリソース定義による Kubernetes API の拡張 を参照してください。

開発者はこれらの CRD を使用して、複雑なユースケースに対応するためにビルディングブロックとして使用できるカスタムリソース (CR) インスタンスを作成します。以下に例を示します。

- サーバーレスコンテナーの迅速なデプロイ
- Pod の自動スケーリング

CR についての詳細は、カスタムリソース定義からのリソースの管理 を参照してください。

5.1.1. Knative Serving CRD

Service

service.serving.knative.dev CRD はワークロードのライフサイクルを自動的に管理し、アプリケーションがネットワーク経由でデプロイされ、到達可能であることを確認します。これは、ユーザーが作成したサービスまたは CR に対して加えられるそれぞれの変更についての ルート、設定、および新規リビジョンを作成します。Knative での開発者の対話のほとんどは、サービスを変更して実行されます。

Revision

revision.serving.knative.dev CRD は、ワークロードに対して加えられるそれぞれの変更についてのコードおよび設定の特定の時点におけるスナップショットです。Revision (リビジョン) はイミュータブル (変更不可) オブジェクトであり、必要な期間保持することができます。

Route

route.serving.knative.dev CRD は、ネットワークのエンドポイントを、1 つ以上のリビジョンにマップします。部分的なトラフィックや名前付きルートなどのトラフィックを複数の方法で管理することができます。

Configuration

configuration.serving.knative.dev CRD は、デプロイメントの必要な状態を維持します。これにより、コードと設定を明確に分離できます。設定を変更すると、新規リビジョンが作成されます。

5.2. KNATIVE EVENTING アーキテクチャー

OpenShift Container Platform 上の Knative Eventing を使用すると、開発者はサーバーレスアプリケーションと共に イベント駆動型のアーキテクチャー を使用できます。イベント駆動型のアーキテクチャーは、イベントを作成するイベントプロデューサーと、イベントを受信するイベント **シンク** またはコンシューマーとの間の切り離された関係の概念に基づいています。

Knative Eventing は、標準の HTTP POST リクエストを使用してイベントプロデューサーとコンシューマー間でイベントを送受信します。これらのイベントは CloudEvents 仕様 に準拠しており、すべてのプログラミング言語でのイベントの作成、解析、および送受信を可能にします。

以下を使用して、イベントを イベントソース から複数のイベントシンクに伝播できます。

- Channel および Subscription または
- Broker および Trigger

イベントは、宛先のシンクが利用できない場合にバッファーされます。

Knative Eventing は以下のシナリオをサポートします。

コンシューマーを作成せずにイベントを公開する

イベントを HTTP POST として Broker に送信し、シンクバインディングを使用してイベントを生成するアプリケーションから宛先設定を分離できます。

パブリッシャーを作成せずにイベントを消費

Trigger を使用して、イベント属性に基づいて Broker からイベントを消費できます。アプリケーションはイベントを HTTP POST として受信します。

5.2.1. イベントシンク

複数のタイプのシンクへの配信を有効にするために、Knative Eventing は複数の Kubernetes リソースで実装できる以下の汎用インターフェイスを定義します。

アドレス指定可能なオブジェクト

HTTP 経由で **status.address.url** フィールドに定義されるアドレスに配信されるイベントを受信し、確認することができます。Kubernetes Service オブジェクトはアドレス指定可能なインターフェイスにも対応します。

呼び出し可能なオブジェクト

HTTP 経由で配信されるイベントを受信し、これを変換できます。HTTP 応答ペイロードで 0 または 1 の新規イベントを返します。返されるイベントは、外部イベントソースからのイベントが処理されるのと同じ方法で処理できます。

第6章 サーバーレスアプリケーションの作成および管理

6.1. KNATIVE サービスを使用した SERVERLESS アプリケーション

OpenShift Serverless でサーバーレスアプリケーションをデプロイするには、**Knative サービス** を作成する必要があります。Knative サービスは、ルートおよび YAML ファイルに含まれる設定によって定義される Kubernetes サービスです。

Knative サービス YAML の例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello 1
  namespace: default 2
spec:
  template:
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift 3
          env:
            - name: RESPONSE 4
              value: "Hello Serverless!"
```

1. アプリケーションの名前
2. アプリケーションが使用する namespace
3. アプリケーションのイメージ
4. サンプルアプリケーションで出力される環境変数

以下の方法のいずれかを使用してサーバーレスアプリケーションを作成できます。

- OpenShift Container Platform Web コンソールからの Knative サービスの作成
- **kn** CLI を使用して Knative サービスを作成します。
- YAML ファイルを作成し、これを適用します。

6.2. OPENSHIFT CONTAINER PLATFORM WEB コンソールでのサーバーレスアプリケーションの作成

OpenShift Container Platform Web コンソールの **Developer** または **Administrator** パースペクティブのいずれかを使用してサーバーレスアプリケーションを作成できます。

6.2.1. Administrator パースペクティブを使用したサーバーレスアプリケーションの作成

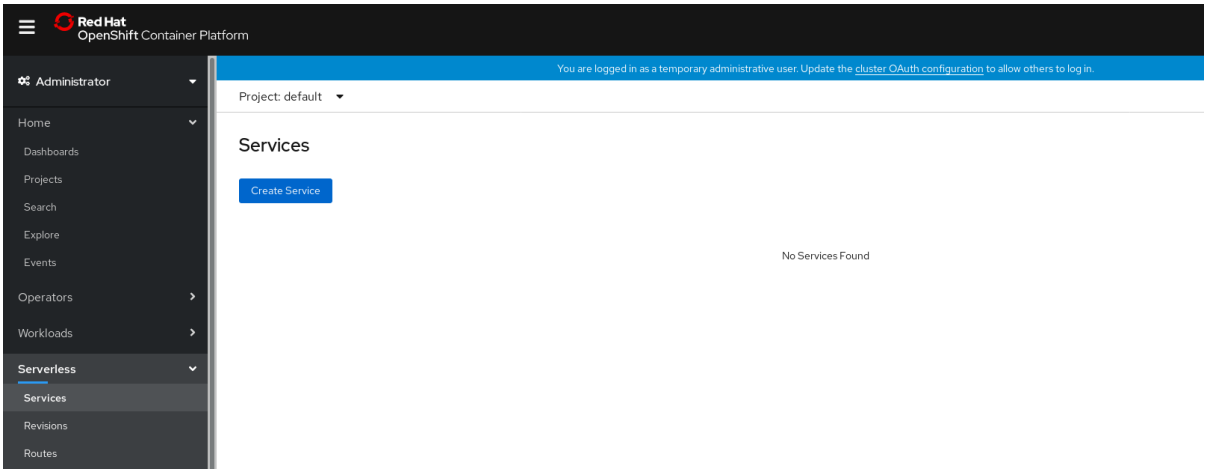
前提条件

Administrator パースペクティブを使用してサーバーレスアプリケーションを作成するには、以下の手順を完了していることを確認してください。

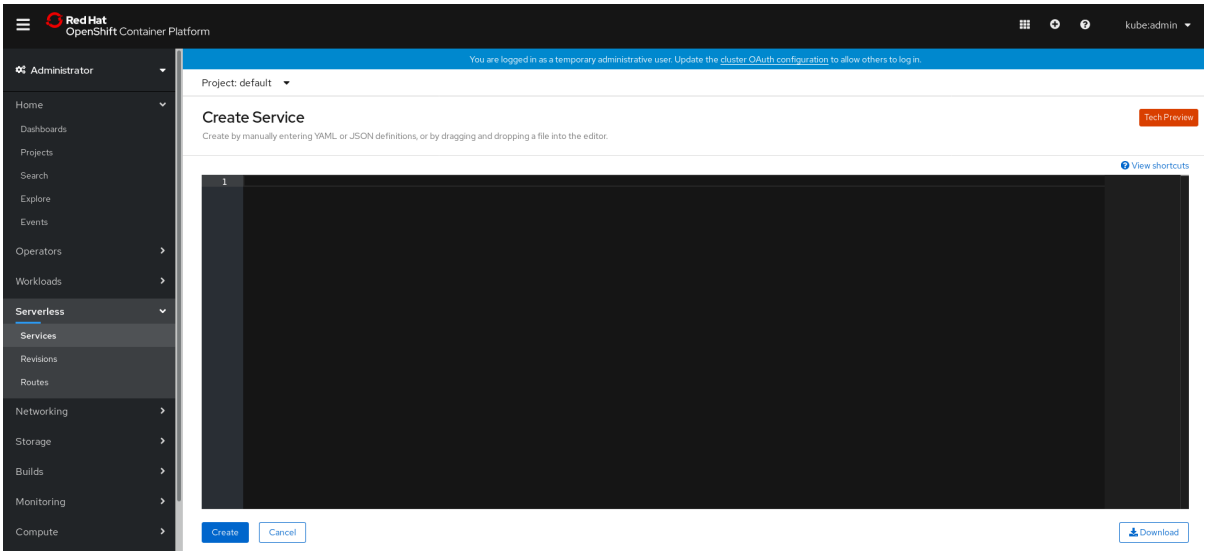
- OpenShift Serverless Operator および Knative Serving がインストールされていること。
- Web コンソールにログインしており、**Administrator** パースペクティブを使用している。

手順

1. **Serverless → Services** ページに移動します。



2. **Create Service** をクリックします。
3. YAML または JSON 定義を手動で入力するか、またはファイルをエディターにドラッグし、ドロップします。



4. **Create** をクリックします。

6.2.2. Developer パースペクティブを使用したサーバーレスアプリケーションの作成

OpenShift Container Platform で **Developer** パースペクティブを使用してアプリケーションを作成する方法についての詳細は、Developer パースペクティブを使用したアプリケーションの作成 ドキュメントを参照してください。

6.3. KN CLI を使用したサーバーレスアプリケーションの作成

以下の手順では、**kn** CLI を使用して基本的なサーバーレスアプリケーションを作成する方法を説明します。

前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされていること。
- **kn** CLI がインストールされていること。

手順

1. 以下のコマンドを入力して Knative サービスを作成します。

```
$ kn service create <service_name> --image <image> --env <key=value>
```

コマンドの例

```
$ kn service create hello --image docker.io/openshift/hello-openshift --env
RESPONSE="Hello Serverless!"
```

出力例

```
Creating service 'hello' in namespace 'default':

  0.271s The Route is still working to reflect the latest desired specification.
  0.580s Configuration "hello" is waiting for a Revision to become ready.
  3.857s ...
  3.861s Ingress has not yet been reconciled.
  4.270s Ready to serve.

Service 'hello' created with latest revision 'hello-bxshg-1' and URL:
http://hello-default.apps-crc.testing
```

6.4. YAML を使用したサーバーレスアプリケーションの作成

YAML を使用してサーバーレスアプリケーションを作成するには、サービスを定義する YAML ファイルを作成し、**oc apply** を使用してこれを適用する必要があります。

手順

1. YAML ファイルを作成してから、以下のサンプルをファイルにコピーします。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift
```

```
          env:
            - name: RESPONSE
              value: "Hello Serverless!"
```

2. YAML ファイルが含まれるディレクトリーに移動し、YAML ファイルを適用してアプリケーションをデプロイします。

```
$ oc apply -f <filename>
```

サービスが作成され、アプリケーションがデプロイされると、Knative はこのバージョンのアプリケーションのイミュータブルなリビジョンを作成します。

また、Knative はネットワークプログラミングを実行し、アプリケーションのルート、ingress、サービスおよびロードバランサーを作成し、アクティブでない Pod を含む Pod をトラフィックに基づいて自動的にスケールアップ/ダウンします。

6.5. サーバーレスアプリケーションのデプロイメントの確認

サーバーレスアプリケーションが正常にデプロイされたことを確認するには、Knative によって作成されたアプリケーション URL を取得してから、その URL に要求を送信し、出力を確認する必要があります。

注記

OpenShift Serverless は HTTP および HTTPS URL の両方の使用をサポートしますが、**oc get ksvc <service_name>** からの出力は常に **http://** 形式を使用して URL を出力します。

手順

1. 以下を入力してアプリケーション URL を検索します。

```
$ oc get ksvc <service_name>
```

出力例

```
NAME          URL                                   LATESTCREATED        LATESTREADY
READY   REASON
hello   http://hello-default.example.com   hello-4wsd2   hello-4wsd2   True
```

2. クラスターに対して要求を実行し、出力を確認します。

HTTP 要求の例

```
$ curl http://hello-default.example.com
```

HTTPS 要求の例

```
$ curl https://hello-default.example.com
```

出力例

```
Hello Serverless!
```

3. オプション。証明書チェーンで自己署名証明書に関連するエラーが発生した場合は、curl コマンドに **--insecure** フラグを追加して、エラーを無視できます。

重要

自己署名証明書は、実稼働デプロイメントでは使用しないでください。この方法は、テスト目的にのみ使用されます。

コマンドの例

```
$ curl https://hello-default.example.com --insecure
```

出力例

```
Hello Serverless!
```

4. オプション。OpenShift Container Platform クラスターが認証局 (CA) で署名されているが、システムにグローバルに設定されていない証明書で設定されている場合、curl コマンドでこれを指定できます。証明書へのパスは、**--cacert** フラグを使用して curl コマンドに渡すことができます。

コマンドの例

```
$ curl https://hello-default.example.com --cacert <file>
```

出力例

```
Hello Serverless!
```

6.6. HTTP2 および GRPC を使用したサーバーレスアプリケーションとの対話

OpenShift Serverless はセキュアでないルートまたは edge termination ルートのみをサポートします。

非セキュアなルートまたは edge termination ルートは OpenShift Container Platform で HTTP2 をサポートしません。gRPC は HTTP2 によって転送されるため、これらのルートは gRPC もサポートしません。

アプリケーションでこれらのプロトコルを使用する場合は、Ingress ゲートウェイを使用してアプリケーションを直接呼び出す必要があります。これを実行するには、Ingress ゲートウェイのパブリックアドレスとアプリケーションの特定のホストを見つける必要があります。

手順

1. アプリケーションホストを検索します。**サーバーレスアプリケーションのデプロイメントの確認**の説明を参照してください。

2. Ingress ゲートウェイのパブリックアドレスを見つけます。

```
$ oc -n knative-serving-ingress get svc kourier
```

出力例:

```
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)
AGE
kourier   LoadBalancer   172.30.51.103   a83e86291bcdd11e993af02b7a65e514-
33544245.us-east-1.elb.amazonaws.com   80:31380/TCP,443:31390/TCP   67m
```

パブリックアドレスは **EXTERNAL-IP** フィールドに表示されます。この場合、**a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com** になります。

3. HTTP 要求のホストヘッダーを手動でアプリケーションのホストに手動で設定しますが、Ingress ゲートウェイのパブリックアドレスに対して要求自体をダイレクトします。
以下は、**サーバーレスアプリケーションのデプロイメントの確認**の手順で記載された情報を使用した例です。

コマンドの例

```
$ curl -H "Host: hello-default.example.com" a83e86291bcdd11e993af02b7a65e514-
33544245.us-east-1.elb.amazonaws.com
```

出力例

```
Hello Serverless!
```

Ingress ゲートウェイに対して要求を直接ダイレクトする間に、権限をアプリケーションのホストに設定して gRPC 要求を行うこともできます。

```
grpc.Dial(
    "a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com:80",
    grpc.WithAuthority("hello-default.example.com:80"),
    grpc.WithInsecure(),
)
```

注記

直前の例のように、それぞれのポート (デフォルトでは 80) を両方のホストに追加します。

第7章 OPENSHIFT SERVERLESS での高可用性

高可用性 (HA) は Kubernetes API の標準的な機能で、中断が生じる場合に API が稼働を継続するのに役立ちます。HA デプロイメントでは、アクティブなコントローラーがクラッシュするか、または削除されると、現在利用できないコントローラーが提供されている API の処理を引き継ぐために別のコントローラーが利用可能になります。

OpenShift Serverless の HA は、リーダーの選択によって利用できます。これは、Knative Serving コントロールプレーンのインストール後にデフォルトで有効になります。

リーダー選択の HA パターンを使用する場合、必要時に備えてコントローラーのインスタンスはスケジュールされ、クラスター内で実行されます。これらのコントローラーインスタンスは、共有リソースの使用に向けて競います。これは、リーダー選択ロックとして知られています。リーダー選択ロックのリソースにアクセスできるコントローラーのインスタンスはリーダーと呼ばれます。

7.1. OPENSHIFT SERVERLESS での高可用性レプリカの設定

高可用性 (HA) 機能は、**autoscaler-hpa**、**controller**、**activator**、**kourier-control**、および **kourier-gateway** コンポーネントについてデフォルトで OpenShift Serverless で利用できます。これらのコンポーネントは、デフォルトで 2 つのレプリカで設定されます。

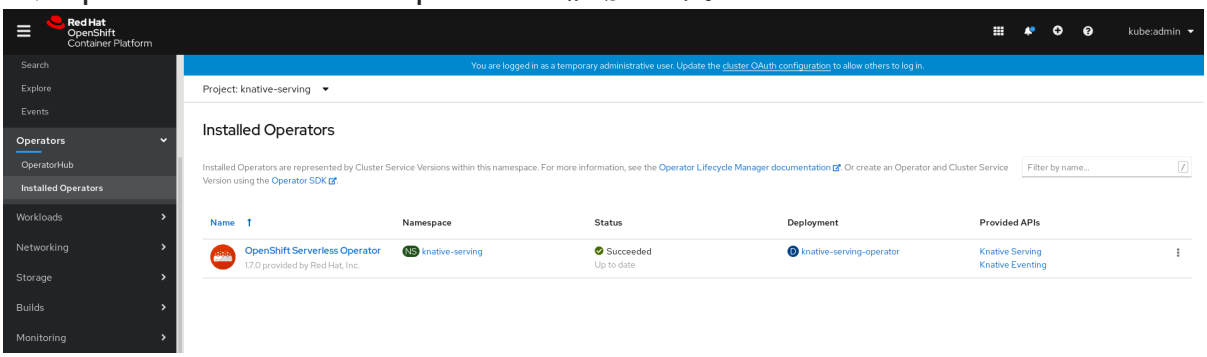
KnativeServing カスタムリソース定義 (CRD) の **KnativeServing.spec.highAvailability** 仕様の設定を変更して、コントローラーごとに作成されるレプリカの数を変更します。

前提条件

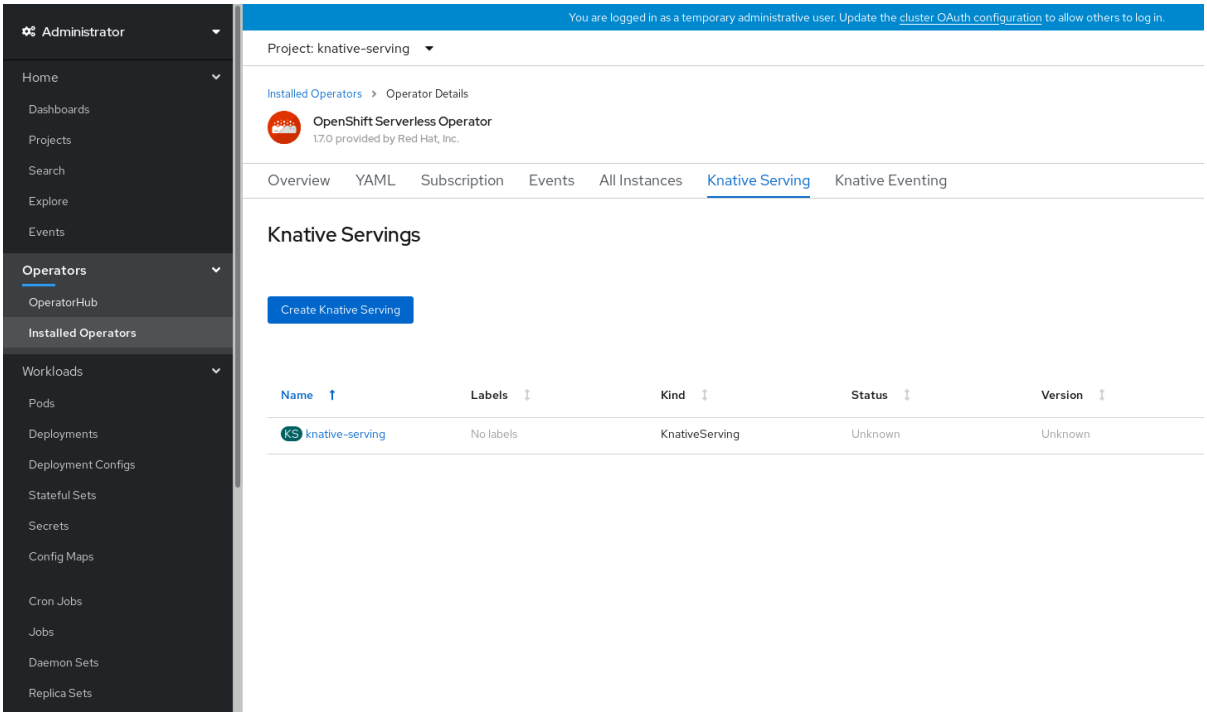
- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウント。
- OpenShift Serverless Operator および Knative Serving がインストールされていること。

手順

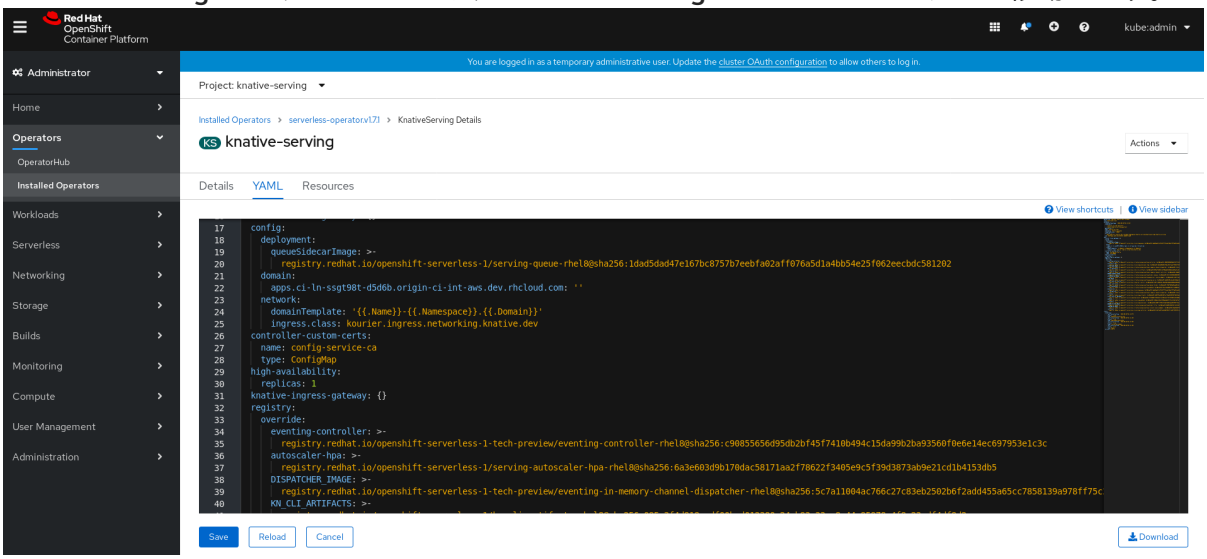
1. OpenShift Container Platform Web コンソールの **Administrator** パースペクティブで、**OperatorHub → Installed Operators** に移動します。



2. **knative-serving** namespace を選択します。
3. OpenShift Serverless Operator の **Provided API** 一覧で **Knative Serving** をクリックし、**Knative Serving** タブに移動します。



4. **knative-serving** をクリックしてから、**knative-serving** ページの **YAML** タブに移動します。



5. カスタムリソース定義 YAML を編集します。

サンプル YAML

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  high-availability:
    replicas: 3
```

重要

config フィールドに含まれる YAML は変更しないでください。このフィールドの設定値の一部は OpenShift Serverless Operator によって挿入され、これらを変更すると、デプロイメントはサポートされなくなります。

- デフォルトの **replicas** 値は **2** です。
- 値を **1** に設定すると HA が無効になります。または、必要に応じてレプリカの数を増やすことができます。上記の設定例は、すべての HA コントローラーのレプリカ数 **3** を指定します。

第8章 JAEGER を使用した要求のトレース

Jaeger を OpenShift Serverless で使用すると、OpenShift Container Platform でのサーバーレスアプリケーションの **分散トレース** を有効にできます。

分散トレースは、アプリケーションを設定する各種のサービスを使用した要求のパスを記録します。

これは、各種の異なる作業単位についての情報を連携させ、分散トランザクションでのイベントチェーン全体を把握できるようにするために使用されます。作業単位は、異なるプロセスまたはホストで実行される場合があります。

開発者は分散トレースを使用し、大規模なアーキテクチャーで呼び出しフローを可視化できます。これは、シリアル化、並行処理、およびレイテンシーのソースについての理解に役立ちます。

Jaeger についての詳細は、Jaeger アーキテクチャー および Jaeger のインストール を参照してください。

8.1. OPENSHIFT SERVERLESS で使用する JAEGER の設定

前提条件

OpenShift Serverless で使用する Jaeger を設定するには、以下が必要です。

- OpenShift Container Platform クラスターでのクラスター管理者パーミッション。
- OpenShift Serverless Operator および Knative Serving がインストールされていること。
- Jaeger Operator をインストールしていること。

手順

1. 以下のサンプル YAML を含む Jaeger カスタムリソース YAML ファイルを作成し、これを適用します。

   Jaeger カスタムリソース YAML

   ```
   apiVersion: jaegertracing.io/v1
   kind: Jaeger
   metadata:
     name: jaeger
     namespace: default
   ```

2. **KnativeServing** リソースを編集し、トレース用に YAML 設定を追加して、Knative Serving のトレースを有効にします。

   トレース用の YAML の例

   ```
   apiVersion: operator.knative.dev/v1alpha1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     config:
       tracing:
   ```

```
    sample-rate: "0.1" 1
    backend: zipkin 2
    zipkin-endpoint: http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans 3
    debug: "false" 4
```

1. **sample-rate** はサンプリングの可能性を定義します。**sample-rate: "0.1"** を使用すると、10 トレースの内の 1 つがサンプリングされます。
2. **backend** は **zipkin** に設定される必要があります。
3. **zipkin-endpoint** は **jaeger-collector** サービスエンドポイントを参照する必要があります。このエンドポイントを取得するには、Jaeger カスタムリソースが適用される namespace を置き換えます。
4. デバッグは **false** に設定する必要があります。**debug: "true"** を設定してデバッグモードを有効にすることで、サンプリングをバイパスしてすべてのスパンがサーバーに送信されるようにします。

検証

Jaeger Web コンソールにアクセスして、トレースデータを表示します。**jaeger** ルートを使用して Jaeger Web コンソールにアクセスできます。

1. **jaeger** ルートのホスト名を取得します。

```
$ oc get route jaeger
```

出力例

```
NAME     HOST/PORT                         PATH   SERVICES      PORT    TERMINATION
WILDCARD
jaeger   jaeger-default.apps.example.com          jaeger-query  <all>   reencrypt     None
```

2. ブラウザーでエンドポイントアドレスを開き、コンソールを表示します。

第9章 KNATIVE SERVING

9.1. KN の使用による SERVING タスクの実行

Knative CLI (**kn**) は **oc** または **kubectl** ツールの機能を拡張し、OpenShift Container Platform の Knative コンポーネントとの対話を可能にします。**kn** を使用すると、開発者は YAML ファイルを直接編集せずにアプリケーションをデプロイし、管理できます。

9.1.1. kn を使用した基本ワークフロー

以下の基本的なワークフローでは、環境変数 **RESPONSE** を読み取る単純な **hello** サービスをデプロイし、その出力を印刷します。

本書は、サービスでの作成、読み取り、更新、削除 (CRUD) 操作を実行する際の参照情報として使用できます。

手順

1. イメージからサービスを **デフォルト** namespace に作成します。

```
$ kn service create hello --image docker.io/openshift/hello-openshift --env
RESPONSE="Hello Serverless!"
```

出力例

```
Creating service 'hello' in namespace 'default':

  0.085s The Route is still working to reflect the latest desired specification.
  0.101s Configuration "hello" is waiting for a Revision to become ready.
 11.590s ...
 11.650s Ingress has not yet been reconciled.
 11.726s Ready to serve.

Service 'hello' created with latest revision 'hello-gsdks-1' and URL:
http://hello-default.apps-crc.testing
```

2. サービスを一覧表示します。

```
$ kn service list
```

出力例

```
NAME    URL                                LATEST         AGE     CONDITIONS   READY
REASON
hello   http://hello-default.apps-crc.testing   hello-gsdks-1   8m35s   3 OK / 3     True
```

3. **curl** サービスエンドポイントコマンドを使用して、サービスが機能しているかどうかを確認します。

```
$ curl http://hello-default.apps-crc.testing
```

出力例

```
Hello Serverless!
```

4. サービスを更新します。

```
$ kn service update hello --env RESPONSE="Hello OpenShift!"
```

出力例

```
Updating Service 'hello' in namespace 'default':

 10.136s Traffic is not yet migrated to the latest revision.
 10.175s Ingress has not yet been reconciled.
 10.348s Ready to serve.

Service 'hello' updated with latest revision 'hello-dghll-2' and URL:
http://hello-default.apps-crc.testing
```

サービスの環境変数 **RESPONSE** は Hello OpenShift! に設定されるようになりました。

5. サービスを記述します。

```
$ kn service describe hello
```

出力例

```
Name:       hello
Namespace:  default
Age:        13m
URL:        http://hello-default.apps-crc.testing

Revisions:
  100%  @latest (hello-dghll-2) [2] (1m)
        Image:  docker.io/openshift/hello-openshift (pinned to 5ea96b)

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 1m
  ++ ConfigurationsReady     1m
  ++ RoutesReady             1m
```

6. サービスを削除します。

```
$ kn service delete hello
```

出力例

```
Service 'hello' successfully deleted in namespace 'default'.
```

7. **hello** サービスに対して **list** を試行し、これが削除されていることを確認しまう。

```
$ kn service list hello
```

出力例

```
No services found.
```

9.1.2. kn を使用した自動スケーリングのワークフロー

YAML ファイルを直接編集せずに **kn** を使用して Knative サービスを変更することで、自動スケーリング機能にアクセスできます。

適切なフラグと共に **service create** および **service update** コマンドを使用して、自動スケーリング動作を設定します。

フラグ	詳細
--concurrency-limit int	単一レプリカによって処理される同時要求のハード制限。
--concurrency-target int	受信する同時要求の数に基づくスケールアップのタイミングの推奨。デフォルトは **--concurrency-limit** に設定されます。
--max-scale int	レプリカの最大数。
--min-scale int	レプリカの最小数。

9.1.3. kn を使用したトラフィック分割

kn は、Knative サービス上でルート指定されたトラフィックを取得するリビジョンを制御するのに役立ちます。

Knative サービスは、トラフィックのマッピングを許可します。これは、サービスのリビジョンのトラフィックの割り当てられた部分へのマッピングです。これは特定のリビジョンに固有の URL を作成するオプションを提供し、トラフィックを最新リビジョンに割り当てる機能を持ちます。

サービスの設定が更新されるたびに、サービスルートがすべてのトラフィックを準備状態にある最新リビジョンにポイントする状態で、新規リビジョンが作成されます。

この動作は、トラフィックの一部を取得するリビジョンを定義して変更することができます。

手順

- **kn service update** コマンドを **--traffic** フラグと共に使用して、トラフィックを更新します。

注記

--traffic RevisionName=Percent は以下の構文を使用します。

- **--traffic** フラグには、等号 (**=**) で区切られた 2 つの値が必要です。
- **RevisionName** 文字列はリビジョンの名前を参照します。
- **Percent** 整数はトラフィックのリビジョンに割り当てられた部分を示します。
- RevisionName の識別子 **@latest** を使用して、サービスの準備状態にある最新のリビジョンを参照します。この識別子は **--traffic** フラグと共に 1 回のみ使用できます。
- **service update** コマンドがトラフィックフラグと共にサービスの設定値を更新する場合、 **@latest** 参照は更新が適用される作成済みリビジョンをポイントします。
- **--traffic** フラグは複数回指定でき、すべてのフラグの **Percent** 値の合計が 100 になる場合にのみ有効です。

注記

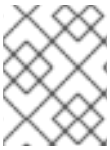
たとえば、すべてのトラフィックを配置する前に 10% のトラフィックを新規リビジョンにルート指定するには、以下のコマンドを使用します。

```
$ kn service update svc --traffic @latest=10 --traffic svc-vwxyz=90
```

9.1.3.1. タグリビジョンの割り当て

サービスのトラフィックブロック内のタグは、参照されるリビジョンをポイントするカスタム URL を作成します。ユーザーは、**http(s)://TAG-SERVICE.DOMAIN** 形式を使用して、カスタム URL を作成するサービスの利用可能なリビジョンの固有タグを定義できます。

指定のタグは、サービスのトラフィックブロックに固有のものである必要があります。**kn** は、**kn service update** コマンドの一環として、サービスのリビジョンのカスタムタグの割り当ておよび割り当て解除に対応します。



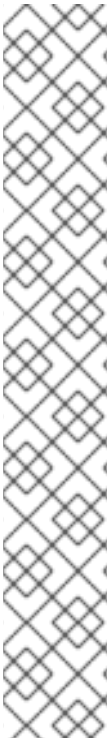
注記

タグを特定のリビジョンに割り当てた場合、ユーザーは、**--traffic** フラグ内で **--traffic Tag=Percent** として示されるタグでこのリビジョンを参照できます。

手順

- 以下のコマンドを使用します。

```
$ kn service update svc --tag @latest=candidate --tag svc-vwxyz=current
```



注記

--tag RevisionName=Tag は以下の構文を使用します。

- **--tag** フラグには、**=** で区切られる 2 つの値が必要です。
- **RevisionName** 文字列は **Revision** の名前を参照します。
- **Tag** 文字列は、このリビジョンに指定されるカスタムタグを示します。
- RevisionName の識別子 **@latest** を使用して、サービスの準備状態にある最新のリビジョンを参照します。この識別子は **--tag** フラグで 1 回のみ使用できます。
- **service update** コマンドがサービスの設定値を (タグフラグと共に) 更新している場合、**@latest** 参照は更新の適用後に作成されるリビジョンをポイントします。
- **--tag** フラグは複数回指定できます。
- **--tag** フラグは、同じリビジョンに複数の異なるタグを割り当てる場合があります。

9.1.3.2. タグリビジョンの割り当て解除

トラフィックブロックのリビジョンに割り当てられたタグは、割り当て解除できます。タグの割り当てを解除すると、カスタム URL が削除されます。

注記

リビジョンのタグが解除され、0% のトラフィックが割り当てられる場合、このリビジョンはトラフィックブロックから完全に削除されます。

手順

- ユーザーは、**kn service update** コマンドを使用してリビジョンのタグの割り当てを解除できます。

```
$ kn service update svc --untag candidate
```

注記

--untag Tag は以下の構文を使用します。

- **--untag** フラグには 1 つの値が必要です。
- **tag** 文字列は、割り当てを解除する必要のあるサービスのトラフィックブロックの固有のタグを示します。これにより、それぞれのカスタム URL も削除されます。
- **--untag** フラグは複数回指定できます。

9.1.3.3. トラフィックフラグ操作の優先順位

すべてのトラフィック関連のフラグは、単一の **kn service update** コマンドを使用して指定できます。**kn** は、これらのフラグの優先順位を定義します。コマンドの使用時に指定されるフラグの順番は考慮に入れられません。

kn で評価されるフラグの優先順位は以下のとおりです。

1. **--untag**: このフラグで参照されるすべてのリビジョンはトラフィックブロックから削除されます。
2. **--tag**: リビジョンはトラフィックブロックで指定されるようにタグ付けされます。
3. **--traffic**: 参照されるリビジョンには、分割されたトラフィックの一部が割り当てられます。

9.1.3.4. トラフィック分割フラグ

kn は **kn service update** コマンドの一環として、サービスのトラフィックブロックでのトラフィック操作に対応します。

以下の表は、トラフィック分割フラグ、値の形式、およびフラグが実行する操作の概要を表示しています。繰り返し列は、フラグの特定の値が **kn service update** コマンドで許可されるかどうかを示します。

フラグ	値	操作	繰り返し
--traffic	**RevisionName=Percent**	**Percent** トラフィックを **RevisionName** に指定します。	はい
--traffic	**Tag=Percent**	**Percent** トラフィックを、**Tag** を持つリビジョンに指定します。	Yes
--traffic	**@latest=Percent**	**Percent** トラフィックを準備状態にある最新のリビジョンに指定します。	No
--tag	**RevisionName=Tag**	**Tag** を **RevisionName** に指定します。	はい
--tag	**@ latest = Tag**	**Tag** を準備状態にある最新リビジョンに指定します。	No
--untag	**Tag**	リビジョンから **Tag** を削除します。	Yes

9.2. KNATIVE SERVING 自動スケーリングの設定

OpenShift Serverless は、Knative Serving 自動スケーリングシステムを OpenShift Container Platform クラスターで有効にすることで、アクティブでない Pod をゼロにスケーリングする機能など、Pod の自動スケーリングの各種機能を提供します。

Knative Serving の自動スケーリングを有効にするには、リビジョンテンプレートで同時実行 (concurrency) およびスケール境界 (scale bound) を設定する必要があります。

注記

リビジョンテンプレートでの制限およびターゲットの設定は、アプリケーションの単一インスタンスに対して行われます。たとえば、**target** アノテーションを **50** に設定することにより、アプリケーションの各インスタンスが一度に 50 要求を処理できるようアプリケーションをスケーリングするように Autoscaler が設定されます。

9.2.1. Knative Serving 自動スケーリングの同時要求の設定

アプリケーションの各インスタンス (リビジョンコンテナー) によって処理される同時要求の数は、リビジョンテンプレートに **target** アノテーションまたは **containerConcurrency** 仕様を追加して指定できます。

リビジョンテンプレートで使用される target アノテーション

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: myapp
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: 50
    spec:
      containers:
      - image: myimage
```

リビジョンテンプレートで使用される containerConcurrency 仕様

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: myapp
spec:
  template:
    metadata:
      annotations:
    spec:
      containerConcurrency: 100
      containers:
      - image: myimage
```

target と **containerConcurrency** の両方の値を追加することにより、同時要求の **target** 数をターゲットとして設定できますが、これにより要求の **containerConcurrency** 数のハード制限も設定されます。

たとえば、**target** 値が 50 で、 **containerConcurrency** 値が 100 の場合、要求のターゲットに設定された数は 50 になりますが、ハード制限は 100 になります。

containerConcurrency 値が **target** 値よりも低い場合、実際に処理できる数よりも多くの要求をターゲットとして設定する必要はないため、**target** 値は小さい値に調整されます。

注記

containerConcurrency は、特定の時点にアプリケーションに到達する要求の数を制限する明らかな必要がある場合にのみ使用する必要があります。**containerConcurrency** は、アプリケーションで同時実行の制約を実行する必要がある場合にのみ使用することを推奨します。

9.2.1.1. ターゲットアノテーションの使用による同時要求の設定

同時要求数のデフォルトターゲットは **100** ですが、リビジョンテンプレートで **autoscaling.knative.dev/target** アノテーション値を追加または変更することによってこの値を上書きできます。

以下は、ターゲットを **50** に設定するためにこのアノテーションをリビジョンテンプレートで使用する方法の例を示しています。

```
autoscaling.knative.dev/target: 50
```

9.2.1.2. containerConcurrency フィールドを使用した同時要求の設定

containerConcurrency は、処理される同時要求数にハード制限を設定します。

```
containerConcurrency: 0 | 1 | 2-N
```

0
: 無制限の同時要求を許可します。

1
: リビジョンコンテナーの所定インスタンスによって一度に処理される要求は 1 つのみであることを保証します。

2 以上
: 同時要求をこの数に制限します。

注記

target アノテーションがない場合、自動スケーリングは、**target** が **containerConcurrency** の値と等しい場合のように設定されます。

9.2.2. Knative Serving 自動スケーリングのスケール境界の設定

minScale および **maxScale** アノテーションは、アプリケーションを提供できる Pod の最小および最大数を設定するために使用できます。これらのアノテーションは、コールドスタートを防いだり、コンピューティングコストをコントロールするために使用できます。

minScale
: **minScale** アノテーションが設定されていない場合、Pod はゼロ (または、enable-scale-to-zero が **ConfigMap** に基づいて false の場合は 1) にスケーリングします。

maxScale
: **maxScale** アノテーションが設定されていない場合、作成される Pod の上限はありません。

minScale および **maxScale** アノテーションは、リビジョンテンプレートで以下のように設定できます。

```
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/minScale: "2"
        autoscaling.knative.dev/maxScale: "10"
```

これらのアノテーションをリビジョンテンプレートで使用することで、この設定が **PodAutoscaler** オブジェクトに伝播します。

注記

これらのアノテーションは、リビジョンの有効期間全体で適用されます。リビジョンがルートで参照されていない場合でも、 **minScale** アノテーションによって指定される最小 Pod 数は依然として指定されます。ルーティングできないリビジョンについては、ガベージコレクションの対象になることに留意してください。これにより、Knative はリソースを回収できます。

9.3. OPENSHIFT SERVERLESS を使用したクラスターロギング

9.3.1. クラスターロギングのデプロイについて

OpenShift Container Platform クラスター管理者は、OpenShift Container Platform Web コンソールまたは CLI コマンドを使用してクラスターロギングをデプロイし、Elasticsearch Operator および Cluster Logging Operator をインストールできます。Operator がインストールされている場合、**ClusterLogging** カスタムリソース (Custom Resource、CR) を作成してクラスターロギング Pod およびクラスターロギングのサポートに必要な他のリソースをスケジュールします。Operator はクラスターロギングのデプロイ、アップグレード、および維持を行います。

ClusterLogging CR は、ログを収集し、保存し、視覚化するために必要なロギングスタックのすべてのコンポーネントを含む完全なクラスターロギング環境を定義します。Cluster Logging Operator は Cluster Logging CR を監視し、ロギングデプロイメントを適宜調整します。

管理者およびアプリケーション開発者は、表示アクセスのあるプロジェクトのログを表示できます。

9.3.2. クラスターロギングのデプロイおよび設定について

OpenShift Container Platform クラスターロギングは、小規模および中規模の OpenShift Container Platform クラスター用に調整されたデフォルト設定で使用されるように設計されています。

以下のインストール方法には、サンプルの **ClusterLogging** カスタムリソース (CR) が含まれます。これを使用して、クラスターロギングインスタンスを作成し、クラスターロギングの環境を設定することができます。

デフォルトのクラスターロギングインストールを使用する必要がある場合は、サンプル CR を直接使用できます。

デプロイメントをカスタマイズする必要がある場合、必要に応じてサンプル CR に変更を加えます。以下では、クラスターロギングのインスタンスをインストール時に実行し、インストール後に変更する設定について説明します。**ClusterLogging** カスタムリソース外で加える変更を含む、各コンポーネントの使用方法については、設定についてのセクションを参照してください。

9.3.2.1. クラスターロギングの設定およびチューニング

クラスターロギング環境は、**openshift-logging** プロジェクトにデプロイされる **ClusterLogging** カスタムリソースを変更することによって設定できます。

インストール時またはインストール後に、以下のコンポーネントのいずれかを変更することができます。

メモリーおよび CPU

resources ブロックを有効なメモリーおよび CPU 値で変更することにより、各コンポーネントの CPU およびメモリーの両方の制限を調整することができます。

```
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu:
          memory: 16Gi
        requests:
          cpu: 500m
          memory: 16Gi
      type: "elasticsearch"
  collection:
    logs:
      fluentd:
        resources:
          limits:
            cpu:
            memory:
          requests:
            cpu:
            memory:
        type: "fluentd"
  visualization:
    kibana:
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu:
          memory:
      type: kibana
  curation:
    curator:
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 200m
          memory: 200Mi
      type: "curator"
```

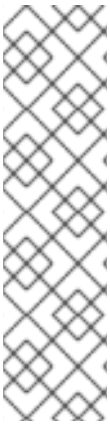
Elasticsearch ストレージ

storageClass name および **size** パラメーターを使用し、Elasticsearch クラスターの永続ストレージのクラスおよびサイズを設定できます。Cluster Logging Operator は、これらのパラメーターに基

づいて、Elasticsearch クラスターの各データノードについて永続ボリューム要求 (PVC) を作成します。

```
spec:
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 3
      storage:
        storageClassName: "gp2"
        size: "200G"
```

この例では、クラスターの各データノードが gp2 ストレージの 200G を要求する PVC にバインドされるように指定します。それぞれのプライマリーシャードは単一のレプリカによってサポートされます。



注記

storage ブロックを省略すると、一時ストレージのみを含むデプロイメントになります。

```
spec:
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 3
      storage: {}
```

Elasticsearch レプリケーションポリシー

Elasticsearch シャードをクラスター内のデータノードにレプリケートする方法を定義するポリシーを設定できます。

- **FullRedundancy**:各インデックスのシャードはすべてのデータノードに完全にレプリケートされます。
- **MultipleRedundancy**:各インデックスのシャードはデータノードの半分に分散します。
- **SingleRedundancy**:各シャードの単一コピー。2 つ以上のデータノードが存在する限り、ログは常に利用可能かつ回復可能です。
- **ZeroRedundancy**:シャードのコピーはありません。ログは、ノードの停止または失敗時に利用不可になる (または失われる) 可能性があります。

Curator スケジュール

Curator のスケジュールを cron 形式 で指定します。

```
spec:
  curation:
  type: "curator"
  resources:
  curator:
    schedule: "30 3 * * *"
```

9.3.2.2. 変更された ClusterLogging カスタムリソースのサンプル

以下は、前述のオプションを使用して変更された **ClusterLogging** カスタムリソースの例です。

変更された ClusterLogging リソースのサンプル

```
apiVersion: "logging.openshift.io/v1"
kind: "ClusterLogging"
metadata:
  name: "instance"
  namespace: "openshift-logging"
spec:
  managementState: "Managed"
  logStore:
    type: "elasticsearch"
    retentionPolicy:
      application:
        maxAge: 1d
      infra:
        maxAge: 7d
      audit:
        maxAge: 7d
    elasticsearch:
      nodeCount: 3
      resources:
        limits:
          memory: 32Gi
        requests:
          cpu: 3
          memory: 32Gi
      storage: {}
      redundancyPolicy: "SingleRedundancy"
  visualization:
    type: "kibana"
    kibana:
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: 500m
          memory: 1Gi
      replicas: 1
  curation:
    type: "curator"
    curator:
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 200m
          memory: 200Mi
      schedule: "*/5 * * * *"
  collection:
    logs:
      type: "fluentd"
      fluentd:
```

```
resources:
  limits:
    memory: 1Gi
  requests:
    cpu: 200m
    memory: 1Gi
```

9.3.3. クラスターロギングの使用による Knative Serving コンポーネントのログの検索

手順

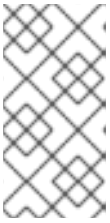
1. Elasticsearch の仮想化ツール Kibana UI を開くには、 以下のコマンドを使用して Kibana ルートを取得します。

```
$ oc -n openshift-logging get route kibana
```

2. ルートの URL を使用して Kibana ダッシュボードに移動し、ログインします。

3. インデックスが **.all** に設定されていることを確認します。インデックスが **.all** に設定されていない場合、OpenShift システムログのみが一覧表示されます。

4. **knative-serving** namespace を使用してログをフィルターできます。**kubernetes.namespace_name:knative-serving** を検索ボックスに入力して結果をフィルターします。



注記

Knative Serving はデフォルトで構造化ロギングを使用します。クラスターロギング Fluentd 設定をカスタマイズしてこれらのログの解析を有効にできます。これにより、ログの検索がより容易になり、ログレベルでのフィルターにより問題を迅速に特定できるようになります。

9.3.4. クラスターロギングを使用した Knative Serving でデプロイされたサービスのログの検索

OpenShift クラスターロギングにより、アプリケーションがコンソールに書き込むログは Elasticsearch で収集されます。以下の手順で、Knative Serving を使用してデプロイされたアプリケーションにこれらの機能を適用する方法の概要を示します。

手順

1. 以下のコマンドを使用して、Kibana の URL を見つけます。

```
$ oc -n cluster-logging get route kibana`
```

2. URL をブラウザーに入力し、Kibana UI を開きます。

3. インデックスが **.all** に設定されていることを確認します。インデックスが **.all** に設定されていない場合、OpenShift システムログのみが一覧表示されます。

4. サービスがデプロイされている Kubernetes namespace を使用してログをフィルターします。フィルターを追加してサービス自体を特定します: **kubernetes.namespace_name:default AND kubernetes.labels.serving_knative_dev\/service:{SERVICE_NAME}**

注記

/configuration または **/revision** を使用してフィルターすることもできます。

5. **kubernetes.container_name:<user-container>** を使用して検索を絞り込み、ご使用のアプリケーションで生成されるログのみを表示することができます。それ以外の場合は、queue-proxy からのログが表示されます。

注記

アプリケーションで JSON ベースの構造化ロギングを使用することで、実稼働環境でのこれらのログの迅速なフィルターを実行できます。

9.4. リビジョン間でのトラフィックの分割

9.4.1. Developer パースペクティブを使用したリビジョン間のトラフィックの分離

サーバーレスアプリケーションの作成後、サーバーレスアプリケーションは **Developer** パースペクティブの **Topology** ビューに表示されます。アプリケーションのリビジョンはノードごとに表示され、サーバーレスリソースサービスには、ノードの周りの四角形のマークが付けられます。

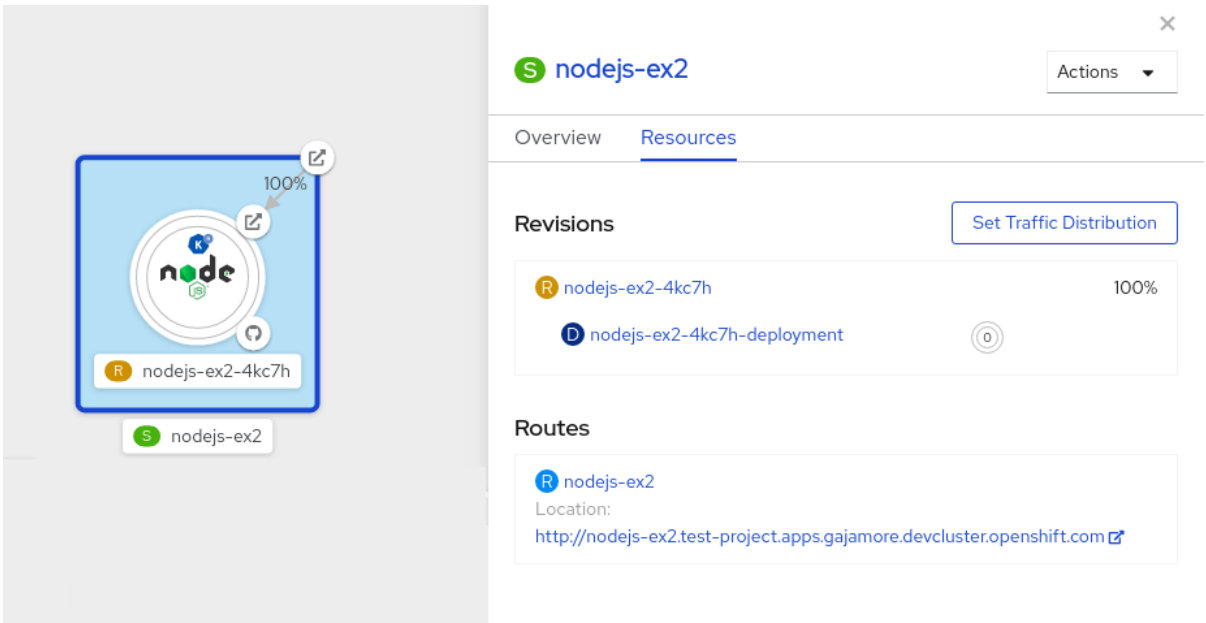
コードまたはサービス設定の新たな変更により、指定のタイミングでリビジョン、コードのスナップショットがトリガーされます。サービスの場合、必要に応じてこれを分割し、異なるリビジョンにルーティングして、サービスのリビジョン間のトラフィックを管理することができます。

手順

Topology ビューでアプリケーションの複数のリビジョン間でトラフィックを分割するには、以下を行います。

1. 四角形のマークが付けられたサーバーレスリソースサービスをクリックし、その概要をサイドパネルに表示します。
2. **Resources** タブをクリックして、サービスの **Revisions** および **Routes** の一覧を表示します。

図9.1 Serverless アプリケーション



3. サイドパネルの上部にある S アイコンで示されるサービスをクリックし、サービスの詳細の概要を確認します。

4. **YAML** タブをクリックし、YAML エディターでサービス設定を変更し、**Save** をクリックします。たとえば、**timeoutseconds** を 300 から 301 に変更します。この設定の変更により、新規リビジョンがトリガーされます。**Topology** ビューでは、最新のリビジョンが表示され、サービスの **Resources** タブに 2 つのリビジョンが表示されるようになります。

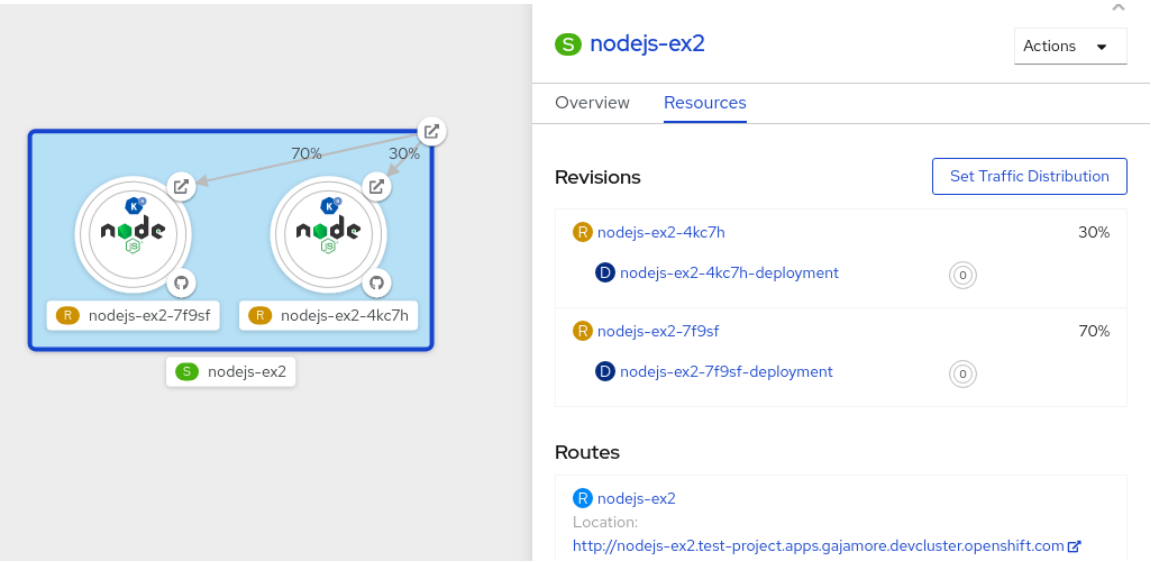
5. **Resources** タブで **Set Traffic Distribution** ボタンをクリックして、トラフィック分配ダイアログボックスを表示します。

   a. **Splits** フィールドに、2 つのリビジョンのそれぞれの分割されたトラフィックパーセンテージを追加します。

   b. 2 つのリビジョンのカスタム URL を作成するタグを追加します。

   c. **Save** をクリックし、Topology ビューで 2 つのリビジョンを表す 2 つのノードを表示します。

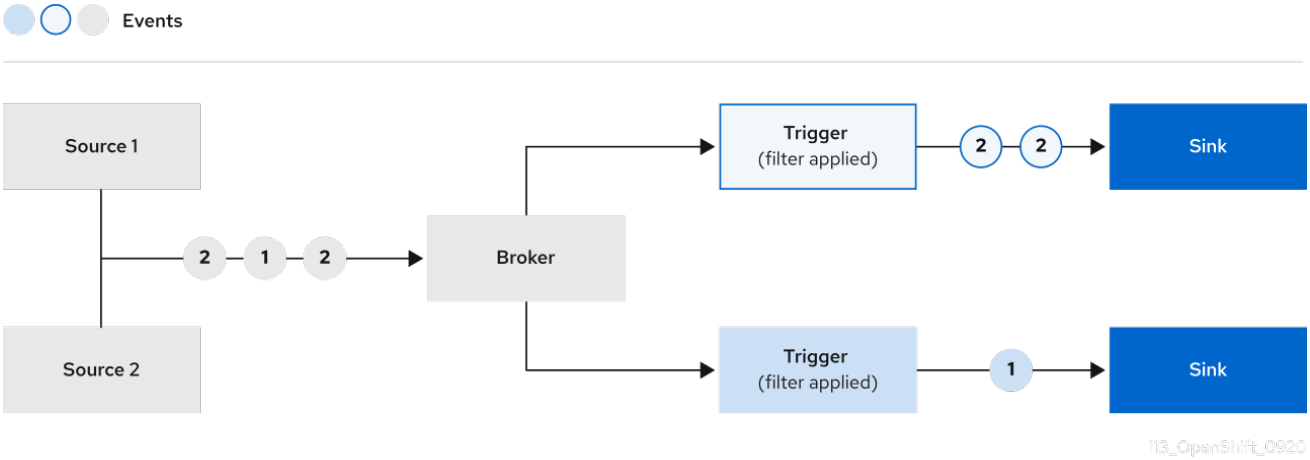
図9.2 Serverless アプリケーションのリビジョン



第10章 イベントワークフロー

10.1. ブローカーおよびトリガーを使用したイベント配信ワークフロー

ブローカーは トリガー と組み合わせて、イベントを イベントソース からイベントシンクに配信できます。



イベントは、HTTP POST リクエストとしてイベントソースからブローカーに送信されます。

イベントがブローカーに送信された後に、それらはトリガーを使用して CloudEvent 属性 でフィルターされ、HTTP POST リクエストとしてイベントシンクに送信できます。

10.1.1. ブローカーの作成

OpenShift Serverless は、Knative CLI を使用して作成できる **default** の Knative ブローカーを提供します。また、クラスター管理者の場合は **eventing.knative.dev/injection=enabled** ラベルを namespace に追加するか、または開発者の場合は **eventing.knative.dev/injection: enabled** アノテーションをトリガーに追加して、**default** ブローカーを作成することもできます。

重要

開発者およびクラスター管理者はどちらも挿入 (injection) によってブローカーを追加できますが、クラスター管理者のみがこの方法を使用して作成されたブローカーを永続的に削除できます。

10.1.1.1. Knative CLI を使用したブローカーの作成

前提条件

- OpenShift Serverless Operator および Knative Eventing が OpenShift Container Platform クラスターにインストールされている。
- **kn** CLI がインストールされている。

手順

- **default** ブローカーを作成します。

```
$ kn broker create default
```

検証

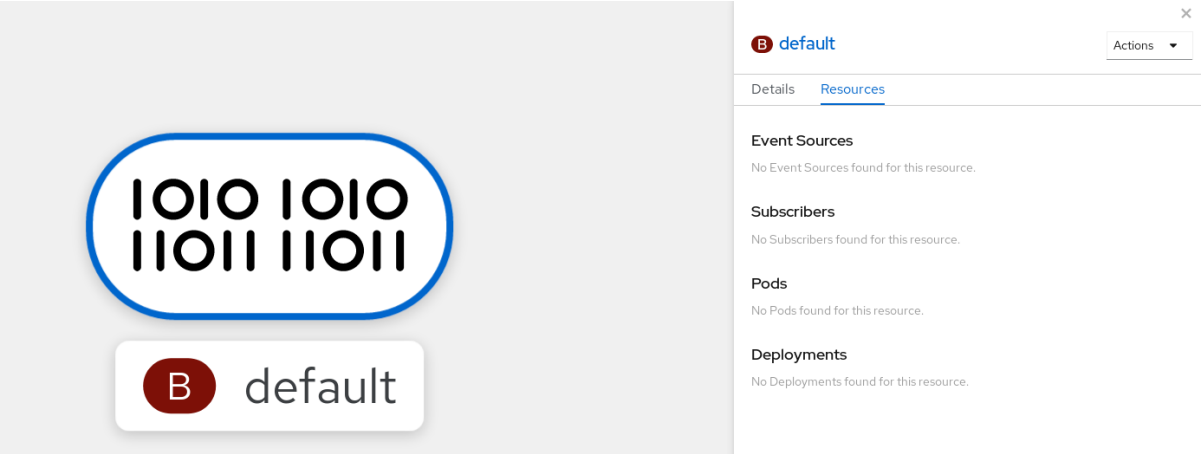
1. **kn** コマンドを使用して、既存のブローカーを一覧表示します。

```
$ kn broker list
```

出力例

```
NAME      URL                                                    AGE   CONDITIONS   READY
REASON
default   http://broker-ingress.knative-eventing.svc.cluster.local/test/default   45s   5 OK / 5
True
```

2. オプション: OpenShift Container Platform Web コンソールを使用している場合、**Developer** パースペクティブの **Topology** ビューに移動し、ブローカーが存在することを確認できます。



10.1.1.2. トリガーのアノテーションによるブローカーの作成

eventing.knative.dev/injection: enabled アノテーションを **Trigger** オブジェクトに追加してブローカーを作成できます。

重要

knative-eventing-injection: enabled アノテーションを使用してブローカーを作成する場合、クラスター管理者パーミッションなしにこのブローカーを削除することはできません。クラスター管理者が最初にこのアノテーションを削除せずにブローカーを削除する場合、ブローカーは削除後に再び作成されます。

前提条件

- OpenShift Serverless Operator および Knative Eventing が OpenShift Container Platform クラスターにインストールされている。

手順

1. **Trigger** オブジェクトを、**eventing.knative.dev/injection: enabled** アノテーションを持つ **.yaml** ファイルとして作成します。

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
```

```
  annotations:
    eventing.knative.dev/injection: enabled
  name: <trigger-name>
spec:
  broker: default
  subscriber: 1
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: <service-name>
```

1 トリガーがイベントを送信するイベントシンクまたは **サブスクライバー** の詳細を指定します。

2. **.yaml** ファイルを適用します。

```
$ oc apply -f <filename>
```

検証

oc CLI を使用してブローカーが正常に作成されていることを確認するか、または Web コンソールの **Topology** ビューでこれを確認できます。

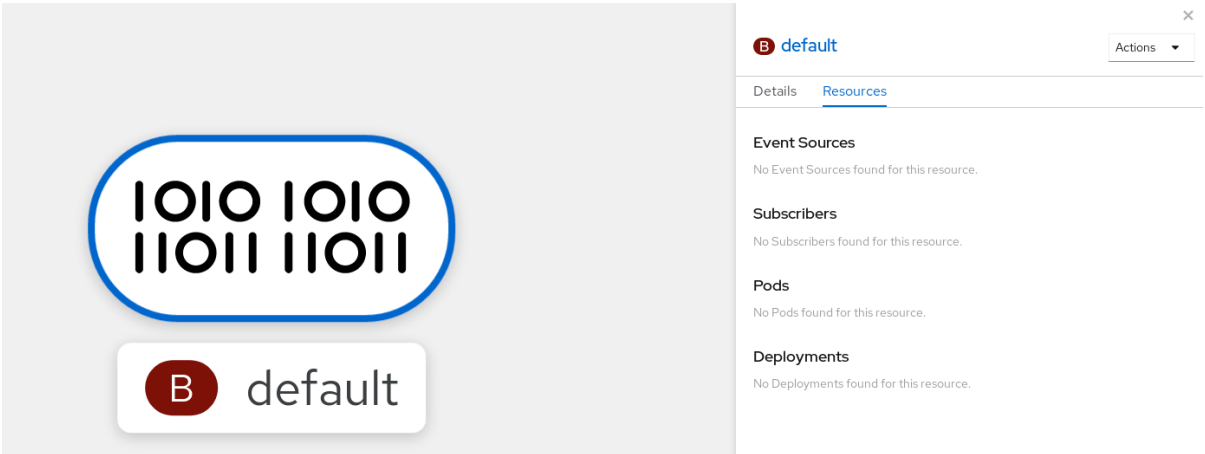
1. **oc** コマンドを使用してブローカーを取得します。

```
$ oc -n <namespace> get broker default
```

出力例

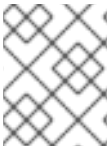
```
NAME      READY     REASON    URL                                                                 AGE
default   True                http://broker-ingress.knative-eventing.svc.cluster.local/test/default   3m56s
```

2. Web コンソールで **Topology** ビューに移動し、ブローカーが存在することを確認します。



10.1.1.3. namespace へのラベル付けによるブローカーの作成

クラスター管理者のパーミッションがある場合は、namespace にラベルを付けて **default** ブローカーを自動的に作成できます。



注記

この方法を使用して作成されたブローカーは、ラベルを削除すると削除されません。これらは手動で削除する必要があります。

前提条件

- OpenShift Serverless Operator および Knative Eventing が OpenShift Container Platform クラスターにインストールされている。
- OpenShift Container Platform のクラスター管理者パーミッションがある。

手順

- **eventing.knative.dev/injection=enabled** で namespace にラベルを付ける。

```
$ oc label namespace <namespace> eventing.knative.dev/injection=enabled
```

検証

oc CLI を使用してブローカーが正常に作成されていることを確認するか、または Web コンソールの **Topology** ビューでこれを確認できます。

1. **oc** コマンドを使用してブローカーを取得します。

```
$ oc -n <namespace> get broker <broker_name>
```

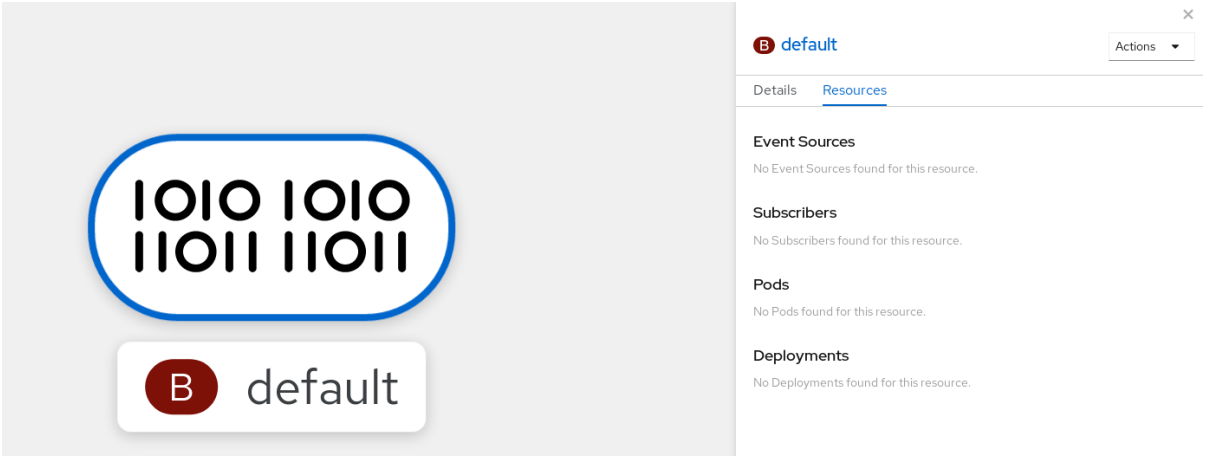
コマンドの例

```
$ oc -n default get broker default
```

出力例

```
NAME    READY    REASON    URL                                                                AGE
default   True             http://broker-ingress.knative-eventing.svc.cluster.local/test/default
3m56s
```

2. Web コンソールで **Topology** ビューに移動し、ブローカーが存在することを確認します。



10.1.2. ブローカーの管理

kn CLI は、ブローカーの一覧表示、説明、更新、および削除に使用できるコマンドを提供します。クラスター管理者は、挿入 (injection) を使用して作成されたブローカーを永続的に削除することもできます。

10.1.2.1. Knative CLI を使用した既存ブローカーの一覧表示

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing が OpenShift Container Platform クラスターにインストールされている。
- **kn** CLI がインストールされている。

手順

- 既存ブローカーの一覧を表示します。

```
$ kn broker list
```

出力例

```
NAME      URL                                                    AGE   CONDITIONS   READY
REASON
default   http://broker-ingress.knative-eventing.svc.cluster.local/test/default   45s   5 OK / 5
True
```

10.1.2.2. Knative CLI を使用した既存ブローカーの記述

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing が OpenShift Container Platform クラスターにインストールされている。
- **kn** CLI がインストールされている。

手順

- 既存ブローカーを記述します。

```
$ kn broker describe <broker_name>
```

デフォルトブローカーを使用したコマンドの例

```
$ kn broker describe default
```

出力例

```
Name:         default
Namespace:    default
Annotations:  eventing.knative.dev/broker.class=MTChannelBasedBroker,
eventing.knative.dev/creato ...
Age:          22s
```

```
Address:
  URL:    http://broker-ingress.knative-eventing.svc.cluster.local/default/default

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 22s
  ++ Addressable           22s
  ++ FilterReady           22s
  ++ IngressReady          22s
  ++ TriggerChannelReady   22s
```

10.1.2.3. 挿入 (injection) によって作成されたブローカーの削除

namespace ラベルまたはトリガーアノテーションを使用して、挿入 (injection) によって作成されたブローカーは、開発者がラベルまたはアノテーションを削除した場合に永続的に削除されません。クラスター管理者のパーミッションを持つユーザーは、これらのブローカーを手動で削除する必要があります。

手順

1. **eventing.knative.dev/injection=enabled** ラベルを namespace から削除します。

   ```
   $ oc label namespace <namespace> eventing.knative.dev/injection-
   ```

   アノテーションを削除すると、Knative では削除後にブローカーを再作成できなくなります。

2. 選択された namespace からブローカーを削除します。

   ```
   $ oc -n <namespace> delete broker <broker_name>
   ```

検証

- **oc** コマンドを使用してブローカーを取得します。

  ```
  $ oc -n <namespace> get broker <broker_name>
  ```

  コマンドの例

  ```
  $ oc -n default get broker default
  ```

  出力例

  ```
  No resources found.
  Error from server (NotFound): brokers.eventing.knative.dev "default" not found
  ```

10.1.3. トリガーを使用したイベントのフィルター

トリガーを使用すると、イベントシンクに配信するためにブローカーからイベントをフィルターできます。

前提条件

トリガーを使用する前に、以下が必要になります。

- Knative Eventing および **kn** がインストールされている。
- **default** ブローカーまたは作成したブローカーのいずれかの利用可能なブローカー。**default** ブローカーは、Knative Eventing でのブローカーの使用 の説明に従うか、またはトリガーの作成時に **--inject-broker** フラグを使用して作成できます。このフラグの使用については、本セクションで説明します。
- Knative サービスなどの利用可能なイベントコンシューマー。

10.1.3.1. Developer パースペクティブを使用したトリガーの作成

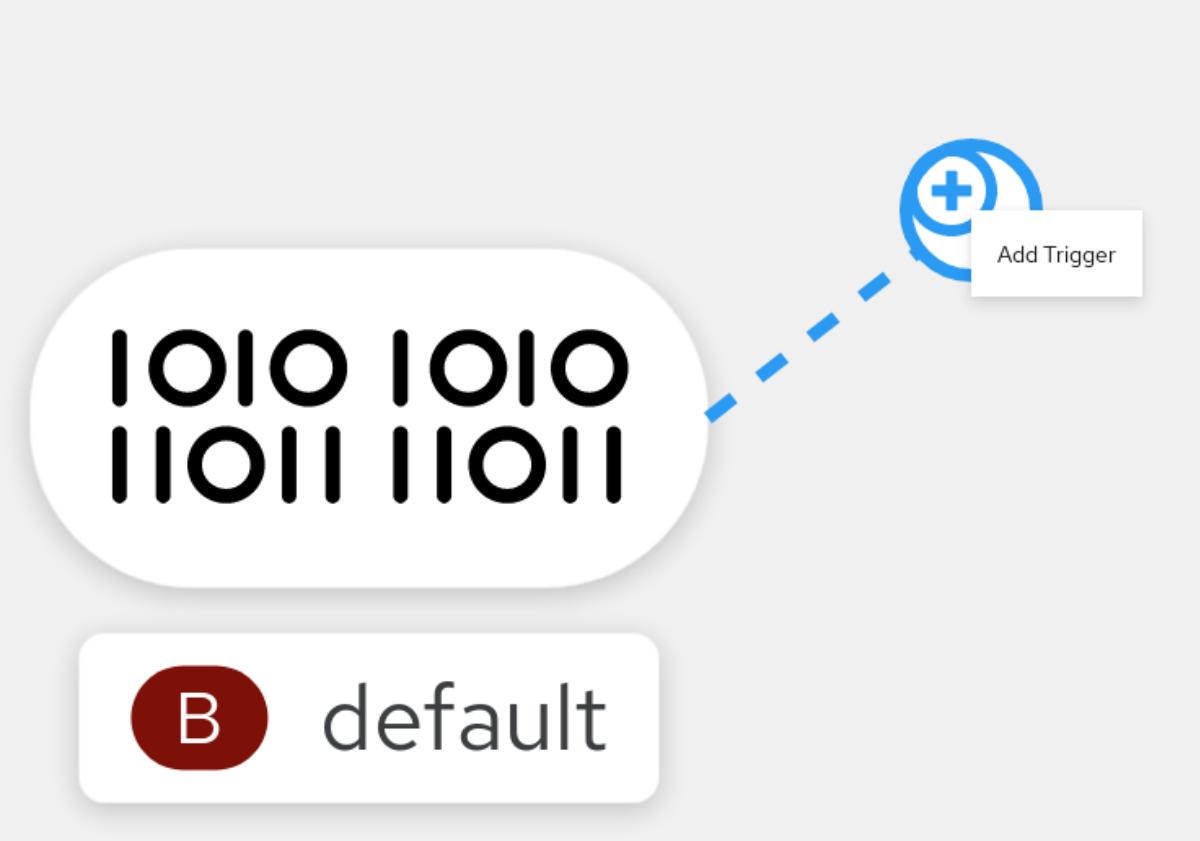
ブローカーの作成後は、Web コンソールの **Developer** パースペクティブでトリガーを作成できます。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing が OpenShift Container Platform クラスターにインストールされている。
- Web コンソールにログインしている。
- **Developer** パースペクティブを使用している。
- OpenShift Container Platform でアプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションを持つプロジェクトにアクセスできる。
- トリガーに接続するために、ブローカーおよび Knative サービスまたは他のイベントシンクを作成している。

手順

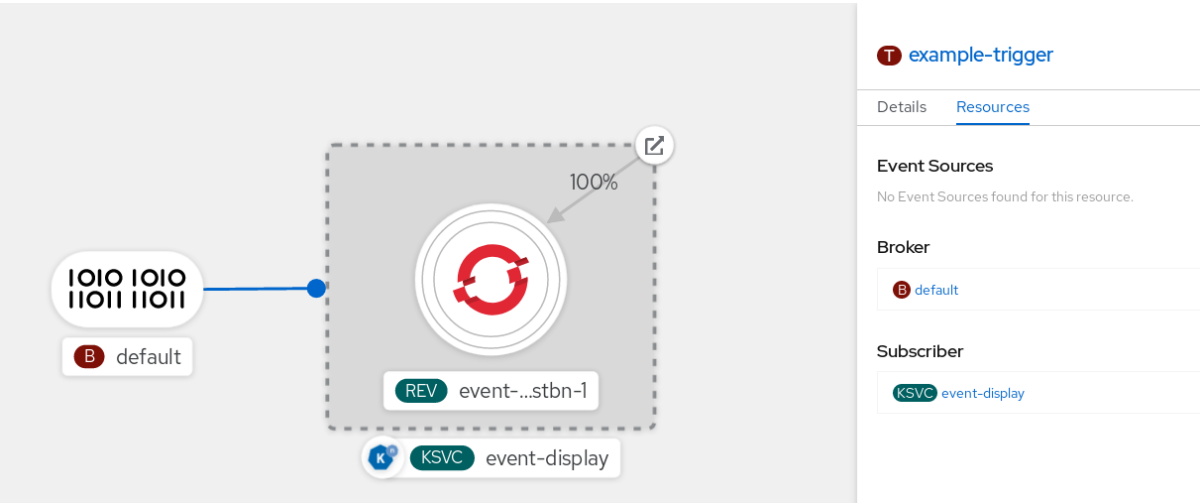
1. **Developer** パースペクティブで、**Topology** ページに移動します。
2. トリガーを作成するブローカーにカーソルを合わせ、矢印をドラッグします。**Add Trigger** オプションが表示されます。



3. **Add Trigger** をクリックします。
4. ドロップダウンリストから、シンクを **Subscriber** として選択します。
5. **Add** をクリックします。

検証

- サブスクリプションの作成後に、これを **Topology** ビューでブローカーをサービスに接続する行として表示できます。



10.1.3.2. Developer パースペクティブを使用したトリガーの削除

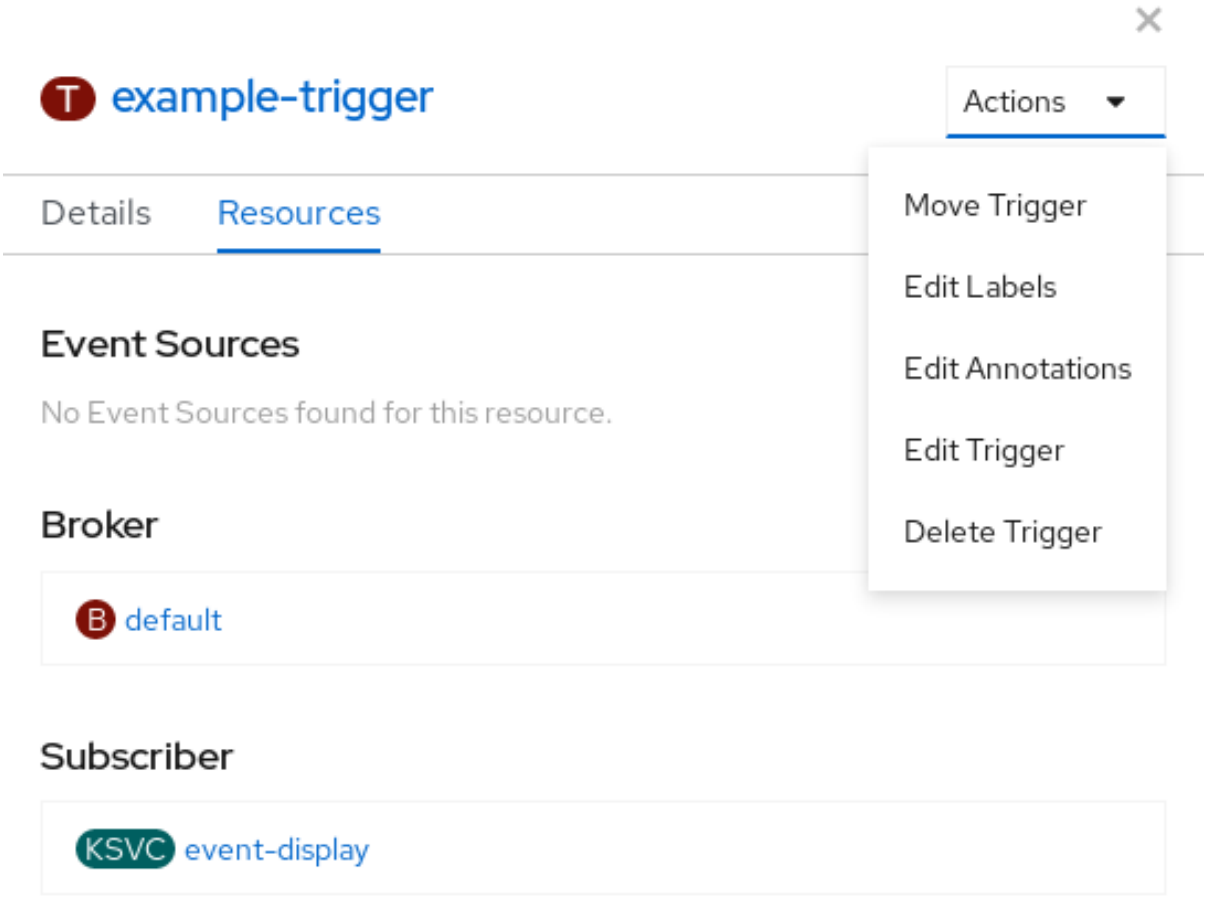
Web コンソールの **Developer** パースペクティブでトリガーを削除できます。

前提条件

- **Developer** パースペクティブを使用してトリガーを削除するには、Web コンソールにログインしている必要があります。

手順

1. **Developer** パースペクティブで、**Topology** ページに移動します。
2. 削除するトリガーをクリックします。
3. **Actions** コンテキストメニューで、**Delete Trigger** を選択します。



10.1.3.3. kn を使用したトリガーの作成

kn trigger create コマンドを使用してトリガーを作成できます。

手順

- トリガーを作成します。

```
$ kn trigger create <trigger_name> --broker <broker_name> --filter <key=value> --sink
<sink_name>
```

または、トリガーを作成し、ブローカー挿入を使用して **default** ブローカーを同時に作成できます。

```
$ kn trigger create <trigger_name> --inject-broker --filter <key=value> --sink <sink_name>
```

デフォルトで、トリガーはブローカーに送信されたすべてのイベントを、そのブローカーにサブスクライブされるシンクに転送します。トリガーの **--filter** 属性を使用すると、ブローカーか

らイベントをフィルターできるため、サブスクライバーは定義された基準に基づくイベントのサブセットのみを受け取ることができます。

10.1.3.4. kn を使用したトリガーの一覧表示

kn trigger list コマンドは利用可能なトリガーの一覧を出力します。

手順

1. 利用可能なトリガーの一覧を出力するには、以下のコマンドを入力します。

```
$ kn trigger list
```

出力例:

```
NAME    BROKER    SINK          AGE   CONDITIONS   READY   REASON
email   default   ksvc:edisplay   4s    5 OK / 5     True
ping    default   ksvc:edisplay   32s   5 OK / 5     True
```

2. オプション: JSON 形式でトリガーの一覧を出力します。

```
$ kn trigger list -o json
```

10.1.3.4.1. kn を使用したトリガーの記述

kn trigger describe コマンドを使用して、トリガーについての情報を出力できます。

手順

- トリガーについての情報を出力するには、以下のコマンドを入力します。

```
$ kn trigger describe <trigger_name>
```

出力例

```
Name:         ping
Namespace:    default
Labels:       eventing.knative.dev/broker=default
Annotations:  eventing.knative.dev/creator=kube:admin,
eventing.knative.dev/lastModifier=kube:admin
Age:          2m
Broker:       default
Filter:
  type:       dev.knative.event

Sink:
  Name:       edisplay
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                 AGE REASON
  ++ Ready                2m
```

```
++ BrokerReady            2m
++ DependencyReady        2m
++ Subscribed             2m
++ SubscriberResolved     2m
```

10.1.3.4.2. トリガーを使用したイベントのフィルター

以下のトリガーの例では、**type: dev.knative.samples.helloworld** 属性のあるイベントのみがイベントコンシューマーに到達します。

```
$ kn trigger create <trigger_name> --broker <broker_name> --filter
type=dev.knative.samples.helloworld --sink ksvc:<service_name>
```

複数の属性を使用してイベントをフィルターすることもできます。以下の例は、type、source、および extension 属性を使用してイベントをフィルターする方法を示しています。

```
$ kn trigger create <trigger_name> --broker <broker_name> --sink ksvc:<service_name> \
--filter type=dev.knative.samples.helloworld \
--filter source=dev.knative.samples/helloworldsource \
--filter myextension=my-extension-value
```

10.1.3.4.3. kn を使用したトリガーの更新

特定のフラグを指定して **kn trigger update** コマンドを使用して、トリガーの属性を迅速に更新できます。

手順

- トリガーを更新します。

```
$ kn trigger update <trigger_name> --filter <key=value> --sink <sink_name> [flags]
```

  - トリガーを、受信イベントに一致するイベント属性をフィルターするように更新できます。たとえば、**type** 属性を使用します。

```
$ kn trigger update mytrigger --filter type=knative.dev.event
```

  - トリガーからフィルター属性を削除できます。たとえば、キー **type** を使用してフィルター属性を削除できます。

```
$ kn trigger update mytrigger --filter type-
```

  - **--sink** パラメーターを使用して、トリガーのイベントシンクを変更できます。

```
$ kn trigger update <trigger_name> --sink ksvc:my-event-sink
```

10.1.3.4.4. kn を使用したトリガーの削除

手順

- トリガーを削除します。

```
$ kn trigger delete <trigger_name>
```

検証

1. 既存のトリガーを一覧表示します。

```
$ kn trigger list
```

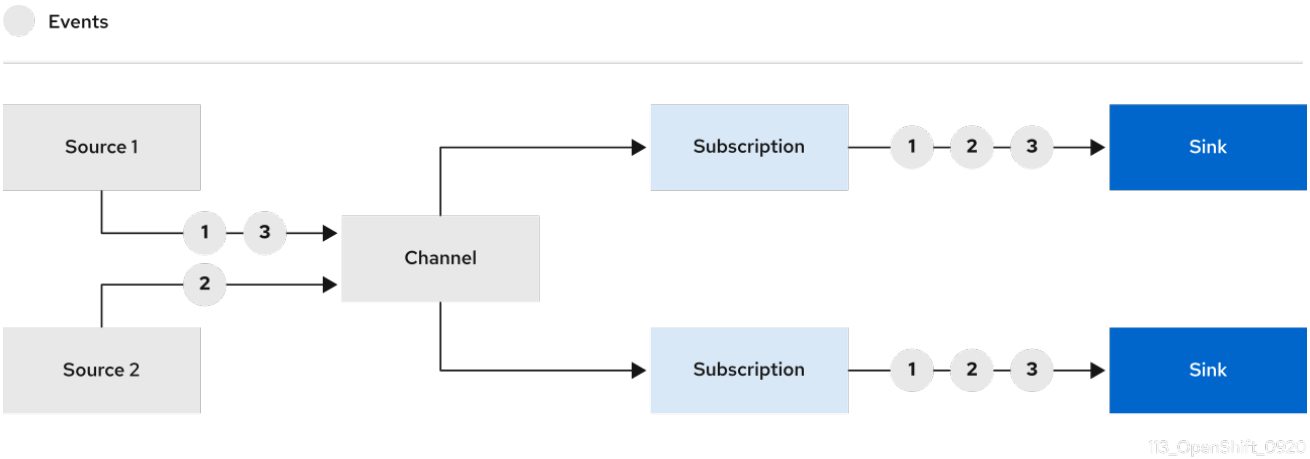
2. トリガーが存在しないことを確認します。

出力例

```
No triggers found.
```

10.2. チャネルを使用したイベント配信ワークフロー

イベント配信のチャネルおよびサブスクリプションを使用して、イベントをソースからシンクに送信できます。



チャネルは、単一のイベント転送および永続レイヤーを定義するカスタムリソースです。

イベントがチャネルに送信された後に、これらのイベントはサブスクリプションを使用して複数の Knative サービスまたは他のシンクに送信できます。

チャネルインスタンスのデフォルト設定は **default-ch-webhook** 設定マップで定義されます。開発者はサポートされている **Channel** オブジェクトをインスタンス化することで、独自のチャネルを直接作成できます。

10.2.1. サポートされているチャネルタイプ

現時点で、OpenShift Serverless は Knative Eventing テクノロジープレビューの一部として **InMemoryChannel** タイプのチャネルの開発目的での使用のみをサポートします。

以下は、**InMemoryChannel** チャネルの制限です。

- イベントの永続性は利用できません。Pod がダウンすると、その Pod のイベントが失われます。

- **InMemoryChannel** チャネルはイベントの順序を実装しないため、チャネルで同時に受信される 2 つのイベントはいずれの順序でもサブスクライバーに配信できます。
- サブスクライバーがイベントを拒否する場合、再配信は試行されません。代わりに、拒否されたイベントは、シンクが存在する場合は **deadLetterSink** オブジェクトに送信されます。これが存在しない場合にはドロップされます。

10.2.1.1. デフォルトの開発チャネル設定の使用

Knative Eventing のインストール時に、以下の **default-ch-webhook** 設定マップが **knative-eventing** namespace に自動的に作成されます。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: default-ch-webhook
  namespace: knative-eventing
data:
  default-ch-config: |
    clusterDefault:
      apiVersion: messaging.knative.dev/v1
      kind: InMemoryChannel
    namespaceDefaults:
      some-namespace:
        apiVersion: messaging.knative.dev/v1
        kind: InMemoryChannel
```

この設定マップは、クラスター全体のデフォルトのチャネル実装または namespace 固有のデフォルトのチャネル実装のいずれかを指定できます。namespace 固有のデフォルトを設定すると、クラスター全体の設定が上書きされます。

Channel オブジェクトが作成されると、変更用の受付 Webhook はデフォルトのチャネル実装に基づいて **Channel** オブジェクトの **spec.channelTemplate** プロパティーのセットを追加します。

spec.channelTemplate プロパティーを持つ Channel オブジェクトの例

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
  name: example-channel
  namespace: default
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1
    kind: InMemoryChannel
```

チャネルコントローラーは、その後に **spec.channelTemplate** 設定に基づいてサポートするチャネルインスタンスを作成します。

注記

spec.channelTemplate プロパティーは作成後に変更できません。それらは、ユーザーではなくデフォルトのチャネルメカニズムで設定されるためです。

このメカニズムが使用される場合、汎用チャネル、および **InMemoryChannel** チャネルなど 2 つのオブジェクトが作成されます。

汎用チャネルは、サブスクリプションを **InMemoryChannel** チャネルにコピーするプロキシーとして機能し、サポートする **InMemoryChannel** チャネルのステータスを反映するようにそのステータスを設定します。

この例のチャネルはデフォルトの namespace で作成されるため、チャネルはクラスターのデフォルト (InMemoryChannel) を使用します。

10.2.2. 開発チャネルの作成

手順

以下の手順を実行して、クラスターのデフォルト設定を使用してチャネルを作成できます。

1. **Channel** オブジェクトを作成します。
   a. YAML ファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
  name: example-channel
  namespace: default
```

   b. 以下を入力して YAML ファイルを適用します。

```
$ oc apply -f <filename>
```

10.2.3. サブスクリプションの作成

Subscription オブジェクトを作成して、チャネルをシンクに接続することができます。以下の手順では、サンプルのシンクは **error-handler** という名前の Knative サービスです。

手順

1. YAML ファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: messaging.knative.dev/v1beta1
kind: Subscription
metadata:
  name: my-subscription (1)
  namespace: default
spec:
  channel: (2)
    apiVersion: messaging.knative.dev/v1beta1
    kind: Channel
    name: example-channel
  delivery: (3)
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
```

```
      name: error-handler
  subscriber: 4
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1 サブスクリプションの名前。

2 サブスクリプションが接続するチャネルの設定。

3 イベント配信の設定。これは、サブスクリプションに対してサブスクライバーに配信できないイベントに何が発生するかについて示します。これが設定されると、使用できないイベントが **deadLetterSink** に送信されます。イベントがドロップされると、イベントの再配信は試行されず、エラーのログがシステムに記録されます。**deadLetterSink** 値は Destination である必要があります。

4 サブスクライバーの設定。これは、イベントがチャネルから送信されるイベントシンクです。

2. YAML ファイルを適用します。

```
$ oc apply -f <FILENAME>
```

第11章 イベントソース

11.1. イベントソースの使用

イベントソース は、イベントプロデューサーをイベント **シンク** またはコンシューマーにリンクするオブジェクトです。シンクには、イベントソースからイベントを受信する Knative サービス、チャネルまたはブローカーを使用できます。

11.1.1. イベントソースの作成

現時点で、OpenShift Serverless は以下のイベントソースタイプをサポートします。

API サーバーソース

APIServerSource オブジェクトを作成して、シンクを Kubernetes API サーバーに接続します。

Ping ソース

一定のペイロードを使用して ping イベントを定期的に送信します。ping ソースはタイマーとして使用でき、**PingSource** オブジェクトとして作成されます。

シンクバインディング もサポートされます。これにより、シンクを使用して **Deployment**、 **Job**、または **StatefulSet** などのコア Kubernetes リソースを接続できます。

OpenShift Container Platform Web コンソール、**kn** CLI を使用するか、または YAML ファイルを適用して **Developer** パースペクティブで Knative イベントソースを作成したり、管理したりできます。

- API サーバーソース を作成します。
- ping ソース を作成します。
- シンクバインディング を作成します。

11.1.2. 追加リソース

- OpenShift Serverless を使用した Eventing ワークフローの詳細は、Knative Eventing アーキテクチャー について参照してください。

11.2. KNATIVE CLI を使用したイベントソースおよびイベントソースタイプの一覧表示

kn CLI を使用して、Knative Eventing で使用するために利用できるイベントソースまたはイベントソースのタイプを一覧表示し、管理できます。

現在、**kn** は以下のイベントソースタイプの管理をサポートします。

API サーバーソース

APIServerSource オブジェクトを作成して、シンクを Kubernetes API サーバーに接続します。

Ping ソース

一定のペイロードを使用して ping イベントを定期的に送信します。ping ソースはタイマーとして使用でき、**PingSource** オブジェクトとして作成されます。

11.2.1. Knative CLI の使用による利用可能なイベントソースタイプの一覧表示

以下のコマンドを使用して、ターミナルに利用可能なイベントソースタイプを一覧表示できます。

```
$ kn source list-types
```

このコマンドのデフォルト出力は以下のようになります。

```
TYPE              NAME                                    DESCRIPTION
ApiServerSource   apiserversources.sources.knative.dev     Watch and send Kubernetes API
events to a sink
PingSource        pingsources.sources.knative.dev         Periodically send ping events to a sink
SinkBinding       sinkbindings.sources.knative.dev        Binding for connecting a PodSpecable to
a sink
```

利用可能なイベントソースタイプを YAML 形式で一覧表示することもできます。

```
$ kn source list-types -o yaml
```

11.2.2. Knative CLI の使用による利用可能なイベントリソースの一覧表示

以下のコマンドを使用して、ターミナルに利用可能なイベントソースを一覧表示できます。

```
$ kn source list
```

出力例

```
NAME  TYPE             RESOURCE                              SINK        READY
a1    ApiServerSource  apiserversources.sources.knative.dev  ksvc:eshow2  True
b1    SinkBinding      sinkbindings.sources.knative.dev      ksvc:eshow3  False
p1    PingSource       pingsources.sources.knative.dev       ksvc:eshow1  True
```

--type フラグを使用して、特定タイプのイベントソースのみを一覧表示できます。

```
$ kn source list --type PingSource
```

出力例

```
NAME  TYPE          RESOURCE                          SINK        READY
p1    PingSource    pingsources.sources.knative.dev   ksvc:eshow1  True
```

11.2.3. 次のステップ

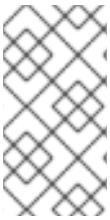
- API サーバーソースの使用 についてのドキュメントを参照してください。
- ping ソースの使用 についてのドキュメントを参照してください。

11.3. API サーバーソースの使用

API サーバーソースは、Knative サービスなどのイベントシンクを Kubernetes API サーバーに接続するために使用できるイベントソースです。API サーバーソースは Kubernetes イベントを監視し、それらを Knative Eventing ブローカーに転送します。

11.3.1. 前提条件

- Knative Serving および Eventing を含む OpenShift Serverless を Container Platform クラスターにインストールしている必要があります。クラスター管理者がこれをインストールできます。
- イベントソースには、イベント **シンク** として使用するサービスが必要です。シンクは、イベントがイベントソースから送信されるサービスまたはアプリケーションです。
- イベントソースのサービスアカウント、ロールおよびロールバインディングを作成または更新する必要があります。



注記

以下の手順の一部では、YAML ファイルの作成が必要になります。

サンプルで使用されたもので YAML ファイルの名前を変更する場合は、必ず対応する CLI コマンドを更新する必要があります。

11.3.2. イベントソースのサービスアカウント、ロールおよびバインディングの作成

手順

1. **authentication.yaml** という名前のファイルを作成し、以下のサンプルコードをこれにコピーして、イベントソースのサービスアカウント、ロールおよびロールバインディングを作成します。

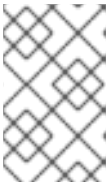
```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default (1)

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default (2)
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default (3)
```

```
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default 4
```

1 2 3 4 この namespace を、イベントソースのインストールに選択した namespace に変更します。



注記

適切なパーミッションを持つ既存のサービスアカウントを再利用する必要がある場合、そのサービスアカウントの **authentication.yaml** を変更する必要があります。

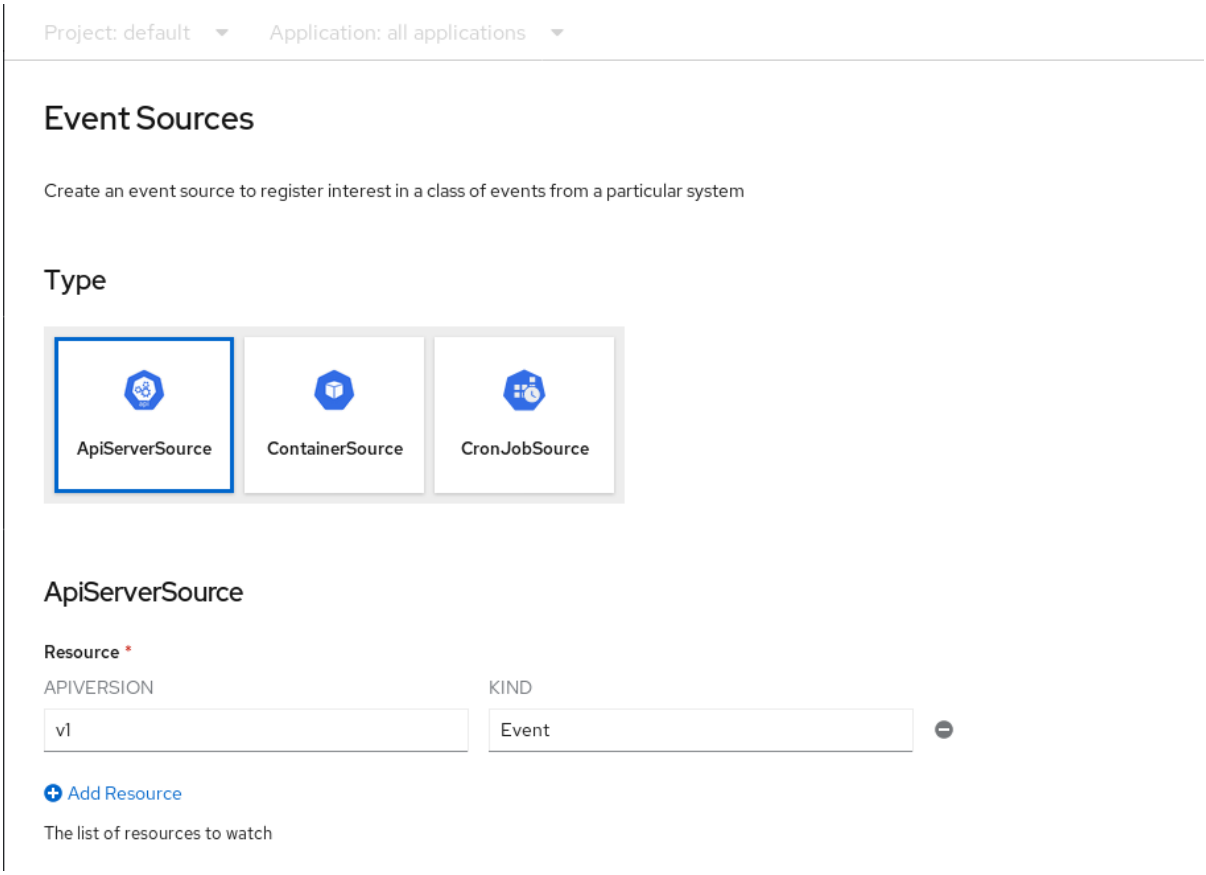
2. 以下のコマンドを入力して、サービスアカウント、ロールバインディング、およびクラスターバインディングを作成します。

```
$ oc apply --filename authentication.yaml
```

11.3.3. Developer パースペクティブを使用した ApiServerSource イベントソースの作成

手順

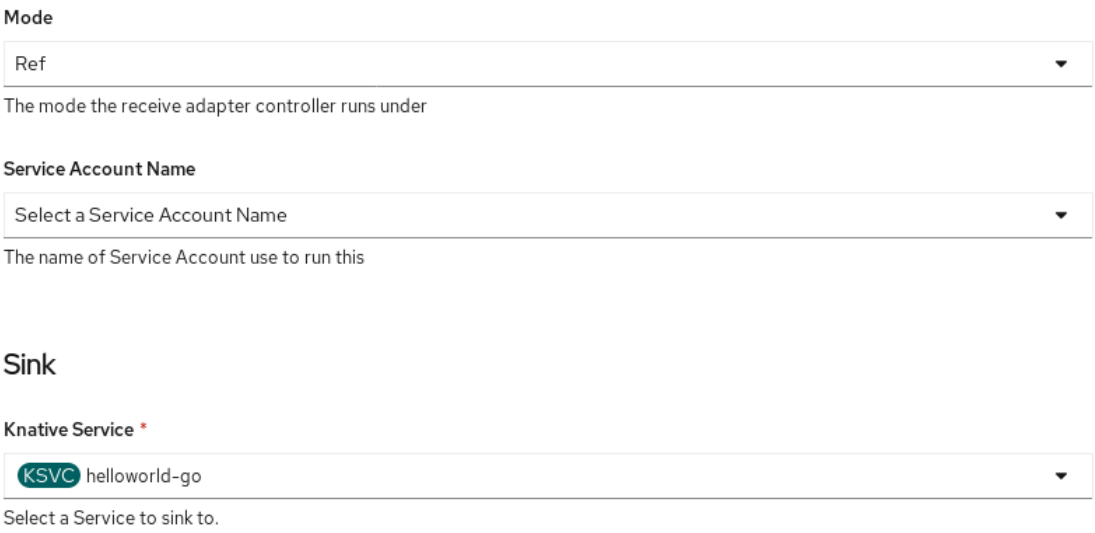
1. **Add** ページに移動し、**Event Source** を選択します。
2. **Event Sources** ページで、**Type** セクションで **ApiServerSource** を選択します。



3. **ApiServerSource** を設定します。

   a. **APIVERSION** に **v1** を、**KIND** に **Event** を入力します。

   b. 作成したサービスアカウントの **Service Account Name** を選択します。

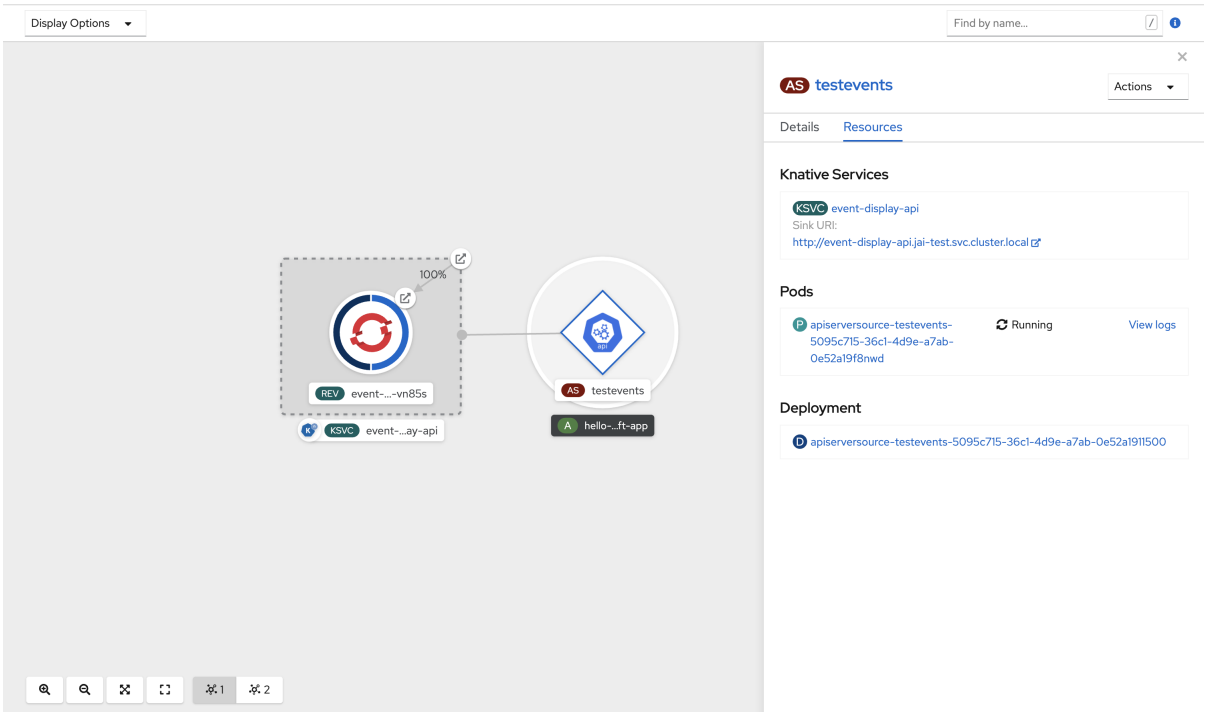


   c. **Sink → Knative Service** のドロップダウンメニューでターゲットに設定された Knative サービスを選択します。

4. **Create** をクリックします。

検証

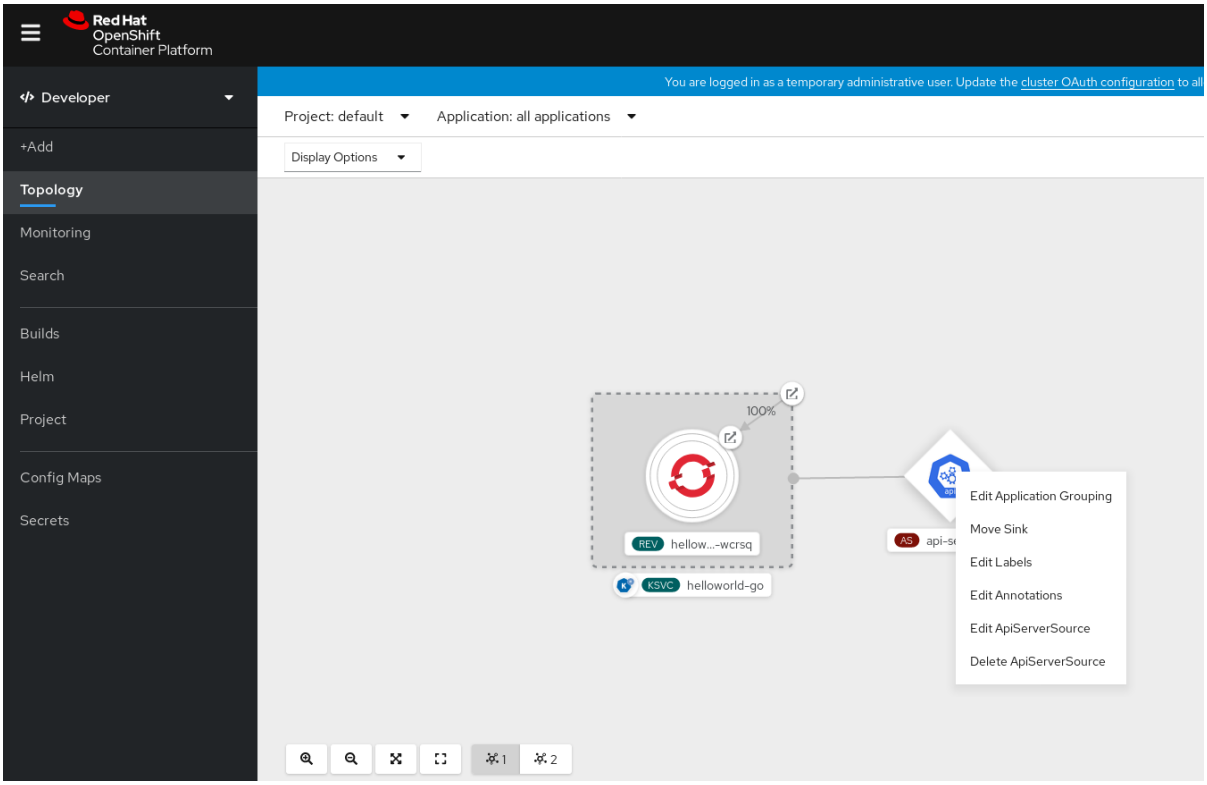
1. ApiServerSource の作成後、これが **Topology** ビューでシンクされるサービスに接続されていることを確認できます。



11.3.4. ApiServerSource の削除

手順

1. **Topology** ビューに移動します。
2. ApiServerSource を右クリックし、**Delete ApiServerSource** を選択します。



11.3.5. Knative CLI での API サーバーソースの使用

以下のセクションでは、**kn** コマンドを使用して **ApiServerSource** オブジェクトを作成するために必要な手順を説明します。

前提条件

- Knative Serving および Eventing がクラスターにインストールされている。
- API サーバーソースがインストールされるのと同じ namespace に **default** ブローカーを作成している必要があります。
- **kn** CLI がインストールされている。

手順

1. **ApiServerSource** オブジェクトのサービスアカウント、ロール、およびロールバインディングを作成します。
authentication.yaml という名前のファイルを作成し、以下のサンプルコードをこれにコピーして、これを実行できます。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default (1)

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default (2)
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default (3)
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default (4)
```

(1)(2)(3)(4) この namespace を、API サーバーのインストール用に選択した namespace に変更します。

注記

適切なパーミッションを持つ既存のサービスアカウントを再利用する必要がある場合、そのサービスアカウントの **authentication.yaml** ファイルを変更する必要があります。

サービスアカウント、ロールバインディング、およびクラスターバインディングを作成します。

```
$ oc apply -f authentication.yaml
```

2. ブローカーをイベントシンクとして使用する **ApiServerSource** オブジェクトを作成します。

```
$ kn source apiserver create <event_source_name> --sink broker:<broker_name> --
resource "event:v1" --service-account <service_account_name> --mode Resource
```

3. 受信メッセージをログにダンプする Knative サービスを作成します。

```
$ kn service create <service_name> --image quay.io/openshift-knative/knative-eventing-
sources-event-display:latest
```

4. **default** ブローカーからサービスにイベントをフィルターするトリガーを作成します。

```
$ kn trigger create <trigger_name> --sink ksvc:<service_name>
```

5. デフォルト namespace で Pod を起動してイベントを作成します。

```
$ oc create deployment hello-node --image=quay.io/openshift-knative/knative-eventing-
sources-event-display
```

6. 以下のコマンドを入力し、生成される出力を検査して、コントローラーが正しくマップされていることを確認します。

```
$ kn source apiserver describe testevents
```

出力例

```
Name:               testevents
Namespace:          default
Annotations:        sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:                3m
ServiceAccountName:  events-sa
Mode:               Resource
Sink:
  Name:       default
  Namespace:  default
  Kind:       Broker (eventing.knative.dev/v1)
Resources:
  Kind:        event (v1)
  Controller:  false
Conditions:
  OK TYPE                  AGE REASON
```

```
++ Ready                 3m
++ Deployed              3m
++ SinkProvided          3m
++ SufficientPermissions   3m
++ EventTypesProvided      3m
```

検証

メッセージダンパー機能ログを確認して、Kubernetes イベントが Knative に送信されていることを確認できます。

1. Pod を取得します。

```
$ oc get pods
```

2. Pod のメッセージダンパー機能ログを表示します。

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

出力例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{hello-node}",
      "kind": "Pod",
      "name": "hello-node",
      "namespace": "default",
       .....
    },
    "kind": "Event",
    "message": "Started container",
    "metadata": {
      "name": "hello-node.159d7608e3a3572c",
      "namespace": "default",
      ....
    },
    "reason": "Started",
    ...
  }
```

11.3.6. Knative CLI を使用した API サーバーソースの削除

本セクションでは、**kn** コマンドおよび **oc** コマンドを使用して **ApiServerSource** オブジェクト、トリガー、サービス、サービスアカウント、クラスターロール、およびクラスターバインディングを削除するために使用される手順について説明します。

前提条件

- **kn** CLI がインストールされていること。

手順

1. トリガーを削除します。

```
$ kn trigger delete <trigger_name>
```

2. サービスを削除します。

```
$ kn service delete <service_name>
```

3. API サーバーソースを削除します。

```
$ kn source apiserver delete <source_name>
```

4. サービスアカウント、クラスターロール、およびクラスターバインディングを削除します。

```
$ oc delete -f authentication.yaml
```

11.3.7. YAML ファイルを使用した API サーバーソースの作成

以下では、YAML ファイルを使用して **ApiServerSource** オブジェクトを作成するために必要な手順を説明します。

前提条件

- Knative Serving および Eventing がクラスターにインストールされている。
- **default** ブローカーを、**ApiServerSource** オブジェクトで定義されるものと同じ namespace に作成している。

手順

1. API サーバーソースのサービスアカウント、ロールおよびロールバインディングを作成するには、**authentication.yaml** という名前のファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default (1)

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```
metadata:
  name: event-watcher
  namespace: default 2
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default 3
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default 4
```

1 2 3 4 この namespace を、API サーバーのインストール用に選択した namespace に変更します。

注記

適切なパーミッションを持つ既存のサービスアカウントを再利用する必要がある場合、そのサービスアカウントの **authentication.yaml** を変更する必要があります。

authentication.yaml ファイルを作成した後に、これを適用します。

```
$ oc apply -f authentication.yaml
```

2. **ApiServerSource** オブジェクトを作成するには、**k8s-events.yaml** という名前のファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  name: testevents
spec:
  serviceAccountName: events-sa
  mode: Resource
  resources:
    - apiVersion: v1
```

```
    kind: Event
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
```

k8s-events.yaml ファイルを作成した後に、これを適用します。

```
$ oc apply -f k8s-events.yaml
```

3. API サーバーソースが正しく設定されていることを確認するには、受信メッセージをログにダンプする Knative サービスを作成します。
以下のサンプル YAML を **service.yaml** という名前のファイルにコピーします。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

service.yaml ファイルを作成した後に、これを適用します。

```
$ oc apply -f service.yaml
```

4. イベントを直前の手順で作成したサービスにフィルターする **default** ブローカーからトリガーを作成するには、**trigger.yaml** という名前のファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: event-display-trigger
  namespace: default
spec:
  broker: default
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

trigger.yaml ファイルを作成した後に、これを適用します。

```
$ oc apply -f trigger.yaml
```

5. イベントを作成するには、**default** namespace で Pod を起動します。

```
$ oc create deployment hello-node --image=quay.io/openshift-knative/knative-eventing-sources-event-display
```

6. コントローラーが正しくマップされていることを確認するには、以下のコマンドを入力し、出力を検査します。

```
$ oc get apiserversource.sources.knative.dev testevents -o yaml
```

出力例

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  annotations:
  creationTimestamp: "2020-04-07T17:24:54Z"
  generation: 1
  name: testevents
  namespace: default
  resourceVersion: "62868"
  selfLink:
/apis/sources.knative.dev/v1alpha1/namespaces/default/apiserversources/testevents2
  uid: 1603d863-bb06-4d1c-b371-f580b4db99fa
spec:
  mode: Resource
  resources:
  - apiVersion: v1
    controller: false
    controllerSelector:
      apiVersion: ""
      kind: ""
      name: ""
      uid: ""
    kind: Event
    labelSelector: {}
  serviceAccountName: events-sa
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
```

検証

Kubernetes イベントが Knative に送信されていることを確認するには、メッセージダンパー機能ログを確認します。

1. Pod を取得します。

```
$ oc get pods
```

2. Pod のメッセージダンパー機能ログを表示します。

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

出力例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{hello-node}",
      "kind": "Pod",
      "name": "hello-node",
      "namespace": "default",
       .....
    },
    "kind": "Event",
    "message": "Started container",
    "metadata": {
      "name": "hello-node.159d7608e3a3572c",
      "namespace": "default",
      ....
    },
    "reason": "Started",
    ...
  }
```

11.3.8. API サーバーソースの削除

本セクションでは、YAML ファイルを削除して、**ApiServerSource** オブジェクト、トリガー、サービス、サービスアカウント、クラスターロール、およびクラスターバインディングを削除する方法について説明します。

手順

1. トリガーを削除します。

```
$ oc delete -f trigger.yaml
```

2. サービスを削除します。

```
$ oc delete -f service.yaml
```

3. API サーバーソースを削除します。

```
$ oc delete -f k8s-events.yaml
```

4. サービスアカウント、クラスターロール、およびクラスターバインディングを削除します。

```
$ oc delete -f authentication.yaml
```

11.4. PING ソースの使用

ping ソースは、一定のペイロードを使用して ping イベントをイベントコンシューマーに定期的に送信するために使用されます。ping ソースを使用すると、以下の例のようにタイマーと同様にイベントの送信をスケジュールできます。

ping ソースの例

```
apiVersion: sources.knative.dev/v1alpha2
kind: PingSource
metadata:
  name: test-ping-source
spec:
  schedule: "*/2 * * * *" 1
  jsonData: '{"message": "Hello world!"}' 2
  sink: 3
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1. CRON 式 を使用して指定されるイベントのスケジュール。
2. JSON でエンコードされたデータ文字列として表現されるイベントメッセージの本体。
3. これらはイベントコンシューマーの詳細です。この例では、**event-display** という名前の Knative サービスを使用しています。

11.4.1. Knative CLI を使用した ping ソースの作成

以下のセクションでは、**kn** CLI を使用して基本的な **PingSource** オブジェクトを作成し、検証し、削除する方法を説明します。

前提条件

- Knative Serving および Eventing がインストールされている。
- **kn** CLI がインストールされている。

手順

1. ping ソースが機能していることを確認するには、受信メッセージをサービスのログにダンプする単純な Knative サービスを作成します。

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

2. 要求する必要のある ping イベントのセットごとに、**PingSource** をイベントコンシューマーと同じ namespace に作成します。

```
$ kn source ping create test-ping-source \
   --schedule "*/2 * * * *" \
   --data '{"message": "Hello world!"}' \
   --sink ksvc:event-display
```

3. 以下のコマンドを入力し、出力を検査して、コントローラーが正しくマップされていることを確認します。

```
$ kn source ping describe test-ping-source
```

出力例

```
Name:         test-ping-source
Namespace:    default
Annotations:  sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:          15s
Schedule:     */2 * * * *
Data:         {"message": "Hello world!"}

Sink:
  Name:       event-display
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE               AGE REASON
  ++ Ready              8s
  ++ Deployed             8s
  ++ SinkProvided         15s
  ++ ValidSchedule        15s
  ++ EventTypeProvided    15s
  ++ ResourcesCorrect     15s
```

検証

シンク Pod のログを確認して、Kubernetes イベントが Knative イベントに送信されていることを確認できます。

デフォルトで、Knative サービスは、トラフィックが 60 秒以内に受信されない場合に Pod を終了します。本書の例では、新たに作成される Pod で各メッセージが確認されるように 2 分ごとにメッセージを送信する **PingSource** オブジェクトを作成します。

1. 作成された新規 Pod を監視します。

```
$ watch oc get pods
```

2. **Ctrl+C** を使用して Pod の監視をキャンセルし、作成された Pod のログを確認します。

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

出力例

```
☁  cloudevents.Event
```

```
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 99e4f4f6-08ff-4bff-acf1-47f61ded68c9
  time: 2020-04-07T16:16:00.000601161Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

11.4.1.1. ping ソースの削除

1. **PingSource** オブジェクトを削除します。

```
$ kn delete pingsources.sources.knative.dev test-ping-source
```

2. **event-display** サービスを削除します。

```
$ kn delete service.serving.knative.dev event-display
```

11.4.2. YAML ファイルを使用した ping ソースの作成

以下のセクションでは、YAML ファイルを使用して基本的な ping ソースを作成し、検証し、削除する方法を説明します。

前提条件

- Knative Serving および Eventing がインストールされている。

注記

以下の手順では、YAML ファイルを作成する必要があります。

サンプルで使用されたもので YAML ファイルの名前を変更する場合は、必ず対応する CLI コマンドを更新する必要があります。

手順

1. PingSource が機能していることを確認するには、受信メッセージをサービスのログにダンプする単純な Knative サービスを作成します。

   a. サンプル YAML を **service.yaml** という名前のファイルにコピーします。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
```

```
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

b. **service.yaml** ファイルを適用します。

```
$ oc apply --filename service.yaml
```

2. 要求する必要のある ping イベントのセットごとに、**PingSource** オブジェクトをイベントコンシューマーと同じ namespace に作成します。

a. サンプル YAML を **ping-source.yaml** という名前のファイルにコピーします。

```
apiVersion: sources.knative.dev/v1alpha2
kind: PingSource
metadata:
  name: test-ping-source
spec:
  schedule: "*/2 * * * *"
  jsonData: '{"message": "Hello world!"}'
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

b. **ping-source.yaml** ファイルを適用します。

```
$ oc apply --filename ping-source.yaml
```

3. 以下のコマンドを入力し、出力を検査して、コントローラーが正しくマップされていることを確認します。

```
$ oc get pingsource.sources.knative.dev test-ping-source -oyaml
```

出力例

```
apiVersion: sources.knative.dev/v1alpha2
kind: PingSource
metadata:
  annotations:
    sources.knative.dev/creator: developer
    sources.knative.dev/lastModifier: developer
  creationTimestamp: "2020-04-07T16:11:14Z"
  generation: 1
  name: test-ping-source
  namespace: default
  resourceVersion: "55257"
  selfLink: /apis/sources.knative.dev/v1alpha2/namespaces/default/pingsources/test-ping-source
  uid: 3d80d50b-f8c7-4c1b-99f7-3ec00e0a8164
spec:
  jsonData: '{ value: "hello" }'
  schedule: '*/2 * * * *'
```

```
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
      namespace: default
```

検証

シンク Pod のログを確認して、Kubernetes イベントが Knative イベントに送信されていることを確認できます。

デフォルトで、Knative サービスは、トラフィックが 60 秒以内に受信されない場合に Pod を終了します。本書の例では、新たに作成される Pod で各メッセージが確認されるように 2 分ごとにメッセージを送信する **PingSource** オブジェクトを作成します。

1. 作成された新規 Pod を監視します。

```
$ watch oc get pods
```

2. **Ctrl+C** を使用して Pod の監視をキャンセルし、作成された Pod のログを確認します。

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

出力例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 042ff529-240e-45ee-b40c-3a908129853e
  time: 2020-04-07T16:22:00.000791674Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

11.4.2.1. PingSource の削除

1. 以下のコマンドを入力してサービスを削除します。

```
$ oc delete --filename service.yaml
```

2. 以下のコマンドを入力して **PingSource** オブジェクトを削除します。

```
$ oc delete --filename ping-source.yaml
```

11.5. シンクバインディングの使用

シンクバインディングは、イベントプロデューサーまたは **イベントソース** を Knative サービスやアプリケーションなどのイベントコンシューマーまたは **イベントシンク** に接続するために使用されます。

重要

開発者がシンクバインディングを使用できるようにするには、クラスター管理者は、**bindings.knative.dev/include:"true"** を使用して **SinkBinding** オブジェクトに設定される namespace にラベルを付ける必要があります。

```
$ oc label namespace <namespace> bindings.knative.dev/include=true
```

11.5.1. Knative CLI によるシンクバインディングの使用

以下に、**kn** CLI を使用して シンクバインディングインスタンスを作成し、管理し、削除するために必要な手順を説明します。

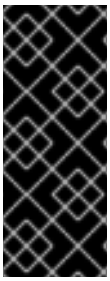
前提条件

- Knative Serving および Eventing がインストールされている。
- **kn** CLI がインストールされている。

注記

以下の手順では、YAML ファイルを作成する必要があります。

サンプルで使用されたもので YAML ファイルの名前を変更する場合は、必ず対応する CLI コマンドを更新する必要があります。



重要

開発者がシンクバインディングを使用できるようにするには、クラスター管理者は、**bindings.knative.dev/include:"true"** を使用して **SinkBinding** オブジェクトに設定される namespace にラベルを付ける必要があります。

```
$ oc label namespace <namespace> bindings.knative.dev/include=true
```

手順

1. シンクバインディングが正しく設定されていることを確認するには、受信メッセージをダンプする Knative イベント表示サービスまたはイベントシンクを作成します。

```
$ kn service create event-display --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

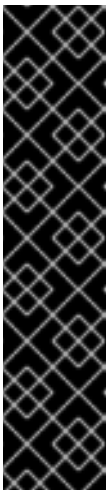
2. イベントをサービスに転送する **SinkBinding** オブジェクトを作成します。

```
$ kn source binding create bind-heartbeat --subject Job:batch/v1:app=heartbeat-cron --sink ksvc:event-display
```

3. CronJob を作成します。

a. **heartbeats-cronjob.yaml** という名前のファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: "true"
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/knative-eventing-sources-heartbeats:latest
              args:
                - --period=1
              env:
                - name: ONE_SHOT
                  value: "true"
                - name: POD_NAME
                  valueFrom:
                    fieldRef:
                      fieldPath: metadata.name
                - name: POD_NAMESPACE
                  valueFrom:
                    fieldRef:
                      fieldPath: metadata.namespace
```



重要

シンクバインディングを使用するには、**bindings.knative.dev/include=true** ラベルを Knative リソースに手動で追加する必要があります。

たとえば、このラベルを **CronJob** オブジェクトに追加するには、以下の行をジョブリソースの YAML 定義に追加します。

```
jobTemplate:
  metadata:
    labels:
      app: heartbeat-cron
      bindings.knative.dev/include: "true"
```

b. **heartbeats-cronjob.yaml** ファイルを作成した後に、これを適用します。

```
$ oc apply --filename heartbeats-cronjob.yaml
```

4. 以下のコマンドを入力し、出力を検査して、コントローラーが正しくマップされていることを確認します。

```
$ kn source binding describe bind-heartbeat
```

出力例

```
Name:         bind-heartbeat
Namespace:    demo-2
Annotations:  sources.knative.dev/creator=minikube-user,
sources.knative.dev/lastModifier=minikub ...
Age:          2m
Subject:
  Resource:   job (batch/v1)
  Selector:
    app:      heartbeat-cron
Sink:
  Name:       event-display
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE     AGE REASON
  ++ Ready    2m
```

検証

メッセージダンパー機能ログを確認して、Kubernetes イベントが Knative イベントシンクに送信されていることを確認できます。

- メッセージダンパー機能ログを表示します。

```
$ oc get pods
```

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

出力例

```
☁   cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
```

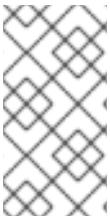
```
  "id": 1,
  "label": ""
}
```

11.5.2. YAML メソッドでのシンクバインディングの使用

以下に、YAML ファイルを使用してシンクバインディングインスタンスを作成し、管理し、削除するために必要な手順を説明します。

前提条件

- Knative Serving および Eventing がインストールされている。



注記

以下の手順では、YAML ファイルを作成する必要があります。

サンプルで使用されたもので YAML ファイルの名前を変更する場合は、必ず対応する CLI コマンドを更新する必要があります。

重要

開発者がシンクバインディングを使用できるようにするには、クラスター管理者は、**bindings.knative.dev/include:"true"** を使用して **SinkBinding** オブジェクトに設定される namespace にラベルを付ける必要があります。

```
$ oc label namespace <namespace> bindings.knative.dev/include=true
```

手順

1. シンクバインディングが正しく設定されていることを確認するには、受信メッセージをダンプする Knative イベント表示サービスまたはイベントシンクを作成します。
   a. 以下のサンプル YAML を **service.yaml** という名前のファイルにコピーします。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

   b. **service.yaml** ファイルを作成した後に、これを適用します。

```
$ oc apply -f service.yaml
```

2. イベントをサービスに転送する **SinkBinding** オブジェクトを作成します。
   a. **sinkbinding.yaml** という名前のファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: sources.knative.dev/v1alpha1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: batch/v1
    kind: Job 1
    selector:
      matchLabels:
        app: heartbeat-cron

  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1 この例では、ラベル **app: heartbeat-cron** を指定したジョブがイベントシンクにバインドされます。

b. **sinkbinding.yaml** ファイルを作成した後に、これを適用します。

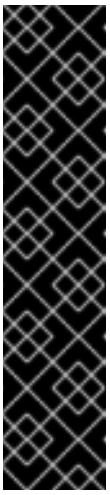
```
$ oc apply -f sinkbinding.yaml
```

3. **CronJob** オブジェクトを作成します。

a. **heartbeats-cronjob.yaml** という名前のファイルを作成し、以下のサンプルコードをこれにコピーします。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: "true"
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/knative-eventing-sources-heartbeats:latest
              args:
                - --period=1
              env:
                - name: ONE_SHOT
```

```
      value: "true"
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
```



重要

シンクバインディングを使用するには、**bindings.knative.dev/include=true** ラベルを Knative リソースに手動で追加する必要があります。

たとえば、このラベルを CronJob インスタンスに追加するには、以下の行を **Job** リソースの YAML 定義に追加します。

```
jobTemplate:
  metadata:
    labels:
      app: heartbeat-cron
      bindings.knative.dev/include: "true"
```

b. **heartbeats-cronjob.yaml** ファイルを作成した後に、これを適用します。

```
$ oc apply -f heartbeats-cronjob.yaml
```

4. 以下のコマンドを入力し、出力を検査して、コントローラーが正しくマップされていることを確認します。

```
$ oc get sinkbindings.sources.knative.dev bind-heartbeat -oyaml
```

出力例

```
spec:
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
      namespace: default
  subject:
    apiVersion: batch/v1
    kind: Job
    namespace: default
    selector:
      matchLabels:
        app: heartbeat-cron
```

検証

メッセージダンパー機能ログを確認して、Kubernetes イベントが Knative イベントシンクに送信されていることを確認できます。

1. メッセージダンパー機能ログを表示します。

```
$ oc get pods
```

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

出力例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

第12章 ネットワーク

12.1. OPENSHIFT SERVERLESS でのサービスメッシュの使用

OpenShift Serverless でサービスメッシュを使用すると、開発者はデフォルトの Kourier 実装で OpenShift Serverless を使用する際にサポートされない追加のネットワークおよびルーティングオプションを設定できます。これらのオプションには、TLS 証明書を使用したカスタムドメインの設定、および JSON Web トークン認証の使用が含まれます。

前提条件

1. OpenShift Serverless Operator および Knative Serving をインストールします。
2. Red Hat OpenShift Service Mesh をインストールします。

手順

1. **default** namespace をメンバーとして ServiceMeshMemberRoll に追加します。

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members:
    - default
```

重要

knative-serving および **knative-serving-ingress** などのシステム namespace の Pod へのサイドカー挿入の追加はサポートされていません。

2. Knative システム Pod から Knative サービスへのトラフィックフローを許可するネットワークポリシーを作成します。
   a. **serving.knative.openshift.io/system-namespace=true** ラベルを **knative-serving** namespace に追加します。

```
$ oc label namespace knative-serving serving.knative.openshift.io/system-
namespace=true
```

   b. **serving.knative.openshift.io/system-namespace=true** ラベルを **knative-serving-ingress** namespace に追加します。

```
$ oc label namespace knative-serving-ingress serving.knative.openshift.io/system-
namespace=true
```

   c. 以下の **NetworkPolicy** リソースを YAML ファイルにコピーします。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
    name: allow-from-serving-system-namespace
    namespace: default
  spec:
    ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            serving.knative.openshift.io/system-namespace: "true"
    podSelector: {}
    policyTypes:
    - Ingress
```

d. **NetworkPolicy** リソースを適用します。

```
$ oc apply -f <filename>
```

12.1.1. Knative サービスのサイドカーコンテナー挿入の有効化

アノテーションを **Service** リソース YAML ファイルに追加し、Knative サービスのサイドカー挿入を有効にすることができます。

手順

1. **sidecar.istio.io/inject="true"** アノテーションを **Service** リソースに追加します。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello-example-1
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 1
    spec:
      containers:
      - image: docker.io/openshift/hello-openshift
        name: container
```

1 **sidecar.istio.io/inject="true"** アノテーションを追加します。

2. **Service** リソースの YAML ファイルを適用します。

```
$ oc apply -f <filename>
```

12.1.2. 追加リソース

- Red Hat OpenShift Service Mesh の詳細は、 Red Hat OpenShift Service Mesh architecture を参照してください。

12.2. サービスメッシュおよび OPENSHIFT SERVERLESS での JSON WEB トークン認証の使用

Knative サービスの JSON Web Token (JWT) 認証を有効にするには、有効な JWT を使用した要求のみを許可するサーバーレスアプリケーションの namespace にポリシーを作成します。

前提条件

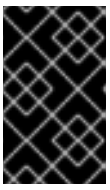
- OpenShift Serverless をインストールします。
- Red Hat OpenShift Service Mesh をインストールします。
- Knative サービスのサイドカーコンテナー挿入を有効化を含め、OpenShift Serverless でサービスメッシュ を設定します。

重要

knative-serving および **knative-serving-ingress** などのシステム namespace の Pod へのサイドカー挿入の追加はサポートされていません。

手順

1. 以下の **Policy** リソースを YAML ファイルにコピーします。



重要

パスの **/metrics** および **/healthz** は、**knative-serving** namespace のシステム Pod からアクセスされるため、**excludedPaths** に組み込まれる必要があります。

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default
spec:
  origins:
  - jwt:
      issuer: testing@secure.istio.io
      jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
1.6/security/tools/jwt/samples/jwks.json"
      triggerRules:
      - excludedPaths:
        - prefix: /metrics
        - prefix: /healthz
  principalBinding: USE_ORIGIN
```

2. **Policy** リソースの YAML ファイルを適用します。

```
$ oc apply -f <filename>
```

検証

1. **curl** 要求トを使用して Knative サービス URL を取得しようとすると、これは拒否されます。

```
$ curl http://hello-example-default.apps.mycluster.example.com/
```

出力例

```
Origin authentication failed.
```

2. 有効な JWT で要求を確認します。

   a. 以下のコマンドを入力して、有効な JWT トークンを取得します。

   ```
   $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --decode -
   ```

   b. **curl** 要求ヘッダーで有効なトークンを使用してサービスにアクセスします。

   ```
   $ curl http://hello-example-default.apps.mycluster.example.com/ -H "Authorization: Bearer $TOKEN"
   ```

   これで要求が許可されます。

   出力例

   ```
   Hello OpenShift!
   ```

12.2.1. 追加リソース

- Red Hat OpenShift Service Mesh architecture を参照してください。
- Knative サービスの検証および **curl** 要求の使用についての詳細は、サーバーレスアプリケーションのデプロイメントの確認 を参照してください。

12.3. サービスメッシュによる KNATIVE サービスのカスタムドメインの使用

デフォルトで、Knative サービスには固定されたドメイン形式があります。

```
<application_name>-<namespace>.<openshift_cluster_domain>
```

サービスをプライベートサービスとして設定し、必要なサービスメッシュリソースを作成して、Knative サービスのドメインをカスタマイズできます。

前提条件

- OpenShift Serverless Operator および Knative Serving をインストールします。
- Red Hat OpenShift Service Mesh をインストールします。
- OpenShift Serverless でのサービスメッシュの使用 の設定手順を完了します。
- 既存の Knative サービスのカスタムドメインを設定するか、または新規サンプルサービスを作成できます。新しいサービスを作成するには、サーバーレスアプリケーションの作成および管理 を参照してください。

12.3.1. クラスター可用性の cluster-local への設定

デフォルトで、Knative サービスはパブリック IP アドレスに公開されます。パブリック IP アドレスに公開されているとは、Knative サービスがパブリックアプリケーションであり、一般にアクセス可能な URL があることを意味します。

一般にアクセス可能な URL は、クラスター外からアクセスできます。ただし、開発者は **プライベートサービス** と呼ばれるクラスター内からのみアクセス可能なバックエンドサービスをビルドする必要がある場合があります。開発者は、クラスター内の個々のサービスに **serving.knative.dev/visibility=cluster-local** ラベルを使用してラベル付けし、それらをプライベートにすることができます。

手順

- **serving.knative.dev/visibility=cluster-local** ラベルを追加して、サービスの可視性を設定します。

```
$ oc label ksvc <service_name> serving.knative.dev/visibility=cluster-local
```

検証

- 以下のコマンドを入力して出力を確認し、サービスの URL の形式が **http://<service_name>.<namespace>.svc.cluster.local** であることを確認します。

```
$ oc get ksvc
```

出力例

```
NAME    URL                                        LATESTCREATED   LATESTREADY   READY   REASON
hello   http://hello.default.svc.cluster.local     hello-tx2g7     hello-tx2g7   True
```

12.3.2. 必要なサービスメッシュリソースの作成

手順

1. トラフィックを受け入れる Istio ゲートウェイを作成します。
   a. YAML ファイルを作成し、以下の YAML をこれにコピーします。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: default-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
```

```
    protocol: HTTP
  hosts:
  - "*"
```

b. YAML ファイルを適用します。

```
$ oc apply -f <filename>
```

2. Istio **VirtualService** オブジェクトを作成し、ホストヘッダーを再作成します。

a. YAML ファイルを作成し、以下の YAML をこれにコピーします。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello
spec:
  hosts:
  - custom-ksvc-domain.example.com
  gateways:
  - default-gateway
  http:
  - rewrite:
      authority: hello.default.svc (1)
    route:
    - destination:
        host: hello.default.svc (2)
        port:
          number: 80
```

(1) (2) Knative サービスは、**<service_name>.<namespace>.svc** 形式の Knative サービスです。

b. YAML ファイルを適用します。

```
$ oc apply -f <filename>
```

3. Istio **ServiceEntry** オブジェクトを作成します。これは、Kourier がサービスメッシュ外にあるため、OpenShift Serverless に必要です。

a. YAML ファイルを作成し、以下の YAML をこれにコピーします。

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: hello.default.svc
spec:
  hosts:
  - hello.default.svc (1)
  location: MESH_EXTERNAL
  endpoints:
  - address: kourier-internal.knative-serving-ingress.svc
  ports:
  - number: 80
```

```
    name: http
    protocol: HTTP
  resolution: DNS
```

1. Knative サービスは、**<service_name>.<namespace>.svc** 形式の Knative サービスです。

b. YAML ファイルを適用します。

```
$ oc apply -f <filename>
```

4. **VirtualService** オブジェクトを参照する OpenShift Container Platform ルートを作成します。

a. YAML ファイルを作成し、以下の YAML をこれにコピーします。

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: hello
  namespace: istio-system 1
spec:
  host: custom-ksvc-domain.example.com
  port:
    targetPort: 8080
  to:
    kind: Service
    name: istio-ingressgateway
```

1. OpenShift Container Platform ルートは ServiceMeshControlPlane と同じ namespace に作成される必要があります。この例では、ServiceMeshControlPlane は **istio-system** namespace にデプロイされます。

a. YAML ファイルを適用します。

```
$ oc apply -f <filename>
```

12.3.3. カスタムドメインを使用したサービスへのアクセス

手順

1. **curl** 要求の **Host** ヘッダーを使用してカスタムドメインにアクセスします。以下は例になります。

```
$ curl -H "Host: custom-ksvc-domain.example.com" http://<ip_address>
```

ここで、**<ip_address>** は、OpenShift Container Platform Ingress ルーターが公開される IP アドレスになります。

出力例

```
Hello OpenShift!
```

12.3.4. 関連資料

- Red Hat OpenShift Service Mesh の詳細は、Understanding Red Hat OpenShift Service Mesh を参照してください。

第13章 OPENSHIFT SERVERLESS でのメータリングの使用

クラスター管理者として、メータリングを使用して OpenShift Serverless クラスターで実行されている内容を分析できます。

OpenShift Container Platform のメータリングについての詳細は、メータリングの概要 を参照してください。

13.1. メータリングのインストール

OpenShift Container Platform でのメータリングのインストールについての詳細は、メータリングのインストール を参照してください。

13.2. KNATIVE SERVING メータリングのデータソース

以下の **ReportDataSources** は、Knative Serving を OpenShift Container Platform メータリングで使用する方法についての例です。

13.2.1. Knative Serving での CPU 使用状況のデータソース

このデータソースは、レポート期間における Knative サービスごとに使用される累積された CPU の秒数を示します。

サンプル YAML ファイル

```
apiVersion: metering.openshift.io/v1
kind: ReportDataSource
metadata:
  name: knative-service-cpu-usage
spec:
  prometheusMetricsImporter:
    query: >
      sum
          by(namespace,
             label_serving_knative_dev_service,
             label_serving_knative_dev_revision)
          (

label_replace(rate(container_cpu_usage_seconds_total{container!="POD",container!="",pod!=""}
[1m]), "pod", "$1", "pod", "(.*)")
            *
            on(pod, namespace)
            group_left(label_serving_knative_dev_service, label_serving_knative_dev_revision)
            kube_pod_labels{label_serving_knative_dev_service!=""}
          )
```

13.2.2. Knative Serving でのメモリー使用状況のデータソース

このデータソースは、レポート期間における Knative サービスごとの平均メモリー消費量を示します。

サンプル YAML ファイル

```
apiVersion: metering.openshift.io/v1
```

```
kind: ReportDataSource
metadata:
  name: knative-service-memory-usage
spec:
  prometheusMetricsImporter:
    query: >
      sum
          by(namespace,
             label_serving_knative_dev_service,
             label_serving_knative_dev_revision)
          (
            label_replace(container_memory_usage_bytes{container!="POD", container!="",pod!=""},
"pod", "$1", "pod", "(.*)")
            *
            on(pod, namespace)
            group_left(label_serving_knative_dev_service, label_serving_knative_dev_revision)
            kube_pod_labels{label_serving_knative_dev_service!=""}
          )
```

13.2.3. Knative Serving メータリングのデータソースの適用

手順

- **ReportDataSources** リソースを YAML ファイルとして適用します。

```
$ oc apply -f <datasource_name>.yaml
```

コマンドの例

```
$ oc apply -f knative-service-memory-usage.yaml
```

13.3. KNATIVE SERVING メータリングのクエリー

以下の **ReportQuery** リソースは、提供されるサンプルの **DataSources** を参照します。

13.3.1. Knative Serving での CPU 使用状況のクエリー

サンプル YAML ファイル

```
apiVersion: metering.openshift.io/v1
kind: ReportQuery
metadata:
  name: knative-service-cpu-usage
spec:
  inputs:
  - name: ReportingStart
    type: time
  - name: ReportingEnd
    type: time
  - default: knative-service-cpu-usage
    name: KnativeServiceCpuUsageDataSource
    type: ReportDataSource
```

```
  columns:
  - name: period_start
    type: timestamp
    unit: date
  - name: period_end
    type: timestamp
    unit: date
  - name: namespace
    type: varchar
    unit: kubernetes_namespace
  - name: service
    type: varchar
  - name: data_start
    type: timestamp
    unit: date
  - name: data_end
    type: timestamp
    unit: date
  - name: service_cpu_seconds
    type: double
    unit: cpu_core_seconds
  query: |
    SELECT
      timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart| prestoTimestamp |}'
AS period_start,
      timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}' AS
period_end,
      labels['namespace'] as project,
      labels['label_serving_knative_dev_service'] as service,
      min("timestamp") as data_start,
      max("timestamp") as data_end,
      sum(amount * "timeprecision") AS service_cpu_seconds
    FROM {| dataSourceTableName .Report.Inputs.KnativeServiceCpuUsageDataSource |}
    WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart
| prestoTimestamp |}'
    AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd |
prestoTimestamp |}'
    GROUP BY labels['namespace'],labels['label_serving_knative_dev_service']
```

13.3.2. Knative Serving でのメモリー使用状況のクエリー

サンプル YAML ファイル

```
apiVersion: metering.openshift.io/v1
kind: ReportQuery
metadata:
  name: knative-service-memory-usage
spec:
  inputs:
  - name: ReportingStart
    type: time
  - name: ReportingEnd
    type: time
  - default: knative-service-memory-usage
    name: KnativeServiceMemoryUsageDataSource
```

```
    type: ReportDataSource
  columns:
  - name: period_start
    type: timestamp
    unit: date
  - name: period_end
    type: timestamp
    unit: date
  - name: namespace
    type: varchar
    unit: kubernetes_namespace
  - name: service
    type: varchar
  - name: data_start
    type: timestamp
    unit: date
  - name: data_end
    type: timestamp
    unit: date
  - name: service_usage_memory_byte_seconds
    type: double
    unit: byte_seconds
  query: |
    SELECT
      timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart| prestoTimestamp |}'
AS period_start,
      timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}' AS
period_end,
      labels['namespace'] as project,
      labels['label_serving_knative_dev_service'] as service,
      min("timestamp") as data_start,
      max("timestamp") as data_end,
      sum(amount * "timeprecision") AS service_usage_memory_byte_seconds
    FROM {| dataSourceTableName .Report.Inputs.KnativeServiceMemoryUsageDataSource |}
    WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart
| prestoTimestamp |}'
    AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd |
prestoTimestamp |}'
    GROUP BY labels['namespace'],labels['label_serving_knative_dev_service']
```

13.3.3. Knative Serving メータリングのクエリーの適用

- クエリーを YAML ファイルとして適用します。

  ```
  $ oc apply -f <query_name>.yaml
  ```

  コマンドの例

  ```
  $ oc apply -f knative-service-memory-usage.yaml
  ```

13.4. KNATIVE SERVING のメータリングレポート

Report リソースを作成し、Knative Serving に対してメータリングレポートを実行できます。レポートを実行する前に、レポート期間の開始日と終了日を指定するために、**Report** リソース内で入力パラメーターを変更する必要があります。

サンプル YAML ファイル

```
apiVersion: metering.openshift.io/v1
kind: Report
metadata:
  name: knative-service-cpu-usage
spec:
  reportingStart: '2019-06-01T00:00:00Z' 1
  reportingEnd: '2019-06-30T23:59:59Z' 2
  query: knative-service-cpu-usage 3
runImmediately: true
```

1. レポートの開始日 (ISO 8601 形式)。
2. レポートの終了日 (ISO 8601 形式)。
3. CPU 使用状況レポートの **knative-service-cpu-usage**、またはメモリー使用状況レポートの **knative-service-memory-usage** のいずれか。

13.4.1. メータリングレポートの実行

1. これを YAML ファイルとして適用してレポートを実行します。

```
$ oc apply -f <report_name>.yaml
```

2. 以下のコマンドを入力してレポートを確認できます。

```
$ oc get report
```

出力例

```
NAME                        QUERY                       SCHEDULE   RUNNING    FAILED   LAST
REPORT TIME       AGE
knative-service-cpu-usage   knative-service-cpu-usage              Finished           2019-06-
30T23:59:59Z   10h
```

第14章 統合

14.1. サーバーレスアプリケーションでの NVIDIA GPU リソースの使用

Nvidia は、OpenShift Container Platform での GPU リソースの実験的な使用をサポートします。OpenShift Container Platform での GPU リソースの設定に関する詳細は、OpenShift Container Platform on NVIDIA GPU accelerated clusters を参照してください。

OpenShift Container Platform クラスターについて GPU リソースが有効にされた後に、**kn** CLI を使用して Knative サービスの GPU 要件を指定できます。

手順

kn を使用して Knative サービスを作成する際に GPU リソース要件を指定できます。

1. サービスを作成します。
2. **nvidia.com/gpu=1** を使用して、GPU リソース要件の制限を **1** に設定します。

```
$ kn service create hello --image docker.io/knativesamples/hellocuda-go --limit
nvidia.com/gpu=1
```

GPU リソース要件の制限が **1** の場合、サービスには専用の GPU リソースが 1 つ必要です。サービスは、GPU リソースを共有しません。GPU リソースを必要とするその他のサービスは、GPU リソースが使用されなくなるまで待機する必要があります。

1 GPU の制限は、1 GPU リソースの使用を超えるアプリケーションが制限されることも意味します。サービスが 2 つ以上の GPU リソースを要求する場合、これは GPU リソース要件を満たしているノードにデプロイされます。

kn を使用した Knative サービスの GPU 要件の更新

- サービスを更新します。**nvidia.com/gpu=3** を使用して、GPU リソース要件の制限を **3** に変更します。

```
$ kn service update hello --limit nvidia.com/gpu=3
```

14.1.1. 追加リソース

- 制限についての詳細は、Setting resource quotas for extended resources を参照してください。