OpenShift Container Platform 4.6

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: 2023-02-23

OpenShift Container Platform 4.6 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 リリースノート

リリースノートには、新機能、非推奨機能、互換性を損なう変更、既知の問題に関する情報が記載されています。以下のリリースノートは、OpenShift Container Platform 上の最新の OpenShift Serverless リリースに適用されます。

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

注記

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 コンポーネントリリースの詳細は、Knative ブログ を参照してください。

1.1. API バージョンについて

API バージョンは、OpenShift Serverless の特定の機能およびカスタムリソースの開発状況を示す重要な指標です。正しい API バージョンを使用していないリソースをクラスター上に作成すると、デプロイメントで問題が発生する可能性があります。

OpenShift Serverless Operator は、最新バージョンを使用するように非推奨の API を使用する古いリソースを自動的にアップグレードします。たとえば、**v1beta1** などの古いバージョンの **ApiServerSource** API を使用するクラスターにリソースを作成した場合、OpenShift Serverless Operator はこれらのリソースを自動的に更新し、これが利用可能な場合に API の **v1** バージョンを使用するように、**v1beta1** バージョンは非推奨になりました。

非推奨となった古いバージョンは、今後のリリースで削除される可能性があります。API の非推奨バージョンを使用すると、リソースが失敗することはありません。ただし、削除された API のバージョンを使用しようとすると、リソースが失敗します。問題を回避するために、マニフェストが最新バージョンを使用するように更新されていることを確認します。

1.2. 一般提供およびテクノロジープレビュー機能

一般提供 (GA) の機能は完全にサポートされており、実稼働での使用に適しています。テクノロジープレビュー (TP) 機能は実験的な機能であり、本番環境での使用を目的としたものではありません。TP 機能の詳細については、Red Hat Customer Portal の Technology Preview のサポート範囲 を参照してください。

次の表は、どの OpenShift Serverless 機能が GA であり、どの機能が TP であるかに関する情報を提供します。

表1.1 一般提供およびテクノロジープレビュー機能トラッカー

機能	1.23	1.24	1.25
kn func	TP	TP	TP
サービスメッシュ mTLS	GA	GA	GA
emptyDir ボリューム	GA	GA	GA
HTTPS リダイレクト	GA	GA	GA

機能	1.23	1.24	1.25
Kafka ブローカー	TP	TP	GA
Kafka シンクコ	TP	TP	GA
Knative サービスの init コンテナーのサポート	TP	GA	GA
Knative サービスの PVC サポート	TP	TP	TP
内部トラフィックの TLS	-	-	TP

1.3. 非推奨および削除された機能

以前のリリースで一般提供 (GA) またはテクノロジープレビュー (TP) であった一部の機能は、非推奨または削除されました。非推奨の機能は依然として OpenShift Serverless に含まれており、引き続きサポートされますが、本製品の今後のリリースで削除されるため、新規デプロイメントでの使用は推奨されません。

OpenShift Serverless で非推奨となり、削除された主な機能の最新の一覧については、以下の表を参照してください。

表1.2 非推奨および削除機能のトラッカー

機能	1.20	1.21	1.22 から 1.25
KafkaBinding API	非推奨	非推奨	廃止
kn func emit (1.21+では **kn func invoke**)	非推奨	廃止	廃止

1.4. RED HAT OPENSHIFT SERVERLESS 1.25.0 のリリースノート

OpenShift Serverless 1.25.0 が公開されました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.4.1. 新機能

- OpenShift Serverless は Knative Serving 1.4 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 1.4 を使用するようになりました。
- OpenShift Serverless は Kourier 1.4 を使用するようになりました。
- OpenShift Serverless は Knative (**kn**) CLI 1.4 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 1.4 を使用するようになりました。
- **kn func** CLI プラグインは **func** 1.7.0 を使用するようになりました。

- 関数を作成およびデプロイするための統合開発環境 (IDE) プラグインが、Visual Studio Code および IntelliJ で利用できるようになりました。
- Knative Kafka ブローカーが一般提供されるようになりました。Knative Kafka ブローカーは、Apache Kafka を直接ターゲットとする、Knative ブローカー API の高性能な実装です。MT-Channel-Broker ではなく、Knative Kafka ブローカーを使用することをお勧めします。
- Knative Kafka シンクが一般提供されるようになりました。**KafkaSink** は **CloudEvent** を取得し、Apache Kafka トピックに送信します。イベントは、構造化コンテンツモードまたはバイナリーコンテンツモードのいずれかで指定できます。
- 内部トラフィックの TLS の有効化がテクノロジープレビューとして利用可能になりました。

1.4.2. 修正された問題

- 以前のバージョンでは、Knative Serving には liveness プローブの失敗後にコンテナーが再起動された場合に readiness プローブが失敗する問題がありました。この問題は修正されています。

1.4.3. 既知の問題

- 連邦情報処理標準 (FIPS) モードは、Kafka ブローカー、Kafka ソース、および Kafka シンクに対して無効になっています。
- **SinkBinding** オブジェクトは、サービスのカスタムリビジョン名をサポートしません。

関連情報

- TLS 認証の設定

1.5. RED HAT OPENSHIFT SERVERLESS 1.24.0 のリリースノート

OpenShift Serverless 1.24.0 が公開されました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.5.1. 新機能

- OpenShift Serverless は Knative Serving 1.3 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 1.3 を使用するようになりました。
- OpenShift Serverless は Kourier 1.3 を使用するようになりました。
- OpenShift Serverless は、Knative (**kn**) CLI 1.3 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 1.3 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.24 を使用するようになりました。
- Knative サービスの初期化コンテナーのサポートが一般提供 (GA) になりました。
- OpenShift Serverless ロジックが開発者プレビューとして利用できるようになりました。これにより、サーバーレスアプリケーションを管理するための宣言型ワークフローモデルを定義できます。

- OpenShift Serverless で Cost Management Service を使用できるようになりました。

1.5.2. 修正された問題

- OpenShift Serverless を Red Hat OpenShift Service Mesh と統合すると、クラスターに存在するシークレットが多すぎると、起動時に **net-istio-controller** Pod がメモリー不足になります。シークレットフィルターリングを有効にできるようになりました。これにより、**net-istio-controller** は、**networking.internal.knative.dev/certificate-uid** ラベルを持つシークレットのみを考慮するようになり、必要なメモリー量が削減されます。

- OpenShift Serverless 機能テクノロジープレビューは、デフォルトで Cloud Native Buildpacks を使用してコンテナーイメージをビルドするようになりました。

1.5.3. 既知の問題

- 連邦情報処理標準 (FIPS) モードは、Kafka ブローカー、Kafka ソース、および Kafka シンクに対して無効になっています。

- OpenShift Serverless 1.23 では、KafkaBindings および kafka **-binding** Webhook のサポートが削除されました。ただし、既存の **kafkabindings.webhook.kafka.sources.knative.dev MutatingWebhookConfiguration** が残り、もはや存在しない **kafka-source-webhook** サービスを指している可能性があります。
  クラスター上の KafkaBindings の特定の仕様については、**kafkabindings.webhook.kafka.sources.knative.dev MutatingWebhookConfiguration** を設定して、Webhook を介して Deployment、Knative Services、または Jobs などのさまざまなリソースに作成および更新イベントを渡すことができます。その後失敗します。

  この問題を回避するには、OpenShift Serverless 1.23 にアップグレードした後、クラスターから **kafkabindings.webhook.kafka.sources.knative.dev MutatingWebhookConfiguration** を手動で削除します。

  ```
  $ oc delete mutatingwebhookconfiguration kafkabindings.webhook.kafka.sources.knative.dev
  ```

1.6. RED HAT OPENSHIFT SERVERLESS 1.23.0 のリリースノート

OpenShift Serverless 1.23.0 が公開されました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.6.1. 新機能

- OpenShift Serverless は Knative Serving 1.2 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 1.2 を使用するようになりました。
- OpenShift Serverless は Kourier 1.2 を使用するようになりました。
- OpenShift Serverless は Knative (**kn**) CLI 1.2 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 1.2 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.24 を使用するようになりました。

- Kafka ブローカーで **kafka.eventing.knative.dev/external.topic** アノテーションを使用できるようになりました。このアノテーションを使用すると、ブローカー自体の内部トピックを作成する代わりに、既存の外部管理トピックを使用できます。

- **kafka-ch-controller** および **kafka-webhook** Kafka コンポーネントが存在しなくなりました。これらのコンポーネントは **kafka-webhook-eventing** コンポーネントに置き換えられました。

- OpenShift Serverless Functions Technology Preview は、デフォルトで Source-to-Image (S2I) を使用してコンテナーイメージをビルドするようになりました。

1.6.2. 既知の問題

- 連邦情報処理標準 (FIPS) モードは、Kafka ブローカー、Kafka ソース、および Kafka シンクに対して無効になっています。

- Kafka ブローカーを含む namespace を削除する場合、ブローカーの **auth.secret.ref.name** シークレットがブローカーの前に削除されると、namespace ファイナライザーが削除されない可能性があります。

- 多数の Knative サービスで OpenShift Serverless を実行すると、Knative アクティベーター Pod がデフォルトのメモリー制限である 600MB 近くで実行される可能性があります。これらの Pod は、メモリー消費がこの制限に達すると再起動される可能性があります。アクティベーターデプロイメントの要求と制限は、**KnativeServing** カスタムリソースを変更することで設定できます。

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  deployments:
  - name: activator
    resources:
    - container: activator
      requests:
        cpu: 300m
        memory: 60Mi
      limits:
        cpu: 1000m
        memory: 1000Mi
```

- 関数のローカルビルド戦略として CloudNativeBuildpack を使用している場合、**kn func** は podman を自動的に起動したり、リモートデーモンへの SSH トンネルを使用したりすることはできません。これらの問題の回避策は、関数をデプロイする前に、ローカル開発コンピューターで Docker または podman デーモンを既に実行していることです。

- 現時点で、クラスター上の関数ビルドが Quarkus および Golang ランタイムで失敗します。これらは Node、Typescript、Python、および Springboot ランタイムで正常に機能します。

関連情報

- Source-to-Image

1.7. RED HAT OPENSHIFT SERVERLESS 1.22.0 のリリースノート

OpenShift Serverless 1.22.0 が公開されました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.7.1. 新機能

- OpenShift Serverless は Knative Serving 1.1 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 1.1 を使用するようになりました。
- OpenShift Serverless は Kourier 1.1 を使用するようになりました。
- OpenShift Serverless は Knative (**kn**) CLI 1.1 を使用するようになりました。
- OpenShift Serverless は KnativeKafka1.1 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.23 を使用するようになりました。
- Knative サービスの初期コンテナーサポートがテクノロジープレビューとして利用できるようになりました。
- Knative サービスの永続ボリュームクレーム (PVC) サポートが、テクノロジープレビューとして利用できるようになりました。
- **knative-serving**、**knative-serving-ingress**、**knative-eventing**、および **knative-kafka** システムネームボックスに、デフォルトで **knative.openshift.io/part-of:"openshift-serverless"** ラベルが付けられるようになりました。
- **Knative Eventing-Kafka Broker/Trigger** ダッシュボードが追加されました。これにより、Web コンソールで Kafka ブローカーとトリガーメトリックを視覚化できます。
- **Knative Eventing-KafkaSink** ダッシュボードが追加されました。これにより、Web コンソールで KafkaSink メトリックを視覚化できます。
- **Knative Eventing-Broker/Trigger** ダッシュボードは、**Knative Eventing-Channel-based Broker/Trigger** と呼ばれるようになりました。
- **knative.openshift.io/part-of: " openshift-serverless "** ラベルが、**knative.openshift.io/system-namespace** ラベルに置き換わりました。
- Knative Serving YAML 設定ファイルの命名スタイルがキャメルケース (**ExampleName**) からハイフンスタイル (**example-name**) に変更されました。このリリース以降、Knative Serving YAML 設定ファイルを作成または編集するときは、ハイフンスタイルの表記を使用してください。

1.7.2. 既知の問題

- 連邦情報処理標準 (FIPS) モードは、Kafka ブローカー、Kafka ソース、および Kafka シンクに対して無効になっています。

1.8. RED HAT OPENSHIFT SERVERLESS 1.21.0 のリリースノート

OpenShift Serverless 1.21.0 が利用可能になりました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.8.1. 新機能

- OpenShift Serverless は Knative Serving 1.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 1.0 を使用するようになりました。
- OpenShift Serverless は Kourier 1.0 を使用するようになりました。
- OpenShift Serverless は、Knative (**kn**) CLI 1.0 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 1.0 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.21 を使用するようになりました。
- Kafka シンクがテクノロジープレビューとして利用できるようになりました。
- Knative オープンソースプロジェクトは、camel-cased 設定キーを廃止し、kebab-cased キーを一貫して使用することを支持し始めました。その結果、OpenShift Serverless 1.18.0 リリースノートで前述した **defaultExternalScheme** キーは非推奨になり、**default-external-scheme** キーに置き換えられました。キーの使用方法は同じです。

1.8.2. 修正された問題

- OpenShift Serverless 1.20.0 では、サービスにイベントを送信するための **kn event send** の使用に影響するイベント配信の問題がありました。この問題は修正されています。
- OpenShift Serverless 1.20.0 (**func**0.20) では、**http** テンプレートを使用して作成された TypeScript 関数をクラスターにデプロイできませんでした。この問題は修正されています。
- OpenShift Serverless 1.20.0 (**func** 0.20) では、**gcr.io** レジストリーを使用した関数のデプロイがエラーで失敗しました。この問題は修正されています。
- OpenShift Serverless 1.20.0 (**func** 0.20) では、**kn func create** コマンドを使用して Springboot 関数プロジェクトディレクトリーを作成してから、**kn func build** コマンドを実行するとエラーメッセージが表示されて失敗しました。この問題は修正されています。
- OpenShift Serverless 1.19.0 (**func** 0.19) では、一部のランタイムが podman を使用して関数をビルドできませんでした。この問題は修正されています。

1.8.3. 既知の問題

- 現在、ドメインマッピングコントローラーは、現在サポートされていないパスを含むブローカーの URI を処理できません。
  つまり、**DomainMapping** カスタムリソース (CR) を使用してカスタムドメインをブローカーにマップする場合は、ブローカーの入力サービスを使用して **DomainMapping** CR を設定し、ブローカーの正確なパスをカスタムドメインに追加する必要があります。

  DomainMappingCR の例

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: <domain-name>
  namespace: knative-eventing
spec:
  ref:
```

```
name: broker-ingress
kind: Service
apiVersion: v1
```

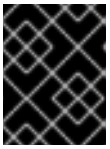
その場合、ブローカーの URI は **<domain-name>/<broker-namespace>/<broker-name>** になります。

1.9. RED HAT OPENSHIFT SERVERLESS 1.20.0 のリリースノート

OpenShift Serverless 1.20.0 が利用可能になりました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.9.1. 新機能

- OpenShift Serverless は Knative Serving 0.26 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.26 を使用するようになりました。
- OpenShift Serverless は Kourier 0.26 を使用するようになりました。
- OpenShift Serverless は、Knative (**kn**) CLI 0.26 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 0.26 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.20 を使用するようになりました。
- Kafka ブローカーがテクノロジープレビュー機能として利用可能になりました。



重要

現在テクノロジープレビューにある Kafka ブローカーは、FIPS ではサポートされていません。

- **kn event** プラグインがテクノロジープレビュー機能として利用可能になりました。
- **knservicecreate** コマンドの **--min-scale** フラグと **--max-scale** フラグは廃止されました。代わりに、**-scale-min** フラグと **--scale-max** フラグを使用してください。

1.9.2. 既知の問題

- OpenShift Serverless は、HTTPS を使用するデフォルトアドレスで Knative サービスをデプロイします。クラスター内のリソースにイベントを送信する場合、送信側にはクラスターの認証局 (CA) が設定されていません。これにより、クラスターがグローバルに受け入れ可能な証明書を使用しない限り、イベント配信は失敗します。
  たとえば、一般にアクセス可能なアドレスへのイベント配信は機能します。

```
$ kn event send --to-url https://ce-api.foo.example.com/
```

一方、サービスがカスタム CA によって発行された HTTPS 証明書でパブリックアドレスを使用する場合、この配信は失敗します。

```
$ kn event send --to Service:serving.knative.dev/v1:event-display
```

ブローカーやチャネルなどの他のアドレス指定可能なオブジェクトへのイベント送信は、この問題の影響を受けず、期待どおりに機能します。

- 現在、Kafka ブローカーは Federal Information Processing Standards (FIPS) モードが有効になっているクラスターでは動作しません。
- **kn func create** コマンドで Springboot 関数プロジェクトディレクトリーを作成する場合、それ以降の **kn func build** コマンドの実行は、以下のエラーメッセージと共に失敗します。

```
[analyzer] no stack metadata found at path ''
[analyzer] ERROR: failed to : set API for buildpack 'paketo-buildpacks/ca-certificates@3.0.2': buildpack API version '0.7' is incompatible with the lifecycle
```

回避策として、関数設定ファイル **func.yaml** で **builder** プロパティーを **gcr.io/paketo-buildpacks/builder:base** に変更します。

- **gcr.io** レジストリーを使用した関数のデプロイは、以下のエラーメッセージと共に失敗します。

```
Error: failed to get credentials: failed to verify credentials: status code: 404
```

回避策として、**quay.io** または **docker.io** などの **gcr.io** 以外のレジストリーを使用します。

- **http** テンプレートで作成された Typescript 関数は、クラスターへのデプロイに失敗します。回避策として、**func.yaml** ファイルで以下のセクションを置き換えます。

```
buildEnvs: []
```

上記を以下のように置き換えます。

```
buildEnvs:
- name: BP_NODE_RUN_SCRIPTS
  value: build
```

- **func** バージョン 0.20 では、一部のランタイムが podman を使用して関数をビルドできない場合があります。以下のようなエラーメッセージが表示される場合があります。

```
ERROR: failed to image: error during connect: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.40/info": EOF
```

  - この問題には、以下の回避策があります。
    a. **--time=0** をサービス **ExecStart** 定義に追加して、podman サービスを更新します。

    サービス設定の例

    ```
    ExecStart=/usr/bin/podman $LOGGING system service --time=0
    ```

    b. 以下のコマンドを実行して podman サービスを再起動します。

    ```
    $ systemctl --user daemon-reload
    ```

    ```
    $ systemctl restart --user podman.socket
    ```

- または、TCP を使用して podman API を公開することもできます。

```
$ podman system service --time=0 tcp:127.0.0.1:5534 &
export DOCKER_HOST=tcp://127.0.0.1:5534
```

1.10. RED HAT OPENSHIFT SERVERLESS 1.19.0 のリリースノート

OpenShift Serverless 1.19.0 が利用可能になりました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.10.1. 新機能

- OpenShift Serverless は Knative Serving 0.25 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.25 を使用するようになりました。
- OpenShift Serverless は Kourier 0.25 を使用するようになりました。
- OpenShift Serverless は Knative (**kn**) CLI 0.25 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 0.25 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.19 を使用するようになりました。
- **KafkaBinding** API は OpenShift Serverless 1.19.0 で非推奨となり、今後のリリースで廃止される予定です。
- HTTPS リダイレクトがサポートされ、クラスターに対してグローバルに設定することも、各 Knative サービスごとに設定することもできるようになりました。

1.10.2. 修正された問題

- 以前のリリースでは、Kafka チャネルディスパッチャーは、ローカルコミットが成功するのを待ってからしか応答していませんでした。これにより、Apache Kafka ノードに障害が発生した場合に、イベントが失われる可能性がありました。Kafka チャネルディスパッチャーは、すべての同期レプリカがコミットするのを待ってから応答するようになりました。

1.10.3. 既知の問題

- **func** バージョン 0.19 では、一部のランタイムが podman を使用して関数をビルドできない場合があります。以下のようなエラーメッセージが表示される場合があります。

```
ERROR: failed to image: error during connect: Get
"http://%2Fvar%2Frun%2Fdocker.sock/v1.40/info": EOF
```

  - この問題には、以下の回避策があります。
    a. **--time=0** をサービス **ExecStart** 定義に追加して、podman サービスを更新します。

    サービス設定の例

    ```
    ExecStart=/usr/bin/podman $LOGGING system service --time=0
    ```

    b. 以下のコマンドを実行して podman サービスを再起動します。

```
$ systemctl --user daemon-reload
```

```
$ systemctl restart --user podman.socket
```

- または、TCP を使用して podman API を公開することもできます。

```
$ podman system service --time=0 tcp:127.0.0.1:5534 &
export DOCKER_HOST=tcp://127.0.0.1:5534
```

1.11. RED HAT OPENSHIFT SERVERLESS 1.18.0 のリリースノート

OpenShift Serverless 1.18.0 が利用可能になりました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.11.1. 新機能

- OpenShift Serverless は Knative Serving 0.24.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.24.0 を使用するようになりました。
- OpenShift Serverless は Kourier 0.24.0 を使用するようになりました。
- OpenShift Serverless は Knative (**kn**) CLI 0.24.0 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 0.24.7 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.18.0 を使用するようになりました。
- 今後の OpenShift Serverless 1.19.0 リリースでは、外部ルートの URL スキームはデフォルトで HTTPS になり、セキュリティーが強化されます。
  この変更をワークロードに適用する必要がない場合は、以下の YAML を **KnativeServing** カスタムリソース (CR) に追加してから 1.19.0 にアップグレードする前にデフォルト設定を上書きできます。

```
...
spec:
  config:
    network:
      defaultExternalScheme: "http"
...
```

  変更を 1.18.0 ですでに適用する必要がある場合には、以下の YAML を追加します。

```
...
spec:
  config:
    network:
      defaultExternalScheme: "https"
...
```

- 今後の OpenShift Serverless 1.19.0 リリースでは、Kourier ゲートウェイが公開されるデフォルトのサービスタイプは **ClusterIP** であり、**LoadBalancer** ではありません。

この変更をワークロードに適用する必要がない場合は、以下の YAML を **KnativeServing** カスタムリソース定義 (CRD) に追加してから 1.19.0 にアップグレードする前にデフォルト設定を上書きできます。

```
...
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...
```

- OpenShift Serverless で **emptyDir** ボリュームを使用できるようになりました。詳細は、Knative Serving に関する OpenShift Serverless ドキュメントを参照してください。
- **kn func** を使用して関数を作成すると、Rust テンプレートが利用できるようになりました。

1.11.2. 修正された問題

- 以前の 1.4 バージョンの Camel-K は OpenShift Serverless 1.17.0 と互換性がありませんでした。Camel-K の問題が修正され、Camel-K バージョン 1.4.1 を OpenShift Serverless 1.17.0 で使用できます。
- 以前のバージョンでは、Kafka チャネルまたは新しい Kafka ソースの新しいサブスクリプションを作成する場合は、新しく作成されたサブスクリプションまたはシンクが準備完了ステータスを報告した後、Kafka データプレーンがメッセージをディスパッチする準備ができるまでに遅延が生じる可能性があります。
その結果、データプレーンが準備完了ステータスを報告していないときに送信されたメッセージは、サブスクライバーまたはシンクに配信されない可能性があります。

  OpenShift Serverless 1.18.0 では、問題が修正され、初期メッセージが失われなくなりました。この問題の詳細は、ナレッジベースの記事 #6343981 を参照してください。

1.11.3. 既知の問題

- Knative **kn** CLI の古いバージョンは、Knative Serving および Knative Eventing API の古いバージョンを使用する可能性があります。たとえば、**kn** CLI のバージョン 0.23.2 は **v1alpha1** API バージョンを使用します。
一方、OpenShift Serverless の新しいリリースでは、古い API バージョンをサポートしない可能性があります。たとえば、OpenShift Serverless 1.18.0 は **kafkasources.sources.knative.dev** API のバージョン **v1alpha1** をサポートしなくなりました。

  そのため、**kn** が古い API を検出できないため、新しい OpenShift Serverless で古いバージョンの Knative **kn** CLI を使用するとエラーが発生する可能性がありました。たとえば、**kn** CLI のバージョン 0.23.2 は OpenShift Serverless 1.18.0 では機能しません。

  問題を回避するには、OpenShift Serverless リリースで利用可能な最新の **kn** CLI バージョンを使用します。OpenShift Serverless 1.18.0 については、Knative **kn** CLI 0.24.0 を使用します。

1.12. RED HAT OPENSHIFT SERVERLESS 1.17.0 のリリースノート

OpenShift Serverless 1.17.0 が利用可能になりました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.12.1. 新機能

- OpenShift Serverless は Knative Serving 0.23.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.23.0 を使用するようになりました。
- OpenShift Serverless は Kourier 0.23.0 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.23.0 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 0.23.0 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.17.0 を使用するようになりました。
- 今後の OpenShift Serverless 1.19.0 リリースでは、外部ルートの URL スキームはデフォルトで HTTPS になり、セキュリティーが強化されます。
  この変更をワークロードに適用する必要がない場合は、以下の YAML を **KnativeServing** カスタムリソース (CR) に追加してから 1.19.0 にアップグレードする前にデフォルト設定を上書きできます。

```
...
spec:
  config:
    network:
      defaultExternalScheme: "http"
...
```

- mTLS 機能は一般に利用可能 (GA) になりました。
- **kn func** を使用して関数を作成すると、Typescript テンプレートが利用できるようになりました。
- Knative Eventing 0.23.0 で API バージョンへの変更
  - OpenShift Serverless バージョン 1.14.0 で非推奨となった **KafkaChannel** API の **v1alpha1** バージョンが削除されました。設定マップの **ChannelTemplateSpec** パラメーターにこの古いバージョンの参照が含まれる場合は、これを仕様のこの部分を更新して、正しい API バージョンを使用する必要があります。

1.12.2. 既知の問題

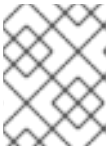
- 新しい OpenShift Serverless リリースで古いバージョンの Knative **kn** CLI の使用を試行する場合は、API が見つからないとエラーが発生します。
  たとえば、バージョン 0.22.0 を使用する **kn** CLI の 1.16.0 リリースと、Knative Serving および Knative Eventing API の 0.23.0 バージョンを使用する 1.17.0 OpenShift Serverless リリースを使用する場合、CLI は、古い 0.22.0 API バージョンを探し続けるため、機能しません。

  問題を回避するために、OpenShift Serverless リリースの最新の **kn** CLI バージョンを使用していることを確認してください。
- 本リリースでは、Kafka チャネルメトリクスは、対応する Web コンソールダッシュボードで監視されず、表示されません。これは、Kafka ディスパッチャーの調整プロセスが大幅に変更されたためです。
- Kafka チャネルまたは新しい Kafka ソースの新しいサブスクリプションを作成する場合は、新しく作成されたサブスクリプションまたはシンクが準備完了ステータスを報告した後、Kafka

データプレーンがメッセージをディスパッチする準備ができるまでに遅延が生じる可能性があります。
その結果、データプレーンが準備完了ステータスを報告していない間に送信されたメッセージは、サブスクライバーまたはシンクに配信されない場合があります。

この問題および可能な回避策に関する詳細は、ナレッジアーティクル #6343981 を参照してください。

- Camel-K 1.4 リリースは、OpenShift Serverless バージョン 1.17.0 と互換性がありません。これは、Camel-K 1.4 が Knative バージョン 0.23.0 で削除された API を使用するためです。現在、この問題に対する回避策はありません。OpenShift Serverless で Camel-K 1.4 を使用する必要がある場合は、OpenShift Serverless バージョン 1.17.0 にアップグレードしないでください。



注記

この問題は修正され、Camel-K バージョン 1.4.1 は OpenShift Serverless 1.17.0 と互換性があります。

1.13. RED HAT OPENSHIFT SERVERLESS 1.16.0 のリリースノート

OpenShift Serverless 1.16.0 が利用可能になりました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.13.1. 新機能

- OpenShift Serverless は Knative Serving 0.22.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.22.0 を使用するようになりました。
- OpenShift Serverless は Kourier 0.22.0 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.22.0 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 0.22.0 を使用するようになりました。
- **kn func** CLI プラグインは **func** 0.16.0 を使用するようになりました。
- **kn func emit** コマンドが関数 **kn** プラグインに追加されました。このコマンドを使用してイベントを送信し、ローカルにデプロイされた機能をテストできます。

1.13.2. 既知の問題

- OpenShift Serverless 1.16.0 にアップグレードする前に、OpenShift Container Platform をバージョン 4.6.30、4.7.11、またはそれ以降にアップグレードする必要があります。
- AMQ Streams Operator は、OpenShift Serverless Operator のインストールまたはアップグレードを妨げる可能性があります。これが生じる場合、以下のエラーが Operator Lifecycle Manager (OLM) によって出力されます。

```
WARNING: found multiple channel heads: [amqstreams.v1.7.2 amqstreams.v1.6.2], please check the `replaces`/`skipRange` fields of the operator bundles.
```

この問題を修正するには、OpenShift Serverless Operator をインストールまたはアップグレードする前に AMQ Streams Operator をアンインストールしてください。その後、AMQ Streams Operator を再インストールできます。

- サービスメッシュが mTLS で有効にされている場合、サービスメッシュが Prometheus のメトリクスの収集を阻止するため、Knative Serving のメトリクスはデフォルトで無効にされます。Service Mesh および mTLS で使用する Knative Serving メトリクスを有効にする方法は、Serverless ドキュメントの Integrating Service Mesh with OpenShift Serverless セクションを参照してください。

- Istio Ingress を有効にしてサービスメッシュ CR をデプロイする場合、**istio-ingressgateway** Pod に以下の警告が表示される可能性があります。

  ```
  2021-05-02T12:56:17.700398Z warning envoy config
  [external/envoy/source/common/config/grpc_subscription_impl.cc:101] gRPC config for
  type.googleapis.com/envoy.api.v2.Listener rejected: Error adding/updating listener(s)
  0.0.0.0_8081: duplicate listener 0.0.0.0_8081 found
  ```

  Knative サービスにもアクセスできない場合があります。

  以下の回避策を使用して、**knative-local-gateway** サービスを再作成することでこの問題を修正できます。

  a. **istio-system** namespace の既存の **knative-local-gateway** サービスを削除します。

  ```
  $ oc delete services -n istio-system knative-local-gateway
  ```

  b. 以下の YAML が含まれる **knative-local-gateway** サービスを作成し、適用します。

  ```
  apiVersion: v1
  kind: Service
  metadata:
   name: knative-local-gateway
   namespace: istio-system
   labels:
     experimental.istio.io/disable-gateway-port-translation: "true"
  spec:
   type: ClusterIP
   selector:
     istio: ingressgateway
   ports:
     - name: http2
       port: 80
       targetPort: 8081
  ```

- 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** カスタムリソース (CR) の状態が **Ready** になった後に最初の新しいサービスを作成すると遅延が生じます。
  3scale-kourier-control サービスは、新しいサービスの作成を処理する前に、既存のすべての Knative サービスを調整します。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** の状態で約 800 秒を費やすことになります。

- Kafka チャネルまたは新しい Kafka ソースの新しいサブスクリプションを作成する場合は、新しく作成されたサブスクリプションまたはシンクが準備完了ステータスを報告した後、Kafka データプレーンがメッセージをディスパッチする準備ができるまでに遅延が生じる可能性があります。
  その結果、データプレーンが準備完了ステータスを報告していない間に送信されたメッセージは、サブスクライバーまたはシンクに配信されない場合があります。

この問題および可能な回避策に関する詳細は、ナレッジアーティクル #6343981 を参照してください。

1.14. RED HAT OPENSHIFT SERVERLESS 1.15.0 のリリースノート

OpenShift Serverless 1.15.0 が公開されました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.14.1. 新機能

- OpenShift Serverless は Knative Serving 0.21.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing Operator 0.21.0 を使用するようになりました。
- OpenShift Serverless は Kourier 0.21.0 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.21.0 を使用するようになりました。
- OpenShift Serverless は Knative Kafka 0.21.1 を使用するようになりました。
- OpenShift Serverless Functions はテクノロジープレビューとして利用可能になりました。

重要

これまでプライベートサービスの作成に使用されていた **serving.knative.dev/visibility** ラベルは非推奨になりました。既存のサービスを更新して、代わりに **networking.knative.dev/visibility** ラベルを使用する必要があります。

1.14.2. 既知の問題

- Kafka チャネルまたは新しい Kafka ソースの新しいサブスクリプションを作成する場合は、新しく作成されたサブスクリプションまたはシンクが準備完了ステータスを報告した後、Kafka データプレーンがメッセージをディスパッチする準備ができるまでに遅延が生じる可能性があります。
その結果、データプレーンが準備完了ステータスを報告していない間に送信されたメッセージは、サブスクライバーまたはシンクに配信されない場合があります。

  この問題および可能な回避策に関する詳細は、ナレッジアーティクル #6343981 を参照してください。

1.15. RED HAT OPENSHIFT SERVERLESS 1.14.0 のリリースノート

OpenShift Serverless 1.14.0 が公開されました。以下では、OpenShift Container Platform 上の OpenShift Serverless に関連する新機能、変更点および既知の問題について説明します。

1.15.1. 新機能

- OpenShift Serverless は Knative Serving 0.20.0 を使用するようになりました。
- OpenShift Serverless は Knative Eventing 0.20.0 を使用しています。
- OpenShift Serverless は Kourier 0.20.0 を使用するようになりました。
- OpenShift Serverless は Knative **kn** CLI 0.20.0 を使用するようになりました。

- OpenShift Serverless は Knative Kafka 0.20.0 を使用するようになりました。
- OpenShift Serverless での Knative Kafka は一般に利用可能 (GA) になりました。

重要

OpenShift Serverless の **KafkaChannel** および **KafkaSource** オブジェクトの API の **v1beta1** バージョンのみがサポートされます。非推奨となった **v1alpha1** バージョンの API は使用しないでください。

- OpenShift Serverless のインストールおよびアップグレード用の Operator チャネルが OpenShift Container Platform 4.6 以降のバージョンで **stable** に更新されました。
- OpenShift Serverless は、IBM Power Systems、IBM Z、および LinuxONE でサポートされるようになりましたが、以下の機能はまだサポートされていません。
  - Knative Kafka の機能。
  - OpenShift Serverless 機能の developer プレビュー。

1.15.2. 既知の問題

- Kafka チャネルのサブスクリプションには **READY** のマークが付けられず、**SubscriptionNotMarkedReadyByChannel** 状態のままになることがあります。これを修正するには、Kafka チャネルの dispatcher を再起動します。
- Kafka チャネルまたは新しい Kafka ソースの新しいサブスクリプションを作成する場合は、新しく作成されたサブスクリプションまたはシンクが準備完了ステータスを報告した後、Kafka データプレーンがメッセージをディスパッチする準備ができるまでに遅延が生じる可能性があります。
  その結果、データプレーンが準備完了ステータスを報告していない間に送信されたメッセージは、サブスクライバーまたはシンクに配信されない場合があります。

  この問題および可能な回避策に関する詳細は、ナレッジアーティクル #6343981 を参照してください。

第2章 発見

2.1. OPENSHIFT SERVERLESS について

OpenShift Serverless は、Kubernetes ネイティブなビルディングブロックを提供します。開発者はこれらを使用して、OpenShift Container Platform 上でサーバーレスのイベント駆動型アプリケーションを作成およびデプロイできます。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対して移植性と一貫性をもたらします。

2.1.1. Knative Serving

Knative Serving は、クラウドネイティブアプリケーション の作成、デプロイ、管理を希望する開発者をサポートします。これにより、オブジェクトのセットが 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) として提供されます。

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

2.1.1.1. Knative Serving リソース

サービス

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カスタムリソース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。

2.1.2. Knative Eventing

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、イベントプロデューサーとイベントコンシューマー間の関係を切り離すという概念に基づいています。

イベントプロデューサーはイベントを作成し、イベント**シンク**またはコンシューマーはイベントを受信

します。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シンク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。

Knative Eventing は以下のユースケースをサポートします。

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 としてブローカーに送信し、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。

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リソース

HTTP 経由でイベントの **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** リソース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。

呼び出し可能なリソース

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。

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。

- Channels and subscriptions、または
- ブローカー と トリガー。

2.1.3. サポートされる構成

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。

2.1.4. スケーラビリティーおよびパフォーマンス

OpenShift Serverless は、3 つのメインノードと 3 つのワーカーノードの設定でテストされています。各ノードには、64 個の CPU、457 GB のメモリー、および 394 GB のストレージがあります。

この設定を使用して作成できる Knative サービスの最大数は 3,000 です。これは、OpenShift Container Platform の Kubernetes サービスの制限である 10,000 に相当します。これは、1 つの Knative サービスが 3 つの Kubernetes サービスを作成するためです。

ゼロ応答時間からの平均スケールは約 3.4 秒で、最大応答時間は 8 秒で、単純な Quarkus アプリケーションの 99.9 パーセンタイルは 4.5 秒でした。これらの時間は、アプリケーションとアプリケーションの実行時間によって異なる場合があります。

2.1.5. 関連情報

- カスタムリソース定義による Kubernetes API の拡張
- カスタムリソース定義からのリソースの管理

- What is serverless?

2.2. OPENSHIFT SERVERLESS FUNCTIONS について

OpenShift Serverless Functions により、開発者は OpenShift Container Platform で Knative サービスとしてステートレスでイベント駆動型の関数を作成およびデプロイできます。**kn func CLI** は Knative **kn** CLI のプラグインとして提供されます。**kn func** CLI を使用して、クラスター上の Knative サービスとしてコンテナーイメージを作成、ビルド、デプロイできます。

重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

2.2.1. 含まれるランタイム

OpenShift Serverless Functions は、以下のランタイムの基本機能を作成するために使用できるテンプレートを提供します。

- Node.js
- Python
- Go
- Quarkus
- TypeScript

2.2.2. 次のステップ

- Getting started with functions

2.3. イベントソース

Knative **イベントソース** には、クラウドイベントの生成またはインポート、これらのイベントの別のエンドポイントへのリレー (**sink** とも呼ばれる) を行う Kubernetes オブジェクトを指定できます。イベントに対応する分散システムを開発するには、イベントのソースが重要になります。

OpenShift Container Platform Web コンソールの **Developer** パースペクティブ、Knative (**kn**) CLI を使用するか、YAML ファイルを適用することで、Knative イベントソースを作成および管理できます。

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

Kubernetes API サーバーイベントを Knative に送ります。API サーバーソースは、Kubernetes リソースが作成、更新、または削除されるたびに新規イベントを送信します。

Ping ソース

指定された cron スケジュールに、固定ペイロードを使用してイベントを生成します。

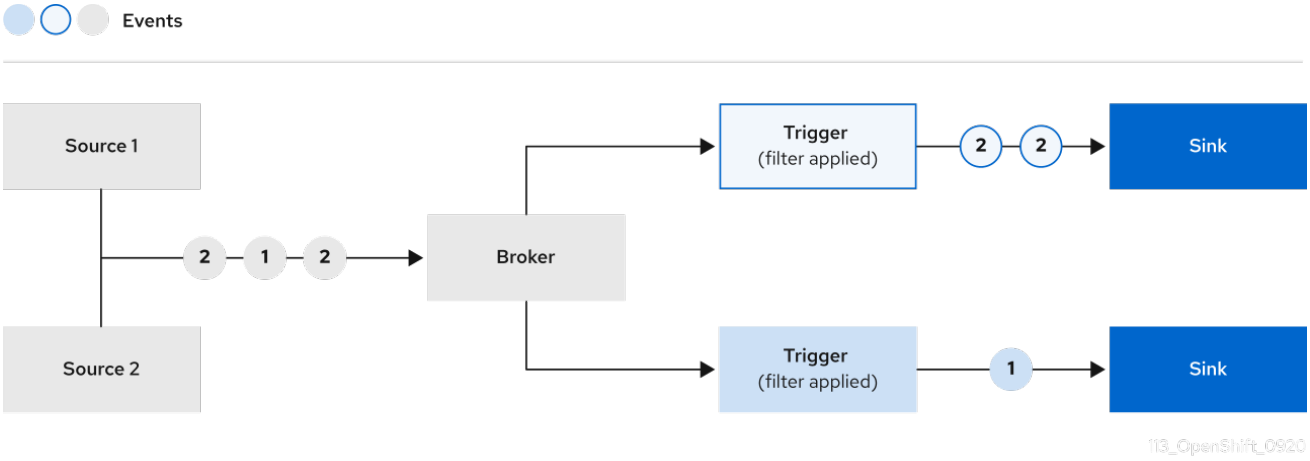
Kafka イベントソース

Kafka クラスターをイベントソースとしてシンクに接続します。

カスタムイベントソース を作成することもできます。

2.4. ブローカー

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。



2.4.1. ブローカータイプ

クラスター管理者は、クラスターのデフォルトブローカー実装を設定できます。ブローカーを作成する場合、**Broker** オブジェクトで設定を指定しない限り、デフォルトのブローカー実装が使用されます。

2.4.1.1. 開発目的でのデフォルトブローカーの実装

Knative は、デフォルトのチャネルベースのブローカー実装を提供します。このチャネルベースのブローカーは、開発およびテストの目的で使用できますが、実稼働環境での適切なイベント配信の保証は提供しません。デフォルトのブローカーは、デフォルトで **InMemoryChannel** チャネル実装によってサポートされています。

Kafka を使用してネットワークホップを削減する場合は、Kafka ブローカーの実装を使用します。チャネルベースのブローカーが **KafkaChannel** チャネル実装によってサポートされるように設定しないでください。

2.4.1.2. 実稼働環境対応の Kafka ブローカーの実装

実稼働環境に対応した Knative Eventing デプロイメントの場合、Red Hat は Knative Kafka ブローカーの実装を使用することをお勧めします。Kafka ブローカーは、Knative ブローカーの Apache Kafka ネイティブ実装であり、CloudEvents を Kafka インスタンスに直接送信します。

重要

Kafka ブローカーの連邦情報処理標準 (FIPS) モードが無効になっています。

Kafka ブローカーは、イベントの保存とルーティングのために Kafka とネイティブに統合されています。これにより、他のブローカータイプよりもブローカーとトリガーモデルの Kafka との統合性が向上し、ネットワークホップを削減することができます。Kafka ブローカー実装のその他の利点は次のとおりです。

- 少なくとも 1 回の配信保証
- CloudEvents パーティショニング拡張機能に基づくイベントの順序付き配信
- コントロールプレーンの高可用性
- 水平方向にスケーラブルなデータプレーン

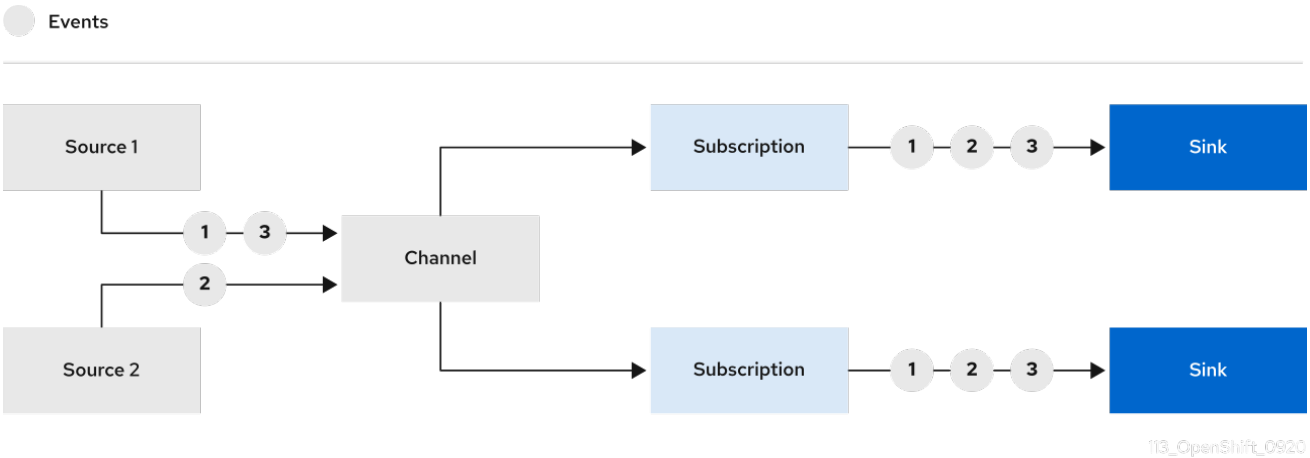
Knative Kafka ブローカーは、バイナリーコンテンツモードを使用して、受信 CloudEvents を Kafka レコードとして保存します。これは、CloudEvent のすべての属性と拡張機能が Kafka レコードのヘッダーとしてマップされ、CloudEvent の **data** 仕様が Kafka レコードの値に対応することを意味します。

2.4.2. 次のステップ

- ブローカーの作成

2.5. チャネルおよびサブスクリプション

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。イベントがイベントソースまたは生成側から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。



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チャネルを作成し、**Subscription** オブジェクトの **delivery** 仕様を変更して再配信の試行を設定できます。

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。たとえば、**InMemoryChannel** のデフォルト実装の場合、**Channel** オブジェクトは以下のようになります。

```
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。

このメカニズムが上記の例で使用される場合、汎用バッキングチャネルおよび **InMemoryChannel** チャネルなど 2 つのオブジェクトが作成されます。別のデフォルトチャネルの実装を使用している場合、**InMemoryChannel** は実装に固有のものに置き換えられます。たとえば、Knative Kafka の場合、**KafkaChannel** チャネルが作成されます。

バッキングチャネルは、サブスクリプションをユーザー作成のチャネルオブジェクトにコピーし、ユーザー作成チャネルオブジェクトのステータスを、バッキングチャネルのステータスを反映するように設定します。

2.5.1. チャネルの実装タイプ

InMemoryChannel および **KafkaChannel** チャネルの実装は、開発目的で OpenShift Serverless で使用できます。

以下は、**InMemoryChannel** タイプの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、再配信はデフォルトで試行されません。**Subscription** オブジェクトの **delivery** 仕様を変更することで、再配信の試行を設定できます。

Kafka チャネルの詳細は、Knative Kafka のドキュメントを参照してください。

2.5.2. 次のステップ

- チャネルを作成します。
- クラスター管理者の場合、チャネルのデフォルト設定を設定できます。チャネルのデフォルトの設定 を参照してください。

第3章 インストール

3.1. OPENSHIFT SERVERLESS OPERATOR のインストール

OpenShift Serverless Operator をインストールすると、OpenShift Container Platform クラスターに Knative Serving、Knative Eventing、Knative Kafka をインストールして使用することができます。OpenShift Serverless Operator は、クラスターの Knative カスタムリソース定義 (CRD) を管理し、各コンポーネントの個別の設定マップを直接修正することなくそれらを設定できるようにします。

3.1.1. 作業を開始する前に

OpenShift Serverless をインストールする前に、サポートされる設定および前提条件についての以下の情報を確認してください。

- 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。
- 現時点で、OpenShift Serverless は単一クラスター上でのマルチテナント設定で使用することはできません。

3.1.1.1. 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 を実行するために必要な総サイズは、インストールされているコンポーネントとデプロイされているアプリケーションに依存し、デプロイメントによって異なる場合があります。

注記

以下の要件は、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。コントロールプレーン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。

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。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。

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。

3.1.1.2. 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。

3.1.2. OpenShift Serverless Operator のインストール

OpenShift Container Platform Web コンソールを使用して、OperatorHub から OpenShift Serverless Operator をインストールできます。この Operator をインストールすることで、Knative コンポーネントをインストールして使用することができます。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift Container Platform Web コンソールにログインしている。

手順

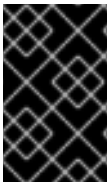
1. OpenShift Container Platform Web コンソールで、**Operators → OperatorHub** ページに移動します。
2. スクロールするか、またはこれらのキーワード **Serverless** を **Filter by keyword ボックス** に入力して OpenShift Serverless Operator を検索します。
3. Operator についての情報を確認してから、**Install** をクリックします。
4. **Install Operator** ページで以下を行います。
    a. **Installation 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-serverless** 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-serverless** です。
    c. **Update Channel** として **stable** チャネルを選択します。**stable** 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-serverless** プロジェクトの Pod のログで確認できます。



重要

OpenShift Serverless で Red Hat 分散トレースを使用する 場合は、KnativeServing または KnativeEventing をインストールする前に、Red Hat 分散トレースをインストールして設定する必要があります。

3.1.3. 関連情報

- ネットワークが制限された環境での Operator Lifecycle Manager の使用
- OpenShift Serverless での高可用性レプリカの設定

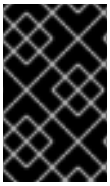
3.1.4. 次のステップ

- OpenShift Serverless Operator のインストール後に、Knative Serving をインストールするか、Knative Eventing をインストールする ことができます。

3.2. KNATIVE SERVING のインストール

Knative Serving をインストールすると、クラスター上で Knative サービスや関数を作成することができます。また、オートスケーリングやネットワークオプションなどの追加機能をアプリケーションに利用することも可能です。

OpenShift Serverless Operator をインストールした後、デフォルトの設定で Knative Serving をインストールするか、**KnativeServing** カスタムリソース (CR) でより高度な設定を行うことが可能です。**KnativeServing** CR の設定オプションの詳細については、 グローバル設定を参照してください。



重要

OpenShift Serverless で Red Hat 分散トレースを使用する 場合は、KnativeServing をインストールする前に、Red Hat 分散トレースをインストールして設定する必要があります。

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

OpenShift Serverless Operator をインストールした後、OpenShift Container Platform の Web コンソールを使用して Knative Serving をインストールします。デフォルトの設定で Knative Serving をインストールするか、**KnativeServing** カスタムリソース (CR) でより詳細な設定を行うことが可能です。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift Container Platform Web コンソールにログインしている。
- OpenShift Serverless Operator がインストールされている。

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** 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

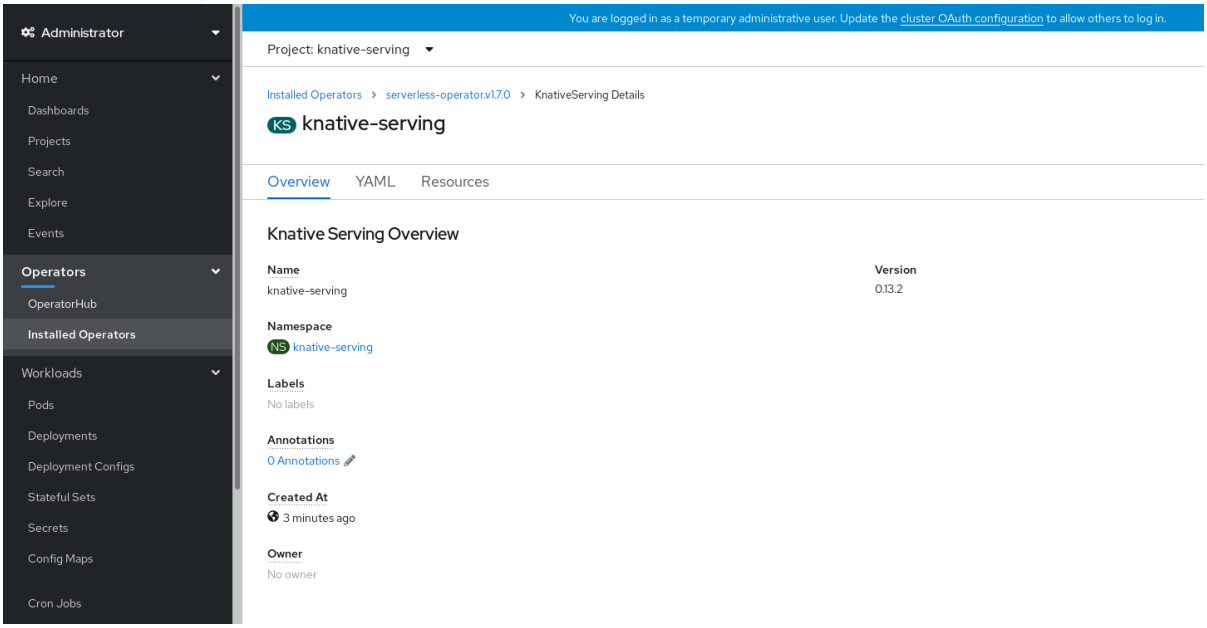
KnativeServing カスタムリソース定義の設定オプションについての詳細は、**高度なインストール設定オプション** についてのドキュメントを参照してください。

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** カスタムリソースが表示されます。

検証

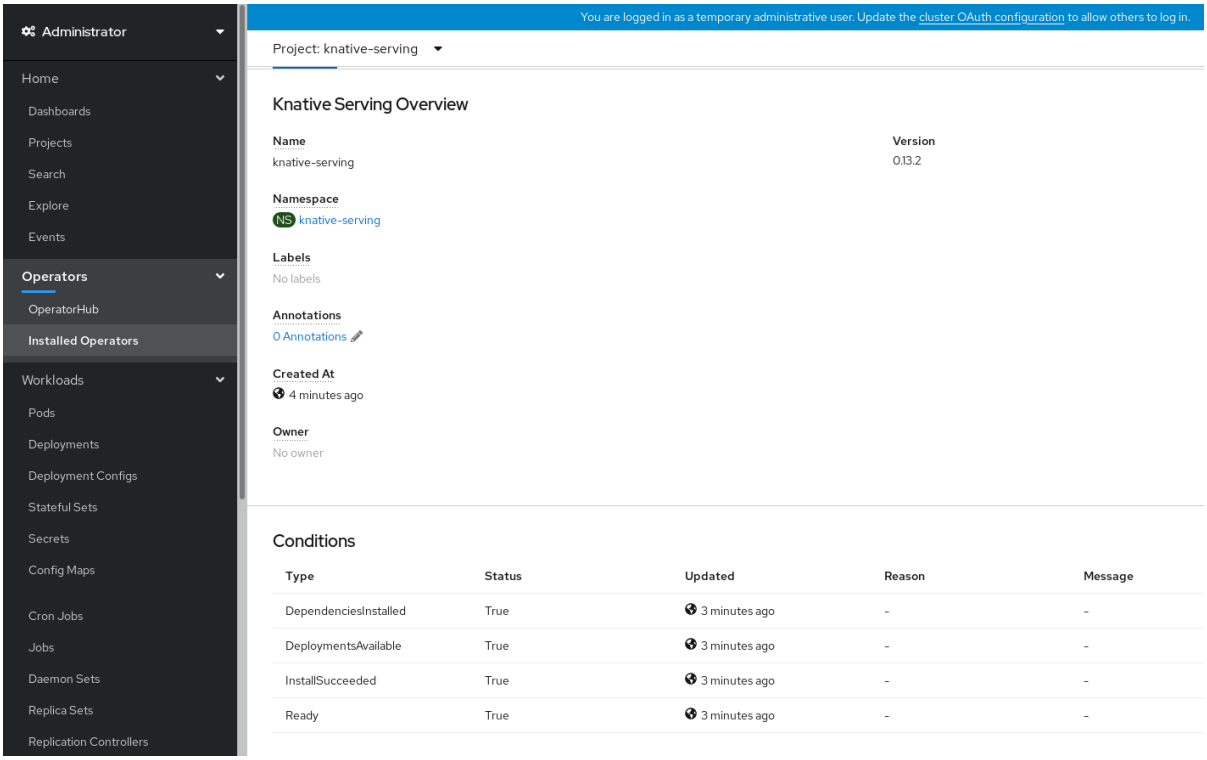
1. **Knative Serving** タブで **knative-serving** カスタムリソースをクリックします。

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。

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

OpenShift Serverless Operator をインストールした後、デフォルトの設定で Knative Serving をインストールするか、**KnativeServing** カスタムリソース (CR) でより高度な設定を行うことが可能です。YAML ファイルと**oc** CLI を利用して、以下の手順で Knative Serving をインストールすることができます。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift Serverless Operator がインストールされている。
- OpenShift CLI (**oc**) をインストールしている。

手順

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

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

2. **serving.yaml** ファイルを適用します。

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

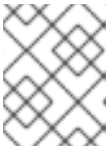
検証

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}}'
```

出力例

```
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。

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。

2. 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
```

出力例

```
NAME                                                READY   STATUS      RESTARTS   AGE
activator-67ddf8c9d7-p7rm5                          2/2     Running     0          4m
activator-67ddf8c9d7-q84fz                          2/2     Running     0          4m
autoscaler-5d87bc6dbf-6nqc6                         2/2     Running     0          3m59s
autoscaler-5d87bc6dbf-h64rl                         2/2     Running     0          3m59s
autoscaler-hpa-77f85f5cc4-lrts7                     2/2     Running     0          3m57s
autoscaler-hpa-77f85f5cc4-zx7hl                     2/2     Running     0          3m56s
controller-5cfc7cb8db-nlccl                         2/2     Running     0          3m50s
controller-5cfc7cb8db-rmv7r                         2/2     Running     0          3m18s
domain-mapping-86d84bb6b4-r746m                     2/2     Running     0          3m58s
domain-mapping-86d84bb6b4-v7nh8                     2/2     Running     0          3m58s
domainmapping-webhook-769d679d45-bkcnj              2/2     Running     0          3m58s
domainmapping-webhook-769d679d45-fff68              2/2     Running     0          3m58s
storage-version-migration-serving-serving-0.26.0--1-6qlkb   0/1     Completed   0          3m56s
webhook-5fb774f8d8-6bqrt                            2/2     Running     0          3m57s
webhook-5fb774f8d8-b8lt5                            2/2     Running     0          3m57s
```

3. 必要なネットワークコンポーネントが、自動的に作成された **knative-serving-ingress** namespace にインストールされていることを確認します。

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

出力例

```
NAME                                    READY   STATUS    RESTARTS   AGE
net-kourier-controller-7d4b6c5d95-62mkf   1/1     Running   0          76s
net-kourier-controller-7d4b6c5d95-qmgm2   1/1     Running   0          76s
3scale-kourier-gateway-6688b49568-987qz   1/1     Running   0          75s
3scale-kourier-gateway-6688b49568-b5tnp   1/1     Running   0          75s
```

3.2.3. 次のステップ

- Knative イベント駆動型アーキテクチャーを使用する必要がある場合は、Knative Eventing をインストール できます。

3.3. KNATIVE EVENTING のインストール

クラスターでイベント駆動型アーキテクチャーを使用するには、Knative Eventing をインストールします。イベントソース、ブローカー、チャネルなどの Knative コンポーネントを作成し、それらを使用してアプリケーションや外部システムにイベントを送信することができます。

OpenShift Serverless Operator をインストールした後、デフォルトの設定で Knative Eventing をインストールするか、**KnativeEventing** カスタムリソース (CR) でより高度な設定を行うことが可能です。**KnativeEventing** CR の設定オプションの詳細については、 グローバル設定 を参照してください。

重要

OpenShift Serverless で Red Hat 分散トレースを使用する 場合は、Knative Eventing をインストールする前に、Red Hat 分散トレースをインストールして設定する必要があります。

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

OpenShift Serverless Operator をインストールした後、OpenShift Container Platform の Web コンソールを使用して Knative Eventing をインストールします。デフォルトの設定で Knative Eventing をインストールするか、**KnativeEventing** カスタムリソース (CR) でより詳細な設定を行うことが可能です。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift Container Platform Web コンソールにログインしている。
- OpenShift Serverless Operator がインストールされている。

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**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。

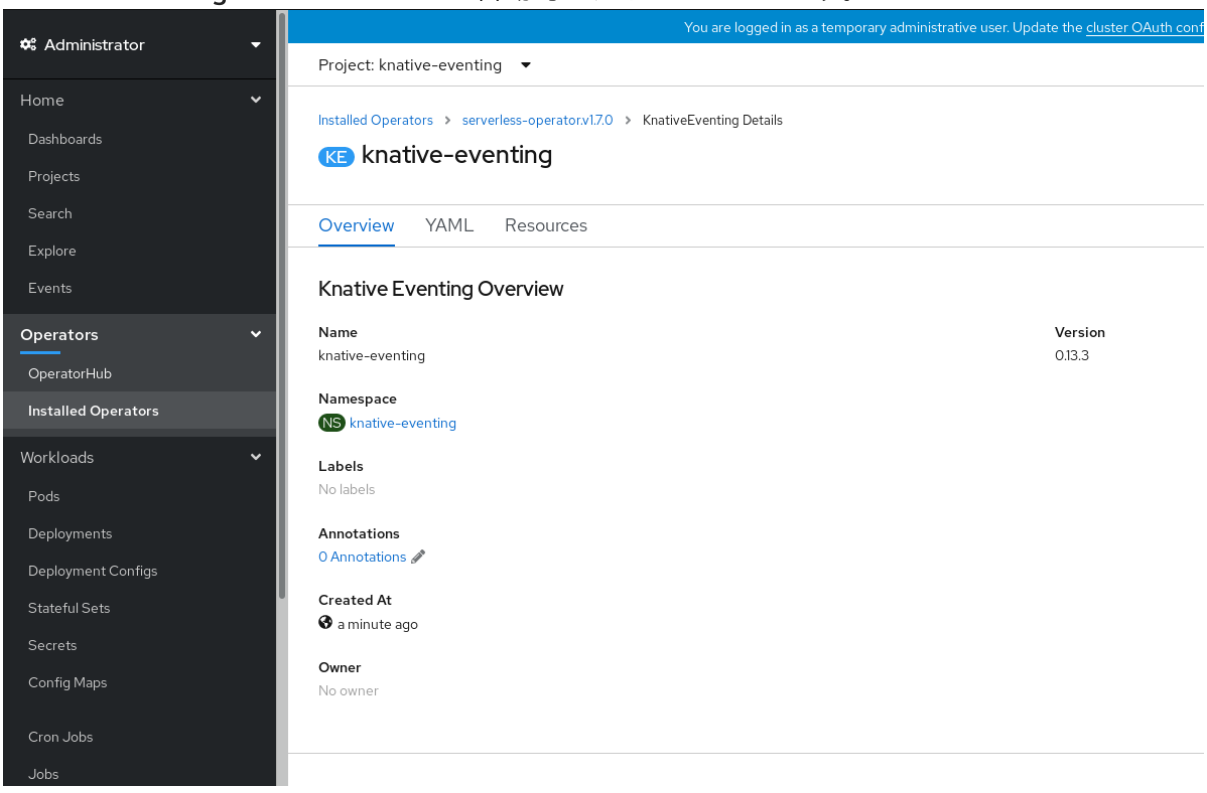
7. **Create** をクリックします。

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** リソースが表示されます。

検証

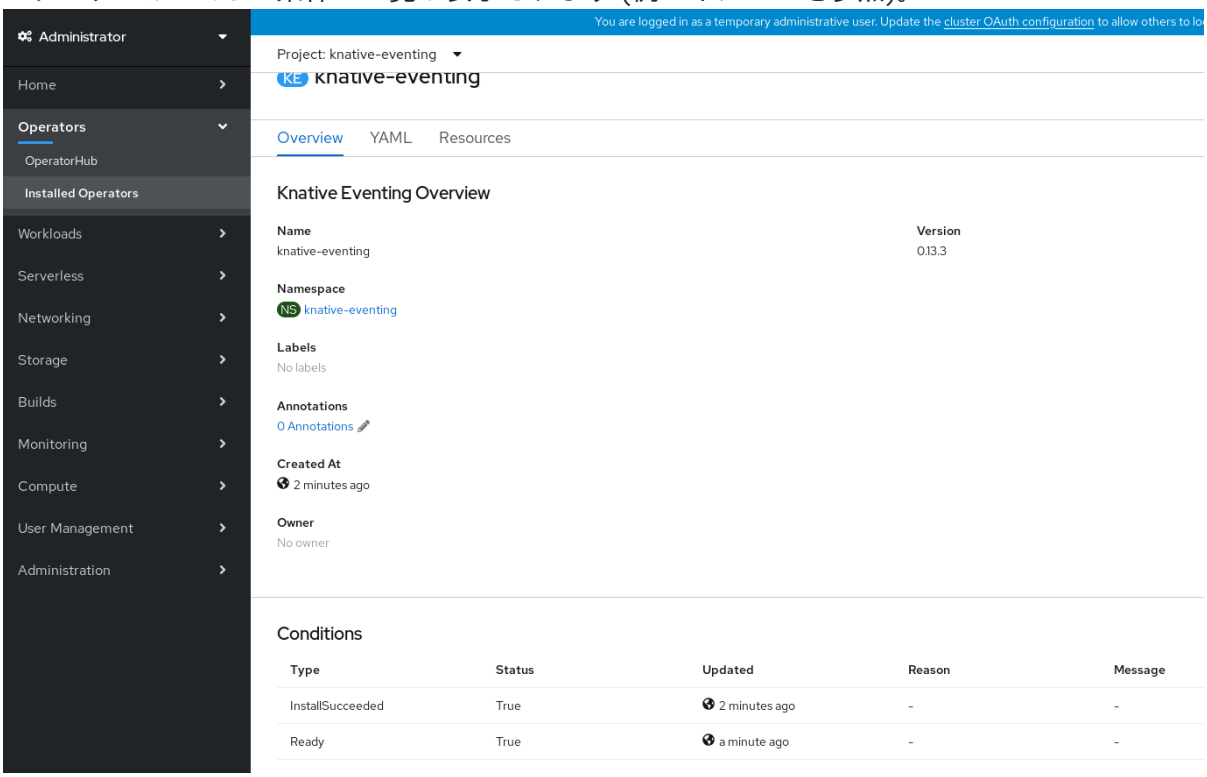
1. **Knative Eventing** タブで **knative-eventing** カスタムリソースをクリックします。

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。

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

OpenShift Serverless Operator をインストールした後、デフォルトの設定で Knative Eventing をインストールするか、**KnativeEventing** カスタムリソース (CR) でより高度な設定を行うことが可能です。YAML ファイルと **oc** CLI を利用して、以下の手順で Knative Eventing をインストールすることができます。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift Serverless Operator がインストールされている。
- OpenShift CLI (**oc**) をインストールしている。

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
```

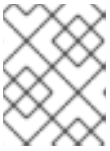
検証

1. 以下のコマンドを入力して出力を確認し、インストールが完了したことを確認します。

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

出力例

```
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
```

出力例

```
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
```

3.3.3. 次のステップ

- Knative サービスを使用する場合は、Knative Serving をインストールできます。

3.4. OPENSHIFT SERVERLESS の削除

必要に応じて OpenShift Serverless をクラスターから削除するには、OpenShift Serverless Operator およびその他の OpenShift Serverless コンポーネントを手動で削除します。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。

3.4.1. Knative Serving のアンインストール

OpenShift Serverless Operator を削除する前に、Knative Serving を削除する必要があります。Knative Serving をアンインストールするには、**KnativeServing** カスタムリソース (CR) を削除してから **knative-serving** namespace を削除する必要があります。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. **KnativeServing** CR を削除します。

```
$ 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、namespace を削除します。

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

3.4.2. Knative Eventing のアンインストール

OpenShift Serverless Operator を削除する前に、Knative Eventing を削除する必要があります。Knative Eventing をアンインストールするには、**KnativeEventing** カスタムリソース (CR) を削除してから **knative-eventing** namespace を削除する必要があります。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. **KnativeEventing** CR を削除します。

```
$ 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、namespace を削除します。

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

3.4.3. OpenShift Serverless Operator の削除

Knative Serving と Knative Eventing を削除した後、OpenShift Serverless Operator を削除することができます。これは、OpenShift Container Platform の Web コンソールまたは **oc** CLI を使用して行うことができます。

3.4.3.1. Web コンソールの使用によるクラスターからの Operator の削除

クラスター管理者は Web コンソールを使用して、選択した namespace からインストールされた Operator を削除できます。

前提条件

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスター Web コンソールにアクセスできること。

手順

1. **Operators → Installed Operators** ページからスクロールするか、または **Filter by name** にキーワードを入力して必要な Operator を見つけます。次に、それをクリックします。

2. **Operator Details** ページの右側で、**Actions** 一覧から **Uninstall Operator** を選択します。**Uninstall Operator?** ダイアログボックスが表示され、以下が通知されます。

   Operator を削除しても、そのカスタムリソース定義や管理リソースは削除されません。Operator がクラスターにアプリケーションをデプロイしているか、またはクラスター外のリソースを設定している場合、それらは引き続き実行され、手動でクリーンアップする必要があります。

   このアクションにより、Operator および Operator のデプロイメントおよび Pod が削除されます (ある場合)。CRD および CR を含む Operator によって管理される Operand およびリソースは削除されません。Web コンソールは、一部の Operator のダッシュボードおよびナビゲーションアイテムを有効にします。Operator のアンインストール後にこれらを削除するには、Operator CRD を手動で削除する必要があります。

3. **Uninstall** を選択します。この Operator は実行を停止し、更新を受信しなくなります。

3.4.3.2. CLI の使用によるクラスターからの Operator の削除

クラスター管理者は CLI を使用して、選択した namespace からインストールされた Operator を削除できます。

前提条件

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。

- **oc** コマンドがワークステーションにインストールされていること。

手順

1. サブスクライブされた Operator (例: **jaeger**) の現行バージョンを **currentCSV** フィールドで確認します。

   ```
   $ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV
   ```

   出力例

```
  currentCSV: jaeger-operator.v1.8.2
```

2. サブスクリプション (例: **jaeger**) を削除します。

```
$ oc delete subscription jaeger -n openshift-operators
```

出力例

```
subscription.operators.coreos.com "jaeger" deleted
```

3. 直前の手順で **currentCSV** 値を使用し、ターゲット namespace の Operator の CSV を削除します。

```
$ oc delete clusterserviceversion jaeger-operator.v1.8.2 -n openshift-operators
```

出力例

```
clusterserviceversion.operators.coreos.com "jaeger-operator.v1.8.2" deleted
```

3.4.3.3. 障害のあるサブスクリプションの更新

Operator Lifecycle Manager (OLM) で、ネットワークでアクセスできないイメージを参照する Operator をサブスクライブする場合、以下のエラーを出して失敗した **openshift-marketplace** namespace でジョブを見つけることができます。

出力例

```
ImagePullBackOff for
Back-off pulling image "example.com/openshift4/ose-elasticsearch-operator-
bundle@sha256:6d2587129c846ec28d384540322b40b05833e7e00b25cca584e004af9a1d292e"
```

出力例

```
rpc error: code = Unknown desc = error pinging docker registry example.com: Get
"https://example.com/v2/": dial tcp: lookup example.com on 10.0.0.1:53: no such host
```

その結果、サブスクリプションはこの障害のある状態のままとなり、Operator はインストールまたはアップグレードを実行できません。

サブスクリプション、クラスターサービスバージョン (CSV) その他の関連オブジェクトを削除して、障害のあるサブスクリプションを更新できます。サブスクリプションを再作成した後に、OLM は Operator の正しいバージョンを再インストールします。

前提条件

- アクセス不可能なバンドルイメージをプルできない障害のあるサブスクリプションがある。
- 正しいバンドルイメージにアクセスできることを確認している。

手順

1. Operator がインストールされている namespace から **Subscription** および **ClusterServiceVersion** オブジェクトの名前を取得します。

```
$ oc get sub,csv -n <namespace>
```

出力例

```
NAME                                         PACKAGE                  SOURCE             CHANNEL
subscription.operators.coreos.com/elasticsearch-operator   elasticsearch-operator   redhat-operators   5.0

NAME                                                  DISPLAY                            VERSION    REPLACES   PHASE
clusterserviceversion.operators.coreos.com/elasticsearch-operator.5.0.0-65   OpenShift Elasticsearch Operator   5.0.0-65              Succeeded
```

2. サブスクリプションを削除します。

```
$ oc delete subscription <subscription_name> -n <namespace>
```

3. クラスターサービスバージョンを削除します。

```
$ oc delete csv <csv_name> -n <namespace>
```

4. **openshift-marketplace** namespace の失敗したジョブおよび関連する設定マップの名前を取得します。

```
$ oc get job,configmap -n openshift-marketplace
```

出力例

```
NAME                                                              COMPLETIONS   DURATION   AGE
job.batch/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb   1/1           26s        9m30s

NAME                                                              DATA   AGE
configmap/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb   3      9m30s
```

5. ジョブを削除します。

```
$ oc delete job <job_name> -n openshift-marketplace
```

これにより、アクセスできないイメージのプルを試行する Pod は再作成されなくなります。

6. 設定マップを削除します。

```
$ oc delete configmap <configmap_name> -n openshift-marketplace
```

7. Web コンソールの OperatorHub を使用した Operator の再インストール

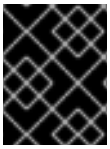
検証

- Operator が正常に再インストールされていることを確認します。

```
$ oc get sub,csv,installplan -n <namespace>
```

3.4.4. OpenShift Serverless カスタムリソース定義の削除

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

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- Knative Serving をアンインストールし、OpenShift Serverless Operator を削除している。
- OpenShift CLI (**oc**) をインストールしている。

手順

- 残りの OpenShift Serverless CRD を削除するには、以下のコマンドを実行します。

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

第4章 KNATIVE CLI

4.1. KNATIVE CLI のインストール

Knative (**kn**) CLI には、独自のログインメカニズムがありません。クラスターにログインするには、OpenShift (**oc**) CLI をインストールし、**oc login** コマンドを使用する必要があります。CLI のインストールオプションは、オペレーティングシステムによって異なる場合があります。

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、OpenShift 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。

Knative (**kn**) CLI を使用して OpenShift Serverless をインストールすることはできません。クラスター管理者は、OpenShift Serverless Operator のインストール のドキュメントで説明されているように、OpenShift Serverless Operator をインストールし、Knative コンポーネントをセットアップする必要があります。

重要

新しい OpenShift Serverless リリースで古いバージョンの Knative (**kn**) CLI の使用を試行する場合は、API が見つからないとエラーが発生します。

たとえば、バージョン 1.2 を使用する Knative (**kn**) CLI の 1.23.0 リリースと、Knative Serving および Knative Eventing API の 1.3 バージョンを使用する 1.24.0 OpenShift Serverless リリースを使用する場合、CLI は古い 1.2 API バージョンを探し続けるため、機能しません。

問題を回避するために、OpenShift Serverless リリースの最新の Knative (**kn**) CLI バージョンを使用していることを確認してください。

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

OpenShift Container Platform Web コンソールを使用すると、Knative (**kn**) CLI をインストールするための合理化された直感的なユーザーインターフェイスが提供されます。\OpenShift Serverless Operator をインストールすると、OpenShift Container PlatformWeb コンソールの **コマンドラインツール** ページから Linux (amd64、s390x、ppc64le)、macOS、または Windows 用の Knative (**kn**) CLI をダウンロードするためのリンクが表示されます。

前提条件

- OpenShift Container Platform Web コンソールにログインしている。
- OpenShift Serverless Operator および Knative Serving が OpenShift Container Platform クラスターにインストールされている。

重要

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
```

- この手順の検証手順を使用する場合は、OpenShift (**oc**) CLI をインストールする必要があります。

手順

1. **Command Line Tools** ページから Knative (**kn**) CLI をダウンロードします。**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、リストの **Command Line Tools** を選択します。
2. アーカイブを展開します。

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

3. **kn** バイナリーを **PATH** 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
```

検証

- 以下のコマンドを実行して、正しい Knative CLI リソースおよびルートが作成されていることを確認します。

```
$ oc get ConsoleCLIDownload
```

出力例

```
NAME                DISPLAY NAME                                      AGE
kn                  kn - OpenShift Serverless Command Line Interface (CLI)   2022-09-
20T08:41:18Z
oc-cli-downloads    oc - OpenShift Command Line Interface (CLI)              2022-09-
20T08:00:20Z
```

```
$ oc get route -n openshift-serverless
```

出力例

```
NAME   HOST/PORT                              PATH   SERVICES                  PORT
TERMINATION     WILDCARD
kn     kn-openshift-serverless.apps.example.com          knative-openshift-metrics-3   http-cli
edge/Redirect   None
```

4.1.2. RPM パッケージマネージャーを使用した Linux 用の Knative CLI のインストール

Red Hat Enterprise Linux (RHEL) の場合、**yum** や **dnf** などのパッケージマネージャーを使用して、Knative (**kn**) CLI を RPM としてインストールできます。これにより、Knative CLI バージョンをシステムで自動的に管理できます。たとえば、**dnf upgrade** のようなコマンドを使用すると、新しいバージョンが利用可能な場合、**kn** を含むすべてのパッケージがアップグレードされます。

前提条件

- お使いの 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。

手順

1. Red Hat Subscription Manager に登録します。

```
# subscription-manager register
```

2. 最新のサブスクリプションデータをプルします。

```
# subscription-manager refresh
```

3. 登録済みのシステムにサブスクリプションを添付します。

```
# subscription-manager attach --pool=<pool_id> 1
```

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

4. Knative (**kn**) CLI に必要なリポジトリーを有効にします。
   - Linux (x86_64, amd64)

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-x86_64-rpms"
```

   - Linux on IBM Z and LinuxONE (s390x)

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-s390x-rpms"
```

   - Linux on IBM Power (ppc64le)

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-ppc64le-rpms"
```

5. パッケージマネージャーを使用して、Knative (**kn**) CLI を RPM としてインストールします。

   yum コマンドの例

```
# yum install openshift-serverless-clients
```

4.1.3. Linux の Knative CLI のインストール

RPM または別のパッケージマネージャーがインストールされていない Linux ディストリビューションを使用している場合は、Knative (**kn**) CLI をバイナリーファイルとしてインストールできます。これを行うには、**tar.gz** アーカイブをダウンロードして解凍し、バイナリーを **PATH** のディレクトリーに追加する必要があります。

前提条件

- RHEL または Fedora を使用していない場合は、ライブラリーパスのディレクトリーに **libc** がインストールされていることを確認してください。

重要

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
```

手順

1. Knative (**kn**) CLI の **tar.gz** アーカイブをダウンロードします。
   - Linux (x86_64, amd64)
   - Linux on IBM Z and LinuxONE (s390x)
   - Linux on IBM Power (ppc64le)
2. アーカイブを展開します。

```
$ tar -xf <filename>
```

3. **kn** バイナリーを **PATH** 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
```

4.1.4. macOS の Knative CLI のインストール

macOS を使用している場合は、Knative (**kn**) CLI をバイナリーファイルとしてインストールできます。これを行うには、**tar.gz** アーカイブをダウンロードして解凍し、バイナリーを **PATH** のディレクトリーに追加する必要があります。

手順

1. Knative (**kn**) **CLItar.gz** アーカイブ をダウンロードします。
2. アーカイブを解凍して解凍します。
3. **kn** バイナリーを **PATH** 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、ターミナルウィンドウを開き、以下を実行します。

```
$ echo $PATH
```

4.1.5. Windows の Knative CLI のインストール

Windows を使用している場合は、Knative (**kn**) CLI をバイナリーファイルとしてインストールできます。これを行うには、ZIP アーカイブをダウンロードして解凍し、バイナリーを **PATH** のディレクトリーに追加する必要があります。

手順

1. Knative (**kn**) CLIZIP アーカイブ をダウンロードします。

2. ZIP プログラムでアーカイブを展開します。
3. **kn** バイナリーを **PATH** 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、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
```

4.2. KNATIVE CLI の設定

config.yaml 設定ファイルを作成することで、Knative (**kn**) CLI セットアップをカスタマイズできます。**--config** フラグを使用してこの設定を指定できます。指定しない場合、設定がデフォルトの場所から選択されます。デフォルトの設定場所は XDGBaseDirectory 仕様 に準拠しており、UNIX システムと Windows システムでは異なります。

UNIX システムの場合:

- **XDG_CONFIG_HOME** 環境変数が設定されている場合、Knative (**kn**) CLI が検索するデフォルト設定の場所は **$XDG_CONFIG_HOME/kn** になります。
- **XDG_CONFIG_HOME** 環境変数が設定されていない場合、Knative (**kn**) CLI は **$HOME/.config/kn/config.yaml** のユーザーのホームディレクトリーにある設定を検索します。

Windows システムの場合、デフォルトの Knative (**kn**) CLI 設定の場所は **%APPDATA%\kn** です。

設定ファイルのサンプル

```
plugins:
  path-lookup: true (1)
  directory: ~/.config/kn/plugins (2)
eventing:
  sink-mappings: (3)
  - prefix: svc (4)
    group: core (5)
    version: v1 (6)
    resource: services (7)
```

(1) Knative (**kn**) CLI が **PATH** 環境変数でプラグインを検索するかどうかを指定します。これはブール型の設定オプションです。デフォルト値は **false** です。

(2) Knative (**kn**) CLI がプラグインを検索するディレクトリーを指定します。前述のように、デフォルトのパスはオペレーティングシステムによって異なります。これには、ユーザーに表示される任意のディレクトリーを指定できます。

(3) **sink-mappings** 仕様は、Knative (**kn**) CLI コマンドで **--sink** フラグを使用する場合に使用される Kubernetes のアドレス可能なリソースを定義します。

(4) シンクの記述に使用する接頭辞。サービスの **svc**、 **channel**、および **broker** は Knative (**kn**) CLI で事前に定義される接頭辞です。

(5) Kubernetes リソースの API グループ。

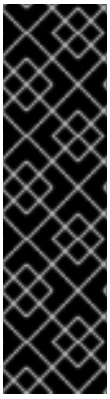
(6) Kubernetes リソースのバージョン。

7 Kubernetes リソースタイプの複数形の名前。例: **services** または **brokers**

4.3. KNATIVE CLI プラグイン

Knative (**kn**) CLI は、プラグインの使用をサポートします。これにより、カスタムコマンドとコアディストリビューションの一部ではない他の共有コマンドを追加でき、**kn** インストールの機能の拡張を可能にします。Knative (**kn**) CLI プラグインは主な **kn** 機能として同じ方法で使用されます。

現在、Red Hat は **kn-source-kafka** プラグインと **kn-event** プラグインをサポートしています。



重要

kn-event プラグインは、テクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

4.3.1. kn-event プラグインを使用してイベントを作成する

kn event build コマンドのビルダーのようなインターフェイスを使用して、イベントをビルドできます。その後、そのイベントを後で送信するか、別のコンテキストで使用できます。

前提条件

- Knative (**kn**) CLI をインストールしている。

手順

- イベントをビルドします。

```
$ kn event build --field <field-name>=<value> --type <type-name> --id <id> --output <format>
```

ここでは、以下のようになります。

  - **--field** フラグは、データをフィールド/値のペアとしてイベントに追加します。これは複数回使用できます。
  - **--type** フラグを使用すると、イベントのタイプを指定する文字列を指定できます。
  - **--id** フラグは、イベントの ID を指定します。
  - **json** または **yaml** 引数を **--output** フラグと共に使用して、イベントの出力形式を変更できます。
    これらのフラグはすべてオプションです。

簡単なイベントのビルド

```
$ kn event build -o yaml
```

YAML 形式のビルドされたイベント

```
data: {}
datacontenttype: application/json
id: 81a402a2-9c29-4c27-b8ed-246a253c9e58
source: kn-event/v0.4.0
specversion: "1.0"
time: "2021-10-15T10:42:57.713226203Z"
type: dev.knative.cli.plugin.event.generic
```

サンプルトランザクションイベントのビルド

```
$ kn event build \
    --field operation.type=local-wire-transfer \
    --field operation.amount=2345.40 \
    --field operation.from=87656231 \
    --field operation.to=2344121 \
    --field automated=true \
    --field signature='FGzCPLvYWdEgsdpb3qXkaVp7Da0=' \
    --type org.example.bank.bar \
    --id $(head -c 10 < /dev/urandom | base64 -w 0) \
    --output json
```

JSON 形式のビルドされたイベント

```
{
  "specversion": "1.0",
  "id": "RjtL8UH66X+UJg==",
  "source": "kn-event/v0.4.0",
  "type": "org.example.bank.bar",
  "datacontenttype": "application/json",
  "time": "2021-10-15T10:43:23.113187943Z",
  "data": {
    "automated": true,
    "operation": {
      "amount": "2345.40",
      "from": 87656231,
      "to": 2344121,
      "type": "local-wire-transfer"
    },
    "signature": "FGzCPLvYWdEgsdpb3qXkaVp7Da0="
  }
}
```

4.3.2. kn-event プラグインを使用したイベントの送信

kn event send コマンドを使用して、イベントを送信できます。イベントは、公開されているアドレス、または Kubernetes サービスや Knative サービス、ブローカー、チャネル等のクラスター内のアドレス指定可能なリソースのいずれかに送信できます。このコマンドは、**kn event build** コマンドと同じビルダーのようなインターフェイスを使用します。

前提条件

- Knative (**kn**) CLI をインストールしている。

手順

- イベントの送信:

```
$ kn event send --field <field-name>=<value> --type <type-name> --id <id> --to-url <url> --to
<cluster-resource> --namespace <namespace>
```

ここでは、以下のようになります。

- **--field** フラグは、データをフィールド/値のペアとしてイベントに追加します。これは複数回使用できます。
- **--type** フラグを使用すると、イベントのタイプを指定する文字列を指定できます。
- **--id** フラグは、イベントの ID を指定します。
- イベントを一般にアクセス可能な宛先に送信する場合は、**--to-url** フラグを使用して URL を指定します。
- イベントをクラスター内の Kubernetes リソースに送信する場合は、**--to** フラグを使用して宛先を指定します。
  - **<Kind>:<ApiVersion>:<name>** 形式を使用して Kubernetes リソースを指定します。
- **--namespace** フラグは namespace を指定します。省略すると、namespace は現在のコンテキストから取得されます。
  --to-url または **--to** のいずれかを使用する必要がある宛先の仕様を除き、これらのフラグはすべてオプションです。

  以下の例は、イベントを URL に送信するケースを示しています。

  コマンドの例

```
$ kn event send \
    --field player.id=6354aa60-ddb1-452e-8c13-24893667de20 \
    --field player.game=2345 \
    --field points=456 \
    --type org.example.gaming.foo \
    --to-url http://ce-api.foo.example.com/
```

  以下の例は、イベントをクラスター内のリソースに送信するケースを示しています。

  コマンドの例

```
$ kn event send \
    --type org.example.kn.ping \
    --id $(uuidgen) \
    --field event.type=test \
    --field event.data=98765 \
    --to Service:serving.knative.dev/v1:event-display
```

4.4. KNATIVE SERVING CLI コマンド

以下の Knative (**kn**) CLI コマンドを使用して、クラスター上の Knative Serving タスクを完了できます。

4.4.1. kn service コマンド

以下のコマンドを使用して Knative サービスを作成し、管理できます。

4.4.1.1. Knative CLI を使用したサーバーレスアプリケーションの作成

Knative (**kn**) CLI を使用してサーバーレスアプリケーションを作成すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。**kn service create** コマンドを使用して、基本的なサーバーレスアプリケーションを作成できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 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。

手順

- Knative サービスを作成します。

```
$ kn service create <service_name> --image <image> --tag <tag-value>
```

詳細は以下のようになります。

- **--image** は、アプリケーションのイメージの URI です。
- **--tag** は、サービスで作成される初期リビジョンにタグを追加するために使用できるオプションのフラグです。

コマンドの例

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

出力例

```
Creating service 'event-display' in namespace 'default':

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

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

4.4.1.2. Knative CLI を使用したサーバーレスアプリケーションの更新

サービスを段階的に構築する際に、コマンドラインで **kn service update** コマンドを使用し、対話式のセッションを使用できます。**kn service apply** コマンドとは対照的に、**kn service update** コマンドを使用する際は、Knative サービスの完全な設定ではなく、更新が必要な変更のみを指定する必要があります。

コマンドの例

- 新規の環境変数を追加してサービスを更新します。

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

- 新しいポートを追加してサービスを更新します。

```
$ kn service update <service_name> --port 80
```

- 新しい要求および制限パラメーターを追加してサービスを更新します。

```
$ kn service update <service_name> --request cpu=500m --limit memory=1024Mi --limit
cpu=1000m
```

- **latest** タグをリビジョンに割り当てます。

```
$ kn service update <service_name> --tag <revision_name>=latest
```

- サービスの最新の **READY** リビジョンについて、**testing** から **staging** にタグを更新します。

```
$ kn service update <service_name> --untag testing --tag @latest=staging
```

- **test** タグをトラフィックの 10% を受信するリビジョンに追加し、残りのトラフィックをサービスの最新の **READY** リビジョンに送信します。

```
$ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90
```

4.4.1.3. サービス宣言の適用

kn service apply コマンドを使用して Knative サービスを宣言的に設定できます。サービスが存在しない場合は、これが作成されますが、それ以外の場合は、既存のサービスが変更されたオプションで更新されます。

kn service apply コマンドは、ユーザーがターゲットの状態を宣言するために単一コマンドでサービスの状態を詳細に指定したい場合など、とくにシェルスクリプトや継続的インテグレーションパイプラインで役に立ちます。

kn service apply を使用する場合は、Knative サービスの詳細な設定を指定する必要があります。これは **kn service update** コマンドとは異なります。このコマンドでは、更新する必要のあるオプションを指定するだけで済みます。

コマンドの例

- サービスを作成します。

```
$ kn service apply <service_name> --image <image>
```

- 環境変数をサービスに追加します。

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

- JSON または YAML ファイルからサービス宣言を読み取ります。

```
$ kn service apply <service_name> -f <filename>
```

4.4.1.4. Knative CLI を使用したサーバーレスアプリケーションの記述

kn service describe コマンドを使用して Knative サービスを記述できます。

コマンドの例

- サービスを記述します。

```
$ kn service describe --verbose <service_name>
```

--verbose フラグは任意ですが、さらに詳細な説明を提供するために追加できます。通常の出力と詳細の出力の違いについては、以下の例に示されます。

--verbose フラグを使用しない出力例

```
Name:       hello
Namespace:  default
Age:        2m
URL:        http://hello-default.apps.ocp.example.com

Revisions:
  100%  @latest (hello-00001) [1] (2m)
        Image:  docker.io/openshift/hello-openshift (pinned to aaea76)

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

--verbose フラグを使用する出力例

```
Name:         hello
Namespace:    default
Annotations:  serving.knative.dev/creator=system:admin
              serving.knative.dev/lastModifier=system:admin
Age:          3m
URL:          http://hello-default.apps.ocp.example.com
Cluster:      http://hello.default.svc.cluster.local

Revisions:
  100%  @latest (hello-00001) [1] (3m)
        Image:  docker.io/openshift/hello-openshift (pinned to aaea76)
        Env:    RESPONSE=Hello Serverless!
```

```
Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 3m
  ++ ConfigurationsReady   3m
  ++ RoutesReady           3m
```

- サービスを YAML 形式で記述します。

```
$ kn service describe <service_name> -o yaml
```

- サービスを JSON 形式で記述します。

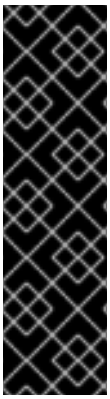
```
$ kn service describe <service_name> -o json
```

- サービス URL のみを出力します。

```
$ kn service describe <service_name> -o url
```

4.4.2. Knative CLI オフラインモードについて

kn service コマンドを実行すると、変更が即座にクラスターに伝播されます。ただし、別の方法として、オフラインモードで **kn service** コマンドを実行できます。オフラインモードでサービスを作成すると、クラスター上で変更は発生せず、代わりにサービス記述子ファイルがローカルマシンに作成されます。



重要

Knative CLI のオフラインモード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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

記述子ファイルの作成後、手動で変更し、バージョン管理システムで追跡できます。記述子ファイルで **kn service create -f**、**kn service apply -f** または **oc apply -f** コマンドを使用して変更をクラスターに伝播することもできます。

オフラインモードには、いくつかの用途があります。

- 記述子ファイルを使用してクラスターで変更する前に、記述子ファイルを手動で変更できます。
- バージョン管理システムでは、サービスの記述子ファイルをローカルで追跡できます。これにより、記述子ファイルを再利用できます。たとえば、継続的インテグレーション (CI) パイプライン、開発環境またはデモなどで、ターゲットクラスター以外の配置が可能になります。
- 作成した記述子ファイルを検証して Knative サービスについて確認できます。特に、生成されるサービスが **kn** コマンドに渡されるさまざまな引数によってどのように影響するかを確認できます。

オフラインモードには、高速で、クラスターへの接続を必要としないという利点があります。ただし、オフラインモードではサーバー側の検証がありません。したがって、サービス名が一意であることや、指定のイメージをプルできることなどを確認できません。

4.4.2.1. オフラインモードを使用したサービスの作成

オフラインモードで **kn service** コマンドを実行すると、クラスター上で変更は発生せず、代わりにサービス記述子ファイルがローカルマシンに作成されます。記述子ファイルを作成した後、クラスターに変更を伝播する前にファイルを変更することができます。

重要

Knative CLI のオフラインモード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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。
- Knative (**kn**) CLI をインストールしている。

手順

1. オフラインモードでは、ローカルの Knative サービス記述子ファイルを作成します。

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

出力例

```
Service 'event-display' created in namespace 'test'.
```

- **--target ./** フラグはオフラインモードを有効にし、**./** を新しいディレクトリーツリーを保存するディレクトリーとして指定します。
  既存のディレクトリーを指定せずに、**--target my-service.yaml** などのファイル名を使用すると、ディレクトリーツリーは作成されません。代わりに、サービス記述子ファイル **my-service.yaml** のみが現在のディレクトリーに作成されます。

  ファイル名には、**.yaml**、**.yml** または **.json** 拡張子を使用できます。**.json** を選択すると、JSON 形式でサービス記述子ファイルが作成されます。

- **--namespace test** オプションは、新規サービスを **テスト** namespace に配置します。

--namespace を使用せずに、OpenShift クラスターにログインしている場合には、記述子ファイルが現在の namespace に作成されます。それ以外の場合は、記述子ファイルが **default** の namespace に作成されます。

2. 作成したディレクトリー構造を確認します。

```
$ tree ./
```

出力例

```
./
└── test
    └── ksvc
        └── event-display.yaml

2 directories, 1 file
```

- **--target** で指定する現在の **./** ディレクトリーには新しい **test/** ディレクトリーが含まれます。このディレクトリーの名前は、指定の namespace をもとに付けられます。
- **test/** ディレクトリーには、リソースタイプの名前が付けられた **ksvc** ディレクトリーが含まれます。
- **ksvc** ディレクトリーには、指定のサービス名に従って命名される記述子ファイル **event-display.yaml** が含まれます。

3. 生成されたサービス記述子ファイルを確認します。

```
$ cat test/ksvc/event-display.yaml
```

出力例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  creationTimestamp: null
  name: event-display
  namespace: test
spec:
  template:
    metadata:
      annotations:
        client.knative.dev/user-image: quay.io/openshift-knative/knative-eventing-sources-event-
display:latest
      creationTimestamp: null
    spec:
      containers:
      - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
        name: ""
        resources: {}
status: {}
```

4. 新しいサービスに関する情報を一覧表示します。

```
$ kn service describe event-display --target ./ --namespace test
```

出力例

```
Name:       event-display
Namespace:  test
Age:
URL:

Revisions:

Conditions:
  OK TYPE    AGE REASON
```

- **--target ./** オプションは、namespace サブディレクトリーを含むディレクトリー構造のルートディレクトリーを指定します。
  または、**--target** オプションで YAML または JSON ファイルを直接指定できます。使用可能なファイルの拡張子は、**.yaml**、**.yml**、および **.json** です。

- **--namespace** オプションは、namespace を指定し、この namespace は必要なサービス記述子ファイルを含むサブディレクトリーの **kn** と通信します。
  --namespace を使用せず、OpenShift クラスターにログインしている場合には、**kn** は現在の namespace をもとに名前が付けられたサブディレクトリーでサービスを検索します。それ以外の場合は、**kn** は **default/** サブディレクトリーで検索します。

5. サービス記述子ファイルを使用してクラスターでサービスを作成します。

```
$ kn service create -f test/ksvc/event-display.yaml
```

出力例

```
Creating service 'event-display' in namespace 'test':

  0.058s The Route is still working to reflect the latest desired specification.
  0.098s ...
  0.168s Configuration "event-display" is waiting for a Revision to become ready.
 23.377s ...
 23.419s Ingress has not yet been reconciled.
 23.534s Waiting for load balancer to be ready
 23.723s Ready to serve.

Service 'event-display' created to latest revision 'event-display-00001' is available at URL:
http://event-display-test.apps.example.com
```

4.4.3. kn container コマンド

以下のコマンドを使用して、Knative サービス仕様で複数のコンテナーを作成し、管理できます。

4.4.3.1. Knative クライアントマルチコンテナーのサポート

kn container add コマンドを使用して、YAML コンテナーの仕様を標準出力に出力できます。このコマンドは、定義を作成するために他の標準の **kn** フラグと共に使用できるため、マルチコンテナーのユースケースに役立ちます。

kn container add コマンドは、**kn service create** コマンドで使用できるコンテナー関連のすべてのフラグを受け入れます。UNIX パイプ (|) を使用して **kn container add** コマンドを連結して、一度に複数のコンテナー定義を作成することもできます。

コマンドの例

- イメージからコンテナーを追加し、標準出力に出力します。

```
$ kn container add <container_name> --image <image_uri>
```

コマンドの例

```
$ kn container add sidecar --image docker.io/example/sidecar
```

出力例

```
containers:
- image: docker.io/example/sidecar
  name: sidecar
  resources: {}
```

- 2 つの **kn container add** コマンドを連結してから、**kn service create** コマンドに渡して、2 つのコンテナーで Knative サービスを作成します。

```
$ kn container add <first_container_name> --image <image_uri> | \
kn container add <second_container_name> --image <image_uri> | \
kn service create <service_name> --image <image_uri> --extra-containers -
```

--extra-containers - は、**kn** が YAML ファイルの代わりにパイプ入力を読み取る特別なケースを指定します。

コマンドの例

```
$ kn container add sidecar --image docker.io/example/sidecar:first | \
kn container add second --image docker.io/example/sidecar:second | \
kn service create my-service --image docker.io/example/my-app:latest --extra-containers -
```

--extra-containers フラグは YAML ファイルへのパスを受け入れることもできます。

```
$ kn service create <service_name> --image <image_uri> --extra-containers <filename>
```

コマンドの例

```
$ kn service create my-service --image docker.io/example/my-app:latest --extra-containers
my-extra-containers.yaml
```

4.4.4. kn domain コマンド

以下のコマンドを使用して、ドメインマッピングを作成および管理できます。

4.4.4.1. Knative CLI を使用したカスタムドメインマッピングの作成

所有するカスタムドメイン名を Knative サービスにマッピングすることで、Knative サービスのドメインをカスタマイズできます。Knative (**kn**) CLI を使用して、Knative サービスまたは Knative ルートなどのアドレス指定可能なターゲット CR にマップする **DomainMapping** カスタムリソース (CR) を作成できます。

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。
- Knative サービスまたはルートを作成し、その CR にマップするカスタムドメインを制御している。

注記

カスタムドメインは OpenShift Container Platform クラスターの DNS を参照する必要があります。

- Knative (**kn**) CLI をインストールしている。
- 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。

手順

- ドメインを現在の namespace の CR にマップします。

```
$ kn domain create <domain_mapping_name> --ref <target_name>
```

コマンドの例

```
$ kn domain create example.com --ref example-service
```

--ref フラグは、ドメインマッピング用のアドレス指定可能なターゲット CR を指定します。

--ref フラグの使用時に接頭辞が指定されていない場合、ターゲットが現在の namespace の Knative サービスであることを前提としています。

- ドメインを指定された namespace の Knative サービスにマップします。

```
$ kn domain create <domain_mapping_name> --ref
<ksvc:service_name:service_namespace>
```

コマンドの例

```
$ kn domain create example.com --ref ksvc:example-service:example-namespace
```

- ドメインを Knative ルートにマップします。

```
$ kn domain create <domain_mapping_name> --ref <kroute:route_name>
```

コマンドの例

```
$ kn domain create example.com --ref kroute:example-route
```

4.4.4.2. Knative CLI を使用したカスタムドメインマッピングの管理

DomainMapping カスタムリソース (CR) の作成後に、既存の CR の一覧表示、既存の CR の情報の表示、CR の更新、または Knative (**kn**) CLI を使用した CR の削除を実行できます。

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。
- 1 つ以上の **DomainMapping** CR を作成している。
- Knative (**kn**) CLI ツールをインストールしている。
- 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。

手順

- 既存の **DomainMapping** CR を一覧表示します。

```
$ kn domain list -n <domain_mapping_namespace>
```

- 既存の **DomainMapping** CR の詳細を表示します。

```
$ kn domain describe <domain_mapping_name>
```

- 新規ターゲットを参照するように **DomainMapping** CR を更新します。

```
$ kn domain update --ref <target>
```

- **DomainMapping** CR を削除します。

```
$ kn domain delete <domain_mapping_name>
```

4.5. KNATIVE EVENTING CLI コマンド

以下の Knative (**kn**) CLI コマンドを使用して、クラスター上で Knative Eventing タスクを実行できます。

4.5.1. kn source コマンド

以下のコマンドを使用して、Knative イベントソースを一覧表示、作成、および管理できます。

4.5.1.1. 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

Knative (**kn**) CLI を使用すると、クラスターで使用可能なイベントソースタイプを表示するための合理的で直感的なユーザーインターフェイスが提供されます。**kn source list-types** CLI コマンドを使用して、クラスターで作成して使用できるイベントソースタイプを一覧表示できます。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。

手順

1. 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します。

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

出力例

```
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
```

2. オプション: 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
```

4.5.1.2. Knative CLI シンクフラグ

Knative (**kn**) CLI を使用してイベントソースを作成する場合、**--sink** フラグを使用して、イベントがリソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能な任意のリソースです。

以下の例では、サービスの **http://event-display.svc.cluster.local** をシンクとして使用するシンクバインディングを作成します。

シンクフラグを使用したコマンドの例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** の **svc** は、シンクが Knative サービスであることを判別します。他のデフォルトのシンクの接頭辞には、**channel** および **broker** が含まれます。

4.5.1.3. Knative CLI を使用したコンテナーソースの作成および管理

kn source container コマンドを使用し、Knative (**kn**) CLI を使用してコンテナーソースを作成および管理できます。イベントソースを作成するために Knative CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

コンテナーソースを作成します。

```
$ kn source container create <container_source_name> --image <image_uri> --sink <sink>
```

コンテナーソースの削除

```
$ kn source container delete <container_source_name>
```

コンテナーソースを記述します。

```
$ kn source container describe <container_source_name>
```

既存のコンテナーソースを一覧表示

```
$ kn source container list
```

既存のコンテナーソースを YAML 形式で一覧表示

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

コンテナーソースを更新します。

このコマンドにより、既存のコンテナーソースのイメージ URI が更新されます。

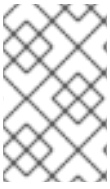
```
$ kn source container update <container_source_name> --image <image_uri>
```

4.5.1.4. Knative CLI を使用した API サーバーソースの作成

kn source apiserver create コマンドを使用し、**kn** CLI を使用して API サーバーソースを作成できます。API サーバーソースを作成するために **kn** CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- 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。
- OpenShift CLI (**oc**) がインストールされている。
- Knative (**kn**) CLI をインストールしている。



手順

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には、既存の **ServiceAccount** リソースを変更して、新規リソースを作成せずに、必要なパーミッションを含めることができます。

1. 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 YAML ファイルとして作成します。

```
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。

2. YAML ファイルを適用します。

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

3. イベントシンクを持つ API サーバーソースを作成します。次の例では、シンクはブローカーです。

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

4. 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。

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

5. ブローカーをイベントシンクとして使用した場合は、トリガーを作成して、**default** のブローカーからサービスへのイベントをフィルターリングします。

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

6. 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:latest
```

7. 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 <source_name>
```

出力例

```
Name:                mysource
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",
    ...
  }
```

API サーバーソースの削除

1. トリガーを削除します。

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

2. イベントソースを削除します。

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

3. 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
```

4.5.1.5. Knative CLI を使用した ping ソースの作成

kn source ping create コマンドを使用し、Knative (**kn**) CLI を使用して ping ソースを作成できます。イベントソースを作成するために Knative CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。
- 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。
- オプション: この手順の検証手順を使用する場合は、OpenShift CLI (**oc**) をインストールします。

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 ping ソースを作成します。

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!"
  }
```

ping ソースの削除

- ping ソースを削除します。

```
$ kn delete pingsources.sources.knative.dev <ping_source_name>
```

4.5.1.6. Knative CLI を使用した Kafka イベントソースの作成

kn source kafka create コマンドを使用し、Knative (**kn**) CLI を使用して Kafka ソースを作成できます。イベントソースを作成するために Knative CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- OpenShift Serverless Operator、Knative Eventing、Knative Serving、および **KnativeKafka** カスタムリソース (CR) がクラスターにインストールされている。
- 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。
- インポートする Kafka メッセージを生成する Red Hat AMQ Streams (Kafka) クラスターにアクセスできる。
- Knative (**kn**) CLI をインストールしている。
- オプション: この手順で検証ステップを使用する場合は、OpenShift CLI (**oc**) をインストールします。

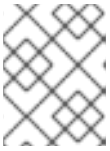
手順

1. Kafka イベント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 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
```

2. **KafkaSource** CR を作成します。

```
$ kn source kafka create <kafka_source_name> \
    --servers <cluster_kafka_bootstrap>.kafka.svc:9092 \
    --topics <topic_name> --consumergroup my-consumer-group \
    --sink event-display
```



注記

このコマンドのプレースホルダー値は、ソース名、ブートストラップサーバー、およびトピックの値に置き換えます。

--servers、**--topics**、および **--consumergroup** オプションは、Kafka クラスターへの接続パラメーターを指定します。**--consumergroup** オプションは任意です。

3. オプション: 作成した **KafkaSource** CR の詳細を表示します。

```
$ kn source kafka describe <kafka_source_name>
```

出力例

```
Name:              example-kafka-source
Namespace:         kafka
Age:               1h
BootstrapServers:  example-cluster-kafka-bootstrap.kafka.svc:9092
Topics:            example-topic
ConsumerGroup:     example-consumer-group

Sink:
```

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

Conditions:
  OK TYPE           AGE REASON
  ++ Ready          1h
  ++ Deployed        1h
  ++ SinkProvided    1h
```

検証手順

1. Kafka インスタンスをトリガーし、メッセージをトピックに送信します。

```
$ oc -n kafka run kafka-producer \
    -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true \
    --restart=Never -- bin/kafka-console-producer.sh \
    --broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic
```

プロンプトにメッセージを入力します。このコマンドは、以下を前提とします。

- Kafka クラスターが **kafka** namespace にインストールされている。
- **KafkaSource** オブジェクトは、**my-topic** トピックを使用するように設定されている。

2. ログを表示して、メッセージが到達していることを確認します。

```
$ 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.kafka.event
  source: /apis/v1/namespaces/default/kafkasources/example-kafka-source#example-topic
  subject: partition:46#0
  id: partition:46/offset:0
  time: 2021-03-10T11:21:49.4Z
Extensions,
  traceparent: 00-161ff3815727d8755848ec01c866d1cd-7ff3916c44334678-00
Data,
  Hello!
```

4.6. 関数コマンド

4.6.1. 関数の作成

関数をビルドし、デプロイする前に、Knative (**kn**) CLI を使用して関数を作成する必要があります。コマンドラインでパス、ランタイム、テンプレート、およびイメージレジストリーをフラグとして指定するか、**-c** フラグを使用してターミナルで対話型エクスペリエンスを開始できます。

重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。
- Knative (**kn**) CLI をインストールしている。

手順

- 関数プロジェクトを作成します。

```
$ kn func create -r <repository> -l <runtime> -t <template> <path>
```

  - 受け入れられるランタイム値には、**quarkus**、**node**、**typescript**、**go**、**python**、**springboot**、および **rust** が含まれます。
  - 受け入れられるテンプレート値には、**http** と **cloudevents** が含まれます。

    コマンドの例

```
$ kn func create -l typescript -t cloudevents examplefunc
```

    出力例

```
Created typescript function in /home/user/demo/examplefunc
```

  - または、カスタムテンプレートを含むリポジトリーを指定することもできます。

    コマンドの例

```
$ kn func create -r https://github.com/boson-project/templates/ -l node -t hello-world
examplefunc
```

    出力例

```
Created node function in /home/user/demo/examplefunc
```

4.6.2. 機能をローカルで実行する

kn func run コマンドを使用して、現在のディレクトリーまたは **--path** フラグで指定されたディレクト

リーで機能をローカルに実行できます。実行している関数が以前にビルドされたことがない場合、またはプロジェクトファイルが最後にビルドされてから変更されている場合、**kn func run** コマンドは、既定で関数を実行する前に関数をビルドします。

現在のディレクトリーで機能を実行するコマンドの例

```
$ kn func run
```

パスとして指定されたディレクトリーで機能を実行するコマンドの例

```
$ kn func run --path=<directory_path>
```

--build フラグを使用して、プロジェクトファイルに変更がなくても、機能を実行する前に既存のイメージを強制的に再構築することもできます。

ビルドフラグを使用した実行コマンドの例

```
$ kn func run --build
```

ビルド フラグを false に設定すると、イメージのビルドが無効になり、以前にビルドされたイメージを使用して機能が実行されます。

ビルドフラグを使用した実行コマンドの例

```
$ kn func run --build=false
```

help コマンドを使用して、**kn func run** コマンドオプションの詳細を確認できます。

help コマンドの構築

```
$ kn func help run
```

4.6.3. 関数のビルド

関数を実行する前に、関数プロジェクトをビルドする必要があります。**kn func run** コマンドを使用している場合、関数は自動的に構築されます。ただし、**kn func build** コマンドを使用すると、実行せずに関数をビルドできます。これは、上級ユーザーやデバッグシナリオに役立ちます。

kn func build は、コンピューターまたは OpenShift Container Platform クラスターでローカルに実行できる OCI コンテナーイメージを作成します。このコマンドは、関数プロジェクト名とイメージレジストリー名を使用して、関数の完全修飾イメージ名を作成します。

4.6.3.1. イメージコンテナーの種類

デフォルトでは、**kn func build** は、Red Hat Source-to-Image (S2I) テクノロジーを使用してコンテナーイメージを作成します。

Red Hat Source-to-Image (S2I) を使用したビルドコマンドの例

```
$ kn func build
```

--builder フラグをコマンドに追加し、**pack** 戦略を指定することで、代わりに CNCF Cloud Native Buildpacks テクノロジーを使用できます。

CNCF Cloud Native Buildpacks を使用したビルドコマンドの例

```
$ kn func build --builder pack
```

4.6.3.2. イメージレジストリーの種類

OpenShift Container Registry は、関数イメージを保存するためのイメージレジストリーとしてデフォルトで使用されます。

OpenShift Container Registry を使用したビルドコマンドの例

```
$ kn func build
```

出力例

```
Building function image
Function image has been built, image: registry.redhat.io/example/example-function:latest
```

--registry フラグを使用して、OpenShift Container Registry をデフォルトのイメージレジストリーとして使用することをオーバーライドできます。

quay.io を使用するように OpenShift Container Registry をオーバーライドするビルドコマンドの例

```
$ kn func build --registry quay.io/username
```

出力例

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

4.6.3.3. Push フラグ

--push フラグを **kn func build** コマンドに追加して、正常にビルドされた後に関数イメージを自動的にプッシュできます。

OpenShift Container Registry を使用したビルドコマンドの例

```
$ kn func build --push
```

4.6.3.4. Help コマンド

kn func build コマンドオプションの詳細については、help コマンドを使用できます。

help コマンドの構築

```
$ kn func help build
```

4.6.4. 関数のデプロイ

kn func deploy コマンドを使用して、関数を Knative サービスとしてクラスターにデプロイできます。ターゲット関数がすでにデプロイされている場合には、コンテナーイメージレジストリーにプッシュされている新規コンテナーイメージで更新され、Knative サービスが更新されます。

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。
- Knative (**kn**) CLI をインストールしている。
- 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。
- デプロイする関数を作成し、初期化している必要がある。

手順

- 関数をデプロイします。

```
$ kn func deploy [-n <namespace> -p <path> -i <image>]
```

出力例

```
Function deployed at: http://func.example.com
```

  - **namespace** が指定されていない場合には、関数は現在の namespace にデプロイされます。
  - この関数は、**パス** が指定されない限り、現在のディレクトリーからデプロイされます。
  - Knative サービス名はプロジェクト名から派生するので、以下のコマンドでは変更できません。

4.6.5. 既存の関数の一覧表示

kn func list を使用して既存の関数を一覧表示できます。Knative サービスとしてデプロイされた関数を一覧表示するには、**kn service list** を使用することもできます。

手順

- 既存の関数を一覧表示します。

```
$ kn func list [-n <namespace> -p <path>]
```

出力例

```
NAME            NAMESPACE  RUNTIME  URL
READY
example-function  default    node     http://example-function.default.apps.ci-ln-g9f36hb-
d5d6b.origin-ci-int-aws.dev.rhcloud.com  True
```

-

- Knative サービスとしてデプロイされた関数を一覧表示します。

```
$ kn service list -n <namespace>
```

出力例

```
NAME            URL                                                                LATEST
AGE   CONDITIONS   READY   REASON
example-function   http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-
aws.dev.rhcloud.com   example-function-gzl4c   16m   3 OK / 3     True
```

4.6.6. 関数の記述

kn func info コマンドは、関数名、イメージ、namespace、Knative サービス情報、ルート情報、イベントサブスクリプションなどのデプロイされた関数に関する情報を出力します。

手順

- 関数を説明します。

```
$ kn func info [-f <format> -n <namespace> -p <path>]
```

コマンドの例

```
$ kn func info -p function/example-function
```

出力例

```
Function name:
  example-function
Function is built in image:
  docker.io/user/example-function:latest
Function is deployed as Knative Service:
  example-function
Function is deployed in namespace:
  default
Routes:
  http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com
```

4.6.7. テストイベントでのデプロイされた関数の呼び出し

kn func invoke CLI コマンドを使用して、ローカルまたは OpenShift Container Platform クラスター上で関数を呼び出すためのテストリクエストを送信できます。このコマンドを使用して、関数が機能し、イベントを正しく受信できることをテストできます。関数をローカルで呼び出すと、関数開発中の簡単なテストに役立ちます。クラスターで関数を呼び出すと、実稼働環境に近いテストに役立ちます。

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。

- Knative (**kn**) CLI をインストールしている。
- 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。
- 呼び出す関数をすでにデプロイしている必要があります。

手順

- 関数を呼び出します。

```
$ kn func invoke
```

  - **kn func invoke** コマンドは、ローカルのコンテナーイメージが実行中の場合や、クラスターにデプロイされた関数がある場合にのみ機能します。
  - **kn func invoke** コマンドは、デフォルトでローカルディレクトリーで実行され、このディレクトリーが関数プロジェクトであると想定します。

4.6.7.1. kn func はオプションのパラメーターを呼び出します

次の **knfuncinvoke** CLI コマンドフラグを使用して、リクエストのオプションのパラメーターを指定できます。

フラグ	説明
-t、--target	呼び出された関数のターゲットインスタンスを指定します。たとえば、**local**、**remote**、**https://staging.example.com/** などです。デフォルトのターゲットは **local** です。
-f、-format	メッセージの形式を指定します (例: **cloudevent** または **http**)。
--id	リクエストの一意の文字列識別子を指定します。
-n、-namespace	クラスターの namespace を指定します。
--source	リクエストの送信者名を指定します。これは、CloudEvent **source** 属性に対応します。
--type	リクエストのタイプを指定します (例: **boson.fn**)。これは、CloudEvent **type** 属性に対応します。
--data	リクエストの内容を指定します。CloudEvent リクエストの場合、これは CloudEvent **data** 属性です。
--file	送信するデータを含むローカルファイルへのパスを指定します。
--content-type	リクエストの MIME コンテンツタイプを指定します。
-p、--path	プロジェクトディレクトリーへのパスを指定します。

フラグ	説明
-c、--confirm	すべてのオプションを対話的に確認するように要求を有効にします。
-v, --verbose	詳細出力の出力を有効にします。
-h、--help	**kn func invoke** の使用法に関する情報を出力します。

4.6.7.1.1. 主なパラメーター

次のパラメーターは、**kn func invoke** コマンドの主なプロパティーを定義します。

イベントターゲット (-t、-target)

呼び出された関数のターゲットインスタンス。ローカルにデプロイされた関数の **local** 値、リモートにデプロイされた関数の **remote** 値、または任意のエンドポイントにデプロイされた関数の URL を受け入れます。ターゲットが指定されていない場合、デフォルトで **local** になります。

イベントメッセージ形式 (-f、--format)

http や **cloudevent** などのイベントのメッセージ形式。これは、デフォルトで、関数の作成時に使用されたテンプレートの形式になります。

イベントタイプ (--type)

送信されるイベントのタイプ。各イベントプロデューサーのドキュメントで設定されている **type** パラメーターに関する情報を見つけることができます。たとえば、API サーバーソースは、生成されたイベントの **type** パラメーターを **dev.knative.apiserver.resource.update** として設定する場合があります。

イベントソース (--source)

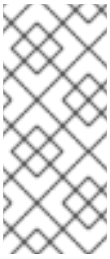
イベントを生成する一意のイベントソース。これは、**https://10.96.0.1/** などのイベントソースの URI、またはイベントソースの名前である可能性があります。

イベント ID (--id)

イベントプロデューサーによって作成されるランダムな一意の ID。

イベントデータ (--data)

kn func invoke コマンドで送信されるイベントの **data** 値を指定できます。たとえば、イベントにこのデータ文字列が含まれるように、**"Hello World"** などの **--data** 値を指定できます。デフォルトでは、**kn func invoke** によって作成されたイベントにデータは含まれません。



注記

クラスターにデプロイされた関数は、**source** および **type** などのプロパティーの値を提供する既存のイベントソースからのイベントに応答できます。多くの場合、これらのイベントには、イベントのドメイン固有のコンテキストをキャプチャーする JSON 形式の **data** 値があります。本書に記載されている CLI フラグを使用して、開発者はローカルテスト用にこれらのイベントをシミュレートできます。

--file フラグを使用してイベントデータを送信し、イベントのデータを含むローカルファイルを指定することもできます。この場合は、**--content-type** を使用してコンテンツタイプを指定します。

データコンテンツタイプ (--content-type)

--data フラグを使用してイベントのデータを追加している場合は、**-content-type** フラグを使用して、イベントによって伝送されるデータのタイプを指定できます。前の例では、データはプレーン

テキストであるため、**kn func invoke --data "Hello world!" --content-type "text/plain"** を指定できます。

4.6.7.1.2. コマンドの例

これは、**kn func invoke** コマンドの一般的な呼び出しです。

```
$ kn func invoke --type <event_type> --source <event_source> --data <event_data> --content-type
<content_type> --id <event_ID> --format <format> --namespace <namespace>
```

たとえば、Hello world! イベントを送信すると、以下を行うことができます。

```
$ kn func invoke --type ping --source example-ping --data "Hello world!" --content-type "text/plain" --
id example-ID --format http --namespace my-ns
```

4.6.7.1.2.1. データを使用したファイルの指定

イベントデータが含まれるディスクにファイルを指定するには、**--file** フラグおよび **--content-type** フラグを使用します。

```
$ kn func invoke --file <path> --content-type <content-type>
```

たとえば、**test.json** ファイルに保存されている JSON データを送信するには、以下のコマンドを使用します。

```
$ kn func invoke --file ./test.json --content-type application/json
```

4.6.7.1.2.2. 関数プロジェクトの指定

--path フラグを使用して、関数プロジェクトへのパスを指定できます。

```
$ kn func invoke --path <path_to_function>
```

たとえば、**./example/example-** function ディレクトリーにある function プロジェクトを使用するには、以下のコマンドを使用します。

```
$ kn func invoke --path ./example/example-function
```

4.6.7.1.2.3. ターゲット関数がデプロイされる場所の指定

デフォルトでは、**kn func invoke** は関数のローカルデプロイメントをターゲットにします。

```
$ kn func invoke
```

別のデプロイメントを使用するには、**--target** フラグを使用します。

```
$ kn func invoke --target <target>
```

たとえば、クラスターにデプロイされた関数を使用するには、**-target remote** フラグを使用します。

```
$ kn func invoke --target remote
```

任意の URL にデプロイされた関数を使用するには、**-target <URL>** フラグを使用します。

```
$ kn func invoke --target "https://my-event-broker.example.com"
```

ローカルデプロイメントを明示的にターゲットとして指定できます。この場合、関数がローカルで実行されていない場合、コマンドは失敗します。

```
$ kn func invoke --target local
```

4.6.8. 関数の削除

kn func delete コマンドを使用して関数を削除できます。これは、関数が不要になった場合に役立ち、クラスターのリソースを節約するのに役立ちます。

手順

- 関数を削除します。

  ```
  $ kn func delete [<function_name> -n <namespace> -p <path>]
  ```

  - 削除する関数の名前またはパスが指定されていない場合には、現在のディレクトリーで **func.yaml** ファイルを検索し、削除する関数を判断します。
  - namespace が指定されていない場合には、**func.yaml** の **namespace** の値にデフォルト設定されます。

第5章 開発

5.1. SERVERLESS アプリケーション

サーバーレスアプリケーションは、ルートと設定で定義され、YAML ファイルに含まれる Kubernetes サービスとして作成およびデプロイされます。OpenShift Serverless を使用してサーバーレスアプリケーションをデプロイするには、Knative **Service** オブジェクトを作成する必要があります。

Knative Service オブジェクトの YAML ファイルの例

```
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 についてのドキュメントを参照してください。
- Knative (**kn**) CLI を使用して Knative サービスを作成します。
- **oc** CLI を使用して、Knative **Service** オブジェクトを YAML ファイルとして作成し、適用します。

5.1.1. Knative CLI を使用したサーバーレスアプリケーションの作成

Knative (**kn**) CLI を使用してサーバーレスアプリケーションを作成すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。**kn service create** コマンドを使用して、基本的なサーバーレスアプリケーションを作成できます。

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。

- Knative (**kn**) CLI をインストールしている。
- 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。

手順

- Knative サービスを作成します。

```
$ kn service create <service_name> --image <image> --tag <tag-value>
```

詳細は以下のようになります。

- **--image** は、アプリケーションのイメージの URI です。
- **--tag** は、サービスで作成される初期リビジョンにタグを追加するために使用できるオプションのフラグです。

コマンドの例

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

出力例

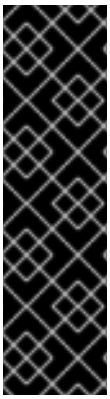
```
Creating service 'event-display' in namespace 'default':

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

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

5.1.2. オフラインモードを使用したサービスの作成

オフラインモードで **kn service** コマンドを実行すると、クラスター上で変更は発生せず、代わりにサービス記述子ファイルがローカルマシンに作成されます。記述子ファイルを作成した後、クラスターに変更を伝播する前にファイルを変更することができます。



重要

Knative CLI のオフラインモード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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。
- Knative (**kn**) CLI をインストールしている。

手順

1. オフラインモードでは、ローカルの Knative サービス記述子ファイルを作成します。

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

出力例

```
Service 'event-display' created in namespace 'test'.
```

- **--target ./** フラグはオフラインモードを有効にし、**./** を新しいディレクトリーツリーを保存するディレクトリーとして指定します。
  既存のディレクトリーを指定せずに、**--target my-service.yaml** などのファイル名を使用すると、ディレクトリーツリーは作成されません。代わりに、サービス記述子ファイル **my-service.yaml** のみが現在のディレクトリーに作成されます。

  ファイル名には、**.yaml**、**.yml** または **.json** 拡張子を使用できます。**.json** を選択すると、JSON 形式でサービス記述子ファイルが作成されます。

- **--namespace test** オプションは、新規サービスを **テスト** namespace に配置します。
  --namespace を使用せずに、OpenShift クラスターにログインしている場合には、記述子ファイルが現在の namespace に作成されます。それ以外の場合は、記述子ファイルが **default** の namespace に作成されます。

2. 作成したディレクトリー構造を確認します。

```
$ tree ./
```

出力例

```
./
└── test
    └── ksvc
        └── event-display.yaml

2 directories, 1 file
```

- **--target** で指定する現在の **./** ディレクトリーには新しい **test/** ディレクトリーが含まれます。このディレクトリーの名前は、指定の namespace をもとに付けられます。
- **test/** ディレクトリーには、リソースタイプの名前が付けられた **ksvc** ディレクトリーが含まれます。

- **ksvc** ディレクトリーには、指定のサービス名に従って命名される記述子ファイル **event-display.yaml** が含まれます。

3. 生成されたサービス記述子ファイルを確認します。

```
$ cat test/ksvc/event-display.yaml
```

出力例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  creationTimestamp: null
  name: event-display
  namespace: test
spec:
  template:
    metadata:
      annotations:
        client.knative.dev/user-image: quay.io/openshift-knative/knative-eventing-sources-event-
display:latest
      creationTimestamp: null
    spec:
      containers:
      - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
        name: ""
        resources: {}
status: {}
```

4. 新しいサービスに関する情報を一覧表示します。

```
$ kn service describe event-display --target ./ --namespace test
```

出力例

```
Name:       event-display
Namespace:  test
Age:
URL:

Revisions:

Conditions:
  OK TYPE    AGE REASON
```

- **--target ./** オプションは、namespace サブディレクトリーを含むディレクトリー構造のルートディレクトリーを指定します。
  または、**--target** オプションで YAML または JSON ファイルを直接指定できます。使用可能なファイルの拡張子は、**.yaml**、**.yml**、および **.json** です。

- **--namespace** オプションは、namespace を指定し、この namespace は必要なサービス記述子ファイルを含むサブディレクトリーの **kn** と通信します。

--namespace を使用せず、OpenShift クラスターにログインしている場合には、**kn** は現在の namespace をもとに名前が付けられたサブディレクトリーでサービスを検索します。それ以外の場合は、**kn** は **default**/ サブディレクトリーで検索します。

5. サービス記述子ファイルを使用してクラスターでサービスを作成します。

```
$ kn service create -f test/ksvc/event-display.yaml
```

出力例

```
Creating service 'event-display' in namespace 'test':

  0.058s The Route is still working to reflect the latest desired specification.
  0.098s ...
  0.168s Configuration "event-display" is waiting for a Revision to become ready.
 23.377s ...
 23.419s Ingress has not yet been reconciled.
 23.534s Waiting for load balancer to be ready
 23.723s Ready to serve.

Service 'event-display' created to latest revision 'event-display-00001' is available at URL:
http://event-display-test.apps.example.com
```

5.1.3. YAML を使用したサーバーレスアプリケーションの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でアプリケーションを宣言的に記述することができます。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、Knative **Service** 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。

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、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。

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。
- 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。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 以下のサンプルコードを含む YAML ファイルを作成します。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-delivery
  namespace: default
```

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

5.1.4. 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。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** からの出力は常に **http://** 形式を使用して URL を出力します。

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。
- **oc** CLI がインストールされている。
- Knative サービスを作成している。

前提条件

- OpenShift CLI (**oc**) をインストールしている。

手順

1. アプリケーション URL を検索します。

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

コマンドの例

```
$ oc get ksvc event-delivery
```

出力例

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

HTTPS 要求の例

```
$ curl https://event-delivery-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。

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

出力例

```
Hello Serverless!
```

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。

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。

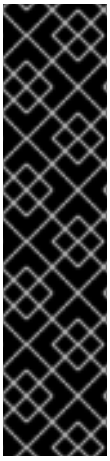
```
$ curl https://event-delivery-default.example.com --cacert <file>
```

出力例

```
Hello Serverless!
```

5.1.5. 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。



重要

この方法は、**LoadBalancer** サービスタイプを使用して Kourier Gateway を公開する必要があります。これは、以下の YAML を **KnativeServing** カスタムリソース定義 (CRD) に追加して設定できます。

```
...
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...
```

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。
- OpenShift CLI (**oc**) をインストールしている。
- Knative サービスを作成している。

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** フィールドで表示され、この場合は **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。

```
$ 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。

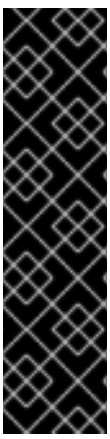
5.1.6. 制限のあるネットワークポリシーを持つクラスターでの Knative アプリケーションとの通信の有効化

複数のユーザーがアクセスできるクラスターを使用している場合、クラスターはネットワークポリシーを使用してネットワーク経由で相互に通信できる Pod、サービス、および namespace を制御する可能性があります。クラスターで制限的なネットワークポリシーを使用する場合は、Knative システム Pod が Knative アプリケーションにアクセスできない可能性があります。たとえば、namespace に、すべての要求を拒否する以下のネットワークポリシーがある場合、Knative システム Pod は Knative アプリケーションにアクセスできません。

namespace へのすべての要求を拒否する NetworkPolicy オブジェクトの例

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
  namespace: example-namespace
spec:
  podSelector:
  ingress: []
```

Knative システム Pod からアプリケーションへのアクセスを許可するには、ラベルを各 Knative システム namespace に追加し、このラベルを持つ他の namespace の namespace へのアクセスを許可する アプリケーション namespace に **NetworkPolicy** オブジェクトを作成する必要があります。



重要

クラスターの非 Knative サービスへの要求を拒否するネットワークポリシーは、これらのサービスへのアクセスを防止するネットワークポリシーです。ただし、Knative システム namespace から Knative アプリケーションへのアクセスを許可することにより、クラスターのすべての namespace から Knative アプリケーションへのアクセスを許可する必要があります。

クラスターのすべての namespace から Knative アプリケーションへのアクセスを許可しない場合は、代わりに **Knative サービスの JSON Web Token 認証**を使用するようにしてください。Knative サービスの JSON Web トークン認証にはサービスメッシュが必要です。

前提条件

- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

1. アプリケーションへのアクセスを必要とする各 Knative システム namespace に **knative.openshift.io/system-namespace=true** ラベルを追加します。
   a. **knative-serving** namespace にラベルを付けます。

   ```
   $ oc label namespace knative-serving knative.openshift.io/system-namespace=true
   ```

   b. **knative-serving-ingress** namespace にラベルを付けます。

   ```
   $ oc label namespace knative-serving-ingress knative.openshift.io/system-namespace=true
   ```

   c. **knative-eventing namespace** にラベルを付けます。

   ```
   $ oc label namespace knative-eventing knative.openshift.io/system-namespace=true
   ```

   d. **knative-kafka namespace** にラベルを付けます。

   ```
   $ oc label namespace knative-kafka knative.openshift.io/system-namespace=true
   ```

2. アプリケーション namespace で **NetworkPolicy** オブジェクトを作成し、**knative.openshift.io/system-namespace** ラベルのある namespace からのアクセスを許可します。

   サンプル NetworkPolicy オブジェクト

   ```
   apiVersion: networking.k8s.io/v1
   kind: NetworkPolicy
   metadata:
     name: <network_policy_name> 1
     namespace: <namespace> 2
   spec:
     ingress:
     - from:
       - namespaceSelector:
           matchLabels:
             knative.openshift.io/system-namespace: "true"
     podSelector: {}
     policyTypes:
     - Ingress
   ```

   1 ネットワークポリシーの名前を指定します。

   2 アプリケーションが存在する namespace。

5.1.7. init コンテナーの設定

Init コンテナー は、Pod 内のアプリケーションコンテナーの前に実行される特殊なコンテナーです。これらは通常、アプリケーションの初期化ロジックを実装するために使用されます。これには、セットアップスクリプトの実行や、必要な設定のダウンロードが含まれる場合があります。

注記

Init コンテナーを使用すると、アプリケーションの起動時間が長くなる可能性があるため、頻繁にスケールアップおよびスケールダウンすることが予想されるサーバーレスアプリケーションには注意して使用する必要があります。

複数の複数の init コンテナーは単一の Knative サービス仕様でサポートされます。テンプレート名が指定されていない場合、Knative はデフォルトの設定可能な名前付けテンプレートを提供します。初期化コンテナーテンプレートは、Knative **Service** オブジェクト仕様に適切な値を追加することで設定できます。

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。
- Knative サービスに init コンテナーを使用する前に、管理者は **kubernetes.podspec-init-containers** フラグを **KnativeServing** カスタムリソース (CR) に追加する必要があります。詳細については、OpenShift Serverless のグローバルコンフィギュレーションドキュメントを参照してください。

手順

- **initContainers** 仕様を **KnativeService** オブジェクトに追加します。

  サービス仕様の例

```
apiVersion: serving.knative.dev/v1
kind: Service
...
spec:
  template:
    spec:
      initContainers:
        - imagePullPolicy: IfNotPresent (1)
          image: <image_uri> (2)
          volumeMounts: (3)
            - name: data
              mountPath: /data
...
```

(1) イメージのダウンロード時の イメージプルポリシー。

(2) init コンテナーイメージの URI。

(3) コンテナーファイルシステム内でボリュームがマウントされる場所。

5.1.8. サービスごとの HTTPS リダイレクト

networking.knative.dev/http-option アノテーションを設定することにより、サービスの HTTPS リダイレクトを有効または無効にできます。次の例は、Knative **Service** YAML オブジェクトでこのアノテーションを使用する方法を示しています。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example
  namespace: default
  annotations:
    networking.knative.dev/http-option: "redirected"
spec:
  ...
```

5.1.9. 関連情報

- Knative Serving CLI コマンド
- Knative サービスの JSON Web Token 認証の設定

5.2. 自動スケーリング

Knative Serving は、アプリケーションが受信要求に一致するように、自動スケーリング (**autoscaling**) を提供します。たとえば、アプリケーションがトラフィックを受信せず、scale-to-zero が有効にされている場合、Knative Serving はアプリケーションをゼロレプリカにスケールダウンします。scale-to-zero が無効になっている場合、アプリケーションはクラスターのアプリケーションに設定された最小のレプリカ数にスケールダウンされます。アプリケーションへのトラフィックが増加したら、要求を満たすようにレプリカをスケールアップすることもできます。

Knative サービスの自動スケーリング設定は、クラスター管理者によって設定されるグローバル設定とすることも、個別サービスに設定されるリビジョンごとの設定とすることもできます。OpenShift Container Platform Web コンソールを使用して、サービスの YAML ファイルを変更するか、または Knative (**kn**) CLI を使用して、サービスのリビジョンごとの設定を変更できます。

注記

サービスに設定した制限またはターゲットは、アプリケーションの単一インスタンスに対して測定されます。たとえば、**target** アノテーションを **50** に設定することにより、各リビジョンが一度に 50 の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。

5.2.1. スケーリング限度

スケーリング限度は、任意の時点でアプリケーションに対応できる最小および最大のレプリカ数を決定します。アプリケーションのスケーリング限度を設定して、コールドスタートを防止したり、コンピューティングコストを制御したりできます。

5.2.1.1. スケーリング下限

アプリケーションにサービスを提供できるレプリカの最小数は、最小 **min-scale** のアノテーションによって決定されます。ゼロへのスケーリングが有効になっていない場合、**min-Scale** 値のデフォルトは **1** になります。

次の条件が満たされた場合、**min-scale** 値はデフォルトで **0** レプリカになります。

- **mi-scale** の注釈が設定されていません
- ゼロへのスケーリングが有効にされている
- **KPA** クラスが使用されている

min-scale アノテーションを使用したサービス仕様の例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/min-scale: "0"
...
```

5.2.1.1.1. Knative CLI を使用した最小スケール注釈の設定

minScale アノテーションを設定するために Knative (**kn**) CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが提供されます。**kn service** コマンドを **--scale-min** フラグと共に使用して、サービスの **--min-scale** 値を作成または変更できます。

前提条件

- Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。

手順

- **--scale-min** フラグを使用して、サービスのレプリカの最小数を設定します。

```
$ kn service create <service_name> --image <image_uri> --scale-min <integer>
```

コマンドの例

```
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --scale-min 2
```

5.2.1.2. スケーリング上限

アプリケーションにサービスを提供できるレプリカの最大数は、**max-scale** アノテーションによって決定されます。**max-scale** アノテーションが設定されていない場合、作成されるレプリカの数に上限はありません。

max-scale アノテーションを使用したサービス仕様の例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/max-scale: "10"
...
```

5.2.1.2.1. Knative CLI を使用した最大スケール注釈の設定

Knative (**kn**) CLI を使用して **max-scale** のアノテーションを設定すると、YAML ファイルを直接変更する場合に比べ、ユーザーインターフェイスがより合理的で直感的です。**--scale-max** フラグを指定して knservice コマンドを使用すると、**kn service** の **max-scale** 値を作成または変更できます。

前提条件

- Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。

手順

- **--scale-max** フラグを使用して、サービスのレプリカの最大数を設定します。

```
$ kn service create <service_name> --image <image_uri> --scale-max <integer>
```

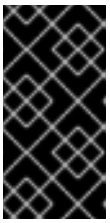
コマンドの例

```
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --scale-max 10
```

5.2.2. 並行処理性

並行処理性は、特定の時点でアプリケーションの各レプリカが処理できる同時リクエストの数を決定します。並行処理性は、**ソフトリミット**または**ハードリミット**のいずれかとして設定できます。

- ソフトリミットは、厳格に強制される限度ではなく、目標となるリクエストの限度です。たとえば、トラフィックの急増が発生した場合、ソフトリミットのターゲットを超過できます。
- ハードリミットは、リクエストに対して厳密に適用される上限です。並行処理がハードリミットに達すると、それ以降のリクエストはバッファー処理され、リクエストを実行するのに十分な空き容量ができるまで待機する必要があります。



重要

ハードリミット設定の使用は、アプリケーションに明確なユースケースがある場合にのみ推奨されます。ハードリミットを低い値に指定すると、アプリケーションのスループットとレイテンシーに悪影響を与える可能性があり、コールドスタートが発生する可能性があります。

ソフトターゲットとハードリミットを追加することは、Autoscaler は同時リクエストのソフトターゲット数を目標とするが、リクエストの最大数にハードリミット値のハードリミットを課すことを意味します。

ハードリミットの値がソフトリミットの値より小さい場合、実際に処理できる数よりも多くのリクエストを目標にする必要がないため、ソフトリミットの値が低減されます。

5.2.2.1. ソフト並行処理ターゲットの設定

ソフトリミットは、厳格に強制される限度ではなく、目標となるリクエストの限度です。たとえば、トラフィックの急増が発生した場合、ソフトリミットのターゲットを超過できます。**autoscaling.knative.dev/target** アノテーションを仕様に設定するか、または正しいフラグを指定して **kn service** コマンドを使用して、Knative サービスにソフト並行処理ターゲットを指定できます。

手順

- オプション:**Service** カスタムリソースの仕様で Knative サービスに **autoscaling.knative.dev/target** アノテーションを設定します。

  サービス仕様の例

  ```
  apiVersion: serving.knative.dev/v1
  kind: Service
  metadata:
    name: example-service
    namespace: default
  spec:
    template:
      metadata:
        annotations:
          autoscaling.knative.dev/target: "200"
  ```

- オプション:**kn service** コマンドを使用して **--concurrency-target** フラグを指定します。

  ```
  $ kn service create <service_name> --image <image_uri> --concurrency-target <integer>
  ```

  並行処理のターゲットを 50 リクエストに設定したサービスを作成するコマンドの例

  ```
  $ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --concurrency-target 50
  ```

5.2.2.2. ハード並行処理リミットの設定

ハード並行処理リミットは、リクエストに対して厳密に適用される上限です。並行処理がハードリミットに達すると、それ以降のリクエストはバッファー処理され、リクエストを実行するのに十分な空き容量ができるまで待機する必要があります。**containerConcurrency** 仕様を変更するか、または正しいフラグを指定して **kn service** コマンドを使用して、Knative サービスにハード並行処理リミットを指定できます。

手順

- オプション:**Service** カスタムリソースの仕様で Knative サービスに **containerConcurrency** 仕様を設定します。

サービス仕様の例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    spec:
      containerConcurrency: 50
```

デフォルト値は **0** です。これは、サービスの 1 つのレプリカに一度に流れることができる同時リクエストの数に制限がないことを意味します。

0 より大きい値は、サービスの 1 つのレプリカに一度に流れることができるリクエストの正確な数を指定します。この例では、50 リクエストのハード並行処理リミットを有効にします。

- オプション:**kn service** コマンドを使用して **--concurrency-limit** フラグを指定します。

```
$ kn service create <service_name> --image <image_uri> --concurrency-limit <integer>
```

並行処理のリミットを 50 リクエストに設定したサービスを作成するコマンドの例

```
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --concurrency-limit 50
```

5.2.2.3. 並行処理ターゲットの使用率

この値は、Autoscaler が実際に目標とする並行処理リミットのパーセンテージを指定します。これは、レプリカが実行する **ホット度** を指定することとも呼ばれます。これにより、Autoscaler は定義されたハードリミットに達する前にスケールアップできるようになります。

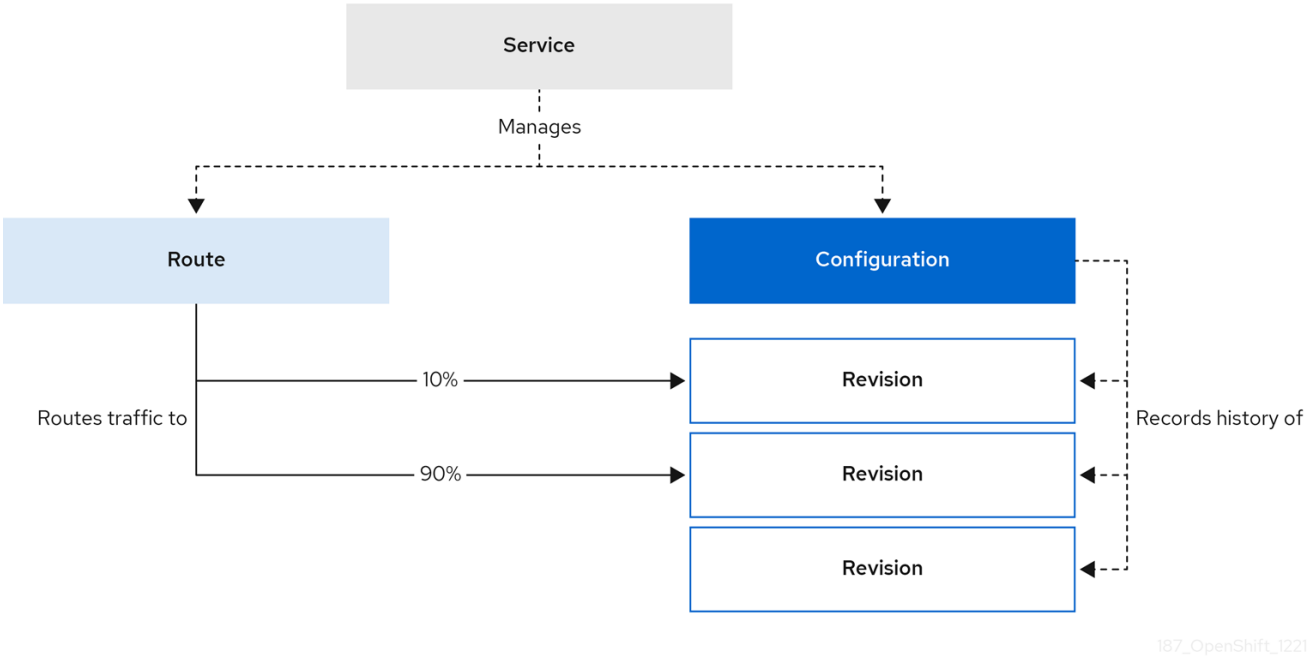
たとえば、**containerConcurrency** 値が 10 に設定され、 **target-utilization-percentage** 値が 70% に設定されている場合、既存のすべてのレプリカの同時リクエストの平均数が 7 に達すると、オートスケーラーは新しいレプリカを作成します。7 から 10 の番号が付けられたリクエストは引き続き既存のレプリカに送信されますが、**containerConcurrency** 値に達した後、必要になることを見越して追加のレプリカが開始されます。

target-utilization-percentage アノテーションを使用して設定されたサービスの例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target-utilization-percentage: "70"
...
```

5.3. トラフィック管理

Knative アプリケーションでは、トラフィック分割を作成することでトラフィックを管理できます。トラフィック分割は、Knative サービスによって管理されるルートの一部として設定されます。



ルートを設定すると、サービスのさまざまなリビジョンにリクエストを送信できます。このルーティングは、**Service** オブジェクトの **traffic** 仕様によって決定されます。

traffic 仕様宣言は、1 つ以上のリビジョンで設定され、それぞれがトラフィック全体の一部を処理する責任があります。各リビジョンにルーティングされるトラフィックの割合は、合計で 100% になる必要があります。これは、Knative 検証によって保証されます。

traffic 仕様で指定されたリビジョンは、固定の名前付きリビジョンにすることも、サービスのすべてのリビジョンのリストの先頭を追跡する最新のリビジョンを指すこともできます。最新のリビジョンは、新しいリビジョンが作成された場合に更新される一種のフローティング参照です。各リビジョンには、そのリビジョンの追加のアクセス URL を作成するタグを付けることができます。

traffic 仕様は次の方法で変更できます。

- **Service** オブジェクトの YAML を直接編集します。
- Knative (**kn**) CLI **--traffic** フラグを使用します。
- OpenShift Container Platform Web コンソールの使用

Knative サービスの作成時に、デフォルトの **traffic** 仕様設定は含まれません。

5.3.1. トラフィックスペックの例

以下の例は、トラフィックの 100% がサービスの最新リビジョンにルーティングされる **traffic** 仕様を示しています。**status** では、**latestRevision** が解決する最新リビジョンの名前を確認できます。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
```

```
  name: example-service
  namespace: default
spec:
...
  traffic:
  - latestRevision: true
    percent: 100
status:
  ...
  traffic:
  - percent: 100
    revisionName: example-service
```

以下の例は、トラフィックの 100% が **current** としてタグ付けされたリビジョンにルーティングされ、そのリビジョンの名前が **example-service** として指定される **traffic** 仕様を示しています。**latest** とタグ付けされたリビジョンは、トラフィックが宛先にルーティングされない場合でも、利用可能な状態になります。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - tag: current
    revisionName: example-service
    percent: 100
  - tag: latest
    latestRevision: true
    percent: 0
```

以下の例は、トラフィックが複数のリビジョン間で分割されるように、**traffic** 仕様のリビジョンの一覧を拡張する方法を示しています。この例では、トラフィックの 50% を、**current** としてタグ付けされたリビジョンに送信します。また、**candidate** としてタグ付けされたリビジョンにトラフィックの 50% を送信します。**latest** とタグ付けされたリビジョンは、トラフィックが宛先にルーティングされない場合でも、利用可能な状態になります。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - tag: current
    revisionName: example-service-1
    percent: 50
  - tag: candidate
    revisionName: example-service-2
    percent: 50
```

```
- tag: latest
  latestRevision: true
  percent: 0
```

5.3.2. Knative CLI トラフィック管理フラグ

Knative (**kn**) CLI 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をサポートします。

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。**Repetition** 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。	はい
--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。	いいえ
--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。	いいえ
--untag	**Tag**	リビジョンから **Tag** を削除します。	はい

5.3.2.1. 複数のフラグおよび順序の優先順位

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。

タグをリビジョンに追加してから、設定したタグに応じてトラフィックを分割することができます。

5.3.2.2. リビジョンのカスタム URL

kn service update コマンドを使用して **--tag** フラグをサービスに割り当てると、サービスの更新時に作成されるリビジョンのカスタム URL が作成されます。カスタム URL は、**https://<tag>-<service_name>-<namespace>.<domain>;** パターンまたは **http://<tag>-<service_name>-<namespace>.<domain>;** パターンに従います。

--tag フラグおよび **--untag** フラグは以下の構文を使用します。

- 1 つの値が必要です。
- サービスのトラフィックブロックに一意のタグを示します。
- 1 つのコマンドで複数回指定できます。

5.3.2.2.1. 例: リビジョンへのタグの割り当て

以下の例では、タグ **latest** を、**example-revision** という名前のリビジョンに割り当てます。

```
$ kn service update <service_name> --tag @latest=example-tag
```

5.3.2.2.2. 例: リビジョンからのタグの削除

--untag フラグを使用して、カスタム URL を削除するタグを削除できます。

注記

リビジョンのタグが削除され、トラフィックの 0% が割り当てられる場合、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。

以下のコマンドは、**example-revision** という名前のリビジョンからすべてのタグを削除します。

```
$ kn service update <service_name> --untag example-tag
```

5.3.3. KnativeCLI を使用してトラフィック分割を作成する

Knative (**kn**) CLI を使用してトラフィック分割を作成すると、YAML ファイルを直接変更するよりも合理的で直感的なユーザーインターフェイスが提供されます。**kn service update** コマンドを使用して、サービスのリビジョン間でトラフィックを分割できます。

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。
- Knative (**kn**) CLI をインストールしている。
- Knative サービスを作成している。

手順

- 標準の **kn service update** コマンドで **--traffic** タグを使用して、サービスのリビジョンとそれにルーティングするトラフィックの割合を指定します。

  コマンドの例

  ```
  $ kn service update <service_name> --traffic <revision>=<percentage>
  ```

  詳細は以下のようになります。

  - **<service_name>** は、トラフィックルーティングを設定する Knative サービスの名前です。
  - **<revision>** は、一定の割合のトラフィックを受信するように設定するリビジョンです。リビジョンの名前、または **--tag** フラグを使用してリビジョンに割り当てたタグのいずれかを指定できます。
  - **<percentage>** は、指定されたリビジョンに送信するトラフィックのパーセンテージです。

- オプション: **--traffic** フラグは、1 つのコマンドで複数回指定できます。たとえば、**@latest** というタグの付いたリビジョンと **stable** という名前のリビジョンがある場合、次のように各リビジョンに分割するトラフィックの割合を指定できます。

  コマンドの例

  ```
  $ kn service update example-service --traffic @latest=20,stable=80
  ```

  複数のリビジョンがあり、最後のリビジョンに分割する必要があるトラフィックの割合を指定しない場合、**-traffic** フラグはこれを自動的に計算できます。たとえば、**example** という名前の 3 番目のリビジョンがあり、次のコマンドを使用する場合:

  コマンドの例

  ```
  $ kn service update example-service --traffic @latest=10,stable=60
  ```

  トラフィックの残りの 30% は、指定されていなくても、**example** リビジョンに分割されます。

5.3.4. OpenShift Container Platform Web コンソールを使用したリビジョン間のトラフィックの管理

サーバーレスアプリケーションの作成後、アプリケーションは OpenShift Container Platform Web コンソールの **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によって表され、Knative サービス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、特定のタイミングでコードのスナップショットである新規リビジョンが作成されます。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。

前提条件

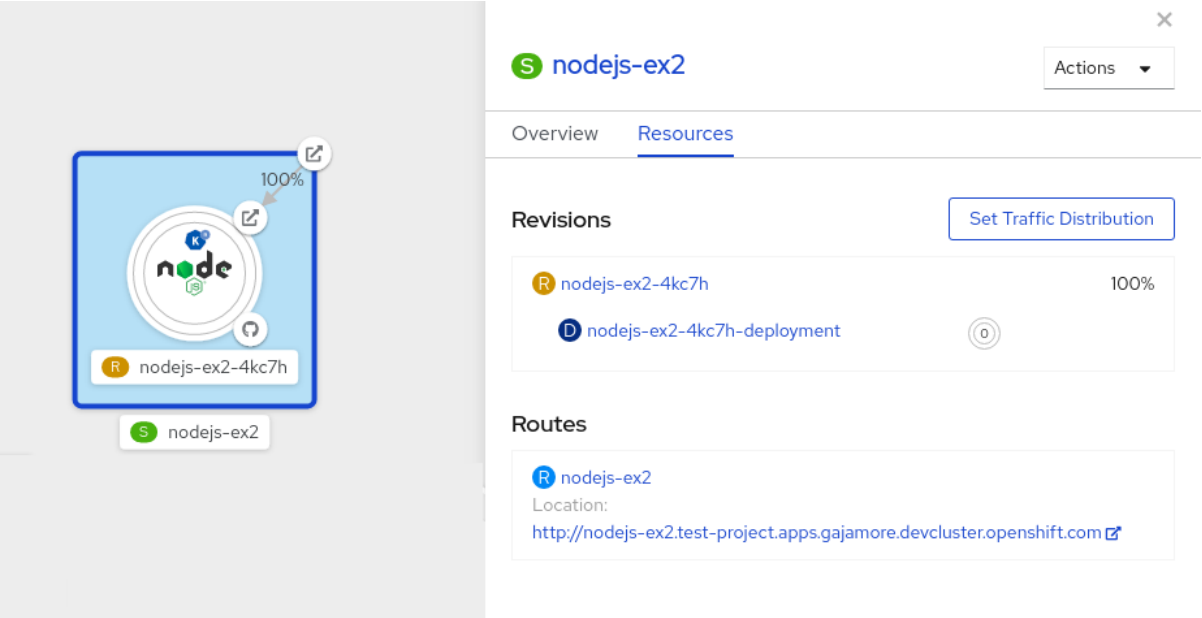
- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- OpenShift Container Platform Web コンソールにログインしている。

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。

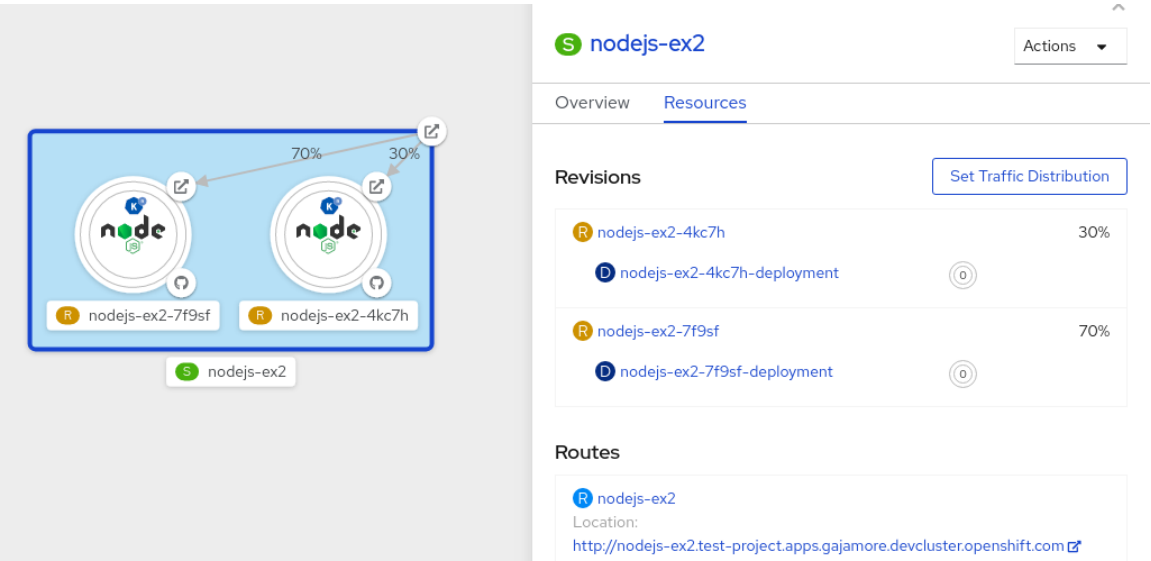
1. Knative サービスをクリックし、サイドパネルの概要を表示します。
2. **Resources** タブをクリックして、サービスの **Revisions** および **Routes** の一覧を表示します。

図5.1 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** 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5.2 Serverless アプリケーションのリビジョン



5.3.5. blue-green デプロイメントストラテジーを使用したトラフィックのルーティングおよび管理

Blue-green デプロイメントストラテジー を使用して、実稼働バージョンのアプリケーションから新規バージョンにトラフィックを安全に再ルーティングすることができます。

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。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. アプリケーションを Knative サービスとして作成し、デプロイします。
2. 以下のコマンドから出力を表示して、サービスのデプロイ時に作成された最初のリビジョンの名前を検索します。

```
$ oc get ksvc <service_name> -o=jsonpath='{.status.latestCreatedRevisionName}'
```

コマンドの例

```
$ oc get ksvc example-service -o=jsonpath='{.status.latestCreatedRevisionName}'
```

出力例

```
$ example-service-00001
```

3. 以下の YAML をサービスの **spec** に追加して、受信トラフィックをリビジョンに送信します。

```
...
spec:
  traffic:
```

```
    - revisionName: <first_revision_name>
      percent: 100 # All traffic goes to this revision
...
```

4. 以下のコマンドを実行して、URL の出力でアプリケーションを表示できることを確認します。

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

5. サービスの **template** 仕様の少なくとも 1 つのフィールドを変更してアプリケーションの 2 番目のリビジョンをデプロイし、これを再デプロイします。たとえば、サービスの **image** や **env** 環境変数を変更できます。サービスの再デプロイは、サービスの YAML ファイルを適用するか、Knative (**kn**) CLI をインストールしている場合は、**kn service update** コマンドを使用します。

6. 以下のコマンドを実行して、サービスを再デプロイする際に作成された 2 番目の最新のリビジョンの名前を見つけます。

```
$ oc get ksvc <service_name> -o=jsonpath='{.status.latestCreatedRevisionName}'
```

この時点で、サービスの最初のバージョンと 2 番目のリビジョンの両方がデプロイされ、実行されます。

7. 既存のサービスを更新して、2 番目のリビジョンの新規テストエンドポイントを作成し、他のすべてのトラフィックを最初のリビジョンに送信します。

テストエンドポイントのある更新されたサービス仕様の例

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 100 # All traffic is still being routed to the first revision
    - revisionName: <second_revision_name>
      percent: 0 # No traffic is routed to the second revision
      tag: v2 # A named route
...
```

YAML リソースを再適用してこのサービスを再デプロイすると、アプリケーションの 番目のリビジョンがステージングされます。トラフィックはメインの URL の 2 番目のリビジョンにルーティングされず、Knative は新たにデプロイされたリビジョンをテストするために **v2** という名前の新規サービスを作成します。

8. 以下のコマンドを実行して、2 番目のリビジョンの新規サービスの URL を取得します。

```
$ oc get ksvc <service_name> --output jsonpath="{.status.traffic[*].url}"
```

この URL を使用して、トラフィックをルーティングする前に、新しいバージョンのアプリケーションが予想通りに機能していることを検証できます。

9. 既存のサービスを再度更新して、トラフィックの 50% が最初のリビジョンに送信され、50% が 2 番目のリビジョンに送信されます。

リビジョン間でトラフィックを 50/50 に分割する更新サービス仕様の例

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 50
    - revisionName: <second_revision_name>
      percent: 50
      tag: v2
...
```

10. すべてのトラフィックを新しいバージョンのアプリケーションにルーティングできる状態になったら、再度サービスを更新して、100% のトラフィックを 2 番目のリビジョンに送信します。

すべてのトラフィックを 2 番目のリビジョンに送信する更新済みのサービス仕様の例

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 0
    - revisionName: <second_revision_name>
      percent: 100
      tag: v2
...
```

ヒント

リビジョンのロールバックを計画しない場合は、これを 0% に設定する代わりに最初のリビジョンを削除できます。その後、ルーティング不可能なリビジョンオブジェクトにはガベージコレクションが行われます。

11. 最初のリビジョンの URL にアクセスして、アプリケーションの古いバージョンに送信されていないことを確認します。

5.4. ROUTING

Knative は OpenShift Container Platform TLS 終端を使用して Knative サービスのルーティングを提供します。Knative サービスが作成されると、OpenShift Container Platform ルートがサービス用に自動的に作成されます。このルートは OpenShift Serverless Operator によって管理されます。OpenShift Container Platform ルートは、OpenShift Container Platform クラスターと同じドメインで Knative サービスを公開します。

OpenShift Container Platform ルーティングの Operator 制御を無効にすることで、Knative ルートを TLS 証明書を直接使用するように設定できます。

Knative ルートは OpenShift Container Platform ルートと共に使用し、トラフィック分割などの詳細なルーティング機能を提供します。

5.4.1. OpenShift Container Platform ルートのラベルおよびアノテーションのカスタマイズ

OpenShift Container Platform ルートは、Knative サービスの **metadata** 仕様を変更して設定できるカス

タムラベルおよびアノテーションの使用をサポートします。カスタムラベルおよびアノテーションはサービスから Knative ルートに伝番され、次に Knative ingress に、最後に OpenShift Container Platform ルートに伝播されます。

前提条件

- OpenShift Serverless Operator および Knative Serving が OpenShift Container Platform クラスターにインストールされている必要があります。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. OpenShift Container Platform ルートに伝播するラベルまたはアノテーションが含まれる Knative サービスを作成します。
   - YAML を使用してサービスを作成するには、以下を実行します。

     YAML を使用して作成されるサービスの例

     ```
     apiVersion: serving.knative.dev/v1
     kind: Service
     metadata:
       name: <service_name>
       labels:
         <label_name>: <label_value>
       annotations:
         <annotation_name>: <annotation_value>
     ...
     ```

   - Knative (**kn**) CLI を使用してサービスを作成するには、次のように入力します。

     kn コマンドを使用して作成されるサービスの例

     ```
     $ kn service create <service_name> \
       --image=<image> \
       --annotation <annotation_name>=<annotation_value> \
       --label <label_value>=<label_value>
     ```

2. 以下のコマンドからの出力を検査して、OpenShift Container Platform ルートが追加したアノテーションまたはラベルで作成されていることを確認します。

   検証のコマンドの例

   ```
   $ oc get routes.route.openshift.io \
       -l serving.knative.openshift.io/ingressName=<service_name> \ 1
       -l serving.knative.openshift.io/ingressNamespace=<service_namespace> \ 2
       -n knative-serving-ingress -o yaml \
           | grep -e "<label_name>: \"<label_value>\""  -e "<annotation_name>:
   <annotation_value>" 3
   ```

   1 サービスの名前を使用します。

   2 サービスが作成された namespace を使用します。

3 ラベルおよびアノテーション名および値の値を使用します。

5.4.2. OpenShift Container Platform ルートでの Knative サービスの設定

Knative サービスを OpenShift Container Platform で TLS 証明書を使用するように設定するには、OpenShift Serverless Operator によるサービスのルートの自動作成を無効にし、代わりにサービスのルートを手動で作成する必要があります。

注記

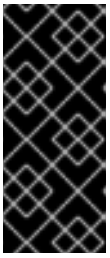
以下の手順を完了すると、**knative-serving-ingress** namespace のデフォルトの OpenShift Container Platform ルートは作成されません。ただし、アプリケーションの Knative ルートはこの namespace に引き続き作成されます。

前提条件

- OpenShift Serverless Operator および Knative Serving コンポーネントが OpenShift Container Platform クラスターにインストールされている。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. **serving.knative.openshift.io/disableRoute=true** アノテーションが含まれる Knative サービスを作成します。



重要

serving.knative.openshift.io/disableRoute=true アノテーションは、OpenShift Serverless に対してルートを自動的に作成しないように指示します。ただし、サービスには URL が表示され、ステータスが **Ready** に達します。URL のホスト名と同じホスト名を使用して独自のルートを作成するまで、この URL は外部では機能しません。

   a. **Service** サービスリソースを作成します。

リソースの例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
  annotations:
    serving.knative.openshift.io/disableRoute: "true"
spec:
  template:
    spec:
      containers:
      - image: <image>
...
```

   b. **Service** リソースを適用します。

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

c. オプション。**kn service create** コマンドを使用して Knative サービスを作成します。

kn コマンドの例

```
$ kn service create <service_name> \
  --image=gcr.io/knative-samples/helloworld-go \
  --annotation serving.knative.openshift.io/disableRoute=true
```

2. サービス用に OpenShift Container Platform ルートが作成されていないことを確認します。

コマンドの例

```
$ $ oc get routes.route.openshift.io \
  -l serving.knative.openshift.io/ingressName=$KSERVICE_NAME \
  -l serving.knative.openshift.io/ingressNamespace=$KSERVICE_NAMESPACE \
  -n knative-serving-ingress
```

以下の出力が表示されるはずです。

```
No resources found in knative-serving-ingress namespace.
```

3. **knative-serving-ingress** namespace で **Route** リソースを作成します。

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 600s (1)
  name: <route_name> (2)
  namespace: knative-serving-ingress (3)
spec:
  host: <service_host> (4)
  port:
    targetPort: http2
  to:
    kind: Service
    name: kourier
    weight: 100
  tls:
    insecureEdgeTerminationPolicy: Allow
    termination: edge (5)
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
```

```
    [...]
    -----END CERTIFICATE----
wildcardPolicy: None
```

1. OpenShift Container Platform ルートのタイムアウト値。**max-revision-timeout-seconds** 設定と同じ値を設定する必要があります (デフォルトでは **600s**)。
2. OpenShift Container Platform ルートの名前。
3. OpenShift Container Platform ルートの namespace。これは **knative-serving-ingress** である必要があります。
4. 外部アクセスのホスト名。これを **<service_name>-<service_namespace>.<domain>** に設定できます。
5. 使用する証明書。現時点で、**edge** termination のみがサポートされています。

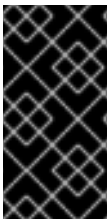
4. **Route** リソースを適用します。

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

5.4.3. クラスターローカルへのクラスター可用性の設定

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 **networking.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。



重要

OpenShift Serverless 1.15.0 以降のバージョンの場合には、**serving.knative.dev/visibility** ラベルは利用できなくなりました。既存のサービスを更新して、代わりに **networking.knative.dev/visibility** ラベルを使用する必要があります。

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。
- Knative サービスを作成している。

手順

- **networking.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> networking.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
```

5.4.4. 関連情報

- ルート固有のアノテーション

5.5. イベントシンク

イベントソースの作成時に、イベントがソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能なリソースです。Knative サービス、チャネル、およびブローカーはすべてシンクのサンプルです。

アドレス指定可能なオブジェクトは、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。特別な場合として、コア 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は、HTTP 経由で配信されるイベントを受信し、そのイベントを変換できます。HTTP 応答で **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。

5.5.1. Knative CLI シンクフラグ

Knative (**kn**) CLI を使用してイベントソースを作成する場合、**--sink** フラグを使用して、イベントがリソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能な任意のリソースです。

以下の例では、サービスの **http://event-display.svc.cluster.local** をシンクとして使用するシンクバインディングを作成します。

シンクフラグを使用したコマンドの例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** の **svc** は、シンクが Knative サービスであることを判別します。他のデフォルトのシンクの接頭辞には、**channel** および **broker** が含まれます。

ヒント

kn のカスタマイズ により、どの CR が Knative (**kn**) CLI コマンドの **--sink** フラグと併用できるかを設定できます。

5.5.2. Developer パースペクティブを使用してイベントソースをシンクに接続します。

OpenShift Container Platform Web コンソールを使用してイベントソースを作成する場合、イベントがリソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能な任意のリソースです。

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おり、**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。
- Knative サービス、チャネル、ブローカーなどのシンクを作成している。

手順

1. **+Add → Event Sources** に移動して任意のタイプのイベントソースを作成し、作成するイベントソースを選択します。
2. **イベントソースの作成** フォームビューの **シンク** セクションで、**リソース** リストからシンクを選択します。
3. **Create** をクリックします。

検証

Topology ページを表示して、イベントソースが作成され、シンクに接続されていることを確認できます。**Developer** パースペクティブで、**Topology** に移動します。

1. イベントソースを表示し、接続されたシンクをクリックし、サイドパネルでシンクの詳細を表示します。

5.5.3. トリガーのシンクへの接続

トリガーをシンクに接続して、シンクへの送信前にブローカーからのイベントがフィルターされるようにします。トリガーに接続されているシンクは、**Trigger** オブジェクトのリソース仕様で **subscriber** として設定されます。

Kafka シンクに接続された Trigger オブジェクトの例

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: <trigger_name> 1
spec:
```

```
...
  subscriber:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: KafkaSink
      name: <kafka_sink_name> 2
```

1 シンクに接続されているトリガーの名前。

2 **KafkaSink** オブジェクトの名前。

5.6. イベント配信

イベントがイベントシンクに配信されなかった場合に適用されるイベント配信パラメーターを設定できます。デッドレターシンクを含むイベント配信パラメーターを設定すると、イベントシンクへの配信に失敗したすべてのイベントが再試行されるようになります。それ以外の場合、未配信のイベントは破棄されます。

5.6.1. チャネルとブローカーのイベント配信動作パターン

さまざまなチャネルとブローカーのタイプには、イベント配信のために従う独自の動作パターンがあります。

5.6.1.1. Knative Kafka のチャネルとブローカー

イベントが Kafka チャネルまたはブローカーレシーバーに正常に配信される場合、受信側は **202** ステータスコードで応答します。つまり、このイベントは Kafka トピック内に安全に保存され、失われることはありません。

受信側がその他のステータスコードを返す場合は、イベントは安全に保存されず、ユーザーがこの問題を解決するために手順を実行する必要があります。

5.6.2. 設定可能なイベント配信パラメーター

以下のパラメーターはイベント配信用に設定できます。

dead letter sink

deadLetterSink 配信パラメーターを設定して、イベントが配信に失敗した場合にこれを指定されたイベントシンクに保存することができます。デッドレターシンクに格納されていない未配信のイベントは破棄されます。デッドレターシンクは、Knative サービス、Kubernetes サービス、または URI など、Knative Eventing シンクコントラクトに準拠する任意のアドレス指定可能なオブジェクトです。

retries

retry 配信パラメーターを整数値で設定することで、イベントが dead letter sink に送信される前に配信を再試行する必要のある最小回数を設定できます。

back off delay

backoffDelay 配信パラメーターを設定し、失敗後にイベント配信が再試行される前の遅延の時間を指定できます。**backoffDelay** パラメーターの期間は ISO 8601 形式を使用して指定されます。たとえば、**PT1S** は 1 秒の遅延を指定します。

back off policy

backoffPolicy 配信パラメーターは再試行バックオフポリシーを指定するために使用できます。ポリ

シーは **linear** または **exponential** のいずれかとして指定できます。**linear** バックオフポリシーを使用する場合、バックオフ遅延は **backoffDelay * <numberOfRetries>** に等しくなります。**exponential** バックオフポリシーを使用する場合、バックオフ遅延は **backoffDelay*2^<numberOfRetries>** と等しくなります。

5.6.3. イベント配信パラメーターの設定例

Broker、**Trigger**、**Channel**、および **Subscription** オブジェクトのイベント配信パラメーターを設定できます。ブローカーまたはチャネルのイベント配信パラメーターを設定すると、これらのパラメーターは、それらのオブジェクト用に作成されたトリガーまたはサブスクリプションに伝播されます。トリガーまたはサブスクリプションのイベント配信パラメーターを設定して、ブローカーまたはチャネルの設定をオーバーライドすることもできます。

Broker オブジェクトの例

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
...
spec:
  delivery:
    deadLetterSink:
      ref:
        apiVersion: eventing.knative.dev/v1alpha1
        kind: KafkaSink
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

Trigger オブジェクトの例

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
...
spec:
  broker: <broker_name>
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

Channel オブジェクトの例

```
apiVersion: messaging.knative.dev/v1
kind: Channel
```

```
metadata:
...
spec:
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

Subscription オブジェクトの例

```
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
...
spec:
  channel:
    apiVersion: messaging.knative.dev/v1
    kind: Channel
    name: <channel_name>
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

5.6.4. トリガーのイベント配信順序の設定

Kafka ブローカーを使用している場合は、トリガーからイベントシンクへのイベントの配信順序を設定できます。

前提条件

- OpenShift Serverless Operator、Knative Eventing、および Knative Kafka が OpenShift Container Platform クラスターにインストールされている。
- Kafka ブローカーがクラスターで使用可能であり、Kafka ブローカーが作成されている。
- 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。
- OpenShift (**oc**) CLI がインストールされている。

手順

1. **Trigger** オブジェクトを作成または変更し、**kafka.eventing.knative.dev/delivery.order** アノテーションを設定します。

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: <trigger_name>
  annotations:
    kafka.eventing.knative.dev/delivery.order: ordered
...
```

サポートされているコンシューマー配信保証は次のとおりです。

unordered

順序付けられていないコンシューマーは、適切なオフセット管理を維持しながら、メッセージを順序付けずに配信するノンブロッキングコンシューマーです。

ordered

順序付きコンシューマーは、CloudEvent サブスクライバーからの正常な応答を待ってから、パーティションの次のメッセージを配信する、パーティションごとのブロックコンシューマーです。
デフォルトの順序保証は **unordered** です。

2. **Trigger** オブジェクトを適用します。

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

5.7. 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

OpenShift Container Platform クラスターに存在する、または使用可能なすべてのイベントソースやイベントソースタイプのリストを表示することができます。OpenShift Container Platform Web コンソールの Knative (**kn**) CLI または **Developer** パースペクティブを使用し、利用可能なイベントソースまたはイベントソースタイプを一覧表示できます。

5.7.1. 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

Knative (**kn**) CLI を使用すると、クラスターで使用可能なイベントソースタイプを表示するための合理的で直感的なユーザーインターフェイスが提供されます。**kn source list-types** CLI コマンドを使用して、クラスターで作成して使用できるイベントソースタイプを一覧表示できます。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。

手順

1. 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します。

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

出力例

```
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
```

2. オプション: 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
```

5.7.2. Developer パースペクティブ内での利用可能なイベントソースタイプの表示

クラスターで使用可能なすべてのイベントソースタイプを一覧表示することができます。OpenShift Container Platform Web コンソールを使用すると、使用可能なイベントソースタイプを表示するための合理的で直感的なユーザーインターフェイスが提供されます。

前提条件

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

手順

1. **Developer** パースペクティブにアクセスします。
2. **+Add** をクリックします。
3. **Event source** をクリックします。
4. 利用可能なイベントソースタイプを表示します。

5.7.3. 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

Knative (**kn**) CLI を使用すると、クラスターの既存イベントソースを表示するための合理的で直感的なユーザーインターフェイスが提供されます。**kn source list** コマンドを使用して、既存のイベントソースを一覧表示できます。

前提条件

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

- Knative (**kn**) CLI をインストールしている。

手順

1. ターミナルにある既存のイベントソースを一覧表示します。

```
$ 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
```

2. オプションで、**--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 <event_source_type>
```

コマンドの例

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

出力例

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

5.8. API サーバーソースの作成

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。

5.8.1. Web コンソールを使用した API サーバーソースの作成

Knative Eventing がクラスターにインストールされると、Web コンソールを使用して API サーバーソースを作成できます。OpenShift Container Platform Web コンソールを使用すると、イベントソースを作成するための合理的で直感的なユーザーインターフェイスが提供されます。

前提条件

- OpenShift Container Platform Web コンソールにログインしている。
- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- 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。

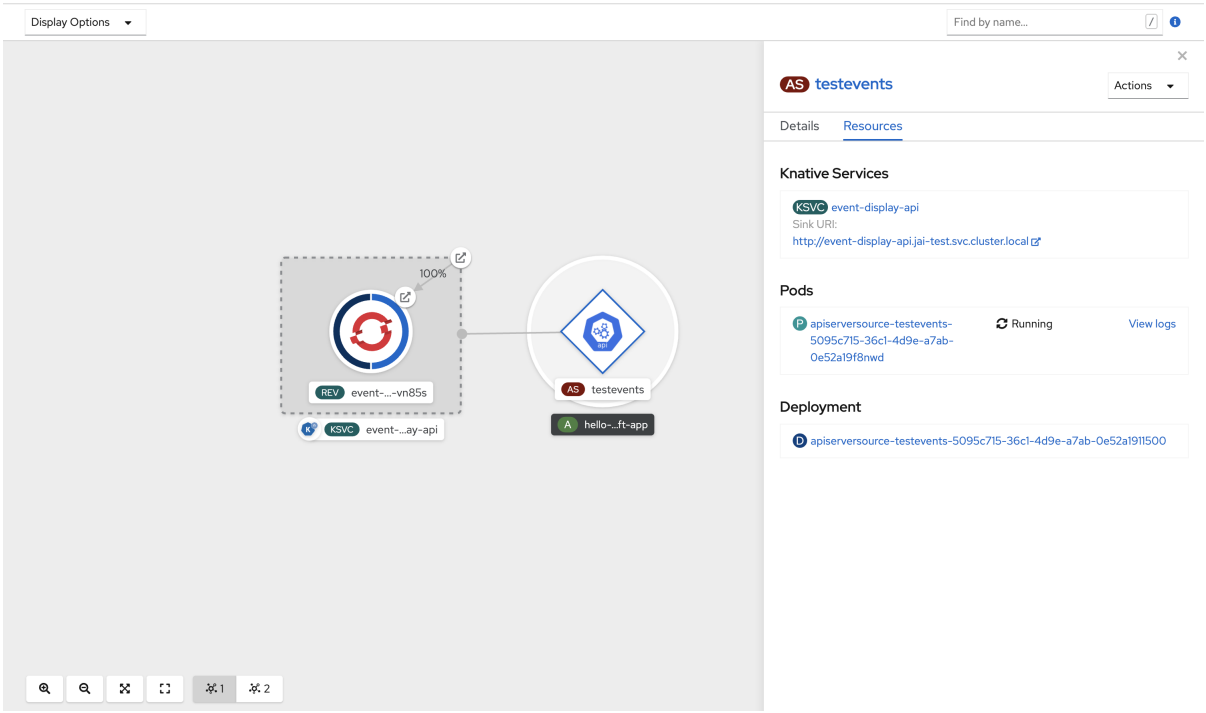
- OpenShift CLI (**oc**) がインストールされている。

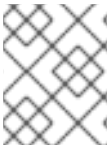
手順

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. イベントソースの **Sink** を選択します。**Sink** は、チャネル、ブローカー、またはサービスなどの **Resource**、または **URI** のいずれかになります。
4. **Create** をクリックします。

検証

- API サーバーソースの作成後、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。



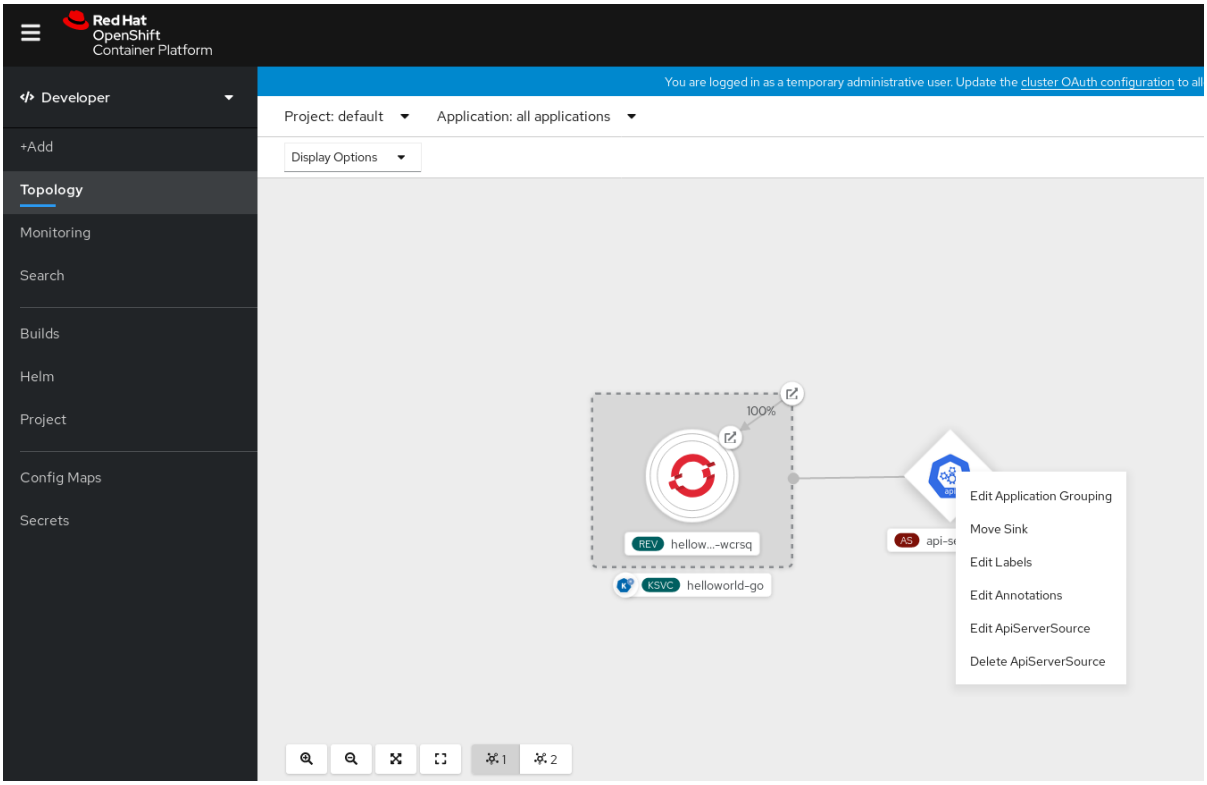


注記

URI シンクが使用される場合、**URI sink → Edit URI** を右クリックして URI を変更します。

API サーバーソースの削除

1. **Topology** ビューに移動します。
2. API サーバーソースを右クリックし、**Delete ApiServerSource** を選択します。



5.8.2. Knative CLI を使用した API サーバーソースの作成

kn source apiserver create コマンドを使用し、**kn** CLI を使用して API サーバーソースを作成できます。API サーバーソースを作成するために **kn** CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- 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。
- OpenShift CLI (**oc**) がインストールされている。
- Knative (**kn**) CLI をインストールしている。

手順

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には、既存の **ServiceAccount** リソースを変更して、新規リソースを作成せずに、必要なパーミッションを含めることができます。

1. 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 YAML ファイルとして作成します。

```
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。

2. YAML ファイルを適用します。

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

3. イベントシンクを持つ API サーバーソースを作成します。次の例では、シンクはブローカーです。

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

4. 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。

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

5. ブローカーをイベントシンクとして使用した場合は、トリガーを作成して、**default** のブローカーからサービスへのイベントをフィルターリングします。

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

6. 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:latest
```

7. 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 <source_name>
```

出力例

```
Name:                mysource
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",
    ...
  }
```

API サーバーソースの削除

1. トリガーを削除します。

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

2. イベントソースを削除します。

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

3. 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
```

5.8.2.1. Knative CLI シンクフラグ

Knative (**kn**) CLI を使用してイベントソースを作成する場合、**--sink** フラグを使用して、イベントがリソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能な任意のリソースです。

以下の例では、サービスの **http://event-display.svc.cluster.local** をシンクとして使用するシンクバインディングを作成します。

シンクフラグを使用したコマンドの例

-

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** の **svc** は、シンクが Knative サービスであることを判別します。他のデフォルトのシンクの接頭辞には、**channel** および **broker** が含まれます。

5.8.3. YAML ファイルを使用した API サーバーソースの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でイベントソースを宣言的に記述することができます。YAML を使用して API サーバーソースを作成するには、**ApiServerSource** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- 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。
- API サーバーソース YAML ファイルで定義されるものと同じ namespace に **default** ブローカーを作成している。
- OpenShift CLI (**oc**) をインストールしている。

手順

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には、既存の **ServiceAccount** リソースを変更して、新規リソースを作成せずに、必要なパーミッションを含めることができます。

1. 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 YAML ファイルとして作成します。

```
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。

2. YAML ファイルを適用します。

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

3. API サーバーソースを YAML ファイルとして作成します。

```
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
```

4. **ApiServerSource** YAML ファイルを適用します。

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

5. 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 YAML ファイルとして作成します。

```
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
```

6. **Service** YAML ファイルを適用します。

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

7. 直接の手順で作成下サービスに、**default** ブローカーからイベントをフィルターする **Trigger** オブジェクトを YAML ファイルとして作成します。

```
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
```

8. **Trigger** YAML ファイルを適用します。

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

9. 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
```

10. 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 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. 以下のコマンドを入力して Pod を取得します。

```
$ oc get pods
```

2. 以下のコマンドを入力して、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",
  ...
}
```

API サーバーソースの削除

1. トリガーを削除します。

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

2. イベントソースを削除します。

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

3. 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
```

5.9. PING ソースの作成

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るイベントソースです。ping ソースを使用すると、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。

5.9.1. Web コンソールを使用した ping ソースの作成

Knative Eventing がクラスターにインストールされると、Web コンソールを使用して ping ソースを作成できます。OpenShift Container Platform Web コンソールを使用すると、イベントソースを作成するための合理的で直感的なユーザーインターフェイスが提供されます。

前提条件

- OpenShift Container Platform Web コンソールにログインしている。
- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がクラスターにインストールされている。
- 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。

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. **Developer** パースペクティブで、**+Add → YAML** に移動します。

   b. サンプル YAML をコピーします。

```
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
```

   c. **Create** をクリックします。

2. 直前の手順で作成したサービスと同じ namespace、またはイベントの送信先となる他のシンクと同じ namespace に ping ソースを作成します。

   a. **Developer** パースペクティブで、**+Add → Event Source** に移動します。

   b. **Ping Source** を選択します。

   c. オプション: **Data** の値を入力できます。これはメッセージのペイロードです。

   d. **Schedule** の値を入力します。この例では、値は ***/2** * * * * であり、2 分ごとにメッセージを送信する ping ソースを作成します。

   e. **Sink** を選択します。これは **Resource** または **URI** のいずれかになります。この例では、直前の手順で作成された **event-display** サービスが **Resources** シンクとして使用されます。

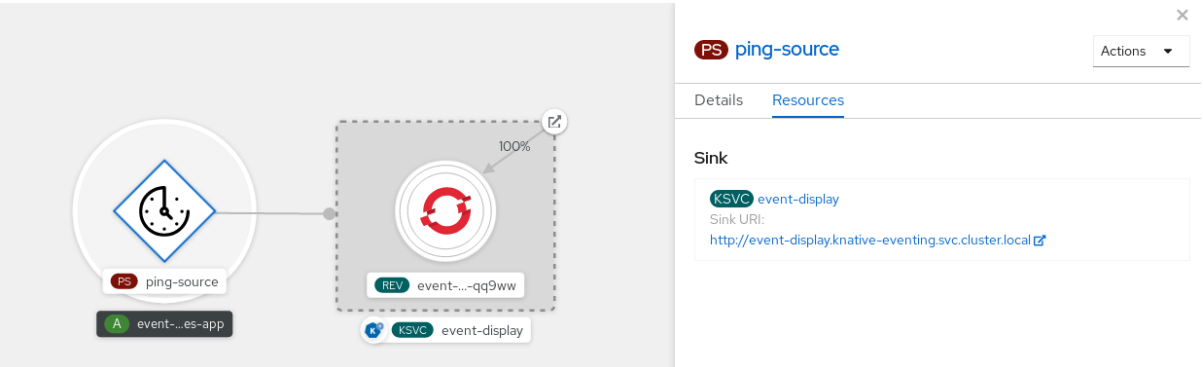
   f. **Create** をクリックします。

検証

Topology ページを表示して、ping ソースが作成され、シンクに接続されていることを確認できます。

1. **Developer** パースペクティブで、**Topology** に移動します。

2. ping ソースおよびシンクを表示します。



ping ソースの削除

1. **Topology** ビューに移動します。

2. API サーバーソースを右クリックし、**Delete Ping Source**を選択します。

5.9.2. Knative CLI を使用した ping ソースの作成

kn source ping create コマンドを使用し、Knative (**kn**) CLI を使用して ping ソースを作成できます。イベントソースを作成するために Knative CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。
- 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。
- オプション: この手順の検証手順を使用する場合は、OpenShift CLI (**oc**) をインストールします。

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 ping ソースを作成します。

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!"
  }
```

ping ソースの削除

- ping ソースを削除します。

```
$ kn delete pingsources.sources.knative.dev <ping_source_name>
```

5.9.2.1. Knative CLI シンクフラグ

Knative (**kn**) CLI を使用してイベントソースを作成する場合、**--sink** フラグを使用して、イベントがリソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能な任意のリソースです。

以下の例では、サービスの **http://event-display.svc.cluster.local** をシンクとして使用するシンクバインディングを作成します。

シンクフラグを使用したコマンドの例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** の **svc** は、シンクが Knative サービスであることを判別します。他のデフォルトのシンクの接頭辞には、**channel** および **broker** が含まれます。

5.9.3. YAML を使用した ping ソースの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でイベントソースを宣言的に記述することができます。YAML を使用してサーバーレス ping を作成するには、**PingSource** オブジェクトを定義する 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。

PingSource オブジェクトの例

```
apiVersion: sources.knative.dev/v1
kind: PingSource
metadata:
  name: test-ping-source
spec:
  schedule: "*/2 * * * *" 1
  data: '{"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。

前提条件

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

- OpenShift CLI (**oc**) をインストールしている。
- 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。

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。
   a. サービス YAML ファイルを作成します。

```
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. サービスを作成します。

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

2. 要求する必要のある ping イベントのセットごとに、ping ソースをイベントコンシューマーと同じ namespace に作成します。
   a. ping ソースの YAML ファイルを作成します。

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

   b. ping ソースを作成します。

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

3. 以下のコマンドを入力し、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 <ping_source_name> -oyaml
```

出力例

```
apiVersion: sources.knative.dev/v1
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/v1/namespaces/default/pingsources/test-ping-source
  uid: 3d80d50b-f8c7-4c1b-99f7-3ec00e0a8164
spec:
  data: '{ 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 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!"
  }
```

ping ソースの削除

- ping ソースを削除します。

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

コマンドの例

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

5.10. カスタムイベントソース

Knative に含まれていないイベントプロデューサーや、**CloudEvent** 形式ではないイベントを生成するプロデューサーからイベントを Ingress する必要がある場合は、カスタムイベントソースを使用してこれを実行できます。カスタムイベントソースは、次のいずれかの方法で作成できます。

- シンクバインディングを作成して、**PodSpecable** オブジェクトをイベントソースとして使用します。
- コンテナーソースを作成して、コンテナーをイベントソースとして使用します。

5.10.1. シンクバインディング

SinkBinding オブジェクトは、イベント生成を配信アドレス指定から切り離すことをサポートします。シンクバインディングは、**イベントプロデューサー** をイベントコンシューマーまたは **シンク** に接続するために使用されます。イベントプロデューサーは、**PodSpec** テンプレートを組み込む Kubernetes リソースであり、イベントを生成します。シンクは、イベントを受信できるアドレス指定可能な Kubernetes オブジェクトです。

SinkBinding オブジェクトは、環境変数をシンクの **PodTemplateSpec** に挿入します。つまり、アプリケーションコードが Kubernetes API と直接対話してイベントの宛先を見つける必要はありません。これらの環境変数は以下のとおりです。

K_SINK
解決されたシンクの URL。

K_CE_OVERRIDES
アウトバウンドイベントの上書きを指定する JSON オブジェクト。

注記

現在、**SinkBinding** オブジェクトはサービスのカスタムリビジョン名をサポートしません。

5.10.1.1. YAML を使用したシンクバインディングの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でイベントソースを宣言的に記述することができます。YAML を使用してシンクバインディングを作成するには、**SinkBinding** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がクラスターにインストールされている。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

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. サービス YAML ファイルを作成します。

      サービス YAML ファイルの例

      ```
      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. サービスを作成します。

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

2. イベントをサービスに転送するシンクバインディングインスタンスを作成します。
   a. シンクバインディング YAML ファイルを作成します。

      サービス YAML ファイルの例

      ```
      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. シンクバインディングを作成します。

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

3. **CronJob** オブジェクトを作成します。

a. cron ジョブの YAML ファイルを作成します。

cron ジョブの YAML ファイルの例

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
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/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. cron ジョブを作成します。

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

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. コマンドを入力します。

```
$ oc get pods
```

2. コマンドを入力します。

```
$ 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.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": ""
  }
```

5.10.1.2. Knative CLI を使用したシンクバインディングの作成

kn source binding create コマンドを使用し、Knative (**kn**) を使用してシンクバインディングを作成できます。イベントソースを作成するために Knative CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がクラスターにインストールされている。
- 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。
- Knative (**kn**) CLI をインストールしている。
- OpenShift CLI (**oc**) をインストールしている。

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. 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シンクバインディングインスタンスを作成します。

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

3. **CronJob** オブジェクトを作成します。

   a. cron ジョブの YAML ファイルを作成します。

cron ジョブの YAML ファイルの例

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
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/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 CR 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** CR に追加するには、以下の行を **Job** CR の YAML 定義に追加します。

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

b. cron ジョブを作成します。

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

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。

- 以下のコマンドを入力して、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": ""
  }
```

5.10.1.2.1. Knative CLI シンクフラグ

Knative (**kn**) CLI を使用してイベントソースを作成する場合、**--sink** フラグを使用して、イベントがリソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能な任意のリソースです。

以下の例では、サービスの **http://event-display.svc.cluster.local** をシンクとして使用するシンクバインディングを作成します。

シンクフラグを使用したコマンドの例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** の **svc** は、シンクが Knative サービスであることを判別します。他のデフォルトのシンクの接頭辞には、**channel** および **broker** が含まれます。

5.10.1.3. Web コンソールを使用したシンクバインディングの作成

Knative Eventing がクラスターにインストールされると、Web コンソールを使用して シンクバインディングを作成できます。OpenShift Container Platform Web コンソールを使用すると、イベントソースを作成するための合理的で直感的なユーザーインターフェイスが提供されます。

前提条件

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

- 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。

手順

1. シンクとして使用する Knative サービスを作成します。
   a. **Developer** パースペクティブで、**+Add → YAML** に移動します。
   b. サンプル YAML をコピーします。

```
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
```

   c. **Create** をクリックします。
2. イベントソースとして使用される **CronJob** リソースを作成し、1分ごとにイベントを送信します。
   a. **Developer** パースペクティブで、**+Add → YAML** に移動します。
   b. サンプル YAML をコピーします。

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
  # Run every minute
  schedule: "*/1 * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: true (1)
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats
              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
```

1. **bindings.knative.dev/include: true** ラベルを含めるようにしてください。OpenShift Serverless のデフォルトの namespace 選択動作は包含モードを使用します。

c. **Create** をクリックします。

3. 直前の手順で作成したサービスと同じ namespace、またはイベントの送信先となる他のシンクと同じ namespace にシンクバインディングを作成します。

a. **Developer** パースペクティブで、**+Add → Event Source** に移動します。**Event Sources** ページが表示されます。

b. オプション: イベントソースに複数のプロバイダーがある場合は、**Providers** 一覧から必要なプロバイダーを選択し、プロバイダーから利用可能なイベントソースをフィルターします。

c. **Sink Binding** を選択し、**Create Event Source**をクリックします。**Create Event Source** ページが表示されます。

d. **apiVersion** フィールドに **batch/v1** を入力します。

e. **Kind** フィールドに **Job** と入力します。

注記

CronJob の種類は OpenShift Serverless シンクバインディングで直接サポートされていないため、**Kind** フィールドは cron ジョブオブジェクト自体ではなく、cron ジョブで作成される **Job** オブジェクトをターゲットにする必要があります。

f. **Sink** を選択します。これは **Resource** または **URI** のいずれかになります。この例では、直前の手順で作成された **event-display** サービスが **Resources** シンクとして使用されます。

g. **Match labels** セクションで以下を実行します。

i. **Name** フィールドに **app** と入力します。

ii. **Value** フィールドに **heartbeat-cron** と入力します。

注記

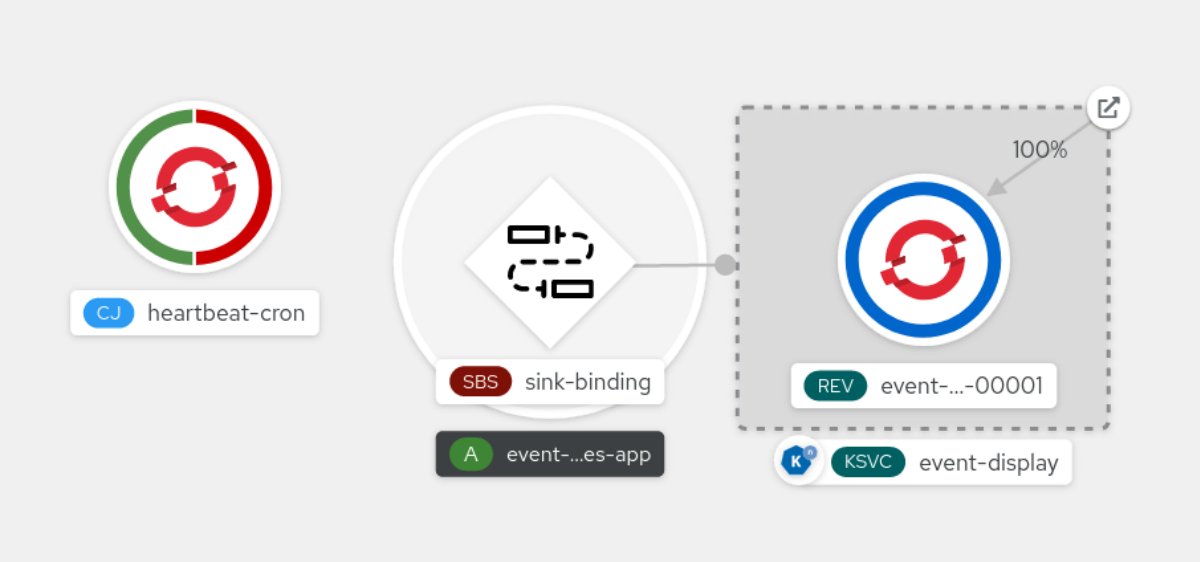
ラベルセレクターは、リソース名ではなくシンクバインディングで cron ジョブを使用する場合に必要になります。これは、cron ジョブで作成されたジョブには予測可能な名前がなく、名前に無作為に生成される文字列が含まれているためです。たとえば、**hearthbeat-cron-1cc23f** になります。

h. **Create** をクリックします。

検証

Topology ページおよび Pod ログを表示して、シンクバインディング、シンク、および cron ジョブが正常に作成され、機能していることを確認できます。

1. **Developer** パースペクティブで、**Topology** に移動します。
2. シンクバインディング、シンク、およびハートビートの cron ジョブを表示します。



3. シンクバインディングが追加されると、正常なジョブが cron ジョブによって登録されていることを確認します。つまり、シンクバインディングは cron ジョブで作成されたジョブが正常に再設定されることを意味します。
4. **event-display** サービス Pod のログを参照し、ハートビート cron ジョブで生成されるイベントを表示します。

5.10.1.4. シンクバインディング参照

シンクバインディングを作成して、**PodSpecable** オブジェクトをイベントソースとして使用できます。**SinkBinding** オブジェクトを作成するときに、複数のパラメーターを設定できます。

SinkBinding オブジェクトは以下のパラメーターをサポートします。

フィールド	説明	必須またはオプション
apiVersion	API バージョンを指定します (例: **sources.knative.dev/v1**)。	必須
kind	このリソースオブジェクトを **SinkBinding** オブジェクトとして特定します。	必須
metadata	**SinkBinding** オブジェクトを一意に識別するメタデータを指定します。たとえば、**name** です。	必須
spec	この **SinkBinding** オブジェクトの設定情報を指定します。	必須

フィールド	説明	必須またはオプション
spec.sink	シンクとして使用する URI に解決するオブジェクトへの参照。	必須
spec.subject	ランタイムコントラクトがバインディング実装によって拡張されるリソースを参照します。	必須
spec.ceOverrides	上書きを定義して、シンクに送信されたイベントへの出力形式および変更を制御します。	任意

5.10.1.4.1. Subject パラメーター

Subject パラメーターは、ランタイムコントラクトがバインディング実装によって拡張されるリソースを参照します。**Subject** 定義に複数のフィールドを設定できます。

Subject 定義は、以下のフィールドをサポートします。

フィールド	説明	必須またはオプション
apiVersion	参照先の API バージョン。	必須
kind	参照先の種類。	必須
namespace	参照先の namespace。省略されている場合、デフォルトはオブジェクトの namespace に設定されます。	任意
name	参照先の名前。	**selector** を設定する場合は、使用しないでください。
selector	参照先のセレクター。	**name** を設定する場合は、使用しないでください。
selector.matchExpressions	ラベルセレクターの要件の一覧です。	**matchExpressions** または **matchLabels** のいずれかのみを使用します。
selector.matchExpressions.key	セレクターが適用されるラベルキー。	**matchExpressions** を使用する場合に必須です。
selector.matchExpressions.operator	キーと値のセットの関係を表します。有効な演算子は **In**、**NotIn**、**Exists**、および **DoesNotExist** です。	**matchExpressions** を使用する場合に必須です。

フィールド	説明	必須またはオプション
selector.matchExpressions.values	文字列値の配列。**operator** パラメーターの値が **In** または **NotIn** の場合、値配列が空でないようにする必要があります。**operator** パラメーターの値が **Exists** または **DoesNotExist** の場合、値の配列は空である必要があります。この配列は、ストラテジーに基づいたマージパッチの適用中に置き換えられます。	**matchExpressions** を使用する場合に必須です。
selector.matchLabels	キーと値のペアのマップ。**matchLabels** マップの各キーと値のペアは **matchExpressions** の要素と同じです。ここで、キーフィールドは **matchLabels.<key>** で、**operator** は **In** で、**values** の配列には **matchLabels.<value>** のみが含まれます。	**matchExpressions** または **matchLabels** のいずれかのみを使用します。

サブジェクトパラメーターの例

以下の YAML の場合は、**default** namespace の **mysubject** という名前の **Deployment** オブジェクトが選択されます。

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: apps/v1
    kind: Deployment
    namespace: default
    name: mysubject
  ...
```

以下の YAML の場合、**default** namespace にラベル **working=example** が設定された **Job** オブジェクトが選択されます。

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: batch/v1
    kind: Job
    namespace: default
```

```
    selector:
      matchLabels:
        working: example
  ...
```

以下の YAML の場合、**default** namespace にラベル **working=example** または **working=sample** が含まれる **Pod** オブジェクトが選択されます。

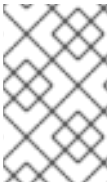
```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: v1
    kind: Pod
    namespace: default
    selector:
      - matchExpression:
        key: working
        operator: In
        values:
          - example
          - sample
  ...
```

5.10.1.4.2. CloudEvent オーバーライド

ceOverrides 定義は、シンクに送信される CloudEvent の出力形式および変更を制御するオーバーライドを提供します。**ceOverrides** 定義に複数のフィールドを設定できます。

ceOverrides の定義は、以下のフィールドをサポートします。

フィールド	説明	必須またはオプション
extensions	アウトバウンドイベントで追加または上書きされる属性を指定します。各 **extensions** のキーと値のペアは、属性拡張機能としてイベントに個別に設定されます。	任意



注記

拡張子として許可されるのは、有効な **CloudEvent** 属性名のみです。拡張機能オーバーライド設定から仕様定義属性を設定することはできません。たとえば、**type** 属性を変更することはできません。

CloudEvent オーバーライドの例

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
```

```
  name: bind-heartbeat
spec:
  ...
  ceOverrides:
    extensions:
      extra: this is an extra attribute
      additional: 42
```

これにより、**subject** に **K_CE_OVERRIDES** 環境変数が設定されます。

出力例

```
{ "extensions": { "extra": "this is an extra attribute", "additional": "42" } }
```

5.10.1.4.3. include ラベル

シンクバインディングを使用するには、**bindings.knative.dev/include: "true"** ラベルをリソースまたはリソースが含まれる namespace のいずれかに割り当てる必要があります。リソース定義にラベルが含まれていない場合には、クラスター管理者は以下を実行してこれを namespace に割り当てることができます。

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

5.10.2. コンテナーソース

コンテナーソースは、イベントを生成し、イベントをシンクに送信するコンテナーイメージを作成します。コンテナーソースを使用して、イメージ URI を使用するコンテナーイメージおよび **ContainerSource** オブジェクトを作成して、カスタムイベントソースを作成できます。

5.10.2.1. コンテナーイメージを作成するためのガイドライン

コンテナーソースコントローラーには、**K_SINK** および **K_CE_OVERRIDES** の 2 つの環境変数が注入されます。これらの変数は、それぞれ **sink** および **ceOverrides** 仕様から解決されます。イベントは、**K_SINK** 環境変数で指定されたシンク URI に送信されます。メッセージは、 **CloudEvent** HTTP 形式を使用して **POST** として送信する必要があります。

コンテナーイメージの例

以下は、ハートビートコンテナーイメージの例になります。

```
package main

import (
 "context"
 "encoding/json"
 "flag"
 "fmt"
 "log"
 "os"
 "strconv"
 "time"

 duckv1 "knative.dev/pkg/apis/duck/v1"
```

```
 cloudevents "github.com/cloudevents/sdk-go/v2"
 "github.com/kelseyhightower/envconfig"
)

type Heartbeat struct {
 Sequence int    `json:"id"`
 Label    string `json:"label"`
}

var (
 eventSource string
 eventType   string
 sink        string
 label       string
 periodStr   string
)

func init() {
 flag.StringVar(&eventSource, "eventSource", "", "the event-source (CloudEvents)")
 flag.StringVar(&eventType, "eventType", "dev.knative.eventing.samples.heartbeat", "the event-type
(CloudEvents)")
 flag.StringVar(&sink, "sink", "", "the host url to heartbeat to")
 flag.StringVar(&label, "label", "", "a special label")
 flag.StringVar(&periodStr, "period", "5", "the number of seconds between heartbeats")
}

type envConfig struct {
 // Sink URL where to send heartbeat cloud events
 Sink string `envconfig:"K_SINK"`

 // CEOverrides are the CloudEvents overrides to be applied to the outbound event.
 CEOverrides string `envconfig:"K_CE_OVERRIDES"`

 // Name of this pod.
 Name string `envconfig:"POD_NAME" required:"true"`

 // Namespace this pod exists in.
 Namespace string `envconfig:"POD_NAMESPACE" required:"true"`

 // Whether to run continuously or exit.
 OneShot bool `envconfig:"ONE_SHOT" default:"false"`
}

func main() {
 flag.Parse()

 var env envConfig
 if err := envconfig.Process("", &env); err != nil {
  log.Printf("[ERROR] Failed to process env var: %s", err)
  os.Exit(1)
 }

 if env.Sink != "" {
  sink = env.Sink
 }
```

```
var ceOverrides *duckv1.CloudEventOverrides
if len(env.CEOverrides) > 0 {
 overrides := duckv1.CloudEventOverrides{}
 err := json.Unmarshal([]byte(env.CEOverrides), &overrides)
 if err != nil {
  log.Printf("[ERROR] Unparseable CloudEvents overrides %s: %v", env.CEOverrides, err)
  os.Exit(1)
 }
 ceOverrides = &overrides
}

p, err := cloudevents.NewHTTP(cloudevents.WithTarget(sink))
if err != nil {
 log.Fatalf("failed to create http protocol: %s", err.Error())
}

c, err := cloudevents.NewClient(p, cloudevents.WithUUIDs(), cloudevents.WithTimeNow())
if err != nil {
 log.Fatalf("failed to create client: %s", err.Error())
}

var period time.Duration
if p, err := strconv.Atoi(periodStr); err != nil {
 period = time.Duration(5) * time.Second
} else {
 period = time.Duration(p) * time.Second
}

if eventSource == "" {
 eventSource = fmt.Sprintf("https://knative.dev/eventing-contrib/cmd/heartbeats/#%s/%s",
env.Namespace, env.Name)
 log.Printf("Heartbeats Source: %s", eventSource)
}

if len(label) > 0 && label[0] == '"' {
 label, _ = strconv.Unquote(label)
}
hb := &Heartbeat{
 Sequence: 0,
 Label:    label,
}
ticker := time.NewTicker(period)
for {
 hb.Sequence++

 event := cloudevents.NewEvent("1.0")
 event.SetType(eventType)
 event.SetSource(eventSource)
 event.SetExtension("the", 42)
 event.SetExtension("heart", "yes")
 event.SetExtension("beats", true)

 if ceOverrides != nil && ceOverrides.Extensions != nil {
  for n, v := range ceOverrides.Extensions {
   event.SetExtension(n, v)
  }
```

```
  }

  if err := event.SetData(cloudevents.ApplicationJSON, hb); err != nil {
   log.Printf("failed to set cloudevents data: %s", err.Error())
  }

  log.Printf("sending cloudevent to %s", sink)
  if res := c.Send(context.Background(), event); !cloudevents.IsACK(res) {
   log.Printf("failed to send cloudevent: %v", res)
  }

  if env.OneShot {
   return
  }

  // Wait for next tick
  <-ticker.C
 }
}
```

以下は、以前のハートビートコンテナーイメージを参照するコンテナーソースの例です。

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  template:
    spec:
      containers:
        # This corresponds to a heartbeats image URI that you have built and published
        - image: gcr.io/knative-releases/knative.dev/eventing/cmd/heartbeats
          name: heartbeats
          args:
            - --period=1
          env:
            - name: POD_NAME
              value: "example-pod"
            - name: POD_NAMESPACE
              value: "event-test"
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: example-service
...
```

5.10.2.2. Knative CLI を使用したコンテナーソースの作成および管理

kn source container コマンドを使用し、Knative (**kn**) CLI を使用してコンテナーソースを作成および管理できます。イベントソースを作成するために Knative CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

コンテナーソースを作成します。

```
$ kn source container create <container_source_name> --image <image_uri> --sink <sink>
```

コンテナーソースの削除

```
$ kn source container delete <container_source_name>
```

コンテナーソースを記述します。

```
$ kn source container describe <container_source_name>
```

既存のコンテナーソースを一覧表示

```
$ kn source container list
```

既存のコンテナーソースを YAML 形式で一覧表示

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

コンテナーソースを更新します。

このコマンドにより、既存のコンテナーソースのイメージ URI が更新されます。

```
$ kn source container update <container_source_name> --image <image_uri>
```

5.10.2.3. Web コンソールを使用したコンテナーソースの作成

Knative Eventing がクラスターにインストールされると、Web コンソールを使用してコンテナーソースを作成できます。OpenShift Container Platform Web コンソールを使用すると、イベントソースを作成するための合理的で直感的なユーザーインターフェイスが提供されます。

前提条件

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

手順

1. **Developer** パースペクティブで、**+Add → Event Source** に移動します。**Event Sources** ページが表示されます。
2. **Container Source** を選択します。
3. **Container Source** 設定を設定します。
    a. **Image** フィールドに、コンテナーソースが作成したコンテナーで実行するイメージの URI を入力します。

b. **Name** フィールドにイメージの名前を入力します。

c. オプション: **Arguments** フィールドで、コンテナーに渡す引数を入力します。

d. オプション: **Environment variables** フィールドで、コンテナーに設定する環境変数を追加します。

e. **Sink** セクションで、コンテナーソースからのイベントがルーティングされるシンクを追加します。

i. **Resource** を選択して、チャネル、ブローカー、またはサービスをイベントソースのシンクとして使用します。

ii. **URI** を選択して、コンテナーソースからのイベントのルーティング先を指定します。

4. コンテナーソースの設定が完了したら、**Create** をクリックします。

5.10.2.4. コンテナーソースのリファレンス

ContainerSource オブジェクトを作成することにより、コンテナーをイベントソースとして使用できます。**ContainerSource** オブジェクトを作成するときに、複数のパラメーターを設定できます。

ContainerSource オブジェクトは以下のフィールドをサポートします。

フィールド	説明	必須またはオプション
apiVersion	API バージョンを指定します (例: **sources.knative.dev/v1**)。	必須
kind	このリソースオブジェクトを **ContainerSource** オブジェクトとして特定します。	必須
metadata	**ContainerSource** オブジェクトを一意に識別するメタデータを指定します。たとえば、**name** です。	必須
spec	この **ContainerSource** オブジェクトの設定情報を指定します。	必須
spec.sink	シンクとして使用する URI に解決するオブジェクトへの参照。	必須
spec.template	**ContainerSource** オブジェクトの **template** 仕様。	必須
spec.ceOverrides	上書きを定義して、シンクに送信されたイベントへの出力形式および変更を制御します。	任意

テンプレートパラメーターの例

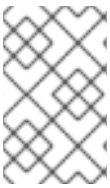
```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/heartbeats:latest
          name: heartbeats
          args:
            - --period=1
          env:
            - name: POD_NAME
              value: "mypod"
            - name: POD_NAMESPACE
              value: "event-test"
  ...
```

5.10.2.4.1. CloudEvent オーバーライド

ceOverrides 定義は、シンクに送信される CloudEvent の出力形式および変更を制御するオーバーライドを提供します。**ceOverrides** 定義に複数のフィールドを設定できます。

ceOverrides の定義は、以下のフィールドをサポートします。

フィールド	説明	必須またはオプション
extensions	アウトバウンドイベントで追加または上書きされる属性を指定します。各 **extensions** のキーと値のペアは、属性拡張機能としてイベントに個別に設定されます。	任意



注記

拡張子として許可されるのは、有効な **CloudEvent** 属性名のみです。拡張機能オーバーライド設定から仕様定義属性を設定することはできません。たとえば、**type** 属性を変更することはできません。

CloudEvent オーバーライドの例

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  ...
  ceOverrides:
```

```
  extensions:
    extra: this is an extra attribute
    additional: 42
```

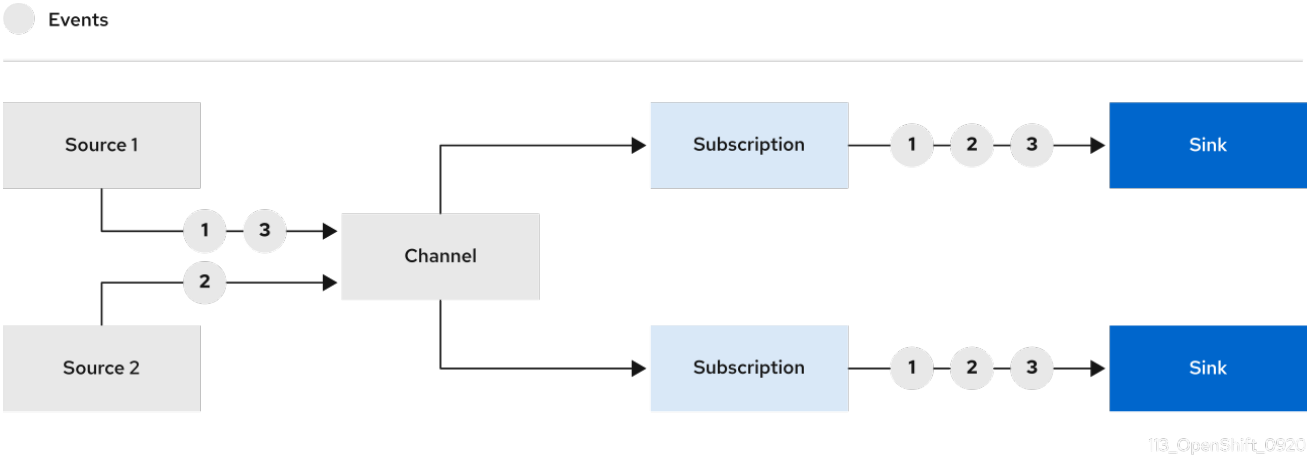
これにより、**subject** に **K_CE_OVERRIDES** 環境変数が設定されます。

出力例

```
{ "extensions": { "extra": "this is an extra attribute", "additional": "42" } }
```

5.11. チャネルの作成

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。イベントがイベントソースまたは生成側から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。



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チャネルを作成し、**Subscription** オブジェクトの **delivery** 仕様を変更して再配信の試行を設定できます。

5.11.1. Web コンソールを使用したチャネルの作成

OpenShift Container Platform Web コンソールを使用すると、チャネルを作成するための合理的で直感的なユーザーインターフェイスが提供されます。Knative Eventing がクラスターにインストールされると、Web コンソールを使用してチャネルを作成できます。

前提条件

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

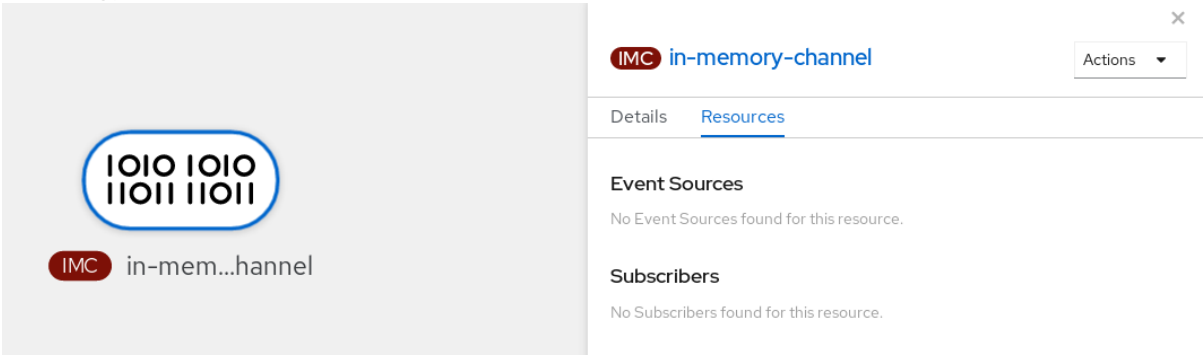
手順

1. **Developer** パースペクティブで、**+Add → Channel** に移動します。

2. **タイプ** リストで、作成する **Channel** オブジェクトのタイプを選択します。
3. **Create** をクリックします。

検証

- **Topology** ページに移動して、チャネルが存在することを確認します。



5.11.2. Knative CLI を使用したチャネルの作成

チャネルを作成するために Knative (**kn**) CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。**kn channel create** コマンドを使用してチャネルを作成できます。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。
- 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。

手順

- チャネルを作成します。

```
$ kn channel create <channel_name> --type <channel_type>
```

チャネルタイプはオプションですが、指定する場合、**Group:Version:Kind** の形式で指定する必要があります。たとえば、**InMemoryChannel** オブジェクトを作成できます。

```
$ kn channel create mychannel --type messaging.knative.dev:v1:InMemoryChannel
```

出力例

```
Channel 'mychannel' created in namespace 'default'.
```

検証

- チャネルが存在することを確認するには、既存のチャネルを一覧表示し、出力を検査します。

```
$ kn channel list
```

出力例

```
kn channel list
NAME        TYPE              URL                                                AGE   READY   REASON
mychannel   InMemoryChannel   http://mychannel-kn-channel.default.svc.cluster.local   93s   True
```

チャネルの削除

- チャネルを削除します。

```
$ kn channel delete <channel_name>
```

5.11.3. YAML を使用したデフォルト実装チャネルの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でチャネルを宣言的に記述することができます。YAML を使用してサーバーレスチャネルを作成するには、**Channel** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

1. **Channel** オブジェクトを YAML ファイルとして作成します。

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

2. YAML ファイルを適用します。

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

5.11.4. YAML を使用した Kafka チャネルの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でチャネルを宣言的に記述することができます。Kafka チャネルを作成することで、Kafka トピックに裏打ちされた Knative Eventing チャネルを作成できます。YAML を使用して Kafka チャネルを

作成するには、**KafkaChannel** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** カスタムリソースは OpenShift Container Platform クラスターにインストールされます。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

1. **KafkaChannel** オブジェクトを YAML ファイルとして作成します。

```
apiVersion: messaging.knative.dev/v1beta1
kind: KafkaChannel
metadata:
  name: example-channel
  namespace: default
spec:
  numPartitions: 3
  replicationFactor: 1
```

重要

OpenShift Serverless 上の **KafkaChannel** オブジェクトの API の **v1beta1** バージョンのみがサポートされます。非推奨となった **v1alpha1** バージョンの API は使用しないでください。

2. **KafkaChannel** YAML ファイルを適用します。

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

5.11.5. 次のステップ

- チャネルの作成後に、イベントシンクがチャネルにサブスクライブしてイベントを受信できるように、サブスクリプションを作成します。
- イベントがイベントシンクに配信されなかった場合に適用されるイベント配信パラメーターを設定します。イベント配信パラメーターの設定例を 参照してください。

5.12. サブスクリプションの作成および管理

チャネルとイベントシンクを作成したら、サブスクリプションを作成してイベント配信を有効にすることができます。サブスクリプションは、イベントを配信するチャネルとシンク (**サブスクライバー**とも呼ばれます) を指定する **Subscription** オブジェクトを設定することによって作成されます。

5.12.1. Web コンソールを使用したサブスクリプションの作成

チャネルとイベントシンクを作成したら、サブスクリプションを作成してイベント配信を有効にすることができます。OpenShift Container Platform Web コンソールを使用すると、サブスクリプションを作成するための合理的で直感的なユーザーインターフェイスが提供されます。

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。
- Knative サービスおよびチャネルなどのイベントシンクを作成している。
- 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。

手順

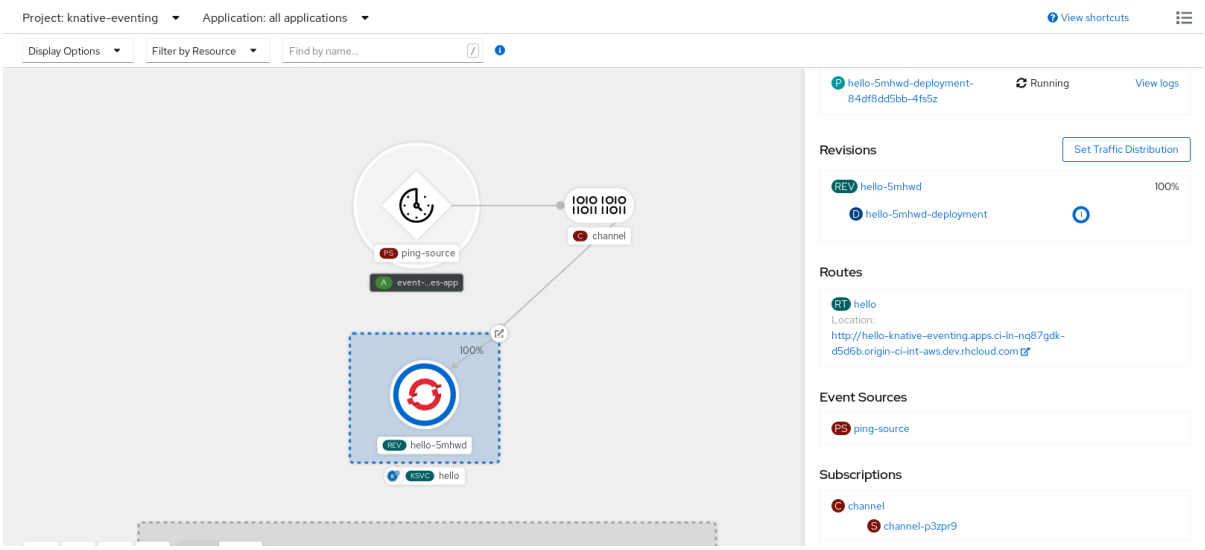
1. **Developer** パースペクティブで、**Topology** ページに移動します。
2. 以下の方法のいずれかを使用してサブスクリプションを作成します。
   a. サブスクリプションを作成するチャネルにカーソルを合わせ、矢印をドラッグします。**Add Subscription** オプションが表示されます。

      i. **Subscriber** 一覧でシンクを選択します。
      ii. **Add** をクリックします。

   b. このサービスが、チャネルと同じ namespace またはプロジェクトにある **Topology** ビューで利用可能な場合は、サブスクリプションを作成するチャネルをクリックし、矢印をサービスに直接ドラッグして、チャネルからそのサービスにサブスクリプションを即時に作成します。

検証

- サブスクリプションの作成後に、これを **Topology** ビューでチャネルをサービスに接続する行として表示できます。



5.12.2. YAML を使用したサブスクリプションの作成

チャネルとイベントシンクを作成したら、サブスクリプションを作成してイベント配信を有効にすることができます。YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でサブスクリプションを宣言的に記述することができます。YAML を使用してサブスクリプションを作成するには、**Subscription** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

- **Subscription** オブジェクトを作成します。
  - 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。

- YAML ファイルを適用します。

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

5.12.3. Knative CLI を使用したサブスクリプションの作成

チャネルとイベントシンクを作成したら、サブスクリプションを作成してイベント配信を有効にすることができます。サブスクリプションを作成するために Knative (**kn**) CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。**kn subscription create** コマンドを適切なフラグとともに使用して、サブスクリプションを作成できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 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。

手順

- サブスクリプションを作成し、シンクをチャネルに接続します。

```
$ kn subscription create <subscription_name> \
  --channel <group:version:kind>:<channel_name> \ 1
  --sink <sink_prefix>:<sink_name> \ 2
  --sink-dead-letter <sink_prefix>:<sink_name> 3
```

1. **--channel** は、処理する必要のあるクラウドイベントのソースを指定します。チャネル名を指定する必要があります。**Channel** カスタムリソースでサポートされるデフォルトの **InMemoryChannel** チャネルを使用しない場合には、チャネル名に指定されたチャネルタイプの **<group:version:kind>** の接頭辞を付ける必要があります。たとえば、これは Kafka 対応チャネルの **messaging.knative.dev:v1beta1:KafkaChannel** のようになります。

2. **--sink** は、イベントが配信されるターゲット宛先を指定します。デフォルトで、**<sink_name>** は、サブスクリプションと同じ namespace でこの名前の Knative サービスとして解釈されます。以下の接頭辞のいずれかを使用して、シンクのタイプを指定できます。

   ksvc
   : Knative サービス

   channel
   : 宛先として使用する必要のあるチャネル。ここで参照できるのは、デフォルトのチャネルタイプのみです。

   broker
   : Eventing ブローカー。

3. オプション: **--sink-dead-letter** は、イベントが配信に失敗する場合にイベントを送信するシンクを指定するために使用できるオプションのフラグです。詳細は、OpenShift Serverless の **Event 配信**についてのドキュメントを参照してください。

コマンドの例

```
$ kn subscription create mysubscription --channel mychannel --sink ksvc:event-display
```

出力例

```
Subscription 'mysubscription' created in namespace 'default'.
```

検証

- サブスクリプションを使用してチャネルがイベントシンクまたは **サブスクライバー** に接続されていることを確認するには、既存のサブスクリプションを一覧表示し、出力を検査します。

```
$ kn subscription list
```

出力例

```
NAME           CHANNEL           SUBSCRIBER          REPLY   DEAD LETTER SINK   READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                         True
```

サブスクリプションの削除

- サブスクリプションを削除します。

```
$ kn subscription delete <subscription_name>
```

5.12.4. Knative CLI を使用したサブスクリプションの記述

kn subscription describe コマンドを使用し、Knative (**kn**) CLI を使用して、端末のサブスクリプションに関する情報を出力できます。サブスクリプションを記述するために Knative CLI を使用すると、YAML ファイルを直接表示するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- Knative (**kn**) CLI をインストールしている。
- クラスターにサブスクリプションを作成している。

手順

- サブスクリプションを記述します。

```
$ kn subscription describe <subscription_name>
```

出力例

```
Name:            my-subscription
Namespace:       default
Annotations:     messaging.knative.dev/creator=openshift-user,
messaging.knative.dev/lastModifier=min ...
Age:             43s
Channel:         Channel:my-channel (messaging.knative.dev/v1)
Subscriber:
  URI:           http://edisplay.default.example.com
Reply:
  Name:          default
  Resource:      Broker (eventing.knative.dev/v1)
DeadLetterSink:
  Name:          my-sink
  Resource:      Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 43s
  ++ AddedToChannel        43s
  ++ ChannelReady          43s
  ++ ReferencesResolved    43s
```

5.12.5. Knative CLI を使用したサブスクリプションの一覧表示

kn subscription list コマンドを使用し、Knative (**kn**) CLI を使用してクラスター内の既存サブスクリプションを一覧表示できます。Knative CLI を使用してサブスクリプションを一覧表示すると、合理的で直感的なユーザーインターフェイスが提供されます。

前提条件

- Knative (**kn**) CLI をインストールしている。

手順

- クラスターのサブスクリプションを一覧表示します。

```
$ kn subscription list
```

出力例

```
NAME            CHANNEL            SUBSCRIBER          REPLY   DEAD LETTER SINK
READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                      True
```

5.12.6. Knative CLI を使用したサブスクリプションの更新

kn subscription update コマンドや適切なフラグを使用し、Knative (**kn**) CLI を使用してサブスクリプションを端末から更新できます。サブスクリプションを更新するために Knative CLI を使用すると、YAML ファイルを直接更新するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- Knative (**kn**) CLI をインストールしている。
- サブスクリプションを作成している。

手順

- サブスクリプションを更新します。

```
$ kn subscription update <subscription_name> \
  --sink <sink_prefix>:<sink_name> \ 1
  --sink-dead-letter <sink_prefix>:<sink_name> 2
```

1. **--sink** は、イベントが配信される、更新されたターゲット宛先を指定します。以下の接頭辞のいずれかを使用して、シンクのタイプを指定できます。

   ksvc
   : Knative サービス

   channel
   : 宛先として使用する必要のあるチャネル。ここで参照できるのは、デフォルトのチャネルタイプのみです。

   broker
   : Eventing ブローカー。

2. オプション: **--sink-dead-letter** は、イベントが配信に失敗する場合にイベントを送信するシンクを指定するために使用できるオプションのフラグです。詳細は、OpenShift Serverless の **Event 配信**についてのドキュメントを参照してください。

コマンドの例

```
$ kn subscription update mysubscription --sink ksvc:event-display
```

5.12.7. 次のステップ

- イベントがイベントシンクに配信されなかった場合に適用されるイベント配信パラメーターを設定します。イベント配信パラメーターの設定例を 参照してください。

5.13. ブローカーの作成

Knative は、デフォルトのチャネルベースのブローカー実装を提供します。このチャネルベースのブローカーは、開発およびテストの目的で使用できますが、実稼働環境での適切なイベント配信の保証は提供しません。

クラスター管理者がデフォルトのブローカータイプとして Kafka を使用するように OpenShift Serverless デプロイメントを設定している場合、デフォルト設定を使用してブローカーを作成すると、Kafka ベースのブローカーが作成されます。

OpenShift Serverless デプロイメントが Kafka ブローカーをデフォルトのブローカータイプとして使用するように設定されていない場合、以下の手順でデフォルト設定を使用すると、チャネルベースのブローカーが作成されます。

5.13.1. Knative CLI を使用したブローカーの作成

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。ブローカーを作成するために Knative (**kn**) CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。**kn broker create** コマンドを使用して、ブローカーを作成できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 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。

手順

- ブローカーを作成します。

```
$ kn broker create <broker_name>
```

検証

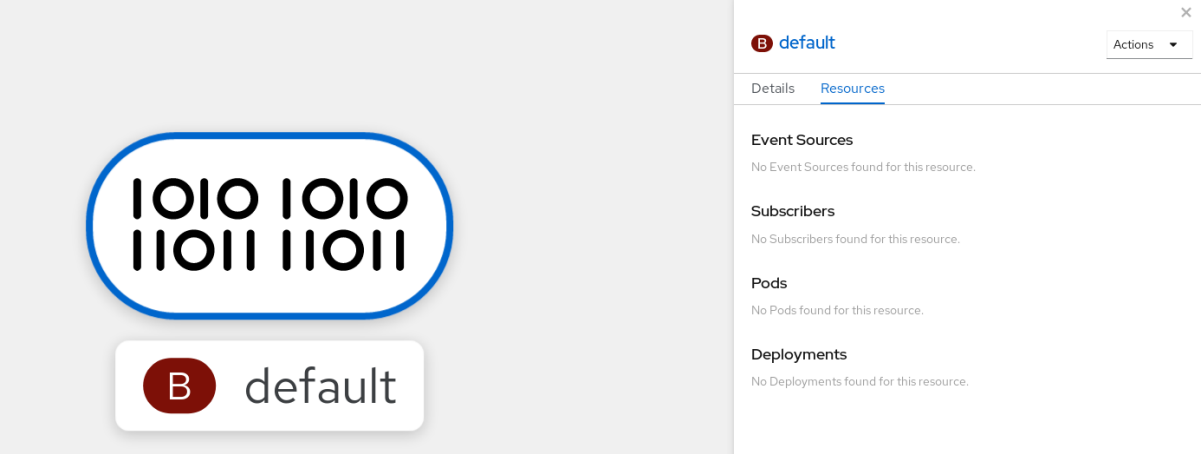
1. **kn** コマンドを使用して、既存のブローカーを一覧表示します。

```
$ 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。



5.13.2. トリガーのアノテーションによるブローカーの作成

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。**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。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

1. **Trigger** オブジェクトを、**eventing.knative.dev/injection: enabled** アノテーションを付けて 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. **Trigger** 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。

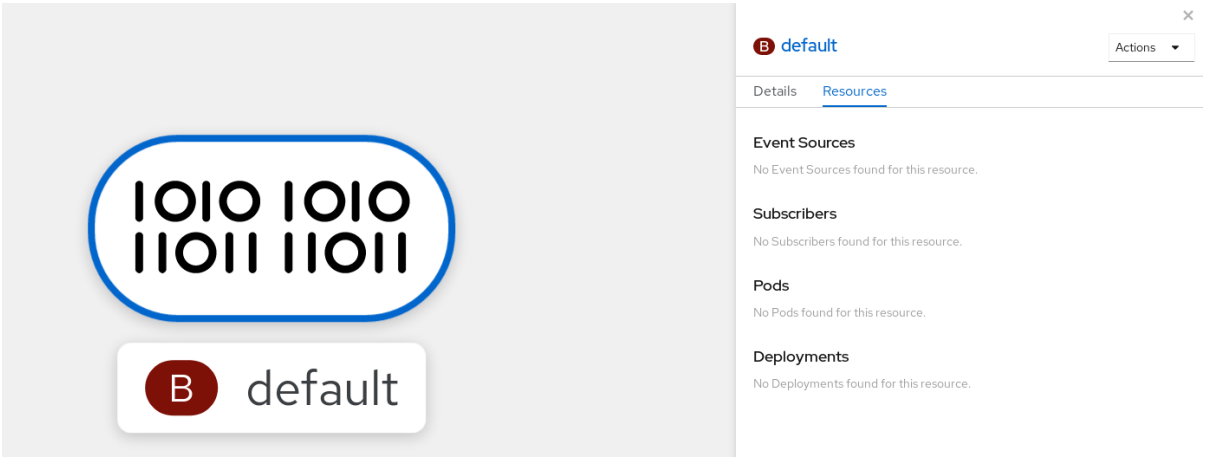
1. 以下の **oc** コマンドを入力してブローカーを取得します。

```
$ 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. 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。



5.13.3. namespace へのラベル付けによるブローカーの作成

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。所有しているか、または書き込みパーミッションのある 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 CLI (**oc**) をインストールしている。
- 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。

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>
```

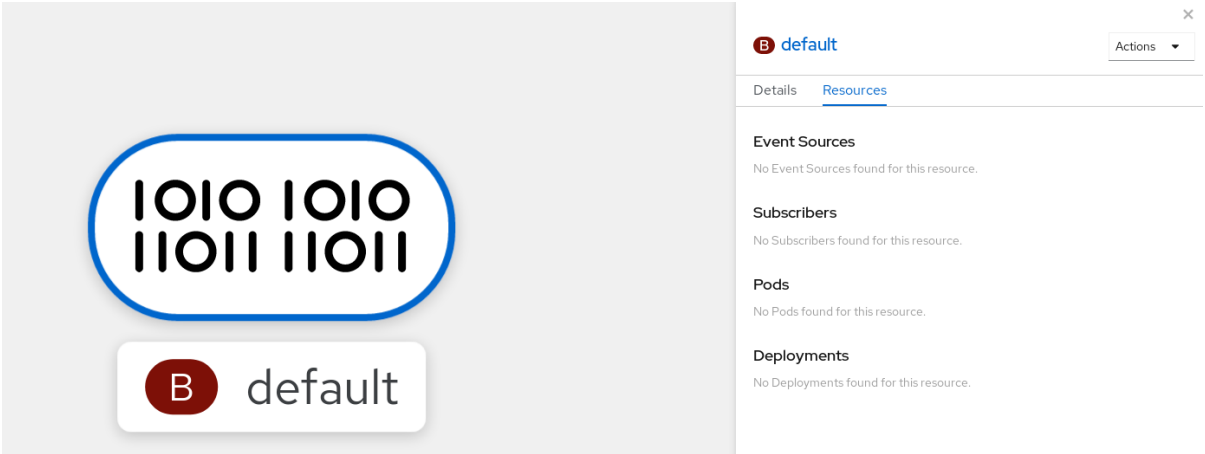
コマンドの例

```
$ 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. 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。



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

挿入によりブローカーを作成し、後でそれを削除する必要がある場合は、手動で削除する必要があります。namespace ラベルまたはトリガーアノテーションを使用して作成されたブローカーは、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。

前提条件

- OpenShift CLI (**oc**) をインストールしている。

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
```

5.13.5. デフォルトのブローカータイプとして設定されていない場合の Kafka ブローカーの作成

OpenShift Serverless デプロイメントがデフォルトのブローカータイプとして Kafka ブローカーを使用するように設定されていない場合は、以下の手順のいずれかを使用して、Kafka ベースのブローカーを作成できます。

5.13.5.1. YAML を使用した Kafka ブローカーの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でアプリケーションを宣言的に記述することができます。YAML を使用して Kafka ブローカーを作成するには、**Broker** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** カスタムリソースは OpenShift Container Platform クラスターにインストールされます。
- 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。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. Kafka ベースのブローカーを YAML ファイルとして作成します。

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: Kafka (1)
  name: example-kafka-broker
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: kafka-broker-config (2)
    namespace: knative-eventing
```

(1) ブローカークラス。指定されていない場合、ブローカーはクラスター管理者の設定に従ってデフォルトクラスを使用します。Kafka ブローカーを使用するには、この値を **Kafka** にする必要があります。

(2) Knative Kafka ブローカーのデフォルトの設定マップ。この設定マップは、クラスター管理者がクラスター上で Kafka ブローカー機能を有効にした場合に作成されます。

2. Kafka ベースのブローカー YAML ファイルを適用します。

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

5.13.5.2. 外部で管理されている Kafka トピックを使用する Kafka ブローカーの作成

独自の内部トピックの作成を許可せずに Kafka ブローカーを使用する場合は、代わりに外部で管理される Kafka トピックを使用できます。これを実行するには、**kafka.eventing.knative.dev/external.topic** アノテーションを使用する Kafka **Broker** オブジェクトを作成する必要があります。

前提条件

- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** カスタムリソースは OpenShift Container Platform クラスターにインストールされます。
- Red Hat AMQ Streams などの Kafka インスタンスにアクセスでき、Kafka トピックを作成しました。
- 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。

- OpenShift CLI (**oc**) がインストールされている。

手順

1. Kafka ベースのブローカーを YAML ファイルとして作成します。

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: Kafka (1)
    kafka.eventing.knative.dev/external.topic: <topic_name> (2)
...
```

(1) ブローカークラス。指定されていない場合、ブローカーはクラスター管理者の設定に従ってデフォルトクラスを使用します。Kafka ブローカーを使用するには、この値を **Kafka** にする必要があります。

(2) 使用する Kafka トピックの名前。

2. Kafka ベースのブローカー YAML ファイルを適用します。

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

5.13.6. ブローカーの管理

Knative (**kn**) CLI は、既存のブローカーを記述およびリストするために使用できるコマンドを提供します。

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

Knative (**kn**) CLI を使用してブローカーを一覧表示すると、合理的で直感的なユーザーインターフェイスが提供されます。**kn broker list** コマンドを使用し、Knative CLI を使用してクラスター内の既存ブローカーを一覧表示できます。

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。
- Knative (**kn**) CLI をインストールしている。

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
```

5.13.6.2. Knative CLI を使用した既存ブローカーの記述

Knative (**kn**) CLI を使用してブローカーを記述すると、合理的で直感的なユーザーインターフェイスが提供されます。**kn broker describe** コマンドを使用し、Knative CLI を使用してクラスター内の既存ブローカーに関する情報を出力できます。

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。
- Knative (**kn**) CLI をインストールしている。

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
```

5.13.7. 次のステップ

- イベントがイベントシンクに配信されなかった場合に適用されるイベント配信パラメーターを設定します。イベント配信パラメーターの設定例を 参照してください。

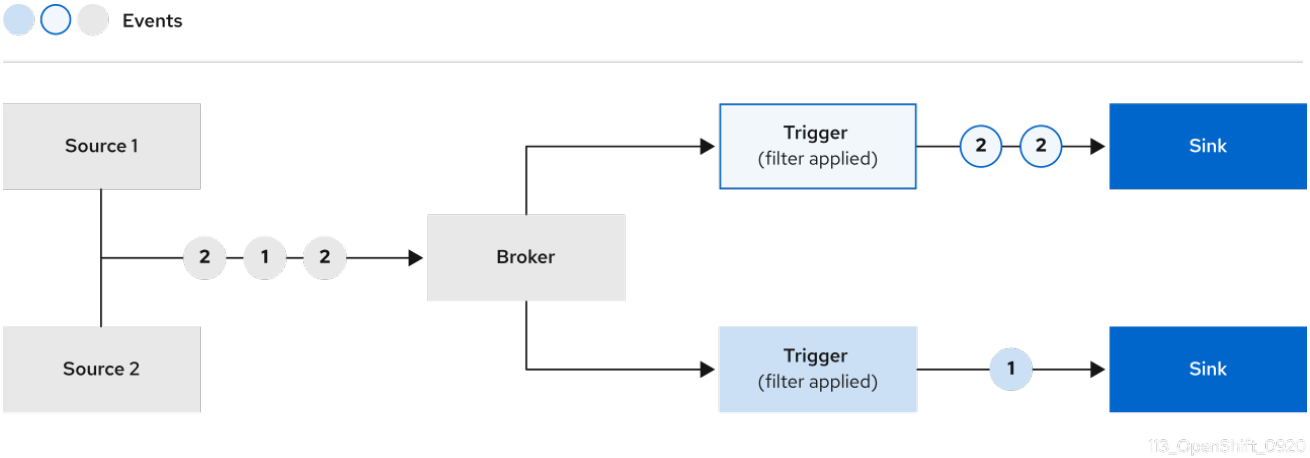
5.13.8. 関連情報

- Configuring the default broker class

- Triggers Event sources
- イベント配信
- Kafka ブローカー
- Knative Kafka の設定

5.14. トリガー

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。



Kafka ブローカーを使用している場合は、トリガーからイベントシンクへのイベントの配信順序を設定できます。トリガーのイベント配信順序の設定 を参照してください。

5.14.1. Web コンソールを使用したトリガーの作成

OpenShift Container Platform Web コンソールを使用すると、トリガーを作成するための合理的で直感的なユーザーインターフェイスが提供されます。Knative Eventing がクラスターにインストールされ、ブローカーが作成されると、Web コンソールを使用して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。
- 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. **Subscriber** 一覧でシンクを選択します。
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** ページで表示できます。ここでは、ブローカーをイベントシンクに接続する線として表されます。

トリガーの削除

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。

5.14.2. Knative CLI を使用したトリガーの作成

トリガーを作成するために Knative (**kn**) CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。**kn trigger create** コマンドを使用して、トリガーを作成できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 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。

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。

5.14.3. Knative CLI の使用によるトリガーの一覧表示

Knative (**kn**) CLI を使用してトリガーを一覧表示すると、合理的で直感的なユーザーインターフェイスが提供されます。**kn trigger list** コマンドを使用して、クラスター内の既存トリガーを一覧表示できます。

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。
- Knative (**kn**) CLI をインストールしている。

手順

1. 利用可能なトリガーの一覧を出力します。

```
$ 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
```

5.14.4. Knative CLI を使用したトリガーの記述

Knative (**kn**) CLI を使用してトリガーを記述すると、合理的で直感的なユーザーインターフェイスが提供されます。**kn trigger describe** コマンドを使用し、Knative CLI を使用してクラスター内の既存トリガーに関する情報を出力できます。

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。
- Knative (**kn**) CLI をインストールしている。
- トリガーを作成している。

手順

- 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
```

5.14.5. Knative CLI を使用したトリガーでのイベントのフィルター

Knative (**kn**) CLI を使用してイベントをフィルターリングすると、合理的で直感的なユーザーインターフェイスが提供されます。**kn trigger create** コマンドを適切なフラグとともに使用し、トリガーを使用してイベントをフィルターリングできます。

以下のトリガーの例では、**type: dev.knative.samples.helloworld** 属性のイベントのみがイベントシンクに送付されます。

```
$ 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
```

5.14.6. Knative CLI を使用したトリガーの更新

Knative (**kn**) CLI を使用してトリガーを更新すると、合理的で直感的なユーザーインターフェイスが提供されます。特定のフラグを指定して **kn trigger update** コマンドを使用して、トリガーの属性を更新できます。

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。
- Knative (**kn**) CLI をインストールしている。

- 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。

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 <trigger_name> --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 <trigger_name> --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
```

5.14.7. Knative CLI を使用したトリガーの削除

Knative (**kn**) CLI を使用してトリガーを削除すると、合理的で直感的なユーザーインターフェイスが提供されます。**kn trigger delete** コマンドを使用してトリガーを削除できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 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。

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.
```

5.14.8. トリガーのイベント配信順序の設定

Kafka ブローカーを使用している場合は、トリガーからイベントシンクへのイベントの配信順序を設定できます。

前提条件

- OpenShift Serverless Operator、Knative Eventing、および Knative Kafka が OpenShift Container Platform クラスターにインストールされている。
- Kafka ブローカーがクラスターで使用可能であり、Kafka ブローカーが作成されている。
- 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。
- OpenShift (**oc**) CLI がインストールされている。

手順

1. **Trigger** オブジェクトを作成または変更し、**kafka.eventing.knative.dev/delivery.order** アノテーションを設定します。

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: <trigger_name>
  annotations:
    kafka.eventing.knative.dev/delivery.order: ordered
...
```

サポートされているコンシューマー配信保証は次のとおりです。

unordered

順序付けられていないコンシューマーは、適切なオフセット管理を維持しながら、メッセージを順序付けずに配信するノンブロッキングコンシューマーです。

ordered

順序付きコンシューマーは、CloudEvent サブスクライバーからの正常な応答を待ってから、パーティションの次のメッセージを配信する、パーティションごとのブロックコンシューマーです。
デフォルトの順序保証は **unordered** です。

2. **Trigger** オブジェクトを適用します。

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

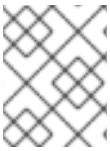
5.14.9. 次のステップ

- イベントがイベントシンクに配信されなかった場合に適用されるイベント配信パラメーターを設定します。イベント配信パラメーターの設定例を 参照してください。

5.15. KNATIVE KAFKA の使用

Knative Kafka は、OpenShift Serverless でサポートされているバージョンの Apache Kafka メッセージストリーミングプラットフォームを使用する統合オプションを提供します。Kafka は、イベントソース、チャネル、ブローカー、およびイベントシンク機能のオプションを提供します。

Knative Kafka 機能は、クラスター管理者が **KnativeKafka** カスタムリソースをインストールしている場合に、OpenShift Serverless インストールで利用できます。



注記

現時点で、Knative Kafka は IBM Z および IBM Power Systems ではサポートされていません。

Knative Kafka は、以下のような追加オプションを提供します。

- Kafka ソース
- Kafka チャネル
- Kafka ブローカー
- Kafka シンクコ

5.15.1. Kafka イベント配信およびリトライ

イベント駆動型のアーキテクチャーで Kafka コンポーネントを使用すると、最低でも 1 度のイベント配信が提供されます。これは、戻りコード値を受け取るまで操作がリトライされることを意味します。これにより、失われたイベントに対してアプリケーションの回復性が強化されます。ただし、重複するイベントが送信されてしまう可能性があります。

Kafka イベントソースでは、デフォルトでイベント配信のリトライ回数が固定されています。Kafka チャネルの場合、Kafka チャネルの **Delivery** 仕様に設定されている場合にのみリトライが実行されます。

配信保証に関する詳細は、イベント配信 のドキュメントを参照してください。

5.15.2. Kafka ソース

Apache Kafka クラスターからイベントを読み取り、これらのイベントをシンクに渡す Kafka ソースを作成できます。Kafka ソースを作成するには、OpenShift Container Platform Web コンソールの Knative (**kn**)CLI を使用するか、**KafkaSource** オブジェクトを YAML ファイルとして直接作成し、OpenShift CLI (**oc**) を使用して適用します。

5.15.2.1. Web コンソールを使用した Kafka イベントソースの作成

Knative Kafka をクラスターにインストールした後、Web コンソールを使用して Kafka ソースを作成できます。OpenShift Container Platform Web コンソールを使用すると、Kafka ソースを作成するための合理的で直感的なユーザーインターフェイスが提供されます。

前提条件

- OpenShift Serverless Operator、Knative Serving、および **KnativeKafka** カスタムリソースが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。
- インポートする Kafka メッセージを生成する Red Hat AMQ Streams (Kafka) クラスターにアクセスできる。
- 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。

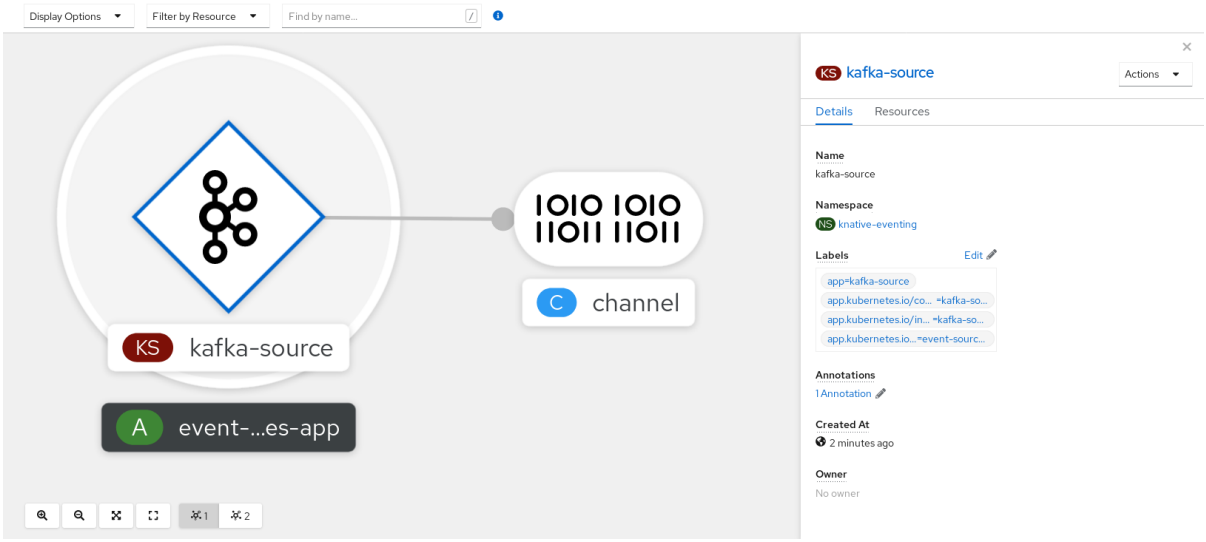
手順

1. **Developer** パースペクティブで、**Add** ページに移動し、**Event Source** を選択します。
2. **Event Sources** ページで、**Type** セクションの **Kafka Source** を選択します。
3. **Kafka Source** 設定を設定します。
   a. **ブートストラップサーバー** のコンマ区切りの一覧を追加します。
   b. **トピック** のコンマ区切りの一覧を追加します。
   c. **コンシューマーグループ** を追加します。
   d. 作成したサービスアカウントの **Service Account Name** を選択します。
   e. イベントソースの **Sink** を選択します。**Sink** は、チャネル、ブローカー、またはサービスなどの **Resource**、または **URI** のいずれかになります。
   f. Kafka イベントソースの **Name** を入力します。
4. **Create** をクリックします。

検証

Topology ページを表示して、Kafka イベントソースが作成され、シンクに接続されていることを確認できます。

1. **Developer** パースペクティブで、**Topology** に移動します。
2. Kafka イベントソースおよびシンクを表示します。



5.15.2.2. Knative CLI を使用した Kafka イベントソースの作成

kn source kafka create コマンドを使用し、Knative (**kn**) CLI を使用して Kafka ソースを作成できます。イベントソースを作成するために Knative CLI を使用すると、YAML ファイルを直接修正するよりも合理的で直感的なユーザーインターフェイスが得られます。

前提条件

- OpenShift Serverless Operator、Knative Eventing、Knative Serving、および **KnativeKafka** カスタムリソース (CR) がクラスターにインストールされている。
- 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。
- インポートする Kafka メッセージを生成する Red Hat AMQ Streams (Kafka) クラスターにアクセスできる。
- Knative (**kn**) CLI をインストールしている。
- オプション: この手順で検証ステップを使用する場合は、OpenShift CLI (**oc**) をインストールします。

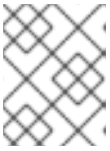
手順

1. Kafka イベント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 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
```

2. **KafkaSource** CR を作成します。

```
$ kn source kafka create <kafka_source_name> \
    --servers <cluster_kafka_bootstrap>.kafka.svc:9092 \
    --topics <topic_name> --consumergroup my-consumer-group \
    --sink event-display
```



注記

このコマンドのプレースホルダー値は、ソース名、ブートストラップサーバー、およびトピックの値に置き換えます。

--servers、**--topics**、および **--consumergroup** オプションは、Kafka クラスターへの接続パラメーターを指定します。**--consumergroup** オプションは任意です。

3. オプション: 作成した **KafkaSource** CR の詳細を表示します。

```
$ kn source kafka describe <kafka_source_name>
```

出力例

```
Name:              example-kafka-source
Namespace:         kafka
Age:               1h
BootstrapServers:  example-cluster-kafka-bootstrap.kafka.svc:9092
Topics:            example-topic
ConsumerGroup:     example-consumer-group

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

Conditions:
  OK TYPE           AGE REASON
  ++ Ready          1h
  ++ Deployed       1h
  ++ SinkProvided   1h
```

検証手順

1. Kafka インスタンスをトリガーし、メッセージをトピックに送信します。

```
$ oc -n kafka run kafka-producer \
    -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true \
    --restart=Never -- bin/kafka-console-producer.sh \
    --broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic
```

プロンプトにメッセージを入力します。このコマンドは、以下を前提とします。

- Kafka クラスターが **kafka** namespace にインストールされている。
- **KafkaSource** オブジェクトは、**my-topic** トピックを使用するように設定されている。

2. ログを表示して、メッセージが到達していることを確認します。

```
$ 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.kafka.event
  source: /apis/v1/namespaces/default/kafkasources/example-kafka-source#example-topic
  subject: partition:46#0
  id: partition:46/offset:0
  time: 2021-03-10T11:21:49.4Z
Extensions,
  traceparent: 00-161ff3815727d8755848ec01c866d1cd-7ff3916c44334678-00
Data,
  Hello!
```

5.15.2.2.1. Knative CLI シンクフラグ

Knative (**kn**) CLI を使用してイベントソースを作成する場合、**--sink** フラグを使用して、イベントがリソースから送信されるシンクを指定できます。シンクは、他のリソースから受信イベントを受信できる、アドレス指定可能または呼び出し可能な任意のリソースです。

以下の例では、サービスの **http://event-display.svc.cluster.local** をシンクとして使用するシンクバインディングを作成します。

シンクフラグを使用したコマンドの例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** の **svc** は、シンクが Knative サービスであることを判別します。他のデフォルトのシンクの接頭辞には、**channel** および **broker** が含まれます。

5.15.2.3. YAML を使用した Kafka イベントソースの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でアプリケーションを宣言的に記述することができます。YAML を使用して Kafka ソースを作成するには、**KafkaSource** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator、Knative Serving、および **KnativeKafka** カスタムリソースがクラスターにインストールされている。
- 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。
- インポートする Kafka メッセージを生成する Red Hat AMQ Streams (Kafka) クラスターにアクセスできる。
- OpenShift CLI (**oc**) をインストールしている。

手順

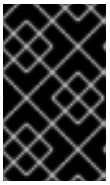
1. **KafkaSource** オブジェクトを YAML ファイルとして作成します。

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: <source_name>
spec:
  consumerGroup: <group_name> 1
  bootstrapServers:
  - <list_of_bootstrap_servers>
  topics:
  - <list_of_topics> 2
  sink:
  - <list_of_sinks> 3
```

1. コンシューマーグループは、同じグループ ID を使用し、トピックからデータを消費するコンシューマーのグループです。

2. トピックは、データの保存先を提供します。各トピックは、1 つまたは複数のパーティションに分割されます。

3. シンクは、イベントがソースから送信される場所を指定します。



重要

OpenShift Serverless 上の **KafkaSource** オブジェクトの API の **v1beta1** バージョンのみがサポートされます。非推奨となった **v1alpha1** バージョンの API は使用しないでください。

KafkaSource オブジェクトの例

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: kafka-source
spec:
  consumerGroup: knative-group
  bootstrapServers:
  - my-cluster-kafka-bootstrap.kafka:9092
  topics:
  - knative-demo-topic
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

2. **KafkaSource** YAML ファイルを適用します。

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

検証

- 以下のコマンドを入力して、Kafka イベントソースが作成されたことを確認します。

```
$ oc get pods
```

出力例

```
NAME                                READY   STATUS    RESTARTS   AGE
kafkasource-kafka-source-5ca0248f-...   1/1     Running   0          13m
```

5.15.3. Kafka ブローカー

実稼働環境に対応した Knative Eventing デプロイメントの場合、Red Hat は Knative Kafka ブローカーの実装を使用することをお勧めします。Kafka ブローカーは、Knative ブローカーの Apache Kafka ネイティブ実装であり、CloudEvents を Kafka インスタンスに直接送信します。

重要

Kafka ブローカーの連邦情報処理標準 (FIPS) モードが無効になっています。

Kafka ブローカーは、イベントの保存とルーティングのために Kafka とネイティブに統合されています。これにより、他のブローカータイプよりもブローカーとトリガーモデルの Kafka との統合性が向上し、ネットワークホップを削減することができます。Kafka ブローカー実装のその他の利点は次のとおりです。

- 少なくとも 1 回の配信保証
- CloudEvents パーティショニング拡張機能に基づくイベントの順序付き配信
- コントロールプレーンの高可用性
- 水平方向にスケーラブルなデータプレーン

Knative Kafka ブローカーは、バイナリーコンテンツモードを使用して、受信 CloudEvents を Kafka レコードとして保存します。これは、CloudEvent のすべての属性と拡張機能が Kafka レコードのヘッダーとしてマップされ、CloudEvent の **data** 仕様が Kafka レコードの値に対応することを意味します。

Kafka ブローカーの使用については、ブローカーの作成 を参照してください。

5.15.4. YAML を使用した Kafka チャネルの作成

YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でチャネルを宣言的に記述することができます。Kafka チャネルを作成することで、Kafka トピックに裏打ちされた Knative Eventing チャネルを作成できます。YAML を使用して Kafka チャネルを作成するには、**KafkaChannel** オブジェクトを定義する YAML ファイルを作成し、**oc apply** コマンドを使用してそれを適用する必要があります。

前提条件

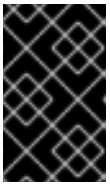
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** カスタムリソースは OpenShift Container Platform クラスターにインストールされます。
- OpenShift CLI (**oc**) をインストールしている。

- 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。

手順

1. **KafkaChannel** オブジェクトを YAML ファイルとして作成します。

```
apiVersion: messaging.knative.dev/v1beta1
kind: KafkaChannel
metadata:
  name: example-channel
  namespace: default
spec:
  numPartitions: 3
  replicationFactor: 1
```



重要

OpenShift Serverless 上の **KafkaChannel** オブジェクトの API の **v1beta1** バージョンのみがサポートされます。非推奨となった **v1alpha1** バージョンの API は使用しないでください。

2. **KafkaChannel** YAML ファイルを適用します。

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

5.15.5. Kafka シンクコ

Kafka シンクは、クラスター管理者がクラスターで Kafka を有効にした場合に使用できるイベントシンクの一種です。Kafka シンクを使用して、イベントソースから Kafka トピックにイベントを直接送信できます。

5.15.5.1. Kafka シンクの使用

Kafka トピックにイベントを送信する Kafka シンクと呼ばれるイベントシンクを作成できます。YAML ファイルを使用して Knative リソースを作成する場合、宣言的 API を使用するため、再現性の高い方法でアプリケーションを宣言的に記述することができます。YAML を使用して Kafka シンクを作成するには、**KafkaSink** オブジェクトを定義する YAML ファイルを作成してから、**ocapply** コマンドを使用してそれを適用する必要があります。

前提条件

- OpenShift Serverless Operator、Knative Serving、および **KnativeKafka** カスタムリソース (CR) がクラスターにインストールされている。
- 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。
- インポートする Kafka メッセージを生成する Red Hat AMQ Streams (Kafka) クラスターにアクセスできる。

- OpenShift CLI (**oc**) をインストールしている。

手順

1. **KafkaSink** オブジェクト定義を YAML ファイルとして作成します。

   Kafka シンク YAML

```
apiVersion: eventing.knative.dev/v1alpha1
kind: KafkaSink
metadata:
  name: <sink-name>
  namespace: <namespace>
spec:
  topic: <topic-name>
  bootstrapServers:
   - <bootstrap-server>
```

2. Kafka シンクを作成するには、**KafkaSink** YAML ファイルを適用します。

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

3. シンクが仕様で指定されるようにイベントソースを設定します。

   API サーバーソースに接続された Kafka シンクの例

```
apiVersion: sources.knative.dev/v1alpha2
kind: ApiServerSource
metadata:
  name: <source-name> 1
  namespace: <namespace> 2
spec:
  serviceAccountName: <service-account-name> 3
  mode: Resource
  resources:
  - apiVersion: v1
    kind: Event
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: KafkaSink
      name: <sink-name> 4
```

1. イベントソースの名前。
2. イベントソースの namespace。
3. イベントソースのサービスアカウント。
4. Kafka シンクの名前。

5.15.6. 関連情報

- Red Hat AMQ Streams のドキュメント
- Kafka における Red Hat AMQ Streams TLS および SASL に関するドキュメントを参照してください。
- イベント配信
- Knative Kafka クラスター管理者ドキュメント

第6章 管理

6.1. グローバル設定

OpenShift Serverless Operator は、**KnativeServing** および **KnativeEventing** カスタムリソースからシステムの 設定マップ への値の反映を含む Knative インストールのグローバル設定を管理します。手動で適用される設定マップの更新は Operator によって上書きされます。ただし、Knative カスタムリソースを変更すると、これらの設定マップの値を設定できます。

Knative には、名前に接頭辞 **config-** が付けられた複数の設定マップがあります。すべての Knative 設定マップは、適用するカスタムリソースと同じ namespace に作成されます。たとえば、**KnativeServing** カスタムリソースが **knative-serving** namespace に作成される場合、すべての Knative Serving 設定マップもこの namespace に作成されます。

Knative カスタムリソースの **spec.config** には、設定マップごとに **config-<name>** という名前の **<name>** エントリーが 1 つあり、設定マップ **data** で使用される値を持ちます。

6.1.1. デフォルトチャネル実装の設定

default-ch-webhook 設定マップを使用して、Knative Eventing のデフォルトのチャネル実装を指定できます。クラスター全体または 1 つ以上の namespace に対して、デフォルトのチャネルの実装を指定できます。現在、**InMemoryChannel** および **KafkaChannel** チャネルタイプがサポートされています。

前提条件

- OpenShift Container Platform に対する管理者権限を持っている。
- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされていること。
- デフォルトのチャネル実装として Kafka チャネルを使用する場合は、クラスターに **KnativeKafka** CR もインストールする必要があります。

手順

- **KnativeEventing** カスタムリソースを変更して、**default-ch-webhook** 設定マップの設定の詳細を追加します。

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config: (1)
    default-ch-webhook: (2)
      default-ch-config: |
        clusterDefault: (3)
          apiVersion: messaging.knative.dev/v1
          kind: InMemoryChannel
          spec:
            delivery:
              backoffDelay: PT0.5S
              backoffPolicy: exponential
```

```
      retry: 5
  namespaceDefaults: 4
    my-namespace:
      apiVersion: messaging.knative.dev/v1beta1
      kind: KafkaChannel
      spec:
        numPartitions: 1
        replicationFactor: 1
```

1. **spec.config** で、変更した設定を追加する設定マップを指定できます。
2. **default-ch-webhook** 設定マップは、クラスターまたは 1 つ以上の namespace のデフォルトチャネルの実装を指定するために使用できます。
3. クラスター全体のデフォルトのチャネルタイプの設定。この例では、クラスターのデフォルトのチャネル実装は **InMemoryChannel** です。
4. namespace スコープのデフォルトのチャネルタイプの設定。この例では、**my-namespace** namespace のデフォルトのチャネル実装は **KafkaChannel** です。

重要

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。

6.1.2. デフォルトのブローカーバッキングチャネルの設定

チャネルベースのブローカーを使用している場合、ブローカーのデフォルトのバッキングチャネルタイプを **InMemoryChannel** または **KafkaChannel** に設定できます。

前提条件

- OpenShift Container Platform に対する管理者権限を持っている。
- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされていること。
- OpenShift (**oc**) CLI がインストールされている。
- Kafka チャネルをデフォルトのバッキングチャネルタイプとして使用する場合は、クラスターに **KnativeKafka** CR もインストールする必要があります。

手順

1. **KnativeEventing** カスタムリソース (CR) を変更して、**config-br-default-channel** 設定マップの設定の詳細を追加します。

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config: 1
```

```
    config-br-default-channel:
      channel-template-spec: |
        apiVersion: messaging.knative.dev/v1beta1
        kind: KafkaChannel (2)
        spec:
          numPartitions: 6 (3)
          replicationFactor: 3 (4)
```

(1) **spec.config** で、変更した設定を追加する設定マップを指定できます。

(2) デフォルトのバッキングチャネルタイプの設定。この例では、クラスターのデフォルトのチャネル実装は **KafkaChannel** です。

(3) ブローカーをサポートする Kafka チャネルのパーティションの数。

(4) ブローカーをサポートする Kafka チャネルのレプリケーションファクター。

2. 更新された **KnativeEventing** CR を適用します。

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

6.1.3. デフォルトブローカークラスの設定

config-br-defaults 設定マップを使用して、Knative Eventing のデフォルトのブローカークラス設定を指定できます。クラスター全体または 1 つ以上の namespace に対して、デフォルトのブローカークラスを指定できます。現在、**MTChannelBasedBroker** および **Kafka** ブローカータイプがサポートされています。

前提条件

- OpenShift Container Platform に対する管理者権限を持っている。
- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされていること。
- Kafka ブローカーをデフォルトのブローカー実装として使用する場合は、クラスターに **KnativeKafka** CR もインストールする必要があります。

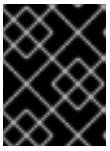
手順

- **KnativeEventing** カスタムリソースを変更して、**config-br-defaults** 設定マップの設定の詳細を追加します。

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  defaultBrokerClass: Kafka (1)
  config: (2)
    config-br-defaults: (3)
      default-br-config: |
```

```
    clusterDefault: 4
      brokerClass: Kafka
      apiVersion: v1
      kind: ConfigMap
      name: kafka-broker-config 5
      namespace: knative-eventing 6
    namespaceDefaults: 7
      my-namespace:
        brokerClass: MTChannelBasedBroker
        apiVersion: v1
        kind: ConfigMap
        name: config-br-default-channel 8
        namespace: knative-eventing 9
...
```

1. Knative Eventing のデフォルトのブローカークラス。
2. **spec.config** で、変更した設定を追加する設定マップを指定できます。
3. **config-br-defaults** 設定マップは、**spec.config** 設定またはブローカークラスを指定しないブローカーのデフォルト設定を指定します。
4. クラスター全体のデフォルトのブローカークラス設定。この例では、クラスターのデフォルトのブローカークラスの実装は **Kafka** です。
5. **kafka-broker-config** 設定マップは、Kafka ブローカーのデフォルト設定を指定します。「関連情報」セクションの「Kafka ブローカー構成の設定」を参照してください。
6. **kafka-broker-config** 設定マップが存在する namespace。
7. namespace スコープのデフォルトブローカクラス設定。この例では、**my-namespace** namespace のデフォルトのブローカークラスの実装は **MTChannelBasedBroker** です。複数の namespace に対してデフォルトのブローカークラスの実装を指定できます。
8. **config-br-default-channel** 設定マップは、ブローカーのデフォルトのバッキングチャネルを指定します。「関連情報」セクションの「デフォルトのブローカーバッキングチャネルの設定」を参照してください。
9. **config-br-default-channel** 設定マップが存在する namespace。



重要

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。

関連情報

- Kafka ブローカー構成の設定
- デフォルトのブローカーバッキングチャネルの設定

6.1.4. scale-to-zero の有効化

Knative Serving は、アプリケーションが受信要求に一致するように、自動スケーリング (**autoscaling**) を提供します。**enable-scale-to-zero** 仕様を使用して、クラスター上のアプリケーションの scale-to-zero をグローバルに有効または無効にすることができます。

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。
- クラスター管理者パーミッションがある。
- デフォルトの Knative Pod Autoscaler を使用している。Kubernetes Horizontal Pod Autoscaler を使用している場合は、ゼロにスケーリングすることはできません。

手順

- **KnativeServing** カスタムリソース (CR) の **enable-scale-to-zero** 仕様を変更します。

  KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      enable-scale-to-zero: "false" 1
```

  1 **enable-scale-to-zero** 仕様は、**true** または **false** のいずれかです。true に設定すると、scale-to-zero が有効にされます。false に設定すると、アプリケーションは設定された **スケーリング下限** にスケールダウンされます。デフォルト値は **"true"** です。

6.1.5. scale-to-zero 猶予期間の設定

Knative Serving は、アプリケーションの Pod をゼロにスケールダウンします。**scale-to-zero-grace-period** 仕様を使用して、アプリケーションの最後のレプリカが削除される前に Knative が scale-to-zero 機構が配置されるのを待機する上限時間を定義できます。

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。
- クラスター管理者パーミッションがある。
- デフォルトの Knative Pod Autoscaler を使用している。Kubernetes Horizontal Pod Autoscaler を使用している場合は、ゼロにスケーリングすることはできません。

手順

- **KnativeServing** カスタムリソース CR の **scale-to-zero-grace-period** 仕様を変更します。

  KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      scale-to-zero-grace-period: "30s" 1
```

1 猶予期間 (秒単位)。デフォルト値は 30 秒です。

6.1.6. システムのデプロイメント設定の上書き

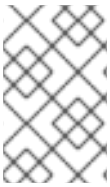
KnativeServing および **KnativeEventing** カスタムリソース (CR) で **deployments** 仕様を変更することにより、一部の特定のデプロイメントのデフォルト設定をオーバーライドできます。

6.1.6.1. Knative Serving システムのデプロイメント設定のオーバーライド

KnativeServing カスタムリソース (CR) の **deployments** 仕様を変更することで、特定のデプロイメントのデフォルト設定を上書きできます。現在、デフォルトの設定設定のオーバーライドは、**resource**、**replica**、**labels**、**annotations**、および **nodeSelector** フィールドでサポートされています。

以下の例では、**KnativeServing** CR は **webhook** デプロイメントをオーバーライドし、以下を確認します。

- デプロイメントには、CPU およびメモリーのリソース制限が指定されています。
- デプロイメントには 3 つのレプリカがあります。
- **example-label:labellabel** が追加されました。
- **example-annotation: annotation** が追加されます。
- **nodeSelector** フィールドは、**disktype: hdd** ラベルを持つノードを選択するように設定されます。



注記

KnativeServing CR ラベルおよびアノテーション設定は、デプロイメント自体と結果として生成される Pod の両方のデプロイメントのラベルおよびアノテーションを上書きします。

KnativeServing CR の例

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

```
  deployments:
  - name: webhook
    resources:
    - container: webhook
      requests:
        cpu: 300m
        memory: 60Mi
      limits:
        cpu: 1000m
        memory: 1000Mi
    replicas: 3
    labels:
      example-label: label
    annotations:
      example-annotation: annotation
    nodeSelector:
      disktype: hdd
```

6.1.6.2. Knative Eventing システムのデプロイメント設定のオーバーライド

KnativeEventing カスタムリソース (CR) の **deployments** 仕様を変更することで、特定のデプロイメントのデフォルト設定を上書きできます。現在、**eventing-controller**、**eventing-webhook**、および **imc-controller** フィールドで、デフォルトの設定設定のオーバーライドがサポートされています。

重要

replicas の仕様は、Horizontal Pod Autoscaler (HPA) を使用するデプロイのレプリカの数をオーバーライドできず、**eventing-webhook** デプロイでは機能しません。

次の例では、**KnativeEventing** CR が **eventing-controller** デプロイメントをオーバーライドして、次のようにします。

- デプロイメントには、CPU およびメモリーのリソース制限が指定されています。
- デプロイメントには 3 つのレプリカがあります。
- **example-label:labellabel** が追加されました。
- **example-annotation: annotation** が追加されます。
- **nodeSelector** フィールドは、**disktype: hdd** ラベルを持つノードを選択するように設定されます。

KnativeEventing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  deployments:
  - name: eventing-controller
    resources:
    - container: eventing-controller
```

```
    requests:
      cpu: 300m
      memory: 100Mi
    limits:
      cpu: 1000m
      memory: 250Mi
  replicas: 3
  labels:
    example-label: label
  annotations:
    example-annotation: annotation
  nodeSelector:
    disktype: hdd
```

注記

KnativeEventing CR ラベルおよびアノテーション設定は、デプロイメント自体と結果として生成される Pod の両方のデプロイメントのラベルおよびアノテーションを上書きします。

6.1.7. EmptyDir 拡張機能の設定

emptyDir ボリュームは、Pod の作成時に作成される空のボリュームであり、一時的な作業ディスク領域を提供するために使用されます。**emptyDir** ボリュームは、それらが作成された Pod が削除されると削除されます。

kubernetes.podspec-volumes-emptydir の拡張は、**emptyDir** ボリュームを Knative Serving で使用できるかどうかを制御します。**emptyDir** ボリュームの使用を有効にするには、**KnativeServing** カスタムリソース (CR) を変更して以下の YAML を追加する必要があります。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    features:
      kubernetes.podspec-volumes-emptydir: enabled
...
```

6.1.8. HTTPS リダイレクトのグローバル設定

HTTPS リダイレクトは、着信 HTTP リクエストのリダイレクトを提供します。これらのリダイレクトされた HTTP リクエストは暗号化されます。**KnativeServing** カスタムリソース (CR) の **httpProtocol** 仕様を設定して、クラスターのすべてのサービスに対して HTTPS リダイレクトを有効にできます。

HTTPS リダイレクトを有効にする KnativeServing CR の例

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

```
spec:
  config:
    network:
      httpProtocol: "redirected"
...
```

6.1.9. 外部ルートの URL スキームの設定

セキュリティーを強化するために、外部ルートの URL スキームはデフォルトで HTTPS に設定されています。このスキームは、**KnativeServing** カスタムリソース (CR) 仕様の **default-external-scheme** キーによって決定されます。

デフォルト仕様

```
...
spec:
  config:
    network:
      default-external-scheme: "https"
...
```

default-external-schemeキーを変更することにより、HTTP を使用するようにデフォルトの仕様をオーバーライドできます。

HTTP オーバーライド仕様

```
...
spec:
  config:
    network:
      default-external-scheme: "http"
...
```

6.1.10. Kourier Gateway サービスタイプの設定

Kourier Gateway は、デフォルトで **ClusterIP** サービスタイプとして公開されます。このサービスタイプは、**KnativeServing** カスタムリソース (CR) の **service-type** 入力仕様によって決定されます。

デフォルト仕様

```
...
spec:
  ingress:
    kourier:
      service-type: ClusterIP
...
```

service-type 仕様を変更することで、デフォルトのサービスタイプをオーバーライドして、代わりにロードバランサーサービスタイプを使用できます。

LoadBalancer オーバーライド仕様

```
...
```

```
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...
```

6.1.11. PVC サポートの有効化

一部のサーバーレスアプリケーションには、永続的なデータストレージが必要です。これを実現するために、Knative サービスの永続ボリュームクレーム (PVC) を設定できます。

重要

Knative サービスの PVC サポートは、テクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

手順

1. Knative Serving が PVC を使用して書き込むことができるようにするには、**KnativeServing** カスタムリソース (CR) を変更して次の YAML を含めます。

   書き込みアクセスで PVC を有効にする

   ```
   ...
   spec:
     config:
       features:
         "kubernetes.podspec-persistent-volume-claim": enabled
         "kubernetes.podspec-persistent-volume-write": enabled
   ...
   ```

   - **kubernetes.podspec-persistent-volume-claim** 拡張機能は、永続ボリューム (PV) を Knative Serving で使用できるかどうかを制御します。
   - **kubernetes.podspec-persistent-volume-write** 拡張機能は、書き込みアクセスで Knative Serving が PV を利用できるかどうかを制御します。

2. PV を要求するには、PV 設定を含めるようにサービスを変更します。たとえば、次の設定で永続的なボリュームクレームがある場合があります。

   注記

   要求しているアクセスモードをサポートするストレージクラスを使用してください。たとえば、**ReadWriteMany** アクセスモードの **ocs-storagecluster-cephfs** クラスを使用できます。

   PersistentVolumeClaim 設定

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pv-claim
  namespace: my-ns
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ocs-storagecluster-cephfs
  resources:
    requests:
      storage: 1Gi
```

この場合、書き込みアクセス権を持つ PV を要求するには、次のようにサービスを変更します。

ネイティブサービス PVC 設定

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  namespace: my-ns
...
spec:
 template:
   spec:
     containers:
         ...
         volumeMounts: (1)
           - mountPath: /data
             name: mydata
             readOnly: false
     volumes:
       - name: mydata
         persistentVolumeClaim: (2)
           claimName: example-pv-claim
           readOnly: false (3)
```

1. ボリュームマウント仕様。
2. 永続的なボリュームクレームの仕様。
3. 読み取り専用アクセスを有効にするフラグ。

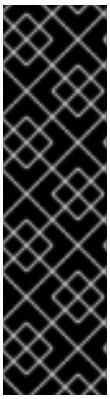
注記

Knative サービスで永続ストレージを正常に使用するには、Knative コンテナーユーザーのユーザー権限などの追加の設定が必要です。

6.1.12. init コンテナーの有効化

Init コンテナー は、Pod 内のアプリケーションコンテナーの前に実行される特殊なコンテナーです。これらは通常、アプリケーションの初期化ロジックを実装するために使用されます。これには、セット

アップスクリプトの実行や、必要な設定のダウンロードが含まれる場合があります。**KnativeServing** カスタムリソース (CR) を変更することにより、Knative サービスの init コンテナーの使用を有効にできます。



重要

Knative サービスの初期化コンテナーはテクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

注記

Init コンテナーを使用すると、アプリケーションの起動時間が長くなる可能性があるため、頻繁にスケールアップおよびスケールダウンすることが予想されるサーバーレスアプリケーションには注意して使用する必要があります。

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。
- クラスター管理者パーミッションがある。

手順

- **KnativeServing** CR に **kubernetes.podspec-init-containers** フラグを追加して、init コンテナーの使用を有効にします。

  KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    features:
      kubernetes.podspec-init-containers: enabled
...
```

6.1.13. タグからダイジェストへの解決

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できる場合、Knative Serving は、サービスのリビジョンを作成するときにイメージタグをダイジェストに解決します。これは**タグからダイジェストへの解決**と呼ばれ、デプロイメントの一貫性を提供するのに役立ちます。

コントローラーに OpenShift Container Platform のコンテナーレジストリーへのアクセスを許可するには、シークレットを作成してから、コントローラーのカスタム証明書を設定する必要がありま

す。**KnativeServing** カスタムリソース (CR) の **controller-custom-certs** 仕様を変更することにより、コントローラーカスタム証明書を設定できます。シークレットは、**KnativeServing** CR と同じ namespace に存在する必要があります。

シークレットが **KnativeServing** CR に含まれていない場合、この設定はデフォルトで公開鍵インフラストラクチャー (PKI) を使用します。PKI を使用する場合、クラスター全体の証明書は、**config-service-sa** 設定マップを使用して KnativeServing コントローラーに自動的に挿入されます。OpenShift Serverless Operator は、**config-service-sa** 設定 マップにクラスター全体の証明書を設定し、設定マップをボリュームとしてコントローラーにマウントします。

6.1.13.1. シークレットを使用したタグからダイジェストへの解決の設定

controller-custom-certs 仕様で **Secret** タイプが使用されている場合、シークレットはシークレットボリュームとしてマウントされます。シークレットに必要な証明書があると仮定すると、ネイティブコンポーネントはシークレットを直接消費します。

前提条件

- OpenShift Container Platform のクラスター管理者パーミッションがある。
- 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。

手順

1. シークレットを作成します。

   コマンドの例

   ```
   $ oc -n knative-serving create secret generic custom-secret --from-file=<secret_name>.crt=
   <path_to_certificate>
   ```

2. **Secret** タイプを使用するように、**KnativeServing** カスタムリソース (CR) で **controller-custom-certs** 仕様を設定します。

   KnativeServing CR の例

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

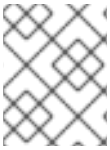
6.1.14. 関連情報

- カスタムリソース定義からのリソースの管理
- 永続ストレージについて
- カスタム PKI の設定

6.2. KNATIVE KAFKA の設定

Knative Kafka は、OpenShift Serverless でサポートされているバージョンの Apache Kafka メッセージストリーミングプラットフォームを使用する統合オプションを提供します。Kafka は、イベントソース、チャネル、ブローカー、およびイベントシンク機能のオプションを提供します。

OpenShift Serverless のコアインストールの一部として提供される Knative Eventing コンポーネントの他に、クラスター管理者は **KnativeKafka** カスタムリソース (CR) をインストールできます。



注記

現時点で、Knative Kafka は IBM Z および IBM Power Systems ではサポートされていません。

KnativeKafka CR は、ユーザーに以下のような追加オプションを提供します。

- Kafka ソース
- Kafka チャネル
- Kafka ブローカー
- Kafka シンクコ

6.2.1. Knative Kafka のインストール

Knative Kafka は、OpenShift Serverless でサポートされているバージョンの Apache Kafka メッセージストリーミングプラットフォームを使用する統合オプションを提供します。**KnativeKafka** カスタムリソースをインストールしている場合、Knative Kafka 機能は OpenShift Serverless インストールで使用できます。

前提条件

- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされていること。
- Red Hat AMQ Streams クラスターにアクセスできる。
- 検証手順を使用する場合は、OpenShift CLI (**oc**) をインストールします。
- OpenShift Container Platform のクラスター管理者パーミッションがある。
- OpenShift Container Platform Web コンソールにログインしている。

手順

1. **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 APIs** の一覧で **Knative Kafka** ボックスを見つけ、**Create Instance** をクリックします。
4. **Create Knative Kafka** ページで **KnativeKafka** オブジェクトを設定します。

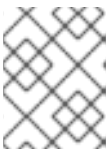
重要

クラスターで Kafka チャネル、ソース、ブローカー、またはシンクを使用するには、使用するオプションの **有効な** スイッチを **true** に切り替える必要があります。これらのスイッチは、デフォルトで **false** に設定されます。さらに、Kafka チャネル、ブローカー、またはシンクを使用するには、ブートストラップサーバーを指定する必要があります。

KnativeKafka カスタムリソースの例

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  channel:
    enabled: true (1)
    bootstrapServers: <bootstrap_servers> (2)
  source:
    enabled: true (3)
  broker:
    enabled: true (4)
    defaultConfig:
      bootstrapServers: <bootstrap_servers> (5)
      numPartitions: <num_partitions> (6)
      replicationFactor: <replication_factor> (7)
  sink:
    enabled: true (8)
```

1. 開発者はクラスターで **KafkaChannel** チャネルを使用できます。
2. AMQ Streams クラスターからのブートストラップサーバーのコンマ区切りの一覧。
3. 開発者はクラスターで **KafkaSource** イベントソースタイプを使用できます。
4. 開発者はクラスターで Knative Kafka ブローカー実装を使用できます。
5. Red Hat AMQ Streams クラスターからのブートストラップサーバーのコンマ区切りリスト。
6. **Broker** オブジェクトでサポートされる Kafka トピックのパーティション数を定義します。デフォルトは **10** です。
7. **Broker** オブジェクトでサポートされる Kafka トピックのレプリケーション係数を定義します。デフォルトは **3** です。
8. 開発者がクラスター内で Kafka シンクを使用できるようにします。



注記

replicationFactor の値は、Red Hat AMQ Streams クラスターのノード数以下である必要があります。

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

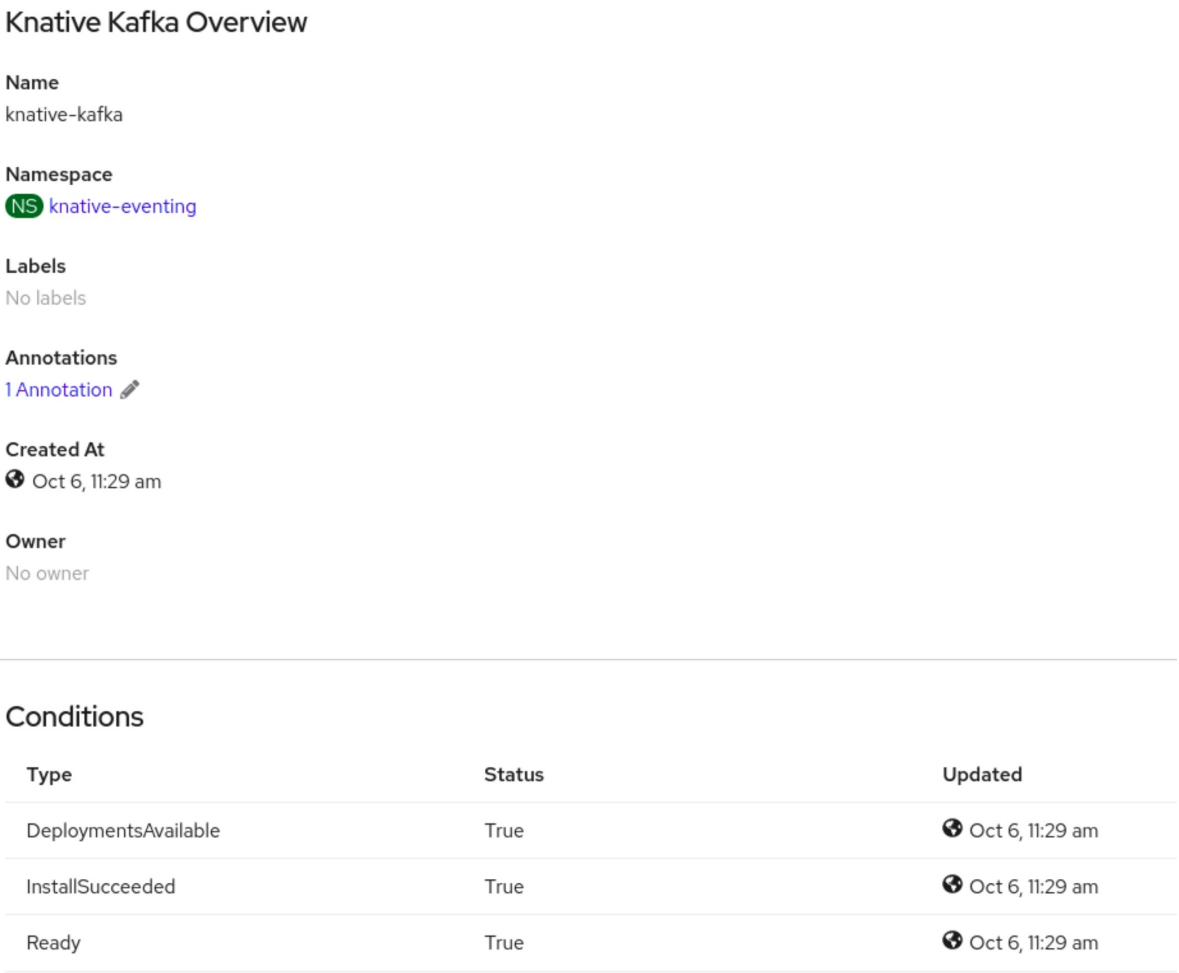
b. **KnativeKafka** オブジェクトの作成を完全に制御する必要のあるより複雑な設定には、YAML の編集が推奨されます。YAML にアクセスするには、**Create Knative Kafka** ページの右上にある **Edit YAML** リンクをクリックします。

5. Kafka のオプションの設定が完了したら、**Create** をクリックします。**Knative Kafka** タブに自動的にダイレクトされます。ここで、**knative-kafka** はリソースの一覧にあります。

検証

1. **Knative Kafka** タブで **knative-kafka** リソースをクリックします。**Knative Kafka Overview** ページに自動的にダイレクトされます。

2. リソースの **Conditions** (状態) の一覧を表示し、それらのステータスが **True** であることを確認します。



状態のステータスが **Unknown** または **False** である場合は、ページを更新するためにしばらく待機します。

3. 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
```

出力例

```
NAME                                      READY   STATUS    RESTARTS   AGE
kafka-broker-dispatcher-7769fbbcbb-xgffn    2/2     Running   0          44s
kafka-broker-receiver-5fb56f7656-fhq8d      2/2     Running   0          44s
kafka-channel-dispatcher-84fd6cb7f9-k2tjv   2/2     Running   0          44s
kafka-channel-receiver-9b7f795d5-c76xr      2/2     Running   0          44s
kafka-controller-6f95659bf6-trd6r           2/2     Running   0          44s
kafka-source-dispatcher-6bf98bdfff-8bcsn    2/2     Running   0          44s
kafka-webhook-eventing-68dc95d54b-825xs     2/2     Running   0          44s
```

6.2.2. Knative Kafka のセキュリティー設定

Kafka クラスターは、通常、TLS または SASL 認証方法を使用して保護されます。TLS または SASL を使用して、保護された Red Hat AMQ Streams クラスターに対して動作するように Kafka ブローカーまたはチャネルを設定できます。

注記

Red Hat は、SASL と TLS の両方を一緒に有効にすることをお勧めします。

6.2.2.1. Kafka ブローカーの TLS 認証の設定

Transport Layer Security (TLS) は、Apache Kafka クライアントおよびサーバーによって、Knative と Kafka 間のトラフィックを暗号化するため、および認証のために使用されます。TLS は、Knative Kafka のトラフィック暗号化でサポートされている唯一の方法です。

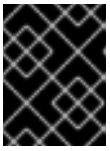
前提条件

- OpenShift Container Platform のクラスター管理者パーミッションがある。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** CR は OpenShift Container Platform クラスターにインストールされます。
- 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。
- **.pem** ファイルとして Kafka クラスター CA 証明書が保存されている。
- Kafka クラスタークライアント証明書とキーが **.pem** ファイルとして保存されている。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 証明書ファイルを **knative-eventing** namespace にシークレットファイルとして作成します。

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```



重要

キー名に **ca.crt**、 **user.crt**、および **user.key** を使用します。これらの値は変更しないでください。

2. **KnativeKafka** CR を編集し、 **broker** 仕様にシークレットへの参照を追加します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
    defaultConfig:
      authSecretName: <secret_name>
...
```

6.2.2.2. Kafka ブローカーの SASL 認証の設定

Simple Authentication and Security Layer(SASL) は、Apache Kafka が認証に使用します。クラスターで SASL 認証を使用する場合、ユーザーは Kafka クラスターと通信するために Knative に認証情報を提供する必要があります。そうしないと、イベントを生成または消費できません。

前提条件

- OpenShift Container Platform のクラスター管理者パーミッションがある。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** CR は OpenShift Container Platform クラスターにインストールされます。
- 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。
- Kafka クラスターのユーザー名およびパスワードがある。
- 使用する SASL メカニズムを選択している (例: **PLAIN**、**SCRAM-SHA-256**、または **SCRAM-SHA-512**)。
- TLS が有効にされている場合、Kafka クラスターの **ca.crt** 証明書ファイルも必要になります。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 証明書ファイルを **knative-eventing** namespace にシークレットファイルとして作成します。

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SASL_SSL \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=user="my-sasl-user"
```

- キー名に **ca.crt**、 **password**、および **sasl.mechanism** を使用します。これらの値は変更しないでください。
- パブリック CA 証明書で SASL を使用する場合は、シークレットの作成時に **ca.crt** 引数ではなく **tls.enabled=true** フラグを使用する必要があります。以下に例を示します。

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-literal=tls.enabled=true \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

2. **KnativeKafka** CR を編集し、 **broker** 仕様にシークレットへの参照を追加します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
    defaultConfig:
      authSecretName: <secret_name>
...
```

6.2.2.3. Kafka チャネルの TLS 認証の設定

Transport Layer Security (TLS) は、Apache Kafka クライアントおよびサーバーによって、Knative と Kafka 間のトラフィックを暗号化するため、および認証のために使用されます。TLS は、Knative Kafka のトラフィック暗号化でサポートされている唯一の方法です。

前提条件

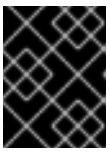
- OpenShift Container Platform のクラスター管理者パーミッションがある。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** CR は OpenShift Container Platform クラスターにインストールされます。
- OpenShift Container Platform でアプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションを持つプロジェクトにアクセスできる。
- **.pem** ファイルとして Kafka クラスター CA 証明書が保存されている。
- Kafka クラスタークライアント証明書とキーが **.pem** ファイルとして保存されている。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 選択された namespace にシークレットとして証明書ファイルを作成します。

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
```

```
--from-file=user.crt=certificate.pem \
--from-file=user.key=key.pem
```



重要

キー名に **ca.crt**、 **user.crt**、および **user.key** を使用します。これらの値は変更しないでください。

2. **KnativeKafka** カスタムリソースの編集を開始します。

```
$ oc edit knativekafka
```

3. シークレットおよびシークレットの namespace を参照します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
  source:
    enabled: true
```

注記

ブートストラップサーバーで一致するポートを指定するようにしてください。

以下に例を示します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: tls-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9094
    enabled: true
  source:
    enabled: true
```

6.2.2.4. Kafka チャネルの SASL 認証の設定

Simple Authentication and Security Layer(SASL) は、Apache Kafka が認証に使用します。クラスターで SASL 認証を使用する場合、ユーザーは Kafka クラスターと通信するために Knative に認証情報を提供する必要があります。そうしないと、イベントを生成または消費できません。

前提条件

- OpenShift Container Platform のクラスター管理者パーミッションがある。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** CR は OpenShift Container Platform クラスターにインストールされます。
- 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。
- Kafka クラスターのユーザー名およびパスワードがある。
- 使用する SASL メカニズムを選択している (例: **PLAIN**、**SCRAM-SHA-256**、または **SCRAM-SHA-512**)。
- TLS が有効にされている場合、Kafka クラスターの **ca.crt** 証明書ファイルも必要になります。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 選択された namespace にシークレットとして証明書ファイルを作成します。

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

   - キー名に **ca.crt**、 **password**、および **sasl.mechanism** を使用します。これらの値は変更しないでください。
   - パブリック CA 証明書で SASL を使用する場合は、シークレットの作成時に **ca.crt** 引数ではなく **tls.enabled=true** フラグを使用する必要があります。以下に例を示します。

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-literal=tls.enabled=true \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

2. **KnativeKafka** カスタムリソースの編集を開始します。

```
$ oc edit knativekafka
```

3. シークレットおよびシークレットの namespace を参照します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
```

```
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
  source:
    enabled: true
```

注記

ブートストラップサーバーで一致するポートを指定するようにしてください。

以下に例を示します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: scram-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9093
    enabled: true
  source:
    enabled: true
```

6.2.2.5. Kafka ソースの SASL 認証の設定

Simple Authentication and Security Layer(SASL) は、Apache Kafka が認証に使用します。クラスターで SASL 認証を使用する場合、ユーザーは Kafka クラスターと通信するために Knative に認証情報を提供する必要があります。そうしないと、イベントを生成または消費できません。

前提条件

- OpenShift Container Platform でクラスターまたは専用の管理者パーミッションを持っている。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** CR は、OpenShift Container Platform クラスターにインストールされている。
- 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。
- Kafka クラスターのユーザー名およびパスワードがある。
- 使用する SASL メカニズムを選択している (例: **PLAIN**、**SCRAM-SHA-256**、または **SCRAM-SHA-512**)。

- TLS が有効にされている場合、Kafka クラスターの **ca.crt** 証明書ファイルも必要になります。
- OpenShift (**oc**) CLI がインストールされている。

手順

1. 選択された namespace にシークレットとして証明書ファイルを作成します。

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \ 1
  --from-literal=user="my-sasl-user"
```

1 SASL タイプは **PLAIN**、**SCRAM-SHA-256**、または **SCRAM-SHA-512** です。

2. Kafka ソースを作成または変更して、次の **spec** 設定が含まれるようにします。

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: example-source
spec:
...
  net:
    sasl:
      enable: true
      user:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: user
      password:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: password
      type:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: saslType
    tls:
      enable: true
      caCert: 1
        secretKeyRef:
          name: <kafka_auth_secret>
          key: ca.crt
...
```

1 Red Hat OpenShift Streams for Apache Kafka などのパブリッククラウド Kafka サービスを使用している場合は、**caCert** 仕様は必要ありません。

6.2.2.6. Kafka シンクのセキュリティーの設定

Transport Layer Security(TLS) は、Apache Kafka クライアントおよびサーバーによって、Knative と Kafka 間のトラフィックを暗号化するため、および認証のために使用されます。TLS は、Knative Kafka のトラフィック暗号化でサポートされている唯一の方法です。

Simple Authentication and Security Layer(SASL) は、Apache Kafka が認証に使用します。クラスターで SASL 認証を使用する場合、ユーザーは Kafka クラスターと通信するために Knative に認証情報を提供する必要があります。そうしないと、イベントを生成または消費できません。

前提条件

- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** カスタムリソース (CR) は OpenShift Container Platform クラスターにインストールされます。
- Kafka シンクは **KnativeKafka** CR で有効になっています。
- 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。
- **.pem** ファイルとして Kafka クラスター CA 証明書が保存されている。
- Kafka クラスタークライアント証明書とキーが **.pem** ファイルとして保存されている。
- OpenShift (**oc**) CLI がインストールされている。
- 使用する SASL メカニズムを選択している (例: **PLAIN**、**SCRAM-SHA-256**、または **SCRAM-SHA-512**)。

手順

1. **KafkaSink** オブジェクトと同じ namespace に証明書ファイルをシークレットとして作成します。

重要

証明書とキーは PEM 形式である必要があります。

- 暗号化なしで SASL を使用した認証の場合:

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SASL_PLAINTEXT \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-literal=user=<username> \
  --from-literal=password=<password>
```

- SASL を使用した認証と TLS を使用した暗号化の場合:

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SASL_SSL \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-file=ca.crt=<my_caroot.pem_file_path> \ 1
  --from-literal=user=<username> \
  --from-literal=password=<password>
```

1 Red Hat OpenShift Streams for Apache Kafka などのパブリッククラウドで管理される Kafka サービスを使用している場合は、システムのルート CA セットを使用するために **ca.crt** を省略できます。

- TLS を使用した認証と暗号化の場合:

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=<my_caroot.pem_file_path> \ 1
  --from-file=user.crt=<my_cert.pem_file_path> \
  --from-file=user.key=<my_key.pem_file_path>
```

1 Red Hat OpenShift Streams for Apache Kafka などのパブリッククラウドで管理される Kafka サービスを使用している場合は、システムのルート CA セットを使用するために **ca.crt** を省略できます。

2. **KafkaSink** オブジェクトを作成または変更し、**auth** 仕様にシークレットへの参照を追加します。

```
apiVersion: eventing.knative.dev/v1alpha1
kind: KafkaSink
metadata:
  name: <sink_name>
  namespace: <namespace>
spec:
...
  auth:
    secret:
      ref:
        name: <secret_name>
...
```

3. **KafkaSink** オブジェクトを適用します。

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

6.2.3. Kafka ブローカー構成の設定

設定マップを作成し、Kafka **Broker** オブジェクトでこの ConfigMap を参照することで、レプリケーション係数、ブートストラップサーバー、および Kafka ブローカーのトピックパーティションの数を設定できます。

前提条件

- OpenShift Container Platform でクラスターまたは専用の管理者パーミッションを持っている。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** カスタムリソース (CR) は OpenShift Container Platform クラスターにインストールされます。
- 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。

- OpenShift CLI (**oc**) がインストールされている。

手順

1. **kafka-broker-config** ConfigMap を変更するか、以下の設定が含まれる独自の ConfigMap を作成します。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: <config_map_name> 1
  namespace: <namespace> 2
data:
  default.topic.partitions: <integer> 3
  default.topic.replication.factor: <integer> 4
  bootstrap.servers: <list_of_servers> 5
```

1. ConfigMap 名。
2. ConfigMap が存在する namespace。
3. Kafka ブローカーのトピックパーティションの数。これは、イベントをブローカーに送信する速度を制御します。パーティションが多い場合には、コンピュートリソースが多く必要です。
4. トピックメッセージのレプリケーション係数。これにより、データ損失を防ぐことができます。レプリケーション係数を増やすには、より多くのコンピュートリソースとストレージが必要になります。
5. ブートストラップサーバーのコンマ区切りリスト。これは、OpenShift Container Platform クラスターの内部または外部にある可能性があり、ブローカーがイベントを受信してイベントを送信する Kafka クラスターのリストです。

重要

default.topic.replication.factor の値は、クラスター内の Kafka ブローカーインスタンスの数以下である必要があります。たとえば、Kafka ブローカーが 1 つしかない場合には、**default.topic.replication.factor** の値は **"1"** を超える値にすることはできません。

Kafka ブローカーの ConfigMap の例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kafka-broker-config
  namespace: knative-eventing
data:
  default.topic.partitions: "10"
  default.topic.replication.factor: "3"
  bootstrap.servers: "my-cluster-kafka-bootstrap.kafka:9092"
```

2. ConfigMap を適用します。

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

3. Kafka **Broker** オブジェクトの ConfigMap を指定します。

Broker オブジェクトの例

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: <broker_name> 1
  namespace: <namespace> 2
  annotations:
    eventing.knative.dev/broker.class: Kafka 3
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: <config_map_name> 4
    namespace: <namespace> 5
...
```

1. ブローカー名。
2. ブローカーが存在する namespace。
3. ブローカークラスアノテーション。この例では、ブローカーはクラス値 Kafka を使用する **Kafka** ブローカーです。
4. ConfigMap 名。
5. ConfigMap が存在する namespace。

4. ブローカーを適用します。

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

関連情報

- ブローカーの作成

6.2.4. 関連情報

- Red Hat AMQ Streams のドキュメント
- Kafka での TLS および SASL

6.3. 管理者の観点から見たサーバーレスコンポーネント

OpenShift Container Platform Web コンソールで **Developer** パースペクティブに切り替えたくない場合、または Knative (**kn**) CLI または YAML ファイルを使用したくない場合は、OpenShift Container PlatformWeb コンソールの **Administator** パースペクティブを使用して Knative コンポーネントを作成できます。

6.3.1. 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

サーバーレスアプリケーションは、ルートと設定で定義され、YAML ファイルに含まれる Kubernetes サービスとして作成およびデプロイされます。OpenShift Serverless を使用してサーバーレスアプリケーションをデプロイするには、Knative **Service** オブジェクトを作成する必要があります。

Knative Service オブジェクトの YAML ファイルの例

```
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

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、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。

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

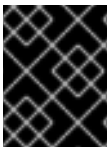
1. **Serverless → Serving** ページに移動します。
2. **Create** 一覧で、**Service** を選択します。
3. YAML または JSON 定義を手動で入力するか、またはファイルをエディターにドラッグし、ドロップします。
4. **Create** をクリックします。

6.3.2. 関連情報

- Serverless アプリケーション

6.4. サービスメッシュと OPENSHIFT SERVERLESS の統合

OpenShift Serverless Operator は、Knative のデフォルト Ingress として Kourier を提供します。ただし、Kourier が有効であるかどうかにかかわらず、OpenShift Serverless でサービスメッシュを使用できます。Kourier を無効にして統合すると、mTLS 機能など、Kourier イングレスがサポートし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。



重要

OpenShift Serverless は、本書で明示的に文書化されている Red Hat OpenShift Service Mesh 機能の使用のみをサポートし、文書化されていない他の機能はサポートしません。

6.4.1. 前提条件

- 以下の手順の例では、ドメイン **example.com** を使用しています。このドメインの証明書のサンプルは、サブドメイン証明書に署名する認証局 (CA) として使用されます。
  お使いのデプロイメントでこの手順を完了し、検証するには、一般に信頼されているパブリック CA によって署名された証明書、または組織が提供する CA のいずれかが必要です。コマンドの例は、ドメイン、サブドメイン、および CA に合わせて調整する必要があります。

- ワイルドカード証明書を OpenShift Container Platform クラスターのドメインに一致するように設定する必要があります。たとえば、OpenShift Container Platform コンソールアドレスが **https://console-openshift-console.apps.openshift.example.com** の場合、ドメインが ***.apps.openshift.example.com** になるようにワイルドカード証明書を設定する必要があります。ワイルドカード証明書の設定に関する詳細は、**着信外部トラフィックを暗号化する証明書の作成**のトピックを参照してください。

- デフォルトの OpenShift Container Platform クラスタードメインのサブドメインではないものを含むドメイン名を使用する必要がある場合は、これらのドメインのドメインマッピングを設定する必要があります。詳細は、OpenShift Serverless ドキュメントのカスタムドメインマッピングの作成を参照してください。

6.4.2. 着信外部トラフィックを暗号化する証明書の作成

デフォルトでは、サービスメッシュ mTLS 機能は、Ingress ゲートウェイとサイドカーを持つ個々の Pod 間で、サービスメッシュ自体内のトラフィックのみを保護します。OpenShift Container Platform クラスターに流入するトラフィックを暗号化するには、OpenShift Serverless とサービスメッシュの統合を有効にする前に証明書を生成する必要があります。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。

- OpenShift Serverless Operator および Knative Serving がインストールされていること。

- OpenShift CLI (**oc**) をインストールしている。

- 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。

手順

1. Knative サービスの証明書に署名する root 証明書と秘密鍵を作成します。

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
    -subj '/O=Example Inc./CN=example.com' \
    -keyout root.key \
    -out root.crt
```

2. ワイルドカード証明書を作成します。

```
$ openssl req -nodes -newkey rsa:2048 \
    -subj "/CN=*.apps.openshift.example.com/O=Example Inc." \
    -keyout wildcard.key \
    -out wildcard.csr
```

3. ワイルドカード証明書を署名します。

```
$ openssl x509 -req -days 365 -set_serial 0 \
    -CA root.crt \
    -CAkey root.key \
    -in wildcard.csr \
    -out wildcard.crt
```

4. ワイルドカード証明書を使用してシークレットを作成します。

```
$ oc create -n istio-system secret tls wildcard-certs \
    --key=wildcard.key \
    --cert=wildcard.crt
```

この証明書は、OpenShift Serverless をサービスメッシュと統合する際に作成されるゲートウェイによって取得され、Ingress ゲートウェイはこの証明書でトラフィックを提供します。

6.4.3. サービスメッシュと OpenShift Serverless の統合

Kourier をデフォルトのイングレスとして使用せずに、Service Mesh を OpenShift Serverless と統合できます。このため、以下の手順を完了する前に、Knative Serving コンポーネントをインストールしないでください。Knative Serving をサービスメッシュと統合するために **KnativeServing** カスタムリソース定義 (CRD) を作成する際に必要な追加の手順があります。これは、一般的な Knative Serving のインストール手順では説明されていません。この手順は、サービスメッシュをデフォルトとして統合し、OpenShift Serverless インストールの唯一のイングレスとして統合する場合に役立ちます。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- 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。
- Red Hat OpenShift Service Mesh Operator をインストールし、**istio-system** namespace に **ServiceMeshControlPlane** リソースを作成します。mTLS 機能を使用する場合は、**ServiceMeshControlPlane** リソースの **spec.security.dataPlane.mtls** フィールドも **true** に設定する必要があります。

重要

Service Mesh での OpenShift Serverless の使用は、Red Hat OpenShift Service Mesh バージョン 2.0.5 以降でのみサポートされます。

- OpenShift Serverless Operator をインストールします。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. サービスメッシュと統合する必要のある namespace をメンバーとして **ServiceMeshMemberRoll** オブジェクトに追加します。

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members: 1
    - knative-serving
    - <namespace>
```

1 サービスメッシュと統合する namespace の一覧。

重要

この namespace の一覧には、**knative-serving** namespace が含まれる必要があります。

2. **ServiceMeshMemberRoll** リソースを適用します。

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

3. サービスメッシュがトラフィックを受け入れることができるように、必要なゲートウェイを作成します。

HTTP を使用した knative-local-gateway オブジェクトの例

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-ingress-gateway
  namespace: knative-serving
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
```

```
      hosts:
        - "*"
      tls:
        mode: SIMPLE
        credentialName: <wildcard_certs> 1
---
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
 name: knative-local-gateway
 namespace: knative-serving
spec:
 selector:
   istio: ingressgateway
 servers:
   - port:
       number: 8081
       name: http
       protocol: HTTP 2
     hosts:
       - "*"
---
apiVersion: v1
kind: Service
metadata:
 name: knative-local-gateway
 namespace: istio-system
 labels:
   experimental.istio.io/disable-gateway-port-translation: "true"
spec:
 type: ClusterIP
 selector:
   istio: ingressgateway
 ports:
   - name: http2
     port: 80
     targetPort: 8081
```

1 ワイルドカード証明書を含むシークレットの名前を追加します。

2 **knative-local-gateway** は HTTP トラフィックに対応します。HTTP を使用するということは、サービスメッシュの外部から来るが、**example.default.svc.cluster.local** などの内部ホスト名を使用するトラフィックは、暗号化されていないことを意味します。別のワイルドカード証明書と、異なる **protocol** 仕様を使用する追加のゲートウェイを作成することで、このパスの暗号化を設定できます。

HTTPS を使用した knative-local-gateway オブジェクトの例

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-local-gateway
  namespace: knative-serving
spec:
```

```
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
      hosts:
        - "*"
      tls:
        mode: SIMPLE
        credentialName: <wildcard_certs>
```

4. **Gateway** リソースを適用します。

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

5. 以下の **KnativeServing** カスタムリソース定義 (CRD) を作成して Knative Serving をインストールします。これにより、Istio 統合も有効化されます。

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  ingress:
    istio:
      enabled: true 1
  deployments: 2
  - name: activator
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: autoscaler
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
```

1 Istio 統合を有効にします。

2 Knative Serving データプレーン Pod のサイドカーの挿入を有効にします。

6. **KnativeServing** リソースを適用します。

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

7. サイドカー挿入が有効で、パススルールートを使用する Knative サービスを作成します。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
```

```
    namespace: <namespace> (1)
    annotations:
      serving.knative.openshift.io/enablePassthrough: "true" (2)
  spec:
    template:
      metadata:
        annotations:
          sidecar.istio.io/inject: "true" (3)
          sidecar.istio.io/rewriteAppHTTPProbers: "true"
      spec:
        containers:
        - image: <image_url>
```

(1) サービスメッシュメンバーロールの一部である namespace。

(2) OpenShift Container Platform のパススルーが有効化されたルートを生成するよう Knative Serving に指示します。これにより、生成した証明書は Ingress ゲートウェイ経由で直接提供されます。

(3) Service Mesh サイドカーは Knative サービス Pod に挿入します。

8. **Service** リソースを適用します。

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

検証

- CA によって信頼されるようになった安全な接続を使用して、サーバーレスアプリケーションにアクセスします。

```
$ curl --cacert root.crt <service_url>
```

コマンドの例

```
$ curl --cacert root.crt https://hello-default.apps.openshift.example.com
```

出力例

```
Hello Openshift!
```

6.4.4. mTLS で Service Mesh を使用する場合の Knative Serving メトリクスの有効化

サービスメッシュが mTLS で有効にされている場合、サービスメッシュが Prometheus のメトリクスの収集を阻止するため、Knative Serving のメトリクスはデフォルトで無効にされます。このセクションでは、Service Mesh および mTLS を使用する際に Knative Serving メトリクスを有効に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。

- mTLS 機能を有効にして Red Hat OpenShift Service Mesh をインストールしています。
- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

1. **prometheus** を Knative Serving カスタムリソース (CR) の **observability** 仕様で **metrics.backend-destination** として指定します。

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    observability:
      metrics.backend-destination: "prometheus"
...
```

   この手順により、メトリクスがデフォルトで無効になることを防ぎます。

2. 以下のネットワークポリシーを適用して、Prometheus namespace からのトラフィックを許可します。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-monitoring-ns
  namespace: knative-serving
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: "openshift-monitoring"
  podSelector: {}
...
```

3. **istio-system** namespace のデフォルトのサービスメッシュコントロールプレーンを変更して再適用し、以下の仕様が含まれるようにします。

```
...
spec:
  proxy:
    networking:
      trafficControl:
        inbound:
```

```
        excludedPorts:
        - 8444
...
```

6.4.5. Kourier が有効にされている場合のサービスメッシュの OpenShift Serverless との統合

Kourier が既に有効になっている場合でも、OpenShift Serverless で Service Mesh を使用できます。この手順は、Kourier を有効にして Knative Serving を既にインストールしているが、後で Service Mesh 統合を追加することにした場合に役立つ可能性があります。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- 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。
- OpenShift CLI (**oc**) をインストールしている。
- OpenShift Serverless Operator と Knative Serving をクラスターに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 with Service Mesh and Kourier は、Red Hat OpenShift Service Mesh バージョン 1.x および 2.x の両方での使用がサポートされています。

手順

1. サービスメッシュと統合する必要のある namespace をメンバーとして **ServiceMeshMemberRoll** オブジェクトに追加します。

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members:
    - <namespace> 1
...
```

1 サービスメッシュと統合する namespace の一覧。

2. **ServiceMeshMemberRoll** リソースを適用します。

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

3. 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. サービスメッシュと統合する必要のある namespace ごとに、**NetworkPolicy** リソースを作成します。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-serving-system-namespace
  namespace: <namespace> 1
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          knative.openshift.io/part-of: "openshift-serverless"
  podSelector: {}
  policyTypes:
  - Ingress
...
```

1 サービスメッシュと統合する必要のある namespace を追加します。

注記

knative.openshift.io/part-of: "openshift-serverless" ラベルが OpenShift Serverless 1.22.0 で追加されました。OpenShift Serverless 1.21.1 以前を使用している場合は、**knative.openshift.io/part-of** ラベルを **knative-serving** および **knative-serving-ingress** ネームスペースに追加します。

knative-serving namespace にラベルを追加します。

```
$ oc label namespace knative-serving knative.openshift.io/part-of=openshift-serverless
```

knative-serving-ingress namespace にラベルを追加します。

```
$ oc label namespace knative-serving-ingress knative.openshift.io/part-of=openshift-serverless
```

b. **NetworkPolicy** リソースを適用します。

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

6.4.6. Service Mesh のシークレットフィルターリングを使用してメモリー使用量を改善する

デフォルトでは、Kubernetes **client-go** ライブラリーの informers の実装は、特定のタイプのすべてのリソースをフェッチします。これにより、多くのリソースが使用可能な場合にかなりのオーバーヘッドが発生する可能性があり、メモリーリークが原因で大規模なクラスターで Knative **net-istio** イングレスコントローラーが失敗する可能性があります。ただし、Knative **net-istio** イングレスコントローラーではフィルターリングメカニズムを使用できます。これにより、コントローラーは Knative 関連のシークレットのみを取得できます。このメカニズムを有効にするには、**KnativeServing** カスタムリソース (CR) にアノテーションを追加します。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- 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。
- Red Hat OpenShift Service Mesh をインストールします。OpenShift Serverless with Service Mesh は、Red Hat OpenShift Service Mesh バージョン 2.0.5 以降での使用でのみサポートされます。
- OpenShift Serverless Operator および Knative Serving をインストールします。
- OpenShift CLI (**oc**) をインストールしている。

手順

- **serverless.openshift.io/enable-secret-informer-filtering** アノテーションを **KnativeServing** CR に追加します。

  KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
  annotations:
    serverless.openshift.io/enable-secret-informer-filtering: "true" (1)
spec:
  ingress:
    istio:
      enabled: true
  deployments:
    - annotations:
        sidecar.istio.io/inject: "true"
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
      name: activator
    - annotations:
        sidecar.istio.io/inject: "true"
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
      name: autoscaler
```

  (1) このアノテーションを追加すると、環境変数 **ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID=true** が **net-istio** コントローラー Pod に挿入されます。

6.5. サーバーレス管理者のメトリクス

メトリクスにより、クラスター管理者は OpenShift Serverless クラスターコンポーネントおよびワークロードのパフォーマンスを監視できます。

OpenShift Container Platform Web コンソールの **Administrator** パースペクティブで Dashboards に移動すると、OpenShift Serverless のさまざまなメトリクスを表示できます。

6.5.1. 前提条件

- クラスターのメトリクスの有効化に関する詳細は、OpenShift Container Platform ドキュメントの メトリクスの管理 を参照してください。
- OpenShift Container Platform で Knative コンポーネントのメトリクスを表示するには、クラスター管理者権限と、Web コンソール**管理者** パースペクティブへのアクセスが必要です。

警告

サービスメッシュが mTLS で有効にされている場合、サービスメッシュが Prometheus のメトリクスの収集を阻止するため、Knative Serving のメトリクスはデフォルトで無効にされます。

この問題の解決については、Enabling Knative Serving metrics when using Service Mesh with mTLS の有効化を参照してください。

メトリクスの収集は、Knative サービスの自動スケーリングには影響しません。これは、収集要求がアクティベーターを通過しないためです。その結果、Pod が実行していない場合に収集が行われることはありません。

6.5.2. コントローラーメトリクス

以下のメトリクスは、コントローラーロジックを実装するコンポーネントによって出力されます。これらのメトリクスは、調整要求がワークキューに追加される調整操作とワークキューの動作に関する詳細を示します。

メトリクス名	説明	タイプ	タグ	単位
work_queue_depth	ワークキューの深さ。	ゲージ	**reconciler**	整数 (単位なし)
reconcile_count	調整操作の数。	カウンター	**reconciler**、**success**	整数 (単位なし)
reconcile_latency	調整操作のレイテンシー。	ヒストグラム	**reconciler**、**success**	ミリ秒
workqueue_adds_total	ワークキューによって処理される追加アクションの合計数。	カウンター	**name**	整数 (単位なし)
workqueue_queue_latency_seconds	アイテムが要求される前にワークキューにとどまる時間の長さ。	ヒストグラム	**name**	秒

メトリクス名	説明	タイプ	タグ	単位
workqueue_retries_total	ワークキューによって処理された再試行回数。	カウンター	**name**	整数 (単位なし)
workqueue_work_duration_seconds	ワークキューからの項目の処理にかかる時間の長さ。	ヒストグラム	**name**	秒
workqueue_unfinished_work_seconds	未処理のワークキュー項目が進行中であった時間の長さ。	ヒストグラム	**name**	秒
workqueue_longest_running_processor_seconds	最も長い間未処理のワークキュー項目が進行中であった時間の長さ。	ヒストグラム	**name**	秒

6.5.3. Webhook メトリクス

Webhook メトリクスは操作に関する有用な情報を表示します。たとえば、多数の操作が失敗する場合は、これはユーザーが作成したリソースに問題があることを示している可能性があります。

メトリクス名	説明	タイプ	タグ	単位
request_count	Webhook にルーティングされる要求の数。	カウンター	**admission_allowed**、**kind_group**、**kind_kind**、**kind_version**、**request_operation**、**resource_group**、**resource_namespace**、**resource_resource**、**resource_version**	整数 (単位なし)
request_latencies	Webhook 要求の応答時間。	ヒストグラム	**admission_allowed**、**kind_group**、**kind_kind**、**kind_version**、**request_operation**、**resource_group**、**resource_namespace**、**resource_resource**、**resource_version**	ミリ秒

6.5.4. Knative Eventing メトリクス

クラスター管理者は、Knative Eventing コンポーネントの以下のメトリクスを表示できます。

HTTP コードからメトリクスを集計することで、イベントは正常なイベント (2xx) および失敗したイベント (5xx) の 2 つのカテゴリーに分類できます。

6.5.4.1. ブローカー Ingress メトリクス

以下のメトリクスを使用してブローカー Ingress をデバッグし、どのように実行されているかを確認し、どのイベントが Ingress コンポーネントによってディスパッチされているかを確認できます。

メトリクス名	説明	タイプ	タグ	単位
event_count	ブローカーによって受信されるイベントの数。	カウンター	**broker_name**、**event_type**、**namespace_name**、**response_code**、**response_code_class**、**unique_name**	整数 (単位なし)
event_dispatch_latencies	イベントのチャネルへのディスパッチにかかる時間。	ヒストグラム	**broker_name**、**event_type**、**namespace_name**、**response_code**、**response_code_class**、**unique_name**	ミリ秒

6.5.4.2. ブローカーフィルターメトリクス

以下のメトリクスを使用してブローカーフィルターをデバッグし、それらがどのように実行されているかを確認し、どのイベントがフィルターによってディスパッチされているかを確認できます。イベントでフィルターリングアクションのレイテンシーを測定することもできます。

メトリクス名	説明	タイプ	タグ	単位
event_count	ブローカーによって受信されるイベントの数。	カウンター	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	整数 (単位なし)

メトリクス名	説明	タイプ	タグ	単位
event_dispatch_latencies	イベントのチャネルへのディスパッチにかかる時間。	ヒストグラム	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	ミリ秒
event_processing_latencies	トリガーサブスクライバーにディスパッチされる前にイベントの処理にかかる時間。	ヒストグラム	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**trigger_name**、**unique_name**	ミリ秒

6.5.4.3. InMemoryChannel dispatcher メトリクス

以下のメトリクスを使用して **InMemoryChannel** チャネルをデバッグし、それらがどのように実行されているかを確認し、どのイベントがチャネルによってディスパッチされているかを確認できます。

メトリクス名	説明	タイプ	タグ	単位
event_count	**InMemoryChannel** チャネルでディスパッチされるイベントの数。	カウンター	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	整数 (単位なし)
event_dispatch_latencies	**InMemoryChannel** チャネルからのイベントのディスパッチにかかる時間。	ヒストグラム	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	ミリ秒

6.5.4.4. イベントソースメトリクス

以下のメトリクスを使用して、イベントがイベントソースから接続されたイベントシンクに配信されていることを確認できます。

メトリクス名	説明	タイプ	タグ	単位
event_count	イベントソースによって送信されるイベントの数。	カウンター	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	整数 (単位なし)
retry_event_count	最初に配信に失敗した後にイベントソースによって送信される再試行イベントの数。	カウンター	**event_source**、**event_type**、**name**、**namespace_name**、**resource_group**、**response_code**、**response_code_class**、**response_error**、**response_timeout**	整数 (単位なし)

6.5.5. Knative Serving メトリクス

クラスター管理者は、Knative Serving コンポーネントの以下のメトリクスを表示できます。

6.5.5.1. activator メトリクス

以下のメトリクスを使用して、トラフィックが activator 経由で渡されるときにアプリケーションがどのように応答するかを理解することができます。

メトリクス名	説明	タイプ	タグ	単位
request_concurrency	activator にルーティングされる同時要求の数、またはレポート期間における平均同時実行数。	ゲージ	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**revision_name**、**service_name**	整数 (単位なし)

メトリクス名	説明	タイプ	タグ	単位
request_count	activator にルーティングされる要求の数。これらは、activator ハンドラーから実行された要求です。	カウンター	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	整数 (単位なし)
request_latencies	実行され、ルーティングされた要求の応答時間 (ミリ秒単位)。	ヒストグラム	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	ミリ秒

6.5.5.2. Autoscaler メトリクス

Autoscaler コンポーネントは、それぞれのリビジョンの Autoscaler の動作に関連する多数のメトリクスを公開します。たとえば、任意の時点で、Autoscaler がサービスに割り当てようとする Pod のターゲット数、安定期間中の 1 秒あたりの要求の平均数、または Knative Pod Autoscaler (KPA) を使用している場合に Autoscaler がパニックモードであるかどうかなどを監視できます。

メトリクス名	説明	タイプ	タグ	単位
desired_pods	Autoscaler がサービスへの割り当てを試みる Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
excess_burst_capacity	stable ウインドウで提供される追加のバースト容量。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
stable_request_concurrency	stable ウィンドウで監視される各 Pod の要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)

メトリクス名	説明	タイプ	タグ	単位
panic_request_concurrency	panic ウィンドウで監視される各 Pod の要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
target_concurrency_per_pod	Autoscaler が各 Pod への送信を試みる同時要求の数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
stable_requests_per_second	stable ウィンドウで監視される各 Pod の 1 秒当たりの要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
panic_requests_per_second	panic ウィンドウで監視される各 Pod の 1 秒当たりの要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
target_requests_per_second	Autoscaler が各 Pod をターゲットとする 1 秒あたりの要求の数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
panic_mode	この値は、Autoscaler がパニックモードの場合は **1** になります。Autoscaler がパニックモードではない場合は **0** になります。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
requested_pods	Autoscaler が Kubernetes クラスターから要求した Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
actual_pods	割り当てられ、現在準備完了状態にある Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)

メトリクス名	説明	タイプ	タグ	単位
not_ready_pods	準備未完了状態の Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
pending_pods	現在保留中の Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
terminating_pods	現在終了中の Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)

6.5.5.3. Go ランタイムメトリクス

各 Knative Serving コントロールプレーンプロセスは、Go ランタイムメモリーの統計を多数出力します (MemStats) 。

注記

各メトリクスの **name** タグは空のタグです。

メトリクス名	説明	タイプ	タグ	単位
go_alloc	割り当てられたヒープオブジェクトのバイト数。このメトリクスは **heap_alloc** と同じです。	ゲージ	**name**	整数 (単位なし)
go_total_alloc	ヒープオブジェクトに割り当てられる累積バイト数。	ゲージ	**name**	整数 (単位なし)
go_sys	オペレーティングシステムから取得したメモリーの合計バイト数。	ゲージ	**name**	整数 (単位なし)

メトリクス名	説明	タイプ	タグ	単位
go_lookups	ランタイムが実行したポインター検索の数。	ゲージ	**name**	整数 (単位なし)
go_mallocs	割り当てられるヒープオブジェクトの累積数。	ゲージ	**name**	整数 (単位なし)
go_frees	解放されているヒープオブジェクトの累積数。	ゲージ	**name**	整数 (単位なし)
go_heap_alloc	割り当てられたヒープオブジェクトのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_sys	オペレーティングシステムから取得したヒープメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_idle	アイドル状態の未使用スパンのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_in_use	現在使用中のスパンのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_released	オペレーティングシステムに返された物理メモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_objects	割り当てられるヒープオブジェクトの数。	ゲージ	**name**	整数 (単位なし)
go_stack_in_use	現在使用中のスタックスパンのバイト数。	ゲージ	**name**	整数 (単位なし)
go_stack_sys	オペレーティングシステムから取得したスタックメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)

メトリクス名	説明	タイプ	タグ	単位
go_mspan_in_use	割り当てられた **mspan** 構造のバイト数。	ゲージ	**name**	整数 (単位なし)
go_mspan_sys	**mspan** 構造のオペレーティングシステムから取得したメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_mcache_in_use	割り当てられた **mcache** 構造のバイト数。	ゲージ	**name**	整数 (単位なし)
go_mcache_sys	**mcache** 構造のためにオペレーティングシステムから取得したメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_bucket_hash_sys	バケットハッシュテーブルのプロファイリングにおけるメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_gc_sys	ガベージコレクションメタデータのメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_other_sys	その他のオフヒープランタイム割り当てのメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_next_gc	次のガベージコレクションサイクルのターゲットヒープサイズ。	ゲージ	**name**	整数 (単位なし)
go_last_gc	最後のガベージコレクションが完了した時間 (Epoch または Unix 時間)。	ゲージ	**name**	ナノ秒

メトリクス名	説明	タイプ	タグ	単位
go_total_gc_pause_ns	プログラム開始以降のガベージコレクションの stop-the-world 停止の累積時間。	ゲージ	**name**	ナノ秒
go_num_gc	完了したガベージコレクションサイクルの数。	ゲージ	**name**	整数 (単位なし)
go_num_forced_gc	ガベージコレクションの機能を呼び出すアプリケーションが原因で強制されたガベージコレクションサイクルの数。	ゲージ	**name**	整数 (単位なし)
go_gc_cpu_fraction	プログラムの開始以降にガベージコレクターによって使用されたプログラムの使用可能な CPU 時間の一部。	ゲージ	**name**	整数 (単位なし)

6.6. OPENSHIFT SERVERLESS でのメータリングの使用

重要

メータリングは非推奨の機能です。非推奨の機能は依然として OpenShift Container Platform に含まれており、引き続きサポートされますが、本製品の今後のリリースで削除されるため、新規デプロイメントでの使用は推奨されません。

OpenShift Container Platform で非推奨となったか、または削除された主な機能の最新の一覧については、OpenShift Container Platform リリースノートの **非推奨および削除された機能**セクションを参照してください。

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。

注記

現時点で、メータリングは IBM Z および IBM Power Systems ではサポートされていません。

6.6.1. 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。

6.6.2. Knative Serving メータリングのデータソースレポート

以下のデータソースレポートは、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。

6.6.2.1. Knative Serving での CPU 使用状況のデータソースレポート

このデータソースレポートは、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!=""}
        )
```

6.6.2.2. Knative Serving でのメモリー使用状況のデータソースレポート

このデータソースレポートは、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!=""}
          )
```

6.6.2.3. Knative Serving メータリングのデータソースレポートの適用

以下のコマンドを使用して、データソースレポートを適用することができます。

```
$ oc apply -f <data_source_report_name>.yaml
```

コマンドの例

```
$ oc apply -f knative-service-memory-usage.yaml
```

6.6.3. 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 **ReportDataSource** リソースを参照します。

Knative Serving での CPU 使用状況のクエリー

```
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']
```

Knative Serving でのメモリー使用状況のクエリー

```
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']
```

6.6.3.1. Knative Serving メータリングのクエリーの適用

1. **ReportQuery** リソースを適用します。

```
$ oc apply -f <query_name>.yaml
```

コマンドの例

```
$ oc apply -f knative-service-memory-usage.yaml
```

6.6.4. 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。

Report リソースの例

```
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。

6.6.4.1. メータリングレポートの実行

1. レポートを実行します。

```
$ oc apply -f <report_name>.yml
```

2. 次に、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
```

6.7. 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または削除された場合、別のコントローラーをすぐに使用できます。このコントローラーは、現在使用できないコントローラーによって処理されていた API の処理を引き継ぎます。

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 または Eventing 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。

6.7.1. Knative Serving の高可用性レプリカの設定

高可用性 (HA) は、デフォルトで Knative Serving **activator**、**autoscaler**、**autoscaler-hpa**、**controller**、**webhook**、**kourier-control**、および **kourier-gateway** コンポーネントで使用できます。これらのコンポーネントは、デフォルトでそれぞれ 2 つのレプリカを持つように設定されています。**KnativeServing** カスタムリソース (CR) の **spec.high-availability.replicas** 値を変更して、これらのコンポーネントのレプリカ数を変更できます。

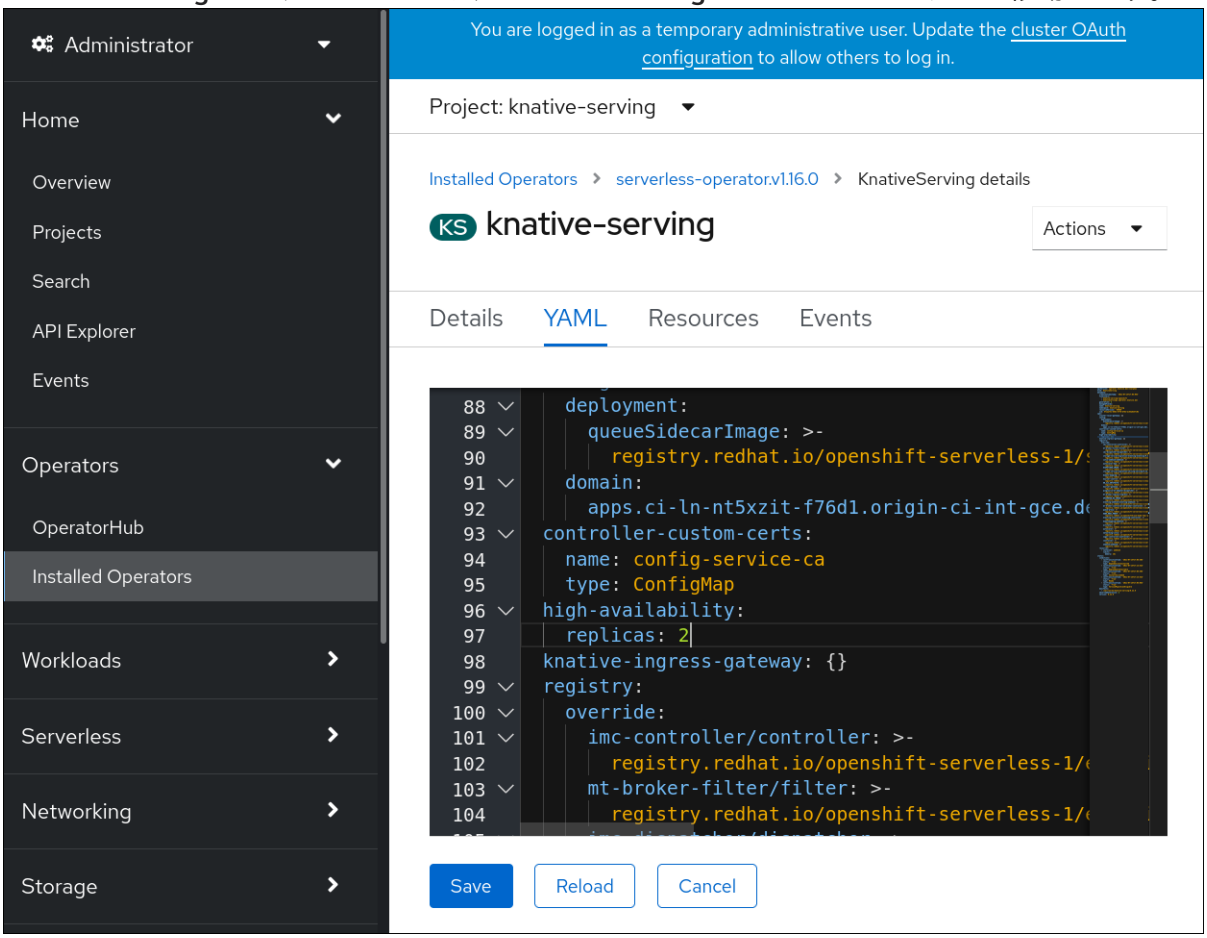
前提条件

- クラスター管理者のパーミッションを持つ OpenShift Container Platform クラスターにアクセスできる。
- 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。

- 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. **KnativeServing** CR のレプリカ数を変更します。

サンプル YAML

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

6.7.2. Knative Eventing の高可用性レプリカの設定

Knative Eventing の **eventing-controller**、**eventing-webhook**、**imc-controller**、**imc-dispatcher**、**mt-broker-controller** コンポーネントは、デフォルトでそれぞれ 2 つのレプリカを持つように設定されており、高可用性 (HA) を利用することができます。**KnativeServing** カスタムリソース (CR) の **spec.high-availability.replicas** 値を変更して、これらのコンポーネントのレプリカ数を変更できます。

注記

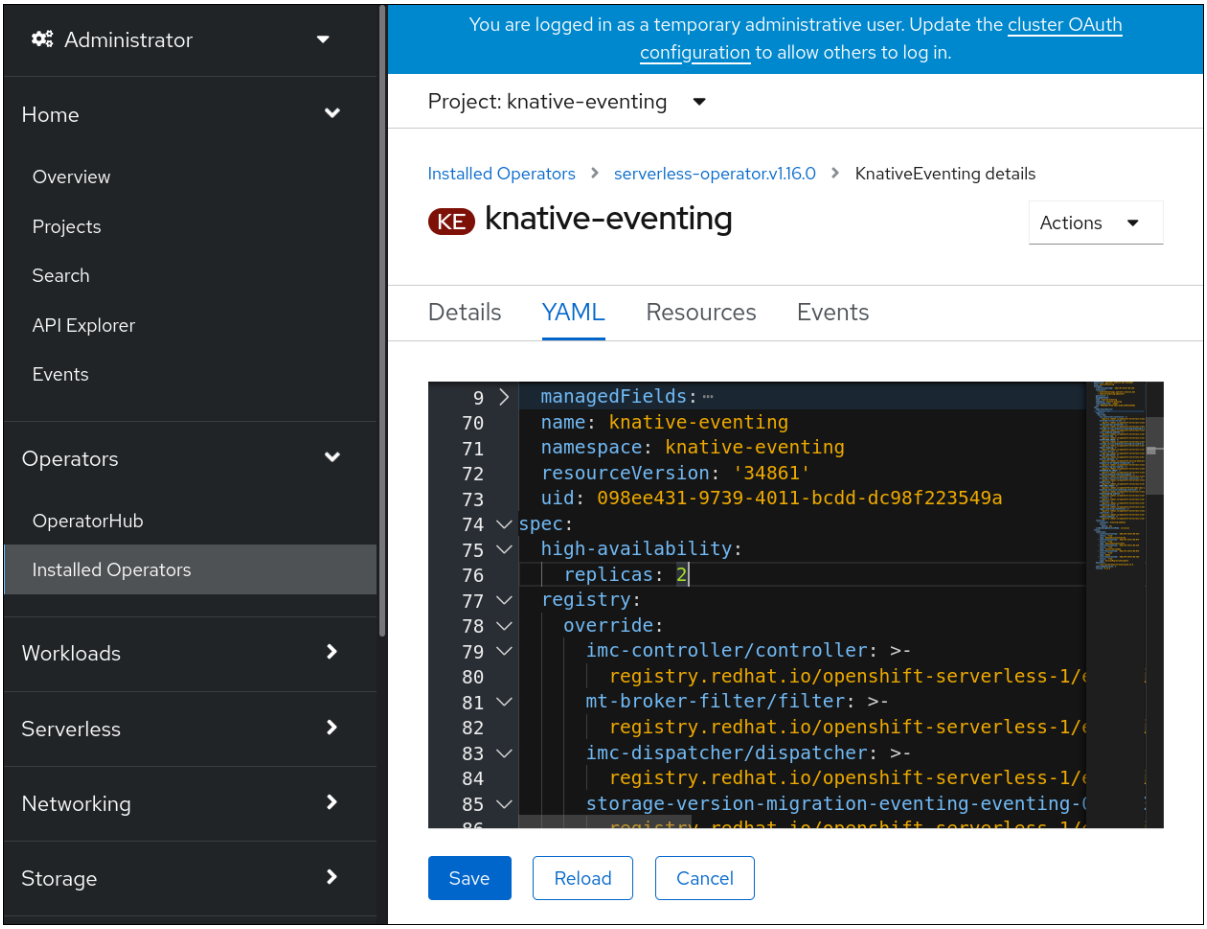
Knative Eventing の場合には、HA では **mt-broker-filter** および **mt-broker-ingress** デプロイメントはスケーリングされません。複数のデプロイメントが必要な場合は、これらのコンポーネントを手動でスケーリングします。

前提条件

- クラスター管理者のパーミッションを持つ OpenShift Container Platform クラスターにアクセスできる。
- OpenShift Serverless Operator および Knative Eventing がクラスターにインストールされている。

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



5. **KnativeEvening** CR のレプリカ数を変更します。

サンプル YAML

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

6.7.3. Knative Kafka の高可用性レプリカの設定

高可用性 (HA) は、デフォルトで Knative Kafka **Kafka-controller** および **kafka-webhook-eventing** コンポーネントで使用できます。これらのコンポーネントは、デフォルトで各レプリカが 2 つあるように設定されています。**KnativeKafka** カスタムリソース (CR) の **spec.high-availability.replicas** 値を変更して、これらのコンポーネントのレプリカ数を変更できます。

前提条件

- クラスター管理者のパーミッションを持つ OpenShift Container Platform クラスターにアクセスできる。
- OpenShift Serverless Operator および Knative Kafka がクラスターにインストールされている。

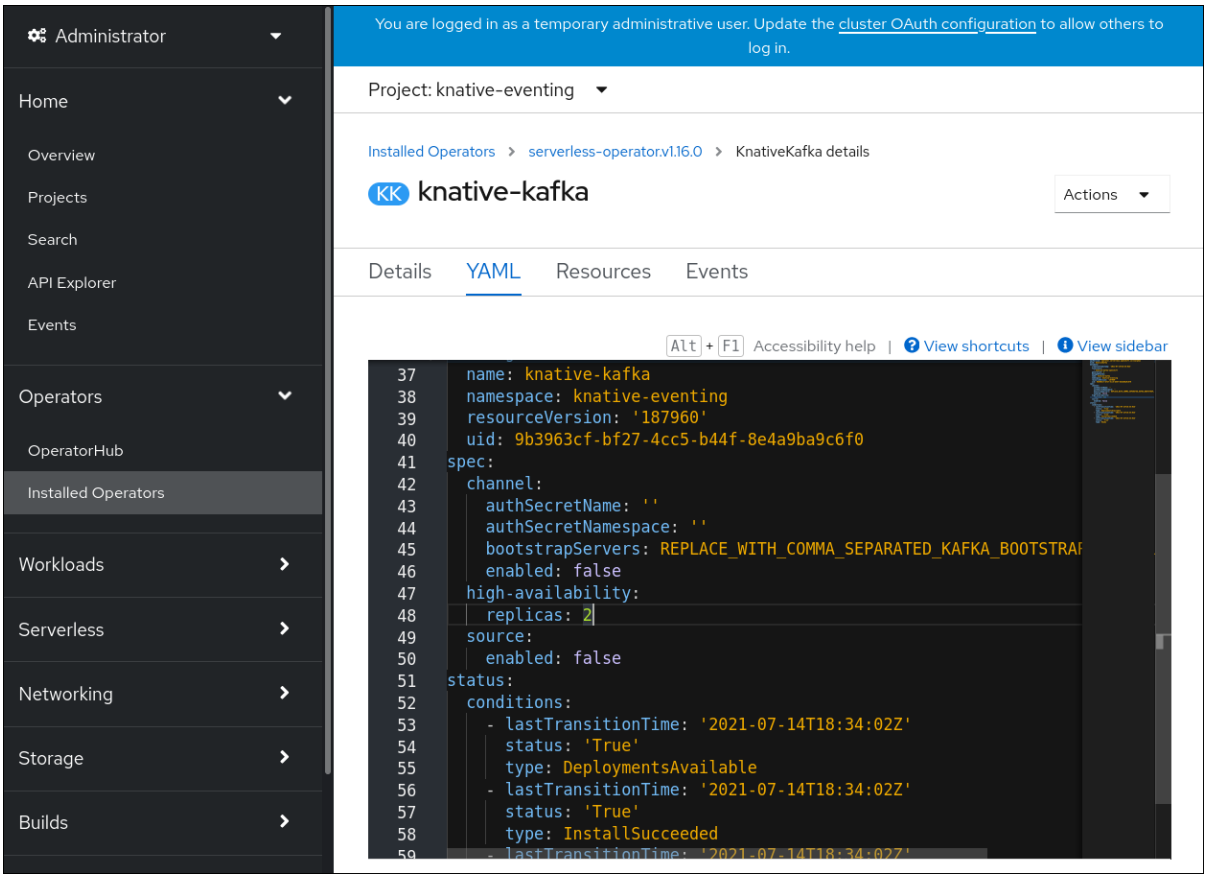
手順

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-eventing** namespace を選択します。

3. OpenShift Serverless Operator の **Provided APIs** の一覧で **Knative Kafka** をクリックし、**Knative Kafka** タブに移動します。

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



5. **KnativeKafka** CR のレプリカ数を変更します。

サンプル YAML

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  high-availability:
    replicas: 3
```

第7章 監視

7.1. OPENSHIFT SERVERLESS での OPENSHIFT LOGGING の使用

7.1.1. 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。

7.1.2. 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。

7.1.2.1. 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:
```

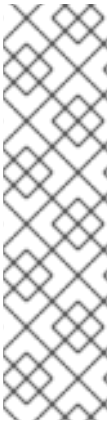
```
        memory: 16Gi
      requests:
        cpu: 500m
        memory: 16Gi
    type: "elasticsearch"
  collection:
    logs:
      fluentd:
        resources:
          limits:
            cpu:
            memory:
          requests:
            cpu:
            memory:
        type: "fluentd"
  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 * * *"
```

7.1.2.2. 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:
        storageClassName: "gp2"
        size: "200G"
    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
```

7.1.3. 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

前提条件

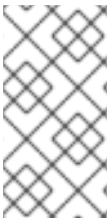
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 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 Container Platform 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します。**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。

7.1.4. 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。

前提条件

- OpenShift CLI (**oc**) をインストールしている。

手順

1. Kibana ルートを取得します。

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

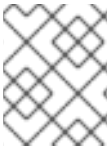
2. ルートの URL を使用して Kibana ダッシュボードに移動し、ログインします。
3. インデックスが **.all** に設定されていることを確認します。インデックスが **.all** に設定されていない場合、OpenShift システムログのみが一覧表示されます。
4. **knative-serving** namespace を使用してログをフィルターします。検索ボックスにサービスのフィルターを入力して、結果をフィルターします。

フィルターの例

```
kubernetes.namespace_name:default AND kubernetes.labels.serving_knative_dev\/service:
{service_name}
```

/**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。

7.2. サーバーレス開発者メトリクス

メトリクスを使用すると、開発者は Knative サービスのパフォーマンスを監視できます。OpenShift Container Platform モニターリングスタックを使用して、Knative サービスのヘルスチェックおよびメトリクスを記録し、表示できます。

OpenShift Container Platform Web コンソール **Developer** パースペクティブの **Dashboards** に移動すると、OpenShift Serverless のさまざまなメトリクスを表示できます。

警告

サービスメッシュが mTLS で有効にされている場合、サービスメッシュが Prometheus のメトリクスの収集を阻止するため、Knative Serving のメトリクスはデフォルトで無効にされます。

この問題の解決については、Enabling Knative Serving metrics when using Service Mesh with mTLS の有効化を参照してください。

メトリクスの収集は、Knative サービスの自動スケーリングには影響しません。これは、収集要求がアクティベーターを通過しないためです。その結果、Pod が実行していない場合に収集が行われることはありません。

7.2.1. デフォルトで公開される Knative サービスメトリクス

表7.1 ポート 9090 の各 Knative サービスについてデフォルトで公開されるメトリクス

メトリクス名、単位、およびタイプ	説明	メトリックのタグ
queue_requests_per_second メトリックの単位: dimensionless メトリックのタイプ: ゲージ	キュープロキシーに到達する、1 秒あたりのリクエスト数。 Formula: **stats.RequestCount / r.reportingPeriodSeconds** **stats.RequestCount** は、指定のレポート期間のネットワーク **pkg** 統計から直接計算されます。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

メトリクス名、単位、およびタイプ	説明	メトリックのタグ
queue_proxied_operations_per_second メトリックの単位: dimensionless メトリックのタイプ: ゲージ	1秒あたりのプロキシー化された要求の数。 Formula: **stats.ProxiedRequestCount / r.reportingPeriodSeconds** **stats.ProxiedRequestCount** は指定されたレポート期間のネットワーク **pkg** 統計から直接計算されます。	
queue_average_concurrent_requests メトリックの単位: dimensionless メトリックのタイプ: ゲージ	この Pod で現在処理されている要求の数。 平均同時実行性は、ネットワークの **pkg** 側で次のように計算されます。 ● **req** の変更が行われると、変更間の時間デルタが計算されます。この結果に基づいて、デルタ上の現在の同時実行数が計算され、現在計算されている同時実行数に追加されます。また、デルタの合計が保持されます。デルタでの現在の同時実行処理は、以下のように計算されます。 **global_concurrency** × デルタ ● レポートが実行されるたびに、合計および現在の計算された同時実行性がリセットされます。 ● 平均同時実行値を報告すると、現在の計算処理はデルタの合計で除算されます。 ● 新しいリクエストが出されると、グローバル同時実行カウンターが増えます。リクエストが完了すると、カウンターが減少します。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

メトリクス名、単位、およびタイプ	説明	メトリックのタグ
queue_average_proxied_concurrent_requests メトリックの単位: dimensionless メトリックのタイプ: ゲージ	この Pod で現在処理されているプロキシー要求の数: **stats.AverageProxiedConcurrency**	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"
process_uptime メトリック単位: 秒 メトリックのタイプ: ゲージ	プロセスが起動している秒数。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

表7.2 ポート 9091 の各 Knative サービスについてデフォルトで公開されるメトリクス

メトリクス名、単位、およびタイプ	説明	メトリックのタグ
request_count メトリックの単位: dimensionless メトリックの型: counter	**queue-proxy** にルーティングされる要求の数。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
request_latencies メトリックの単位: ミリ秒 メトリックのタイプ: histogram	応答時間 (ミリ秒単位)。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

メトリクス名、単位、およびタイプ	説明	メトリックのタグ
app_request_count メトリックの単位: dimensionless メトリックの型: counter	**user-container** にルーティングされる要求の数。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
app_request_latencies メトリックの単位: ミリ秒 メトリックのタイプ: histogram	応答時間 (ミリ秒単位)。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
queue_depth メトリックの単位: dimensionless メトリックのタイプ: ゲージ	提供および待機キューの現在の項目数。無制限の同時実行の場合は報告されません。**breaker.inFlight** が使用されます。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

7.2.2. カスタムアプリケーションメトリクスを含む Knative サービス

Knative サービスによってエクスポートされるメトリクスのセットを拡張できます。正確な実装は、使用するアプリケーションと言語によって異なります。

以下のリストは、処理されたイベントカスタムメトリクスの数をエクスポートするサンプル Go アプリケーションを実装します。

```
package main

import (
  "fmt"
  "log"
  "net/http"
```

```
  "os"

  "github.com/prometheus/client_golang/prometheus" 1
  "github.com/prometheus/client_golang/prometheus/promauto"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
  opsProcessed = promauto.NewCounter(prometheus.CounterOpts{ 2
    Name: "myapp_processed_ops_total",
    Help: "The total number of processed events",
  })
)


func handler(w http.ResponseWriter, r *http.Request) {
  log.Print("helloworld: received a request")
  target := os.Getenv("TARGET")
  if target == "" {
    target = "World"
  }
  fmt.Fprintf(w, "Hello %s!\n", target)
  opsProcessed.Inc() 3
}

func main() {
  log.Print("helloworld: starting server...")

  port := os.Getenv("PORT")
  if port == "" {
    port = "8080"
  }

  http.HandleFunc("/", handler)

  // Separate server for metrics requests
  go func() { 4
    mux := http.NewServeMux()
    server := &http.Server{
      Addr: fmt.Sprintf(":%s", "9095"),
      Handler: mux,
    }
    mux.Handle("/metrics", promhttp.Handler())
    log.Printf("prometheus: listening on port %s", 9095)
    log.Fatal(server.ListenAndServe())
  }()

   // Use same port as normal requests for metrics
  //http.Handle("/metrics", promhttp.Handler()) 5
  log.Printf("helloworld: listening on port %s", port)
  log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```

1 Prometheus パッケージの追加。

2 **opsProcessed** メトリクスの定義。

3 **opsProcessed** メトリクスのインクリメント。

4 メトリクス要求に別のサーバーを使用するように設定。

5 メトリクスおよび **metrics** サブパスの通常の要求と同じポートを使用するように設定。

7.2.3. カスタムメトリクスの収集の設定

カスタムメトリクスの収集は、ユーザーワークロードのモニターリング用に設計された Prometheus のインスタンスで実行されます。ユーザーのワークロードのモニターリングを有効にしてアプリケーションを作成した後に、モニターリングスタックがメトリクスを収集する方法を定義する設定が必要になります。

以下のサンプル設定は、アプリケーションの **ksvc** を定義し、サービスモニターを設定します。正確な設定は、アプリケーションおよびメトリクスのエクスポート方法によって異なります。

```
apiVersion: serving.knative.dev/v1 1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
      annotations:
    spec:
      containers:
      - image: docker.io/skonto/helloworld-go:metrics
        resources:
          requests:
            cpu: "200m"
        env:
        - name: TARGET
          value: "Go Sample v1"
---
apiVersion: monitoring.coreos.com/v1 2
kind: ServiceMonitor
metadata:
  labels:
  name: helloworld-go-sm
spec:
  endpoints:
  - port: queue-proxy-metrics
    scheme: http
  - port: app-metrics
    scheme: http
  namespaceSelector: {}
  selector:
    matchLabels:
      name:  helloworld-go-sm
---
```

```
apiVersion: v1 3
kind: Service
metadata:
  labels:
    name:  helloworld-go-sm
  name:  helloworld-go-sm
spec:
  ports:
  - name: queue-proxy-metrics
    port: 9091
    protocol: TCP
    targetPort: 9091
  - name: app-metrics
    port: 9095
    protocol: TCP
    targetPort: 9095
  selector:
    serving.knative.dev/service: helloworld-go
  type: ClusterIP
```

1. アプリケーション仕様。
2. アプリケーションのメトリクスが収集される設定。
3. メトリクスの収集方法の設定。

7.2.4. サービスのメトリックの検証

メトリクスとモニターリングスタックをエクスポートするようにアプリケーションを設定したら、Web コンソールでメトリクスを検査できます。

前提条件

- OpenShift Container Platform Web コンソールにログインしている。
- 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. オプション: メトリクスに表示できるアプリケーションに対する要求を実行します。

```
$ hello_route=$(oc get ksvc helloworld-go -n ns1 -o jsonpath='{.status.url}') && \
    curl $hello_route
```

出力例

```
Hello Go Sample v1!
```

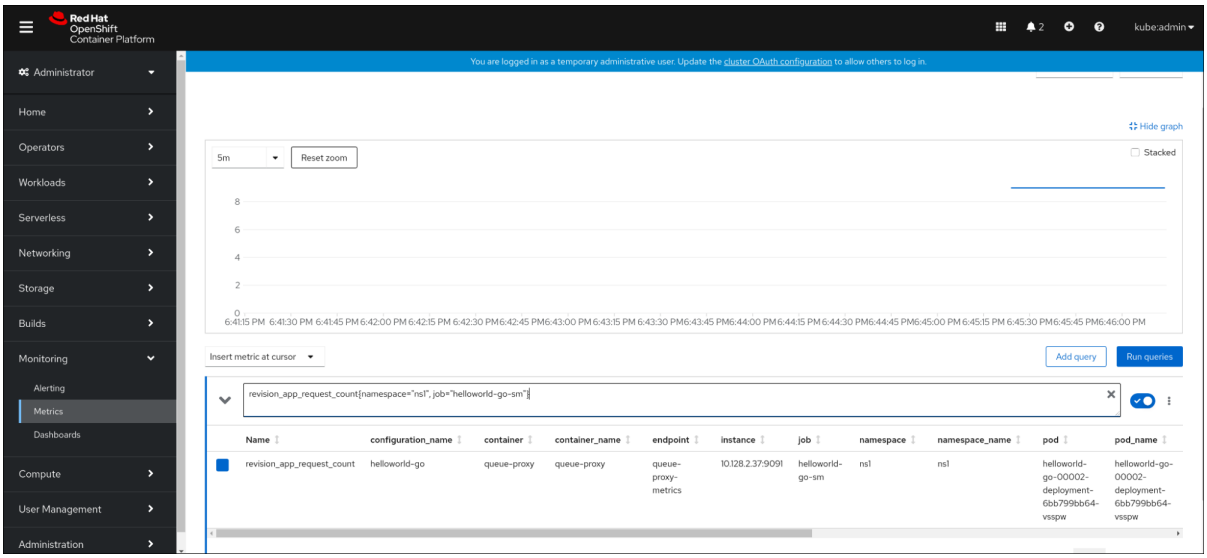
2. Web コンソールで、**Monitoring → Metrics** インターフェイスに移動します。
3. 入力フィールドに、監視するメトリクスのクエリーを入力します。以下に例を示します。

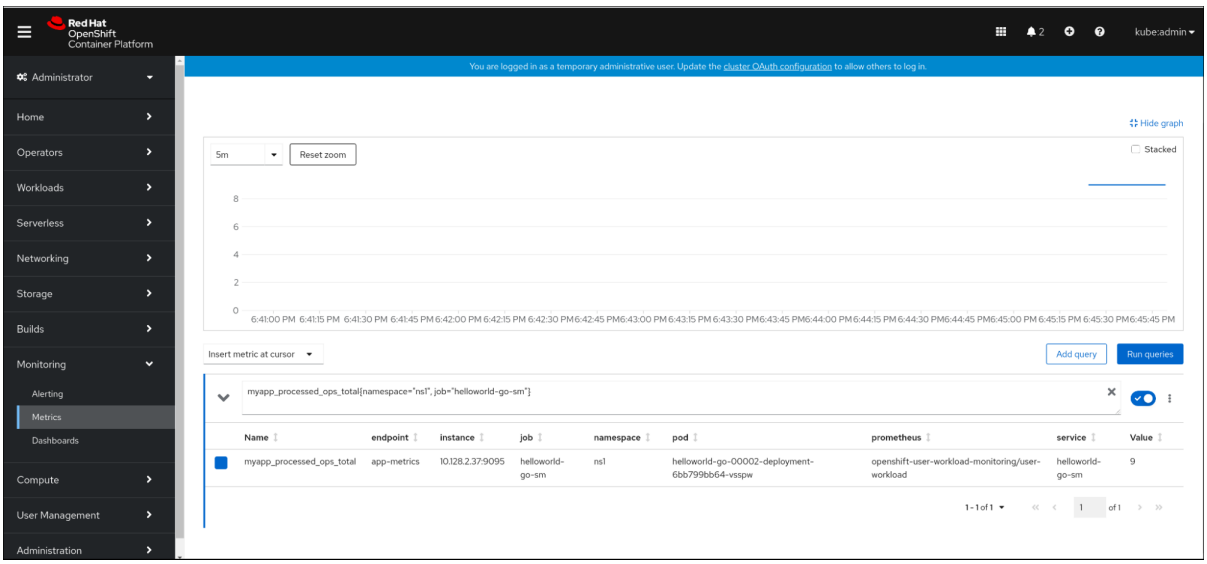
```
revision_app_request_count{namespace="ns1", job="helloworld-go-sm"}
```

別の例:

```
myapp_processed_ops_total{namespace="ns1", job="helloworld-go-sm"}
```

4. 可視化されたメトリクスを確認します。





7.2.4.1. キュープロキシーメトリクス

各 Knative サービスには、アプリケーションコンテナーへの接続をプロキシーするプロキシーコンテナーがあります。キュープロキシーのパフォーマンスについて多くのメトリクスが報告されます。

以下のメトリクスを使用して、要求がプロキシー側でキューに入れられているかどうか、およびアプリケーション側で要求を処理する際の実際の遅延を測定できます。

メトリクス名	説明	タイプ	タグ	単位

メトリクス名	説明	タイプ	タグ	単位
revision_request_count	**queue-proxy** Pod にルーティングされる要求の数。	カウンター	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	整数 (単位なし)
revision_request_latencies	リビジョン要求の応答時間。	ヒストグラム	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	ミリ秒
revision_app_request_count	**user-container** Pod にルーティングされる要求の数。	カウンター	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	整数 (単位なし)
revision_app_request_latencies	リビジョンアプリケーション要求の応答時間。	ヒストグラム	**configuration_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	ミリ秒

メトリクス名	説明	タイプ	タグ	単位
revision_queue_depth	**serving** および **waiting** キューの現在の項目数。無制限の同時実行が設定されている場合には、このメトリクスは報告されません。	ゲージ	**configuration_name**、**event-display**、**container_name**、**namespace_name**、**pod_name**、**response_code_class**、**revision_name**、**service_name**	整数 (単位なし)

7.2.5. ダッシュボードでのサービスのメトリクスの検証

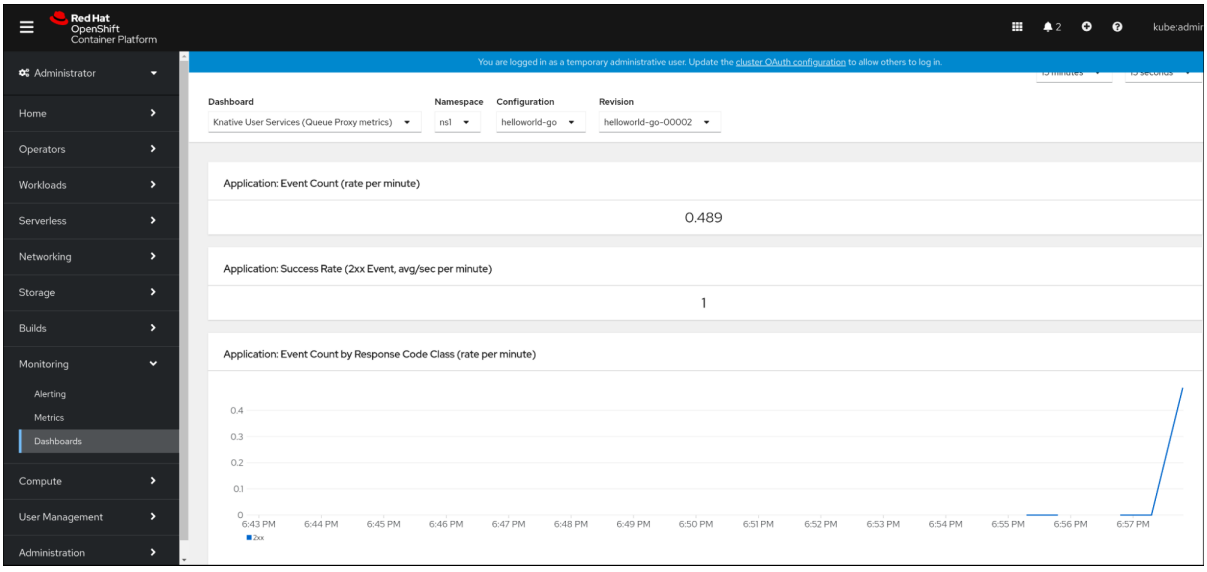
namespace でキュープロキシーメトリクスを集約する専用のダッシュボードを使用してメトリクスを検査できます。

前提条件

- OpenShift Container Platform Web コンソールにログインしている。
- 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. Web コンソールで、**Monitoring** → **Metrics** インターフェイスに移動します。
2. **Knative User Services (Queue Proxy metrics)** ダッシュボードを選択します。
3. アプリケーションに対応する **Namespace**、**Configuration**、および **Revision** を選択します。
4. 可視化されたメトリクスを確認します。



7.2.6. 関連情報

- モニタリングの概要

- ユーザー定義プロジェクトのモニタリングの有効化
- サービスのモニター方法の指定

第8章 リクエストのトレース

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。

8.1. 分散トレースの概要

サービスの所有者は、分散トレースを使用してサービスをインストルメント化し、サービスアーキテクチャーに関する洞察を得ることができます。分散トレースを使用して、現代的なクラウドネイティブのマイクロサービスベースのアプリケーションにおける、コンポーネント間の対話の監視、ネットワークプロファイリング、およびトラブルシューティングを行うことができます。

分散トレースを使用すると、以下の機能を実行できます。

- 分散トランザクションの監視
- パフォーマンスとレイテンシーの最適化
- 根本原因分析の実行

Red Hat OpenShift の分散トレースは、2 つの主要コンポーネントで設定されています。

- **Red Hat OpenShift 分散トレースプラットフォーム**: このコンポーネントは、オープンソースの Jaeger プロジェクト に基づいています。
- **Red Hat OpenShift 分散トレースデータ収集**: このコンポーネントは、オープンソースの OpenTelemetry プロジェクト に基づいています。

これらのコンポーネントは共に、特定のベンダーに依存しない OpenTracing API およびインストルメンテーションに基づいています。

8.2. RED HAT 分散トレースを使用して分散トレースを有効にする

Red Hat OpenShift 分散トレースは、複数のコンポーネントで設定されており、トレースデータを収集し、保存し、表示するためにそれらが連携します。OpenShift Serverless で Red Hat 分散トレースを使用して、サーバーレスアプリケーションを監視およびトラブルシューティングできます。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift Serverless Operator および Knative Serving がまだインストールされていません。これらは Red Hat OpenShift 分散トレースのインストール後にインストールする必要があります。
- OpenShift Container Platform の分散トレーシングのインストールのドキュメントに従って、Red Hat OpenShift の分散トレーシングをインストールしている。
- OpenShift CLI (**oc**) がインストールされている。
- 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。

手順

1. **OpenTelemetryCollector** カスタムリソース (CR) を作成します。

   OpenTelemetryCollector CR の例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: cluster-collector
  namespace: <namespace>
spec:
  mode: deployment
  config: |
    receivers:
      zipkin:
    processors:
    exporters:
      jaeger:
        endpoint: jaeger-all-in-one-inmemory-collector-headless.tracing-system.svc:14250
        tls:
          ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
      logging:
    service:
      pipelines:
        traces:
          receivers: [zipkin]
          processors: []
          exporters: [jaeger, logging]
```

2. Red Hat 分散トレースがインストールされているネームスペースで 2 つの Pod が実行されていることを確認します。

```
$ oc get pods -n <namespace>
```

   出力例

```
NAME                                  READY   STATUS    RESTARTS   AGE
cluster-collector-collector-85c766b5c-b5g99   1/1     Running   0          5m56s
jaeger-all-in-one-inmemory-ccbc9df4b-ndkl5    2/2     Running   0          15m
```

3. 次のヘッドレスサービスが作成されていることを確認します。

```
$ oc get svc -n <namespace> | grep headless
```

   出力例

```
cluster-collector-collector-headless           ClusterIP   None           <none>        9411/TCP
7m28s
jaeger-all-in-one-inmemory-collector-headless   ClusterIP   None           <none>
9411/TCP,14250/TCP,14267/TCP,14268/TCP   16m
```

   これらのサービスは、Jaeger および KnativeServing を設定するために使用されます。Jaeger サービスの名前は異なる場合があります。

4. OpenShift Serverless Operator のインストールのドキュメントに従って、OpenShift Serverless Operator をインストールします。

5. 以下の **KnativeServing** CR を作成して Knative Serving をインストールします。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
    name: knative-serving
    namespace: knative-serving
spec:
  config:
    tracing:
      backend: "zipkin"
      zipkin-endpoint: "http://cluster-collector-collector-headless.tracing-
system.svc:9411/api/v2/spans"
      debug: "true"
      sample-rate: "0.1" 1
```

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 1 つがサンプリングされます。

6. Knative サービスを作成します。

サービスの例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
      annotations:
        autoscaling.knative.dev/minScale: "1"
        autoscaling.knative.dev/target: "1"
    spec:
      containers:
      - image: quay.io/openshift-knative/helloworld:v1.2
        imagePullPolicy: Always
        resources:
          requests:
            cpu: "200m"
        env:
        - name: TARGET
          value: "Go Sample v1"
```

7. サービスにいくつかのリクエストを行います。

HTTPS 要求の例

```
$ curl https://helloworld-go.example.com
```

8. Jaeger Web コンソールの URL を取得します。

コマンドの例

```
$ oc get route jaeger-all-in-one-inmemory  -o jsonpath='{.spec.host}' -n <namespace>
```

Jaeger コンソールを使用してトレースを検証できるようになりました。

8.3. JAEGER を使用して分散トレースを有効にする

Red Hat OpenShift 分散トレースのすべてのコンポーネントをインストールしたくない場合でも、OpenShift Serverless を使用する OpenShift Container Platform で分散トレースを使用できます。これを行うには、Jaeger をスタンドアロン統合としてインストールおよび設定する必要があります。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。
- OpenShift Serverless Operator および Knative Serving がインストールされていること。
- Red Hat 分散トレースプラットフォーム Operator をインストールしました。
- OpenShift CLI (**oc**) がインストールされている。
- 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。

手順

1. 以下を含む **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 CR

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

2. **KnativeServing** CR 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 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 CR 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 ルートを使用して Jaeger Web コンソールにアクセスし、追跡データを表示できます。

1. **jaeger** ルートのホスト名を取得します。

```
$ oc get route jaeger -n default
```

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。

8.4. 関連情報

- Red Hat OpenShift 分散トレースのアーキテクチャー
- 分散トレースのインストール

第9章 OPENSHIFT SERVERLESS のサポート

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。Red Hat Customer Portal を使用して、Red Hat 製品に関するテクニカルサポート記事の Red Hat ナレッジベースを検索または閲覧できます。Red Hat Global Support Services (GSS) にサポートケースを送信したり、他の製品ドキュメントにアクセスしたりすることもできます。

このガイドを改善するための提案がある場合、またはエラーを見つけた場合は、最も関連性の高いドキュメントコンポーネントの Jira イシュー を送信できます。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。

9.1. RED HAT ナレッジベースについて

Red Hat ナレッジベース は、お客様が Red Hat の製品やテクノロジーを最大限に活用できるようにするための豊富なコンテンツを提供します。Red Hat ナレッジベースは、Red Hat 製品のインストール、設定、および使用に関する記事、製品ドキュメント、および動画で設定されています。さらに、簡潔な根本的な原因についての説明や修正手順を説明した既知の問題のソリューションを検索できます。

9.2. RED HAT ナレッジベースの検索

OpenShift Container Platform の問題が発生した場合には、初期検索を実行して、解決策を Red Hat ナレッジベース内ですでに見つけることができるかどうかを確認できます。

前提条件

- Red Hat カスタマーポータルのアカウントがある。

手順

1. Red Hat カスタマーポータル にログインします。
2. 主な Red Hat カスタマーポータルの検索フィールドには、問題に関連する入力キーワードおよび文字列を入力します。これらには、以下が含まれます。
   - OpenShift Container Platform コンポーネント (**etcd** など)
   - 関連する手順 (**installation** など)
   - 明示的な失敗に関連する警告、エラーメッセージ、およびその他の出力
3. **Search** をクリックします。
4. **OpenShift Container Platform** 製品フィルターを選択します。
5. **ナレッジベース** のコンテンツタイプフィルターを選択します。

9.3. サポートケースの送信

前提条件

- OpenShift CLI (**oc**) がインストールされている。
- Red Hat カスタマーポータルのアカウントがある。

- OpenShift Cluster Manager にアクセスできる。

手順

1. Red Hat カスタマーポータル にログインし、**SUPPORT CASES → Open a case** を選択します。
2. 問題の該当するカテゴリー (**Defect / Bug** など)、製品 (**OpenShift Container Platform**)、および製品バージョン (すでに自動入力されていない場合は **4.6**) を選択します。
3. 報告されている問題に対する一致に基づいて提案される Red Hat ナレッジベースソリューションの一覧を確認してください。提案されている記事が問題に対応していない場合は、**Continue** をクリックします。
4. 問題についての簡潔で説明的な概要と、確認されている現象および予想される動作についての詳細情報を入力します。
5. 報告されている問題に対する一致に基づいて提案される Red Hat ナレッジベースソリューションの更新された一覧を確認してください。ケース作成プロセスでより多くの情報を提供すると、この一覧の絞り込みが行われます。提案されている記事が問題に対応していない場合は、**Continue** をクリックします。
6. アカウント情報が予想通りに表示されていることを確認し、そうでない場合は適宜修正します。
7. 自動入力された OpenShift Container Platform クラスター ID が正しいことを確認します。正しくない場合は、クラスター ID を手動で取得します。
   - OpenShift Container Platform Web コンソールを使用してクラスター ID を手動で取得するには、以下を実行します。
     a. **Home → Dashboards → Overview** に移動します。
     b. **Details** セクションの **Cluster ID** フィールドで値を見つけます。
   - または、OpenShift Container Platform Web コンソールで新規サポートケースを作成し、クラスター ID を自動的に入力することができます。
     a. ツールバーから、**(?) Help → Open Support Case** に移動します。
     b. **Cluster ID** 値が自動的に入力されます。
   - OpenShift CLI (**oc**) を使用してクラスター ID を取得するには、以下のコマンドを実行します。

     ```
     $ oc get clusterversion -o jsonpath='{.items[].spec.clusterID}{"\n"}'
     ```

8. プロンプトが表示されたら、以下の質問に入力し、**Continue** をクリックします。
   - 動作はどこで発生しているか ?どの環境を使用しているか ?
   - 動作はいつ発生するか ?頻度は ?繰り返し発生するか ?特定のタイミングで発生するか ?
   - 時間枠およびビジネスへの影響について提供できるどのような情報があるか ?

9. 関連する診断データファイルをアップロードし、**Continue** をクリックします。まず **oc adm must-gather** コマンドを使用して収集されるデータと、そのコマンドによって収集されない問題に固有のデータを含めることが推奨されます。

10. 関連するケース管理の詳細情報を入力し、**Continue** をクリックします。

11. ケースの詳細をプレビューし、**Submit** をクリックします。

9.4. 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。

9.4.1. 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
- サービスログ

デフォルトで、**oc adm must-gather** コマンドはデフォルトのプラグインイメージを使用し、**./must-gather.local** に書き込みを行います。

または、以下のセクションで説明されているように、適切な引数を指定してコマンドを実行すると、特定の情報を収集できます。

- 1 つ以上の特定の機能に関連するデータを収集するには、以下のセクションに示すように、イメージと共に **--image** 引数を使用します。
  以下に例を示します。

  ```
  $ oc adm must-gather  --image=registry.redhat.io/container-native-virtualization/cnv-must-gather-rhel8:v4.9.0
  ```

- 監査ログを収集するには、以下のセクションで説明されているように **--** **/usr/bin/gather_audit_logs** 引数を使用します。
  以下に例を示します。

  ```
  $ oc adm must-gather -- /usr/bin/gather_audit_logs
  ```

  注記

  ファイルのサイズを小さくするために、監査ログはデフォルトの情報セットの一部として収集されません。

oc adm must-gather を実行すると、ランダムな名前を持つ新規 Pod がクラスターの新規プロジェクトに作成されます。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。

以下に例を示します。

```
NAMESPACE                  NAME              READY   STATUS      RESTARTS      AGE
...
openshift-must-gather-5drcj    must-gather-bklx4    2/2     Running     0          72s
openshift-must-gather-5drcj    must-gather-s8sdh    2/2     Running     0          72s
...
```

9.4.2. 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。

前提条件

- OpenShift CLI (**oc**) をインストールしている。

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

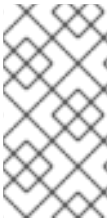
```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-rhel8:1.14.0
```

第10章 セキュリティー

10.1. TLS 認証の設定

Transport Layer Security (TLS) を使用して、Knative トラフィックを暗号化し、認証することができます。

TLS は、Knative Kafka のトラフィック暗号化でサポートされている唯一の方法です。Red Hat は、Knative Kafka リソースに SASL と TLS の両方を一緒に使用することを推奨しています。

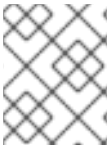


注記

Red Hat OpenShift Service Mesh 統合で内部 TLS を有効にする場合は、以下の手順で説明する内部暗号化の代わりに、mTLS で Service Mesh を有効にする必要があります。mTLS で Service Mesh を使用する場合の Knative Serving メトリクスの有効化 に関するドキュメントを参照してください。

10.1.1. 内部トラフィックの TLS 認証を有効にする

OpenShift Serverless はデフォルトで TLS エッジターミネーションをサポートしているため、エンドユーザーからの HTTPS トラフィックは暗号化されます。ただし、OpenShift ルートの背後にある内部トラフィックは、プレーンデータを使用してアプリケーションに転送されます。内部トラフィックに対して TLS を有効にすることで、コンポーネント間で送信されるトラフィックが暗号化され、このトラフィックがより安全になります。



注記

Red Hat OpenShift Service Mesh 統合で内部 TLS を有効にする場合は、以下の手順で説明する内部暗号化の代わりに、mTLS で Service Mesh を有効にする必要があります。

重要

内部 TLS 暗号化のサポートは、テクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

前提条件

- OpenShift Serverless Operator および Knative Serving がインストールされていること。
- OpenShift (**oc**) CLI がインストールされている。

手順

1. 仕様に **internal-encryption: "true"** フィールドを含む Knative サービスを作成します。

```
...
```

```
spec:
  config:
    network:
      internal-encryption: "true"
...
```

2. **knative-serving** namespace でアクティベーター Pod を再起動して、証明書を読み込みます。

```
$ oc delete pod -n knative-serving --selector app=activator
```

10.1.2. クラスターローカルサービスの TLS 認証の有効化

クラスターローカルサービスの場合、Kourier ローカルゲートウェイ **kourier-internal** が使用されます。Kourier ローカルゲートウェイに対して TLS トラフィックを使用する場合は、ローカルゲートウェイで独自のサーバー証明書を設定する必要があります。

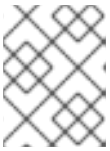
前提条件

- OpenShift Serverless Operator および Knative Serving がインストールされていること。
- 管理者権限がある。
- OpenShift (**oc**) CLI がインストールされている。

手順

1. サーバー証明書を **knative-serving-ingress** namespace にデプロイします。

```
$ export san="knative"
```



注記

これらの証明書が **<app_name>.<namespace>.svc.cluster.local** への要求を処理できるように、Subject Alternative Name (SAN) の検証が必要です。

2. ルートキーと証明書を生成します。

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
    -subj '/O=Example/CN=Example' \
    -keyout ca.key \
    -out ca.crt
```

3. SAN 検証を使用するサーバーキーを生成します。

```
$ openssl req -out tls.csr -newkey rsa:2048 -nodes -keyout tls.key \
  -subj "/CN=Example/O=Example" \
  -addext "subjectAltName = DNS:$san"
```

4. サーバー証明書を作成します。

```
$ openssl x509 -req -extfile <(printf "subjectAltName=DNS:$san") \
  -days 365 -in tls.csr \
  -CA ca.crt -CAkey ca.key -CAcreateserial -out tls.crt
```

5. Courier ローカルゲートウェイのシークレットを設定します。

   a. 前の手順で作成した証明書から、**knative-serving-ingress** namespace にシークレットをデプロイします。

   ```
   $ oc create -n knative-serving-ingress secret tls server-certs \
       --key=tls.key \
       --cert=tls.crt --dry-run=client -o yaml | oc apply -f -
   ```

   b. **KnativeServing** カスタムリソース (CR) 仕様を更新して、Kourier ゲートウェイによって作成されたシークレットを使用します。

   KnativeServing CR の例

   ```
   ...
   spec:
     config:
       kourier:
         cluster-cert-secret: server-certs
   ...
   ```

Kourier コントローラーはサービスを再起動せずに証明書を設定するため、Pod を再起動する必要はありません。

クライアントから **ca.crt** をマウントして使用することにより、ポート **443** 経由で TLS を使用して Kourier 内部サービスにアクセスできます。

関連情報

- mTLS で Service Mesh を使用する場合の Knative Serving メトリクスの有効化

10.1.3. TLS 証明書を使用してカスタムドメインでサービスを保護する

Knative サービスのカスタムドメインを設定したら、TLS 証明書を使用して、マップされたサービスを保護できます。これを行うには、Kubernetes TLS シークレットを作成してから、作成した TLS シークレットを使用するように **DomainMapping** CR を更新する必要があります。

前提条件

- Knative サービスのカスタムドメインを設定し、有効な **DomainMapping** CR がある。
- 認証局プロバイダーからの TLS 証明書または自己署名証明書がある。
- 認証局プロバイダーまたは自己署名証明書から **cert** ファイルおよび **key** ファイルを取得している。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. Kubernetes TLS シークレットを作成します。

```
$ oc create secret tls <tls_secret_name> --cert=<path_to_certificate_file> --key=
<path_to_key_file>
```

2. Red Hat OpenShift Service Mesh を OpenShift Serverless インストールのイングレスとして使用している場合は、Kubernetes TLS シークレットに次のラベルを付けます。

```
“networking.internal.knative.dev/certificate-uid": “<value>”
```

cert-manager などのサードパーティーのシークレットプロバイダーを使用している場合は、Kubernetes TLS シークレットに自動的にラベルを付けるようにシークレットマネージャーを設定できます。Cert-manager ユーザーは、提供されたシークレットテンプレートを使用して、正しいラベルを持つシークレットを自動的に生成できます。この場合、シークレットのフィルターリングはキーのみに基づいて行われますが、この値には、シークレットに含まれる証明書 ID などの有用な情報が含まれている可能性があります。

注記

{cert-manager-operator} はテクノロジープレビュー機能です。詳細は、**{cert-manager-operator} のインストール** に関するドキュメントを参照してください。

3. 作成した TLS シークレットを使用するように **DomainMapping** CR を更新します。

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: <domain_name>
  namespace: <namespace>
spec:
  ref:
    name: <service_name>
    kind: Service
    apiVersion: serving.knative.dev/v1
# TLS block specifies the secret to be used
  tls:
    secretName: <tls_secret_name>
```

検証

1. **DomainMapping** CR のステータスが **True** であることを確認し、出力の **URL** 列に、マップされたドメインをスキームの **https** で表示していることを確認します。

```
$ oc get domainmapping <domain_name>
```

出力例

```
NAME                URL                         READY   REASON
example.com         https://example.com         True
```

2. オプション: サービスが公開されている場合は、以下のコマンドを実行してこれが利用可能であることを確認します。

```
$ curl https://<domain_name>
```

証明書が自己署名されている場合は、**curl** コマンドに **-k** フラグを追加して検証を省略します。

10.1.4. Kafka ブローカーの TLS 認証の設定

Transport Layer Security (TLS) は、Apache Kafka クライアントおよびサーバーによって、Knative と Kafka 間のトラフィックを暗号化するため、および認証のために使用されます。TLS は、Knative Kafka のトラフィック暗号化でサポートされている唯一の方法です。

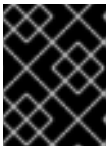
前提条件

- OpenShift Container Platform のクラスター管理者パーミッションがある。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** CR は OpenShift Container Platform クラスターにインストールされます。
- 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。
- **.pem** ファイルとして Kafka クラスター CA 証明書が保存されている。
- Kafka クラスタークライアント証明書とキーが **.pem** ファイルとして保存されている。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 証明書ファイルを **knative-eventing** namespace にシークレットファイルとして作成します。

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```



重要

キー名に **ca.crt**、 **user.crt**、および **user.key** を使用します。これらの値は変更しないでください。

2. **KnativeKafka** CR を編集し、**broker** 仕様にシークレットへの参照を追加します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
```

```
    defaultConfig:
      authSecretName: <secret_name>
...
```

10.1.5. Kafka チャネルの TLS 認証の設定

Transport Layer Security (TLS) は、Apache Kafka クライアントおよびサーバーによって、Knative と Kafka 間のトラフィックを暗号化するため、および認証のために使用されます。TLS は、Knative Kafka のトラフィック暗号化でサポートされている唯一の方法です。

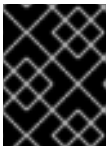
前提条件

- OpenShift Container Platform のクラスター管理者パーミッションがある。
- OpenShift Serverless Operator、Knative Eventing、および **KnativeKafka** CR は OpenShift Container Platform クラスターにインストールされます。
- 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。
- **.pem** ファイルとして Kafka クラスター CA 証明書が保存されている。
- Kafka クラスタークライアント証明書とキーが **.pem** ファイルとして保存されている。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. 選択された namespace にシークレットとして証明書ファイルを作成します。

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```



重要

キー名に **ca.crt**、 **user.crt**、および **user.key** を使用します。これらの値は変更しないでください。

2. **KnativeKafka** カスタムリソースの編集を開始します。

```
$ oc edit knativekafka
```

3. シークレットおよびシークレットの namespace を参照します。

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
```

```
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
  source:
    enabled: true
```

注記

ブートストラップサーバーで一致するポートを指定するようにしてください。

以下に例を示します。

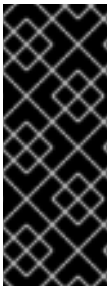
```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: tls-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9094
    enabled: true
  source:
    enabled: true
```

10.2. KNATIVE サービスの JSON WEB TOKEN 認証の設定

OpenShift Serverless には現在、ユーザー定義の承認機能がありません。ユーザー定義の承認をデプロイメントに追加するには、OpenShift Serverless を Red Hat OpenShift Service Mesh と統合してから、Knative サービスの JSON Web Token (JWT) 認証とサイドカーインジェクションを設定する必要があります。

10.2.1. Service Mesh 2.x および OpenShift Serverless での JSON Web トークン認証の使用

Service Mesh 2.x と OpenShift Serverless を使用して、Knative サービスで JSON Web Token (JWT) 認証を使用できます。これを行うには、**ServiceMeshMemberRoll** オブジェクトのメンバーであるアプリケーション namespace に認証要求とポリシーを作成する必要があります。サービスのサイドカーインジェクションも有効にする必要があります。



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、Kourier が有効化されている場合はサポートされません。

これらの namespace の Pod にサイドカーの挿入が必要な場合は、**サービスメッシュと OpenShift Serverless のネイティブに統合**に関する OpenShift Serverless のドキュメントを参照してください。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Red Hat OpenShift Service Mesh をクラスターにインストールしました。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

1. **sidecar.istio.io/inject="true"** アノテーションをサービスに追加します。

   サービスの例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 1
        sidecar.istio.io/rewriteAppHTTPProbers: "true" 2
...
```

   1 **sidecar.istio.io/inject="true"** アノテーションを追加します。

   2 OpenShift Serverless バージョン 1.14.0 以降では、HTTP プローブをデフォルトで Knative サービスの readiness プローブとして使用することから、Knative サービスでアノテーション **sidecar.istio.io/rewriteAppHTTPProbers: "true"** を設定する必要があります。

2. **Service** リソースを適用します。

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

3. **ServiceMeshMemberRoll** オブジェクトのメンバーである各サーバーレスアプリケーション namespace に **RequestAuthentication** リソースを作成します。

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: jwt-example
  namespace: <namespace>
spec:
  jwtRules:
  - issuer: testing@secure.istio.io
    jwksUri: https://raw.githubusercontent.com/istio/istio/release-
1.8/security/tools/jwt/samples/jwks.json
```

4. **RequestAuthentication** リソースを適用します。

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

5. 以下の **AuthorizationPolicy** リソースを作成して、**ServiceMeshMemberRoll** オブジェクトのメンバーである各サーバーレスアプリケーション namespace のシステム Pod からの **RequestAuthenticaton** リソースへのアクセスを許可します。

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allowlist-by-paths
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - to:
    - operation:
        paths:
        - /metrics 1
        - /healthz 2
```

1. システム Pod でメトリクスを収集するためのアプリケーションのパス。
2. システム Pod でプローブするアプリケーションのパス。

6. **AuthorizationPolicy** リソースを適用します。

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

7. **ServiceMeshMemberRoll** オブジェクトのメンバーであるサーバーレスアプリケーション namespace ごとに、以下の **AuthorizationPolicy** リソースを作成します。

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - from:
    - source:
        requestPrincipals: ["testing@secure.istio.io/testing@secure.istio.io"]
```

8. **AuthorizationPolicy** リソースを適用します。

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

検証

1. **curl** 要求を使用して 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。

コマンドの例

-

```
$ curl http://hello-example-1-default.apps.mycluster.example.com/
```

出力例

```
RBAC: access denied
```

2. 有効な JWT で要求を確認します。

   a. 有効な JWT トークンを取得します。

   ```
   $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.8/security/tools/jwt/samples/demo.jwt -s) && echo "$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 -H "Authorization: Bearer $TOKEN"  http://hello-example-1-default.apps.example.com
   ```

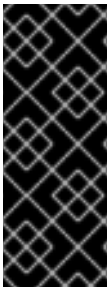
   これで要求が許可されます。

   出力例

   ```
   Hello OpenShift!
   ```

10.2.2. Service Mesh 1.x および OpenShift Serverless での JSON Web トークン認証の使用

Service Mesh 1.x と OpenShift Serverless を使用して、Knative サービスで JSON Web Token (JWT) 認証を使用できます。これを行うには、**ServiceMeshMemberRoll** オブジェクトのメンバーであるアプリケーション namespace にポリシーを作成する必要があります。サービスのサイドカーインジェクションも有効にする必要があります。



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、Kourier が有効化されている場合はサポートされません。

これらの namespace の Pod にサイドカーの挿入が必要な場合は、**サービスメッシュと OpenShift Serverless のネイティブに統合**に関する OpenShift Serverless のドキュメントを参照してください。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Red Hat OpenShift Service Mesh をクラスターにインストールしました。
- OpenShift CLI (**oc**) をインストールしている。
- 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。

手順

1. **sidecar.istio.io/inject="true"** アノテーションをサービスに追加します。

   サービスの例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 1
        sidecar.istio.io/rewriteAppHTTPProbers: "true" 2
...
```

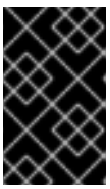
   1 **sidecar.istio.io/inject="true"** アノテーションを追加します。

   2 OpenShift Serverless バージョン 1.14.0 以降では、HTTP プローブをデフォルトで Knative サービスの readiness プローブとして使用することから、Knative サービスでアノテーション **sidecar.istio.io/rewriteAppHTTPProbers: "true"** を設定する必要があります。

2. **Service** リソースを適用します。

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

3. 有効な JSON Web Tokens (JWT) の要求のみを許可する **ServiceMeshMemberRoll** オブジェクトのメンバーであるサーバーレスアプリケーション namespace でポリシーを作成します。



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
  namespace: <namespace>
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 1
        - prefix: /healthz 2
  principalBinding: USE_ORIGIN
```

1. システム Pod でメトリクスを収集するためのアプリケーションのパス。
2. システム Pod でプローブするアプリケーションのパス。

4. **Policy** リソースを適用します。

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

検証

1. **curl** 要求を使用して 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. 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!
```

10.3. KNATIVE サービスのカスタムドメインの設定

Knative サービスには、クラスターの設定に基づいてデフォルトのドメイン名が自動的に割り当てられます。例: **<service_name>-<namespace>.example.com**.所有するカスタムドメイン名を Knative サービスにマッピングすることで、Knative サービスのドメインをカスタマイズできます。

これを行うには、サービスの **DomainMapping** リソースを作成します。複数の **DomainMapping** を作成して、複数のドメインおよびサブドメインを単一サービスにマップすることもできます。

10.3.1. カスタムドメインマッピングの作成

所有するカスタムドメイン名を Knative サービスにマッピングすることで、Knative サービスのドメインをカスタマイズできます。カスタムドメイン名をカスタムリソース (CR) にマッピングするには、Knative サービスまたは Knative ルートなどのアドレス指定可能なターゲット CR にマッピングする **DomainMapping** CR を作成する必要があります。

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。
- OpenShift CLI (**oc**) をインストールしている。
- 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。
- Knative サービスを作成し、そのサービスにマップするカスタムドメインを制御できる。

注記

カスタムドメインは OpenShift Container Platform クラスターの IP アドレスを参照する必要があります。

手順

1. マップ先となるターゲット CR と同じ namespace に **DomainMapping** CR が含まれる YAML ファイルを作成します。

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
 name: <domain_name> 1
 namespace: <namespace> 2
spec:
 ref:
   name: <target_name> 3
   kind: <target_type> 4
   apiVersion: serving.knative.dev/v1
```

1. ターゲット CR にマップするカスタムドメイン名。
2. **DomainMapping** CR とターゲット CR の両方の namespace。
3. カスタムドメインにマップするサービス名。
4. カスタムドメインにマップされる CR のタイプ。

サービスドメインマッピングの例

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
 name: example.com
 namespace: default
spec:
 ref:
   name: example-service
   kind: Service
   apiVersion: serving.knative.dev/v1
```

ルートドメインマッピングの例

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
 name: example.com
 namespace: default
spec:
 ref:
   name: example-route
   kind: Route
   apiVersion: serving.knative.dev/v1
```

2. **DomainMapping** CR を YAML ファイルとして適用します。

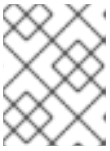
```
$ oc apply -f <filename>
```

10.3.2. Knative CLI を使用したカスタムドメインマッピングの作成

所有するカスタムドメイン名を Knative サービスにマッピングすることで、Knative サービスのドメインをカスタマイズできます。Knative (**kn**) CLI を使用して、Knative サービスまたは Knative ルートなどのアドレス指定可能なターゲット CR にマップする **DomainMapping** カスタムリソース (CR) を作成できます。

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。
- Knative サービスまたはルートを作成し、その CR にマップするカスタムドメインを制御している。



注記

カスタムドメインは OpenShift Container Platform クラスターの DNS を参照する必要があります。

- Knative (**kn**) CLI をインストールしている。
- 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。

手順

- ドメインを現在の namespace の CR にマップします。

```
$ kn domain create <domain_mapping_name> --ref <target_name>
```

コマンドの例

```
$ kn domain create example.com --ref example-service
```

--ref フラグは、ドメインマッピング用のアドレス指定可能なターゲット CR を指定します。

--ref フラグの使用時に接頭辞が指定されていない場合、ターゲットが現在の namespace の Knative サービスであることを前提としています。

- ドメインを指定された namespace の Knative サービスにマップします。

```
$ kn domain create <domain_mapping_name> --ref
<ksvc:service_name:service_namespace>
```

コマンドの例

```
$ kn domain create example.com --ref ksvc:example-service:example-namespace
```

- ドメインを Knative ルートにマップします。

```
$ kn domain create <domain_mapping_name> --ref <kroute:route_name>
```

コマンドの例

```
$ kn domain create example.com --ref kroute:example-route
```

10.3.3. TLS 証明書を使用してカスタムドメインでサービスを保護する

Knative サービスのカスタムドメインを設定したら、TLS 証明書を使用して、マップされたサービスを保護できます。これを行うには、Kubernetes TLS シークレットを作成してから、作成した TLS シークレットを使用するように **DomainMapping** CR を更新する必要があります。

前提条件

- Knative サービスのカスタムドメインを設定し、有効な **DomainMapping** CR がある。
- 認証局プロバイダーからの TLS 証明書または自己署名証明書がある。
- 認証局プロバイダーまたは自己署名証明書から **cert** ファイルおよび **key** ファイルを取得している。
- OpenShift CLI (**oc**) をインストールしている。

手順

1. Kubernetes TLS シークレットを作成します。

```
$ oc create secret tls <tls_secret_name> --cert=<path_to_certificate_file> --key=
<path_to_key_file>
```

2. Red Hat OpenShift Service Mesh を OpenShift Serverless インストールのイングレスとして使用している場合は、Kubernetes TLS シークレットに次のラベルを付けます。

```
“networking.internal.knative.dev/certificate-uid": “<value>”
```

cert-manager などのサードパーティーのシークレットプロバイダーを使用している場合は、Kubernetes TLS シークレットに自動的にラベルを付けるようにシークレットマネージャーを設

定できます。Cert-manager ユーザーは、提供されたシークレットテンプレートを使用して、正しいラベルを持つシークレットを自動的に生成できます。この場合、シークレットのフィルターリングはキーのみに基づいて行われますが、この値には、シークレットに含まれる証明書 ID などの有用な情報が含まれている可能性があります。

注記

{cert-manager-operator} はテクノロジープレビュー機能です。詳細は、**{cert-manager-operator} のインストール** に関するドキュメントを参照してください。

3. 作成した TLS シークレットを使用するように **DomainMapping** CR を更新します。

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: <domain_name>
  namespace: <namespace>
spec:
  ref:
    name: <service_name>
    kind: Service
    apiVersion: serving.knative.dev/v1
# TLS block specifies the secret to be used
  tls:
    secretName: <tls_secret_name>
```

検証

1. **DomainMapping** CR のステータスが **True** であることを確認し、出力の **URL** 列に、マップされたドメインをスキームの **https** で表示していることを確認します。

```
$ oc get domainmapping <domain_name>
```

出力例

```
NAME                URL                         READY   REASON
example.com         https://example.com         True
```

2. オプション: サービスが公開されている場合は、以下のコマンドを実行してこれが利用可能であることを確認します。

```
$ curl https://<domain_name>
```

証明書が自己署名されている場合は、**curl** コマンドに **-k** フラグを追加して検証を省略します。

第11章 関数

11.1. OPENSHIFT SERVERLESS FUNCTIONS の設定

重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

アプリケーションコードのデプロイプロセスを改善するために、OpenShift Serverless を使用して、ステートレスでイベント駆動型の関数を Knative サービスとして OpenShift Container Platform にデプロイできます。関数を開発する場合は、セットアップ手順を完了する必要があります。

11.1.1. 前提条件

クラスターで OpenShift Serverless Functions の使用を有効にするには、以下の手順を実行する必要があります。

- 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。

注記

関数は Knative サービスとしてデプロイされます。関数でイベント駆動型のアーキテクチャーを使用する必要がある場合は、Knative Eventing もインストールする必要があります。

- **oc** CLI CLI がインストールされている。
- Knative (**kn**) CLI がインストールされている。Knative CLI をインストールすると、関数の作成および管理に使用できる **kn func** コマンドを使用できます。
- Docker Container Engine または podman バージョン 3.4.7 以降がインストールされており、OpenShift Container Registry などの利用可能なイメージレジストリーにアクセスできる。
- Quay.io をイメージレジストリーとして使用する場合は、リポジトリーがプライベートではないか確認するか、OpenShift Container Platform ドキュメント Pod が他のセキュアなレジストリーからイメージを参照できるようにする設定 に従っていることを確認する必要があります。
- OpenShift Container レジストリーを使用している場合には、クラスター管理者は レジストリーを公開する 必要があります。

11.1.2. podman の設定

高度なコンテナー管理機能を使用するには、OpenShift Serverless Functions で podman を使用することをお勧めします。そのためには、podman サービスを開始し、それに接続するように Knative (**kn**) CLI を設定する必要があります。

手順

1. **${XDG_RUNTIME_DIR}/podman/podman.sock** で、UNIX ソケットで Docker API を提供する podman サービスを起動します。

```
$ systemctl start --user podman.socket
```

注記

多くのシステムでは、このソケットは **/run/user/$(id -u)/podman/podman.sock** にあります。

2. 関数のビルドに使用する環境変数を確立します。

```
$ export DOCKER_HOST="unix://${XDG_RUNTIME_DIR}/podman/podman.sock"
```

3. **-v** フラグを指定して、関数プロジェクトディレクトリー内で build コマンドを実行し、詳細な出力を表示します。ローカルの UNIX ソケットへの接続が表示されるはずです。

```
$ kn func build -v
```

11.1.3. macOS での podman のセットアップ

高度なコンテナー管理機能を使用するには、OpenShift Serverless Functions で podman を使用することをお勧めします。macOS でこれを行うには、podman マシンを起動し、それに接続するように Knative (**kn**) CLI を設定する必要があります。

手順

1. podman マシンを作成します。

```
$ podman machine init --memory=8192 --cpus=2 --disk-size=20
```

2. UNIX ソケットで Docker API を提供する podman マシンを開始します。

```
$ podman machine start
Starting machine "podman-machine-default"
Waiting for VM ...
Mounting volume... /Users/myuser:/Users/user

[...truncated output...]

You can still connect Docker API clients by setting DOCKER_HOST using the
following command in your terminal session:

 export
DOCKER_HOST='unix:///Users/myuser/.local/share/containers/podman/machine/podman-
```

```
machine-default/podman.sock'

Machine "podman-machine-default" started successfully
```

注記

ほとんどの macOS システムでは、このソケットは **/Users/myuser/.local/share/containers/podman/machine/podman-machine-default/podman.sock** にあります。

3. 関数のビルドに使用する環境変数を確立します。

```
$ export
DOCKER_HOST='unix:///Users/myuser/.local/share/containers/podman/machine/podman-
machine-default/podman.sock'
```

4. **-v** フラグを指定して、関数プロジェクトディレクトリー内で build コマンドを実行し、詳細な出力を表示します。ローカルの UNIX ソケットへの接続が表示されるはずです。

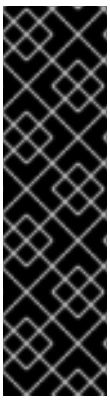
```
$ kn func build -v
```

11.1.4. 次のステップ

- Docker Container Engine または podman の詳細は、コンテナービルドツールのオプション を参照してください。
- 関数を使い始めるを参照してください。

11.2. 関数を使い始める

関数のライフサイクル管理には、関数の作成、構築、デプロイが含まれます。必要に応じて、デプロイされた関数を呼び出してテストすることもできます。これらの操作はすべて、**kn func** ツールを使用して OpenShift Serverless で実行できます。



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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。

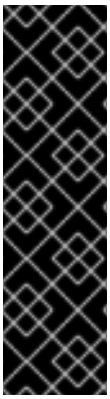
Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

11.2.1. 前提条件

以下の手順を実行する前に、OpenShift Serverless 機能の設定の要件タスクをすべて完了している必要があります。

11.2.2. 関数の作成

関数をビルドし、デプロイする前に、Knative (**kn**) CLI を使用して関数を作成する必要があります。コマンドラインでパス、ランタイム、テンプレート、およびイメージレジストリーをフラグとして指定するか、**-c** フラグを使用してターミナルで対話型エクスペリエンスを開始できます。



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。
- Knative (**kn**) CLI をインストールしている。

手順

- 関数プロジェクトを作成します。

```
$ kn func create -r <repository> -l <runtime> -t <template> <path>
```

  - 受け入れられるランタイム値には、**quarkus**、**node**、**typescript**、**go**、**python**、**springboot**、および **rust** が含まれます。
  - 受け入れられるテンプレート値には、**http** と **cloudevents** が含まれます。

    コマンドの例

```
$ kn func create -l typescript -t cloudevents examplefunc
```

    出力例

```
Created typescript function in /home/user/demo/examplefunc
```

  - または、カスタムテンプレートを含むリポジトリーを指定することもできます。

    コマンドの例

```
$ kn func create -r https://github.com/boson-project/templates/ -l node -t hello-world
examplefunc
```

    出力例

```
Created node function in /home/user/demo/examplefunc
```

11.2.3. 機能をローカルで実行する

kn func run コマンドを使用して、現在のディレクトリーまたは **--path** フラグで指定されたディレクトリーで機能をローカルに実行できます。実行している関数が以前にビルドされたことがない場合、またはプロジェクトファイルが最後にビルドされてから変更されている場合、**kn func run** コマンドは、既定で関数を実行する前に関数をビルドします。

現在のディレクトリーで機能を実行するコマンドの例

```
$ kn func run
```

パスとして指定されたディレクトリーで機能を実行するコマンドの例

```
$ kn func run --path=<directory_path>
```

--build フラグを使用して、プロジェクトファイルに変更がなくても、機能を実行する前に既存のイメージを強制的に再構築することもできます。

ビルドフラグを使用した実行コマンドの例

```
$ kn func run --build
```

ビルド フラグを false に設定すると、イメージのビルドが無効になり、以前にビルドされたイメージを使用して機能が実行されます。

ビルドフラグを使用した実行コマンドの例

```
$ kn func run --build=false
```

help コマンドを使用して、**kn func run** コマンドオプションの詳細を確認できます。

help コマンドの構築

```
$ kn func help run
```

11.2.4. 関数のビルド

関数を実行する前に、関数プロジェクトをビルドする必要があります。**kn func run** コマンドを使用している場合、関数は自動的に構築されます。ただし、**kn func build** コマンドを使用すると、実行せずに関数をビルドできます。これは、上級ユーザーやデバッグシナリオに役立ちます。

kn func build は、コンピューターまたは OpenShift Container Platform クラスターでローカルに実行できる OCI コンテナーイメージを作成します。このコマンドは、関数プロジェクト名とイメージレジストリー名を使用して、関数の完全修飾イメージ名を作成します。

11.2.4.1. イメージコンテナーの種類

デフォルトでは、**kn func build** は、Red Hat Source-to-Image (S2I) テクノロジーを使用してコンテナーイメージを作成します。

Red Hat Source-to-Image (S2I) を使用したビルドコマンドの例

```
$ kn func build
```

--builder フラグをコマンドに追加し、**pack** 戦略を指定することで、代わりに CNCF Cloud Native Buildpacks テクノロジーを使用できます。

CNCF Cloud Native Buildpacks を使用したビルドコマンドの例

```
$ kn func build --builder pack
```

11.2.4.2. イメージレジストリーの種類

OpenShift Container Registry は、関数イメージを保存するためのイメージレジストリーとしてデフォルトで使用されます。

OpenShift Container Registry を使用したビルドコマンドの例

```
$ kn func build
```

出力例

```
Building function image
Function image has been built, image: registry.redhat.io/example/example-function:latest
```

--registry フラグを使用して、OpenShift Container Registry をデフォルトのイメージレジストリーとして使用することをオーバーライドできます。

quay.io を使用するように OpenShift Container Registry をオーバーライドするビルドコマンドの例

```
$ kn func build --registry quay.io/username
```

出力例

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

11.2.4.3. Push フラグ

--push フラグを **kn func build** コマンドに追加して、正常にビルドされた後に関数イメージを自動的にプッシュできます。

OpenShift Container Registry を使用したビルドコマンドの例

```
$ kn func build --push
```

11.2.4.4. Help コマンド

kn func build コマンドオプションの詳細については、help コマンドを使用できます。

help コマンドの構築

```
$ kn func help build
```

11.2.5. 関数のデプロイ

kn func deploy コマンドを使用して、関数を Knative サービスとしてクラスターにデプロイできます。ターゲット関数がすでにデプロイされている場合には、コンテナーイメージレジストリーにプッシュされている新規コンテナーイメージで更新され、Knative サービスが更新されます。

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。
- Knative (**kn**) CLI をインストールしている。
- 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。
- デプロイする関数を作成し、初期化している必要がある。

手順

- 関数をデプロイします。

  ```
  $ kn func deploy [-n <namespace> -p <path> -i <image>]
  ```

  出力例

  ```
  Function deployed at: http://func.example.com
  ```

  - **namespace** が指定されていない場合には、関数は現在の namespace にデプロイされます。
  - この関数は、**パス** が指定されない限り、現在のディレクトリーからデプロイされます。
  - Knative サービス名はプロジェクト名から派生するので、以下のコマンドでは変更できません。

11.2.6. テストイベントでのデプロイされた関数の呼び出し

kn func invoke CLI コマンドを使用して、ローカルまたは OpenShift Container Platform クラスター上で関数を呼び出すためのテストリクエストを送信できます。このコマンドを使用して、関数が機能し、イベントを正しく受信できることをテストできます。関数をローカルで呼び出すと、関数開発中の簡単なテストに役立ちます。クラスターで関数を呼び出すと、実稼働環境に近いテストに役立ちます。

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。
- Knative (**kn**) CLI をインストールしている。

- 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。
- 呼び出す関数をすでにデプロイしている必要があります。

手順

- 関数を呼び出します。

```
$ kn func invoke
```

  - **kn func invoke** コマンドは、ローカルのコンテナーイメージが実行中の場合や、クラスターにデプロイされた関数がある場合にのみ機能します。
  - **kn func invoke** コマンドは、デフォルトでローカルディレクトリーで実行され、このディレクトリーが関数プロジェクトであると想定します。

11.2.7. 関数の削除

kn func delete コマンドを使用して関数を削除できます。これは、関数が不要になった場合に役立ち、クラスターのリソースを節約するのに役立ちます。

手順

- 関数を削除します。

```
$ kn func delete [<function_name> -n <namespace> -p <path>]
```

  - 削除する関数の名前またはパスが指定されていない場合には、現在のディレクトリーで **func.yaml** ファイルを検索し、削除する関数を判断します。
  - namespace が指定されていない場合には、**func.yaml** の **namespace** の値にデフォルト設定されます。

11.2.8. 関連情報

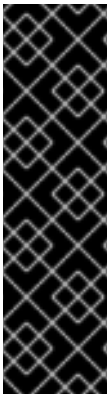
- Exposing a default registry manually
- Marketplace page for the Intellij Knative plug-in
- Marketplace page for the Visual Studio Code Knative plug-in

11.3. クラスター上での機能の構築とデプロイ

関数をローカルでビルドする代わりに、クラスターで直接関数をビルドできます。このワークフローをローカル開発マシンで使用する場合は、関数のソースコードのみを操作する必要があります。これは、たとえば、docker や podman などのクラスター上の関数構築ツールをインストールできない場合に役立ちます。

11.3.1. クラスターでの関数のビルドとデプロイ

Knative (**kn**) CLI を使用して、関数プロジェクトのビルドを開始し、関数をクラスターに直接デプロイできます。この方法で関数プロジェクトをビルドするには、関数プロジェクトのソースコードが、クラスターにアクセスできる Git リポジトリーブランチに存在する必要があります。



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

前提条件

- Red Hat OpenShift パイプラインをクラスターにインストールする必要がある。
- OpenShift CLI (**oc**) がインストールされている。
- Knative (**kn**) CLI をインストールしている。

手順

1. パイプラインを実行して関数をデプロイする各 namespace で、次のリソースを作成する必要があります。
   a. パイプラインで Source-to-Image を使用できるように **s2i** Tekton タスクを作成します。

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.25.0/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
```

   b. **kn func** deploy Tekton タスクを作成して、パイプラインに関数をデプロイできるようにします。

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.25.0/pipelines/resources/tekton/task/func-deploy/0.1/func-deploy.yaml
```

2. 関数を作成します。

```
$ kn func create <function_name> -l <runtime>
```

3. 新しい関数プロジェクトを作成したら、プロジェクトを Git リポジトリーに追加し、リポジトリーがクラスターで使用可能であることを確認する必要があります。この Git リポジトリーに関する情報は、次のステップで **func.yaml** ファイルを更新するために使用されます。

4. 関数プロジェクトの **func.yaml** ファイルの設定を更新して、Git リポジトリーのクラスター上ビルドを有効にします。

```
...
git:
  url: <git_repository_url> 1
```

```
    revision: main 2
    contextDir: <directory_path> 3
...
```

1. 必須。関数のソースコードを含む Git リポジトリーを指定します。
2. オプション。使用する Git リポジトリーのリビジョンを指定します。これは、ブランチ、タグ、またはコミットにすることができます。
3. オプション。関数が Git リポジトリーのルートフォルダーにない場合は、関数のディレクトリーパスを指定します。

5. 関数のビジネスロジックを実装します。次に、Git を使用して変更をコミットしてプッシュします。
6. 関数をデプロイします。

```
$ kn func deploy --remote
```

関数設定で参照されているコンテナーレジストリーにログインしていない場合は、関数イメージをホストするリモートコンテナーレジストリーの資格情報を入力するように求められます。

出力例とプロンプト

```
  Creating Pipeline resources
Please provide credentials for image registry used by Pipeline.
? Server: https://index.docker.io/v1/
? Username: my-repo
? Password: ********
  Function deployed at URL: http://test-function.default.svc.cluster.local
```

7. 関数を更新するには、Git を使用して新しい変更をコミットしてプッシュしてから、**kn func deploy --remote** コマンドを再度実行します。

11.3.2. 関数リビジョンの指定

関数をビルドしてクラスターにデプロイするときは、リポジトリー内の Git リポジトリー、ブランチ、およびサブディレクトリーを指定して、関数コードの場所を指定する必要があります。**main** ブランチを使用する場合は、ブランチを指定する必要はありません。同様に、関数がリポジトリーのルートにある場合、サブディレクトリーを指定する必要はありません。これらのパラメーターは、**func.yaml** 設定ファイルで指定するか、**kn func deploy** コマンドでフラグを使用して指定できます。

前提条件

- Red Hat OpenShift パイプラインをクラスターにインストールする必要がある。
- OpenShift (**oc**) CLI がインストールされている。
- Knative (**kn**) CLI をインストールしている。

手順

- 関数をデプロイします。

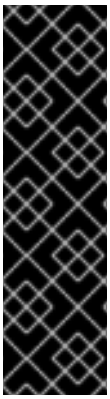
```
$ kn func deploy --remote \ 1
                 --git-url <repo-url> \ 2
                 [--git-branch <branch>] \ 3
                 [--git-dir <function-dir>] 4
```

1. **--remote** フラグを使用すると、ビルドはリモートで実行されます。
2. **<repo-url>** を Git リポジトリーの URL に置き換えます。
3. **<branch>** を Git ブランチ、タグ、またはコミットに置き換えます。**main** ブランチで最新のコミットを使用している場合は、このフラグをスキップできます。
4. **<function-dir>** がリポジトリーのルートディレクトリーと異なる場合は、関数を含むディレクトリーに置き換えます。

以下に例を示します。

```
$ kn func deploy --remote \
                 --git-url https://example.com/alice/myfunc.git \
                 --git-branch my-feature \
                 --git-dir functions/example-func/
```

11.4. NODE.JS 関数の開発



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Node.js 関数プロジェクトを作成 したら、指定のテンプレートを変更して、関数にビジネスロジックを追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

11.4.1. 前提条件

- 関数を開発する前に、OpenShift Serverless 機能の設定の手順を実行する必要があります。

11.4.2. Node.js 関数テンプレート構造

Knative (**kn**) CLI を使用して Node.js 関数を作成すると、プロジェクトディレクトリーは典型的な Node.js プロジェクトのようになります。唯一の例外は、関数の設定に使用される追加の **func.yaml** ファイルです。

http および **event** トリガー関数のテンプレート構造はいずれも同じです。

テンプレート構造

```
.
├── func.yaml 1
├── index.js 2
├── package.json 3
├── README.md
└── test 4
    ├── integration.js
    └── unit.js
```

1. **func.yaml** 設定ファイルは、イメージ名とレジストリーを判断するために使用されます。
2. プロジェクトに関数を 1 つエクスポートする **index.js** ファイルを追加する必要があります。
3. テンプレート **package.json** ファイルにある依存関係に限定されるわけではありません。他の Node.js プロジェクトと同様に、別の依存関係を追加できます。

   npm 依存関係の追加例

   ```
   npm install --save opossum
   ```

   デプロイメント用にプロジェクトをビルドすると、これらの依存関係は作成したランタイムコンテナーイメージに含まれます。
4. 統合およびテストスクリプトは、関数テンプレートの一部として提供されます。

11.4.3. Node.js 関数の呼び出しについて

Knative (**kn**) CLI を使用して関数プロジェクトを作成する場合に、CloudEvents に応答するプロジェクト、または単純な HTTP 要求に応答するプロジェクトを生成できます。Knative の CloudEvents は HTTP 経由で POST 要求として転送されるため、関数タイプはいずれも受信 HTTP イベントをリッスンして応答します。

Node.js 関数は、単純な HTTP 要求で呼び出すことができます。受信要求を受け取ると、関数は **context** オブジェクトで最初のパラメーターとして呼び出されます。

11.4.3.1. Node.js コンテキストオブジェクト

関数は、**context** オブジェクトを最初のパラメーターとして渡して呼び出されます。このオブジェクトは、受信 HTTP 要求情報へのアクセスを提供します。

コンテキストオブジェクトの例

```
function handle(context, data)
```

この情報には、HTTP リクエストメソッド、リクエストと共に送信されたクエリー文字列またはヘッダー、HTTP バージョン、およびリクエスト本文が含まれます。**CloudEvent** の受信インスタンスが含まれる受信要求はコンテキストオブジェクトにアタッチし、**context.cloudevent** を使用してアクセスできるようにします。

11.4.3.1.1. コンテキストオブジェクトメソッド

context オブジェクトには、データの値を受け入れ、CloudEvent を返す **cloudEventResponse()** メソッドが 1 つあります。

Knative システムでは、サービスとしてデプロイされた関数が CloudEvent を送信するイベントブローカーによって呼び出される場合に、ブローカーが応答を確認します。応答が CloudEvent の場合には、このイベントはブローカーにが処理します。

コンテキストオブジェクトメソッドの例

```
// Expects to receive a CloudEvent with customer data
function handle(context, customer) {
  // process the customer
  const processed = handle(customer);
  return context.cloudEventResponse(customer)
    .source('/handle')
    .type('fn.process.customer')
    .response();
}
```

11.4.3.1.2. CloudEvent data

受信要求が CloudEvent の場合は、CloudEvent に関連付けられたデータがすべてイベントから抽出され、2 番目のパラメーターとして提供されます。たとえば、以下のように data プロパティーに JSON 文字列が含まれる CloudEvent が受信された場合に、以下のようになります。

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

呼び出されると、関数に対して **context** オブジェクト後に来る 2 番目のパラメーターは、JavaScript オブジェクトで、このオブジェクトには **customerId** と **productId** プロパティーが含まれます。

署名の例

```
function handle(context, data)
```

この例の **data** パラメーターは、**customerId** および **productId** プロパティーが含まれる JavaScript オブジェクトです。

11.4.4. Node.js 関数の戻り値

この関数は有効な JavaScript タイプを返すことができます。がそれ以外は戻り値を持たせないようにすることもできます。関数に戻り値が指定されておらず、失敗を指定しないと、呼び出し元は **204 No Content** 応答を受け取ります。

関数は、CloudEvent または **Message** オブジェクトを返してイベントを Knative Eventing システムにプッシュすることもできます。この場合に、開発者は CloudEvent メッセージング仕様の理解や実装は必要ありません。返された値からのヘッダーおよびその他の関連情報は抽出され、応答で送信されます。

例

```
function handle(context, customer) {
```

```
  // process customer and return a new CloudEvent
  return new CloudEvent({
    source: 'customer.processor',
    type: 'customer.processed'
  })
}
```

11.4.4.1. 返されるヘッダー

headers プロパティーを **return** オブジェクトに追加して応答ヘッダーを設定できます。これらのヘッダーは抽出され、呼び出し元に応答して送信されます。

応答ヘッダーの例

```
function handle(context, customer) {
  // process customer and return custom headers
  // the response will be '204 No content'
  return { headers: { customerid: customer.id } };
}
```

11.4.4.2. 返されるステータスコード

statusCode プロパティーを **return** オブジェクトに追加して、呼び出し元に返されるステータスコードを設定できます。

ステータスコード

```
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    return { statusCode: 451 }
  }
}
```

ステータスコードは、関数で作成および出力されるエラーに対して設定することもできます。

エラーステータスコードの例

```
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
    err.statusCode = 451;
    throw err;
  }
}
```

11.4.5. Node.js 関数のテスト

Node.js 関数は、コンピューターに対してローカルでテストできます。**kn func create** を使用して関数を作成する際に作成されるデフォルトプロジェクトには、簡単なユニットテストおよびインテグレーションテストが含まれる **test** フォルダーがあります。

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。
- Knative (**kn**) CLI をインストールしている。
- **kn func create** を使用して関数を作成している。

手順

1. 関数の **test** フォルダーに移動します。
2. テストを実行します。

```
$ npm test
```

11.4.6. 次のステップ

- Node.js コンテキストオブジェクトの参照ドキュメントを参照してください。
- 関数を構築 して デプロイ します。

11.5. TYPESCRIPT 関数の開発

重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

TypeScript 関数プロジェクトを作成 したら、指定のテンプレートを変更して、関数にビジネスロジックを追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

11.5.1. 前提条件

- 関数を開発する前に、OpenShift Serverless 機能の設定の手順を実行する必要があります。

11.5.2. Typescript 関数テンプレートの構造

Knative (**kn**) CLI を使用して TypeScript 関数を作成すると、プロジェクトディレクトリーは典型的な TypeScript プロジェクトのようになります。唯一の例外は、関数の設定に使用される追加の **func.yaml** ファイルです。

http および **event** トリガー関数のテンプレート構造はいずれも同じです。

テンプレート構造

```
.
├── func.yaml 1
├── package.json 2
├── package-lock.json
├── README.md
├── src
│   └── index.ts 3
├── test 4
│   ├── integration.ts
│   └── unit.ts
└── tsconfig.json
```

1. **func.yaml** 設定ファイルは、イメージ名とレジストリーを判断するために使用されます。
2. テンプレート **package.json** ファイルにある依存関係に限定されるわけではありません。他の TypeScript プロジェクトと同様に、別の依存関係を追加できます。

   npm 依存関係の追加例

   ```
   npm install --save opossum
   ```

   デプロイメント用にプロジェクトをビルドすると、これらの依存関係は作成したランタイムコンテナーイメージに含まれます。
3. プロジェクトには、**handle** という名前の関数をエクスポートする **src/index.js** ファイルが含まれている必要があります。
4. 統合およびテストスクリプトは、関数テンプレートの一部として提供されます。

11.5.3. TypeScript 関数の呼び出しについて

Knative (**kn**) CLI を使用して関数プロジェクトを作成する場合に、CloudEvents に応答するプロジェクト、または単純な HTTP 要求に応答するプロジェクトを生成できます。Knative の CloudEvents は HTTP 経由で POST 要求として転送されるため、関数タイプはいずれも受信 HTTP イベントをリッスンして応答します。

Typescript 関数は、単純な HTTP 要求で呼び出すことができます。受信要求を受け取ると、関数は **context** オブジェクトで最初のパラメーターとして呼び出されます。

11.5.3.1. Typescript コンテキストオブジェクト

関数を呼び出すには、**context** オブジェクトを最初のパラメーターとして指定します。**context** オブジェクトのプロパティーにアクセスすると、着信 HTTP 要求に関する情報を提供できます。

コンテキストオブジェクトの例

```
function handle(context:Context): string
```

この情報には、HTTP リクエストメソッド、リクエストと共に送信されたクエリー文字列またはヘッダー、HTTP バージョン、およびリクエスト本文が含まれます。**CloudEvent** の受信インスタンスが含

まれる受信要求はコンテキストオブジェクトにアタッチし、**context.cloudevent** を使用してアクセスできるようにします。

11.5.3.1.1. コンテキストオブジェクトメソッド

context オブジェクトには、データの値を受け入れ、CloudEvent を返す **cloudEventResponse()** メソッドが 1 つあります。

Knative システムでは、サービスとしてデプロイされた関数が CloudEvent を送信するイベントブローカーによって呼び出される場合に、ブローカーが応答を確認します。応答が CloudEvent の場合には、このイベントはブローカーにが処理します。

コンテキストオブジェクトメソッドの例

```
// Expects to receive a CloudEvent with customer data
export function handle(context: Context, cloudevent?: CloudEvent): CloudEvent {
  // process the customer
  const customer = cloudevent.data;
  const processed = processCustomer(customer);
  return context.cloudEventResponse(customer)
    .source('/customer/process')
    .type('customer.processed')
    .response();
}
```

11.5.3.1.2. コンテキストタイプ

TypeScript タイプの定義ファイルは、関数で使用する以下のタイプをエクスポートします。

エクスポートタイプの定義

```
// Invokable is the expeted Function signature for user functions
export interface Invokable {
    (context: Context, cloudevent?: CloudEvent): any
}

// Logger can be used for structural logging to the console
export interface Logger {
  debug: (msg: any) => void,
  info:  (msg: any) => void,
  warn:  (msg: any) => void,
  error: (msg: any) => void,
  fatal: (msg: any) => void,
  trace: (msg: any) => void,
}

// Context represents the function invocation context, and provides
// access to the event itself as well as raw HTTP objects.
export interface Context {
    log: Logger;
    req: IncomingMessage;
    query?: Record<string, any>;
    body?: Record<string, any>|string;
    method: string;
    headers: IncomingHttpHeaders;
```

```
    httpVersion: string;
    httpVersionMajor: number;
    httpVersionMinor: number;
    cloudevent: CloudEvent;
    cloudEventResponse(data: string|object): CloudEventResponse;
}

// CloudEventResponse is a convenience class used to create
// CloudEvents on function returns
export interface CloudEventResponse {
    id(id: string): CloudEventResponse;
    source(source: string): CloudEventResponse;
    type(type: string): CloudEventResponse;
    version(version: string): CloudEventResponse;
    response(): CloudEvent;
}
```

11.5.3.1.3. CloudEvent data

受信要求が CloudEvent の場合は、CloudEvent に関連付けられたデータがすべてイベントから抽出され、2 番目のパラメーターとして提供されます。たとえば、以下のように data プロパティーに JSON 文字列が含まれる CloudEvent が受信された場合に、以下のようになります。

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

呼び出されると、関数に対して **context** オブジェクト後に来る 2 番目のパラメーターは、JavaScript オブジェクトで、このオブジェクトには **customerId** と **productId** プロパティーが含まれます。

署名の例

```
function handle(context: Context, cloudevent?: CloudEvent): CloudEvent
```

この例の **cloudevent** パラメーターは、**customerId** および **productId** プロパティーが含まれる JavaScript オブジェクトです。

11.5.4. Typescript 関数の戻り値

この関数は有効な JavaScript タイプを返すことができます。がそれ以外は戻り値を持たせないようにすることもできます。関数に戻り値が指定されておらず、失敗を指定しないと、呼び出し元は **204 No Content** 応答を受け取ります。

関数は、CloudEvent または **Message** オブジェクトを返してイベントを Knative Eventing システムにプッシュすることもできます。この場合に、開発者は CloudEvent メッセージング仕様の理解や実装は必要ありません。返された値からのヘッダーおよびその他の関連情報は抽出され、応答で送信されます。

例

```
export const handle: Invokable = function (
  context: Context,
  cloudevent?: CloudEvent
```

```
): Message {
  // process customer and return a new CloudEvent
  const customer = cloudevent.data;
  return HTTP.binary(
    new CloudEvent({
      source: 'customer.processor',
      type: 'customer.processed'
    })
  );
};
```

11.5.4.1. 返されるヘッダー

headers プロパティーを **return** オブジェクトに追加して応答ヘッダーを設定できます。これらのヘッダーは抽出され、呼び出し元に応答して送信されます。

応答ヘッダーの例

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer and return custom headers
  const customer = cloudevent.data as Record<string, any>;
  return { headers: { 'customer-id': customer.id } };
}
```

11.5.4.2. 返されるステータスコード

statusCode プロパティーを **return** オブジェクトに追加して、呼び出し元に返されるステータスコードを設定できます。

ステータスコード

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
    return {
      statusCode: 451
    }
  }
  // business logic, then
  return {
    statusCode: 240
  }
}
```

ステータスコードは、関数で作成および出力されるエラーに対して設定することもできます。

エラーステータスコードの例

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, string> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
```

```
    err.statusCode = 451;
    throw err;
  }
}
```

11.5.5. TypeScript 関数のテスト

Typescript 機能は、お使いのコンピューターでローカルでテストできます。**kn func create** を使用した関数の作成時に作成される default プロジェクトには、**test** フォルダーがあり、一部の単純なユニットおよび統合テストが含まれます。

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。
- Knative (**kn**) CLI をインストールしている。
- **kn func create** を使用して関数を作成している。

手順

1. テストを実行していない場合は、最初に依存関係をインストールします。

```
$ npm install
```

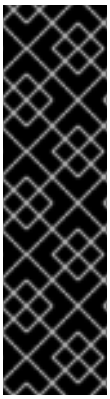
2. 関数の **test** フォルダーに移動します。
3. テストを実行します。

```
$ npm test
```

11.5.6. 次のステップ

- TypeScript コンテキストオブジェクトの参照 ドキュメントを参照してください。
- 関数を構築 して デプロイ します。
- 関数に関するロギングの詳細は、Pino API のドキュメント を参照してください。

11.6. GO 関数の開発



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Go 関数プロジェクトを作成 したら、指定のテンプレートを変更して、関数にビジネスロジックを追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

11.6.1. 前提条件

- 関数を開発する前に、OpenShift Serverless 機能の設定の手順を実行する必要があります。

11.6.2. Go 関数テンプレートの構造

Knative (**kn**) CLI を使用して Go 関数を作成すると、プロジェクトディレクトリーは典型的な Go プロジェクトのようになります。唯一の例外は、イメージの指定に使用される追加の **func.yaml** 設定ファイルです。

Go 関数にはほとんど制限がありません。唯一の要件として、プロジェクトが **function** モジュールで定義する必要があり、**Handle()** 関数をエクスポートする必要があります。

http および **event** トリガー関数のテンプレート構造はいずれも同じです。

テンプレート構造

```
fn
├── README.md
├── func.yaml 1
├── go.mod 2
├── go.sum
├── handle.go
└── handle_test.go
```

1. **func.yaml** 設定ファイルは、イメージ名とレジストリーを判断するために使用されます。
2. 必要な依存関係を **go.mod** ファイルに追加できます。このファイルには、追加のローカル Go ファイルを含めることができます。デプロイメント用にプロジェクトをビルドすると、これらの依存関係は作成したランタイムコンテナーイメージに含まれます。

   依存関係の追加例

   ```
   $ go get gopkg.in/yaml.v2@v2.4.0
   ```

11.6.3. Go 関数の呼び出しについて

Knative (**kn**) CLI を使用して関数プロジェクトを作成する場合に、CloudEvents に応答するプロジェクト、または単純な HTTP 要求に応答するプロジェクトを生成できます。Go 関数は、HTTP 要求、CloudEvent のいずれかでトリガーされるかどうかによって、異なる方法を使用して呼び出されます。

11.6.3.1. HTTP 要求でトリガーされる関数

受信 HTTP リクエストを受信すると、関数は標準の Go コンテキスト を最初のパラメーターとして使用して呼び出され、その後に **http.ResponseWriter** および **http.Request** パラメーターが続きます。標準の Go 手法を使用してリクエストにアクセスし、関数に対応する HTTP レスポンスを設定できます。

HTTP 応答の例

```
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
  // Read body
  body, err := ioutil.ReadAll(req.Body)
  defer req.Body.Close()
  if err != nil {
 http.Error(res, err.Error(), 500)
 return
  }
  // Process body and function logic
  // ...
}
```

11.6.3.2. CloudEvent でトリガーされた関数

受信クラウドイベントが受信されると、そのイベントは CloudEvents Go SDK によって呼び出されます。この呼び出しでは、**Event** タイプをパラメーターとして使用します。

サポート対象の関数署名のリストが示すように、Go Context を関数契約のオプションのパラメーターとして使用できます。

サポート対象の関数署名

```
Handle()
Handle() error
Handle(context.Context)
Handle(context.Context) error
Handle(cloudevents.Event)
Handle(cloudevents.Event) error
Handle(context.Context, cloudevents.Event)
Handle(context.Context, cloudevents.Event) error
Handle(cloudevents.Event) *cloudevents.Event
Handle(cloudevents.Event) (*cloudevents.Event, error)
Handle(context.Context, cloudevents.Event) *cloudevents.Event
Handle(context.Context, cloudevents.Event) (*cloudevents.Event, error)
```

11.6.3.2.1. CloudEvent トリガーの例

CloudEvent が受信され、これには data プロパティーに JSON 文字列が含まれます。

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

このデータにアクセスするには、CloudEvent データのプロパティーをマッピングし、受信イベントからデータを取得する構造を定義する必要があります。以下の例では、**Purchase** 構造を使用します。

```
type Purchase struct {
  CustomerId string `json:"customerId"`
  ProductId  string `json:"productId"`
}
func Handle(ctx context.Context, event cloudevents.Event) (err error) {

  purchase := &Purchase{}
```

```
  if err = event.DataAs(purchase); err != nil {
 fmt.Fprintf(os.Stderr, "failed to parse incoming CloudEvent %s\n", err)
 return
  }
  // ...
}
```

または、Go **encoding/json** パッケージを使用して、バイトアレイ形式で直接 JSON として CloudEvent にアクセスできます。

```
func Handle(ctx context.Context, event cloudevents.Event) {
  bytes, err := json.Marshal(event)
  // ...
}
```

11.6.4. Go 関数の戻り値

HTTP リクエストによってトリガーされる関数は、レスポンスを直接設定できます。Go http.ResponseWriter を使用して、これを行うように関数を設定できます。

HTTP 応答の例

```
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
  // Set response
  res.Header().Add("Content-Type", "text/plain")
  res.Header().Add("Content-Length", "3")
  res.WriteHeader(200)
  _, err := fmt.Fprintf(res, "OK\n")
  if err != nil {
 fmt.Fprintf(os.Stderr, "error or response write: %v", err)
  }
}
```

CloudEvent によってトリガーされる関数は、何も返さないか、**error** または **CloudEvent** を返し、Knative Eventing システムにイベントをプッシュする場合があります。この場合、CloudEvent に一意の **ID**、適切な **ソース** および **種別** を設定する必要があります。データは、定義した構造または **マップ** から入力できます。

CloudEvent 応答の例

```
func Handle(ctx context.Context, event cloudevents.Event) (resp *cloudevents.Event, err error) {
  // ...
  response := cloudevents.NewEvent()
  response.SetID("example-uuid-32943bac6fea")
  response.SetSource("purchase/getter")
  response.SetType("purchase")
  // Set the data from Purchase type
  response.SetData(cloudevents.ApplicationJSON, Purchase{
 CustomerId: custId,
 ProductId:  prodId,
  })
  // OR set the data directly from map
  response.SetData(cloudevents.ApplicationJSON, map[string]string{"customerId": custId, "productId":
prodId})
```

```
    // Validate the response
    resp = &response
    if err = resp.Validate(); err != nil {
   fmt.Printf("invalid event created. %v", err)
    }
    return
  }
```

11.6.5. Go 関数のテスト

Go 機能は、お使いのコンピューターのローカルでテストできます。**kn func create** を使用した関数の作成時に作成される default プロジェクトには、一部の基本的なテストが含まれる **handle_test.go** ファイルがあります。これらのテストは、必要に応じて拡張できます。

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。
- Knative (**kn**) CLI をインストールしている。
- **kn func create** を使用して関数を作成している。

手順

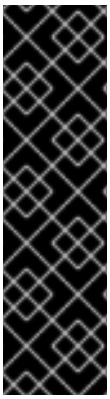
1. 関数の **test** フォルダーに移動します。
2. テストを実行します。

```
$ go test
```

11.6.6. 次のステップ

- 関数を構築 して デプロイ します。

11.7. PYTHON 関数の開発



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Python 関数プロジェクトを作成 したら、指定したテンプレートファイルを変更して、ビジネスロジックを機能に追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

11.7.1. 前提条件

- 関数を開発する前に、OpenShift Serverless 機能の設定の手順を実行する必要があります。

11.7.2. Python 関数テンプレート構造

Knative (**kn**) CLI を使用して Python 機能を作成する場合の、プロジェクトディレクトリーは通常の Python プロジェクトに似ています。Python 関数にはいくつかの制限があります。プロジェクトの要件として唯一、**main()** 関数と **func.yaml** 設定ファイルで設定される **func.py** が含まれることが挙げられます。

開発者は、テンプレート **requirements.txt** ファイルにある依存関係しか使用できないわけではありません。その他の依存関係は、他の Python プロジェクトに配置されるように追加できます。デプロイメント用にプロジェクトをビルドすると、これらの依存関係は作成したランタイムコンテナーイメージに含まれます。

http および **event** トリガー関数のテンプレート構造はいずれも同じです。

テンプレート構造

```
fn
├── func.py 1
├── func.yaml 2
├── requirements.txt 3
└── test_func.py 4
```

1. **main()** 関数が含まれます。
2. イメージ名とレジストリーを判断するために使用されます。
3. その他の依存関係は、他の Python プロジェクトにあるため、**requirements.txt** ファイルに追加できます。
4. 関数のローカルでのテストに使用できる単純なユニットテストが含まれます。

11.7.3. Python 関数の呼び出しについて

Python 関数は、単純な HTTP 要求で呼び出すことができます。受信要求を受け取ると、関数は **context** オブジェクトで最初のパラメーターとして呼び出されます。

context オブジェクトは、2 つの属性を持つ Python クラスです。

- **request** 属性。この属性常に存在し、Flask **request** オブジェクトが含まれます。
- 2 番目の属性 **cloud_event**。受信した要求が **CloudEvent** オブジェクトの場合に設定されます。

開発者はコンテキストオブジェクトから **CloudEvent** データすべてにアクセスできます。

コンテキストオブジェクトの例

```
def main(context: Context):
    """
     The context parameter contains the Flask request object and any
```

```
    CloudEvent received with the request.
    """
    print(f"Method: {context.request.method}")
    print(f"Event data {context.cloud_event.data}")
    # ... business logic here
```

11.7.4. Python 関数の戻り値

関数は、Flask でサポートされている任意の値を返すことができます。これは、呼び出しフレームワークがこれらの値を Flask サーバーに直接プロキシーするためです。

例

```
def main(context: Context):
    body = { "message": "Howdy!" }
    headers = { "content-type": "application/json" }
    return body, 200, headers
```

関数は、関数呼び出しの 2 番目および 3 番目の応答値として、ヘッダーと応答コードの両方を設定できます。

11.7.4.1. Returning CloudEvents

開発者は **@event** デコレーターを使用して、呼び出し元に対して、応答を送信する前に関数の戻り値を CloudEvent に変換する必要があることを指示できます。

例

```
@event("event_source"="/my/function", "event_type"="my.type")
def main(context):
    # business logic here
    data = do_something()
    # more data processing
    return data
```

この例では、タイプが **"my.type"**、ソースが **"/my/function"** の応答値として CloudEvent を送信します。CloudEvent **data** プロパティー は、返された **data** 変数に設定されます。**event_source** および **event_type** デコレーター属性は任意です。

11.7.5. Python 関数のテスト

Python 機能は、お使いのコンピューターのローカルにテストできます。デフォルトプロジェクトには、**test_func.py** ファイルが含まれており、関数の単純なユニットテストを提供します。

注記

Python 関数のデフォルトのテストフレームワークは **unittest** です。必要に応じて、別のテストフレームワークを使用できます。

前提条件

- Python 関数テストをローカルで実行するには、必要な依存関係をインストールする必要があります。

```
$ pip install -r requirements.txt
```

手順

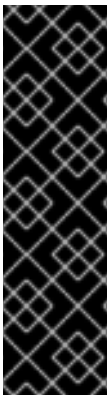
1. **test_func.py** ファイルが含まれる関数のフォルダーに移動します。
2. テストを実行します。

```
$ python3 test_func.py
```

11.7.6. 次のステップ

- 関数を構築 して デプロイ します。

11.8. QUARKUS 関数の開発



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Quarkus 関数プロジェクトを作成 したら、指定のテンプレートを変更して、関数にビジネスロジックを追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

11.8.1. 前提条件

- 関数を開発する前に、OpenShift Serverless 機能の設定の設定手順を完了する必要があります。

11.8.2. Quarkus 関数テンプレートの構造

Knative (**kn**) CLI を使用して Quarkus 機能を作成する場合の、プロジェクトディレクトリーは通常の Maven プロジェクトに似ています。さらに、プロジェクトには、関数の設定に使用される **func.yaml** ファイルが含まれています。

http および **event** トリガー関数のテンプレート構造はいずれも同じです。

テンプレート構造

```
.
├── func.yaml 1
├── mvnw
├── mvnw.cmd
├── pom.xml 2
├── README.md
```

```
└── src
    ├── main
    │   ├── java
    │   │   └── functions
    │   │       ├── Function.java 3
    │   │       ├── Input.java
    │   │       └── Output.java
    │   └── resources
    │       └── application.properties
    └── test
        └── java
            └── functions 4
                ├── FunctionTest.java
                └── NativeFunctionIT.java
```

1 イメージ名とレジストリーを判断するために使用されます。

2 プロジェクトオブジェクトモデル (POM) ファイルには、依存関係に関する情報などのプロジェクト設定が含まれています。このファイルを変更して、別の依存関係を追加できます。

追加の依存関係の例

```
...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>3.8.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
...
```

依存関係は、最初のコンパイル時にダウンロードされます。

3 関数プロジェクトには、**@Funq** アノテーションが付けられた Java メソッドが含まれている必要があります。このメソッドは **Function.java** クラスに配置できます。

4 関数のローカルでのテストに使用できる単純なテストケースが含まれます。

11.8.3. Quarkus 関数の呼び出しについて

CloudEvents に応答する Quarkus プロジェクトや、簡単な HTTP 要求に応答する Quarkus プロジェクトを作成できます。Knative の CloudEvents は HTTP 経由で POST 要求として転送されるため、いずれかの関数タイプは受信 HTTP 要求をリッスンして応答します。

受信要求が受信されると、Quarkus 関数は使用可能なタイプのインスタンスと合わせて呼び出されます。

表11.1 関数呼び出しオプション

呼び出しメソッド	インスタンスに含まれるデータタイプ	データの例
HTTP POST 要求	要求のボディーに含まれる JSON オブジェクト	**{ "customerId": "0123456", "productId": "6543210" }**
HTTP GET 要求	クエリー文字列のデータ	**?customerId=0123456&productId=6543210**
CloudEvent	**data** プロパティーの JSON オブジェクト	**{ "customerId": "0123456", "productId": "6543210" }**

以下の例は、以前の表に記載されている **customerId** および **productId** の購入データを受信して処理する関数です。

Quarkus 関数の例

```
public class Functions {
    @Funq
    public void processPurchase(Purchase purchase) {
        // process the purchase
    }
}
```

購入データが含まれる、該当の **Purchase** JavaBean クラスは以下のようになります。

クラスの例

```
public class Purchase {
    private long customerId;
    private long productId;
    // getters and setters
}
```

11.8.3.1. StorageLocation の例

以下のコード例は、**withBeans**、**withCloudEvent**、および **withBinary** の 3 つの関数を定義します。

例

```
import io.quarkus.funqy.Funq;
import io.quarkus.funqy.knative.events.CloudEvent;

public class Input {
    private String message;

    // getters and setters
}
```

```
public class Output {
    private String message;

    // getters and setters
}

public class Functions {
    @Funq
    public Output withBeans(Input in) {
        // function body
    }

    @Funq
    public CloudEvent<Output> withCloudEvent(CloudEvent<Input> in) {
        // function body
    }

    @Funq
    public void withBinary(byte[] in) {
        // function body
    }
}
```

Functions クラスの **withBeans** 機能は、以下にで呼び出すことができます。

- JSON ボディーが含まれる HTTP POST 要求:

```
$ curl "http://localhost:8080/withBeans" -X POST \
    -H "Content-Type: application/json" \
    -d '{"message": "Hello there."}'
```

- クエリーパラメーターが含まれる HTTP GET 要求:

```
$ curl "http://localhost:8080/withBeans?message=Hello%20there." -X GET
```

- バイナリーエンコーディングの **CloudEvent** オブジェクト:

```
$ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/json" \
  -H "Ce-SpecVersion: 1.0" \
  -H "Ce-Type: withBeans" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  -d '{"message": "Hello there."}'
```

- 構造化されたエンコーディングでの **CloudEvent** オブジェクト:

```
$ curl http://localhost:8080/ \
    -H "Content-Type: application/cloudevents+json" \
    -d '{ "data": {"message":"Hello there."},
          "datacontenttype": "application/json",
          "id": "42",
          "source": "curl",
          "type": "withBeans",
          "specversion": "1.0"}'
```

Functions クラスの **withCloudEvent** 機能は、**withBeans** 関数と同様に **CloudEvent** オブジェクトを使用して呼び出すことができます。ただし、**withBeans** とは異なり、**withCloudEvent** はプレーン HTTP 要求で呼び出すことはできません。

Functions クラスの **withBinary** 関数は、以下にで呼び出すことができます。

- バイナリーエンコーディングの **CloudEvent** オブジェクト:

```
$ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/octet-stream" \
  -H "Ce-SpecVersion: 1.0"\
  -H "Ce-Type: withBinary" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  --data-binary '@img.jpg'
```

- 構造化されたエンコーディングでの **CloudEvent** オブジェクト:

```
$ curl http://localhost:8080/ \
  -H "Content-Type: application/cloudevents+json" \
  -d "{ \"data_base64\": \"$(base64 --wrap=0 img.jpg)\",
       \"datacontenttype\": \"application/octet-stream\",
       \"id\": \"42\",
       \"source\": \"curl\",
       \"type\": \"withBinary\",
       \"specversion\": \"1.0\"}"
```

11.8.4. CloudEvent 属性

type または **subject** など CloudEvent の属性を読み取るか、書き込む必要がある場合は、**CloudEvent<T>** 汎用インターフェイスおよび **CloudEventBuilder** ビルダーを使用できます。**<T>** タイプパラメーターは使用可能なタイプのいずれかでなければなりません。

以下の例では、**CloudEventBuilder** を使用して、購入処理の成功または失敗を返します。

```
public class Functions {

    private boolean _processPurchase(Purchase purchase) {
        // do stuff
    }

    public CloudEvent<Void> processPurchase(CloudEvent<Purchase> purchaseEvent) {
        System.out.println("subject is: " + purchaseEvent.subject());

        if (!_processPurchase(purchaseEvent.data())) {
            return CloudEventBuilder.create()
                    .type("purchase.error")
                    .build();
        }
        return CloudEventBuilder.create()
                .type("purchase.success")
                .build();
    }
}
```

11.8.5. Quarkus 関数の戻り値

関数は、許可された型のリストから任意の型のインスタンスを返すことができます。または、**Uni<T>** 型を返すこともできます。ここで、**<T>** 型パラメーターは、許可されている型の任意の型にすることができます。

Uni<T> タイプは、返されるオブジェクトが受信したオブジェクトと同じ形式でシリアライズされるため、関数が非同期 API を呼び出す場合に便利です。以下に例を示します。

- 関数が HTTP 要求を受信すると、返されるオブジェクトが HTTP 応答のボディーに送信されます。
- 関数がバイナリーエンコーディングで **CloudEvent** オブジェクトを受信する場合に、返されるオブジェクトはバイナリーエンコードされた **CloudEvent** オブジェクトの data プロパティーで送信されます。

以下の例は、購入リストを取得する関数を示しています。

コマンドの例

```
public class Functions {
    @Funq
    public List<Purchase> getPurchasesByName(String name) {
      // logic to retrieve purchases
    }
}
```

- HTTP 要求経由でこの関数を呼び出すと、応答のボディーに購入された一覧が含まれる HTTP 応答が生成されます。
- 受信 **CloudEvent** オブジェクト経由でこの関数を呼び出すと、**data** プロパティーの購入リストが含まれる **CloudEvent** 応答が生成されます。

11.8.5.1. 使用可能なタイプ

関数の入力と出力は、**void**、**String**、または **byte[]** 型のいずれかです。さらに、プリミティブ型とそのラッパー (**int** や **Integer** など) にすることもできます。これらは、Javabean、マップ、リスト、配列、および特殊な **CloudEvents<T>** タイプの複合オブジェクトにすることもできます。

マップ、リスト、配列、**CloudEvents<T>** 型の **<T>** 型パラメーター、および Javabeans の属性は、ここにリストされている型のみにすることができます。

例

```
public class Functions {
    public List<Integer> getIds();
    public Purchase[] getPurchasesByName(String name);
    public String getNameById(int id);
    public Map<String,Integer> getNameIdMapping();
    public void processImage(byte[] img);
}
```

11.8.6. Quarkus 関数のテスト

Quarkus 関数は、コンピューターに対してローカルでテストできます。**kn func create** を使用して関数を作成するときに作成されるデフォルトプロジェクトには、基本的な Maven テストを含む **src/test/** ディレクトリーがあります。これらのテストは、必要に応じて拡張できます。

前提条件

- Quarkus 関数を作成している。
- Knative (**kn**) CLI をインストールしている。

手順

1. 関数のプロジェクトフォルダーに移動します。
2. Maven テストを実行します。

```
$ ./mvnw test
```

11.8.7. 次のステップ

- 関数を 構築 して デプロイ します。

11.9. FUNC.YAML の関数プロジェクト設定

func.yaml ファイルには、関数プロジェクトの設定が含まれます。**kn func** コマンドを実行すると、**func.yaml** に指定された値が使用されます。たとえば、**kn func build** コマンドを実行すると、**build** フィールドの値が使用されます。一部のケースでは、この値はコマンドラインフラグまたは環境変数で上書きできます。

11.9.1. func.yaml の設定可能なフィールド

func.yaml のフィールドの多くは、関数の作成、ビルド、およびデプロイ時に自動的に生成されます。ただし、関数名またはイメージ名などの変更用に手動で変更するフィールドもあります。

11.9.1.1. buildEnvs

buildEnvs フィールドを使用すると、関数をビルドする環境で利用できる環境変数を設定できます。**envs** を使用して設定する変数とは異なり、**buildEnv** を使用して設定する変数は、関数の実行時には使用できません。

buildEnv 変数を値から直接設定できます。以下の例では、**EXAMPLE1** という名前の **buildEnv** 変数に値 **one** が直接割り当てられます。

```
buildEnvs:
- name: EXAMPLE1
  value: one
```

また、ローカルの環境変数から **buildEnv** 変数を設定することもできます。以下の例では、**EXAMPLE2** という名前の **buildEnv** 変数にローカル環境変数 **LOCAL_ENV_VAR** の値が割り当てられます。

```
buildEnvs:
- name: EXAMPLE1
  value: '{{ env:LOCAL_ENV_VAR }}'
```

11.9.1.2. envs

envs フィールドを使用すると、ランタイム時に関数で使用できるように環境変数を設定できます。環境変数は、複数の異なる方法で設定できます。

1. 値から直接設定します。
2. ローカル環境変数に割り当てられた値から設定します。詳細は、func.yaml フィールドからのローカル環境変数の参照のセクションを参照してください。
3. シークレットまたは設定マップに格納されているキーと値のペアから設定します。
4. 作成された環境変数の名前として使用されるキーを使用して、シークレットまたは設定マップに格納されているすべてのキーと値のペアをインポートすることもできます。

以下の例は、環境変数を設定するさまざまな方法を示しています。

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE1 (1)
  value: value
- name: EXAMPLE2 (2)
  value: '{{ env:LOCAL_ENV_VALUE }}'
- name: EXAMPLE3 (3)
  value: '{{ secret:mysecret:key }}'
- name: EXAMPLE4 (4)
  value: '{{ configMap:myconfigmap:key }}'
- value: '{{ secret:mysecret2 }}' (5)
- value: '{{ configMap:myconfigmap2 }}' (6)
```

(1) 値から直接設定された環境変数。

(2) ローカル環境変数に割り当てられた値から設定された環境変数。

(3) シークレットに格納されているキーと値のペアから割り当てられた環境変数。

(4) 設定マップに保存されるキーと値のペアから割り当てられる環境変数。

(5) シークレットのキーと値のペアからインポートされた環境変数のセット。

(6) 設定マップのキーと値のペアからインポートされた環境変数のセット。

11.9.1.3. builder

builder フィールドは、機能がイメージを構築するために使用する戦略を指定します。**pack** または **s2i** の値を受け入れます。

11.9.1.4. build

build フィールドは、機能を構築する方法を示します。値 **local** は、機能がマシン上でローカルに構築されていることを示します。値 **git** は、機能が **git** フィールドで指定された値を使用してクラスター上に構築されていることを示します。

11.9.1.5. volumes

以下の例のように、**volumes** フィールドを使用すると、指定したパスで関数にアクセスできるボリュームとしてシークレットと設定マップをマウントできます。

```
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret (1)
  path: /workspace/secret
- configMap: myconfigmap (2)
  path: /workspace/configmap
```

1. **mysecret** シークレットは、**/workspace/secret** にあるボリュームとしてマウントされます。
2. **myconfigmap** 設定マップは、**/workspace/configmap** にあるボリュームとしてマウントされます。

11.9.1.6. オプション

options フィールドを使用すると、自動スケーリングなど、デプロイされた関数の Knative Service プロパティーを変更できます。これらのオプションが設定されていない場合は、デフォルトのオプションが使用されます。

これらのオプションを利用できます。

- **scale**
  - **min**: レプリカの最小数。負ではない整数でなければなりません。デフォルトは 0 です。
  - **max**: レプリカの最大数。負ではない整数でなければなりません。デフォルトは 0 で、これは制限がないことを意味します。
  - **metric**: Autoscaler によって監視されるメトリクスタイプを定義します。これは、デフォルトの **concurrency**、または **rps** に設定できます。
  - **target**: 同時に受信する要求の数に基づくスケールアップのタイミングの推奨。**target** オプションは、0.01 より大きい浮動小数点値を指定できます。**options.resources.limits.concurrency** が設定されていない限り、デフォルトは 100 になります。この場合、**target** はデフォルトでその値になります。
  - **utilization**: スケールアップする前に許可された同時リクエスト使用率のパーセンテージ。1 から 100 までの浮動小数点値を指定できます。デフォルトは 70 です。
- **resources**
  - **requests**
    - **cpu**: デプロイされた関数を持つコンテナーの CPU リソース要求。

- **memory**: デプロイされた関数を持つコンテナーのメモリーリソース要求。

- **limits**
  - **cpu**: デプロイされた関数を持つコンテナーの CPU リソース制限。
  - **memory**: デプロイされた関数を持つコンテナーのメモリーリソース制限。
  - **concurrency**: 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。0 以上の整数値を指定できます。デフォルトは 0 です (制限なしを意味します)。

これは、**scale** オプションの設定例です。

```
name: test
namespace: ""
runtime: go
...
options:
  scale:
    min: 0
    max: 10
    metric: concurrency
    target: 75
    utilization: 75
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 1000m
      memory: 256Mi
      concurrency: 100
```

11.9.1.7. image

image フィールドは、関数がビルドされた後の関数のイメージ名を設定します。このフィールドは必要に応じて変更できます。変更する場合、次に **kn func build** または **kn func deploy** を実行すると、関数イメージは新しい名前で作成されます。

11.9.1.8. imageDigest

imageDigest フィールドには、関数のデプロイ時のイメージマニフェストの SHA256 ハッシュが含まれます。この値は変更しないでください。

11.9.1.9. labels

labels フィールドを使用すると、デプロイされた関数にラベルを設定できます。

値から直接ラベルを設定できます。以下の例では、**role** キーを持つラベルに **backend** の値が直接割り当てられます。

```
labels:
- key: role
  value: backend
```

ローカル環境変数からラベルを設定することもできます。以下の例では、**author** キーの付いたラベルに **USER** ローカル環境変数の値が割り当てられます。

```
labels:
- key: author
  value: '{{ env:USER }}'
```

11.9.1.10. name

name フィールドは、関数の名前を定義します。この値は、デプロイ時に Knative サービスの名前として使用されます。このフィールドを変更して、後続のデプロイメントで関数の名前を変更できます。

11.9.1.11. namespace

namespace フィールドは、関数がデプロイされる namespace を指定します。

11.9.1.12. runtime

runtime フィールドは、関数の言語ランタイムを指定します (例: **python**)。

11.9.2. func.yaml フィールドからのローカル環境変数の参照

API キーなどの機密情報を関数設定に保存したくない場合は、ローカル環境で使用可能な環境変数への参照を追加できます。これを行うには、**func.yaml** ファイルの **envs** フィールドを変更します。

前提条件

- 関数プロジェクトを作成する必要があります。
- ローカル環境には、参照する変数が含まれている必要があります。

手順

- ローカル環境変数を参照するには、以下の構文を使用します。

  ```
  {{ env:ENV_VAR }}
  ```

  ENV_VAR を、使用するローカル環境の変数の名前に置き換えます。

  たとえば、ローカル環境で **API_KEY** 変数が利用可能な場合があります。その値を **MY_API_KEY** 変数に割り当てることができます。これにより、関数内で直接使用できます。

  関数の例

  ```
  name: test
  namespace: ""
  runtime: go
  ...
  envs:
  - name: MY_API_KEY
    value: '{{ env:API_KEY }}'
  ...
  ```

11.9.3. 関連情報

- 関数を使い始める
- Serverless 関数からのシークレットおよび設定マップへのアクセス
- 自動スケーリングに関する Knative ドキュメント
- コンテナーのリソースの管理に関する Kubernetes のドキュメント
- 並行性の設定に関する Knative ドキュメント

11.10. 関数からのシークレットおよび設定マップへのアクセス

関数がクラスターにデプロイされた後に、それらはシークレットおよび設定マップに保存されているデータにアクセスできます。このデータはボリュームとしてマウントすることも、環境変数に割り当てることもできます。Knative CLI を使用して、このアクセスを対話的に設定するか、関数設定 YAML ファイルを編集して手動で設定できます。

重要

シークレットおよび設定マップにアクセスするには、関数をクラスターにデプロイする必要があります。この機能は、ローカルで実行している関数では利用できません。

シークレットまたは設定マップの値にアクセスできない場合、デプロイメントは失敗し、アクセスできない値を指定するエラーメッセージが表示されます。

11.10.1. シークレットおよび設定マップへの関数アクセスの対話的な変更

kn func config 対話型ユーティリティーを使用して、関数がアクセスするシークレットおよび設定マップを管理できます。使用可能な操作には、config map とシークレットに環境変数として保存されている値の一覧表示、追加、および削除、およびボリュームの一覧表示、追加、および削除が含まれます。この機能を使用すると、クラスターに保存されているどのデータを関数からアクセスできるかを管理できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数プロジェクトディレクトリーで以下のコマンドを実行します。

```
$ kn func config
```

   あるいは、**--path** または **-p** オプションを使用して、関数プロジェクトディレクトリーを指定できます。

2. 対話型インターフェイスを使用して必要な操作を実行します。たとえば、ユーティリティーを使用して設定したボリュームの一覧を表示すると、以下のような出力が生成されます。

```
$ kn func config
? What do you want to configure? Volumes
? What operation do you want to perform? List
Configured Volumes mounts:
- Secret "mysecret" mounted at path: "/workspace/secret"
- Secret "mysecret2" mounted at path: "/workspace/secret2"
```

このスキームは、対話型ユーティリティーで利用可能なすべての操作と、それらに移動する方法を示しています。

```
kn func config
  ├─> Environment variables
  │             ├─> Add
  │             │    ├─> ConfigMap: Add all key-value pairs from a config map
  │             │    ├─> ConfigMap: Add value from a key in a config map
  │             │    ├─> Secret: Add all key-value pairs from a secret
  │             │    └─> Secret: Add value from a key in a secret
  │             ├─> List: List all configured environment variables
  │             └─> Remove: Remove a configured environment variable
  └─> Volumes
        ├─> Add
        │    ├─> ConfigMap: Mount a config map as a volume
        │    └─> Secret: Mount a secret as a volume
        ├─> List: List all configured volumes
        └─> Remove: Remove a configured volume
```

3. オプション。変更を反映させるため、関数をデプロイします。

```
$ kn func deploy -p test
```

11.10.2. 特殊なコマンドを使用したシークレットおよび設定マップへの関数アクセスの対話的な変更

kn func config ユーティリティーを実行するたびにダイアログ全体を移動して、直前のセクションで示されているように、必要な操作を選択する必要があります。ステップを保存するには、**kn func config** コマンドのより具体的なフォームを実行することで、特定の操作を直接実行します。

- 設定した環境変数を一覧表示するには、以下を実行します。

```
$ kn func config envs [-p <function-project-path>]
```

- 関数設定に環境変数を追加するには、以下を実行します。

```
$ kn func config envs add [-p <function-project-path>]
```

- 関数設定から環境変数を削除するには、以下を実行します。

```
$ kn func config envs remove [-p <function-project-path>]
```

- 設定したボリュームを一覧表示するには、以下を実行します。

```
$ kn func config volumes [-p <function-project-path>]
```

- 関数設定にボリュームを追加するには、以下を実行します。

```
$ kn func config volumes add [-p <function-project-path>]
```

- 関数設定からボリュームを削除するには、以下を実行します。

```
$ kn func config volumes remove [-p <function-project-path>]
```

11.10.3. シークレットおよび設定マップへの関数アクセスの手動による追加

シークレットおよび設定マップにアクセスするための設定を手動で関数に追加できます。これは、既存の設定スニペットがある場合などに、**kn func config** 対話型ユーティリティーとコマンドを使用するよりも望ましい場合があります。

11.10.3.1. シークレットのボリュームとしてのマウント

シークレットをボリュームとしてマウントできます。シークレットがマウントされると、関数から通常のファイルとしてアクセスできます。これにより、関数がアクセスする必要がある URI のリストなど、関数が必要とするデータをクラスターに格納できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数の **func.yaml** ファイルを開きます。
2. ボリュームとしてマウントするシークレットごとに、以下の YAML を **volumes** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret
  path: /workspace/secret
```

   - **mysecret** をターゲットシークレットの名前に置き換えます。
   - **/workspace/secret** は、シークレットをマウントするパスに置き換えます。
   たとえば、**addresses** シークレットをマウントするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
```

```
volumes:
- configMap: addresses
  path: /workspace/secret-addresses
```

3. 設定を保存します。

11.10.3.2. 設定マップのボリュームとしてのマウント

設定マップをボリュームとしてマウントできます。設定マップがマウントされると、関数から通常のファイルとしてアクセスできます。これにより、関数がアクセスする必要がある URI のリストなど、関数が必要とするデータをクラスターに格納できます。

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。
- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数の **func.yaml** ファイルを開きます。
2. ボリュームとしてマウントする設定マップごとに、以下の YAML を **volumes** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: myconfigmap
  path: /workspace/configmap
```

- **myconfigmap** をターゲット設定マップの名前に置き換えます。
- **/workspace/configmap** は、設定マップをマウントするパスに置き換えます。
  たとえば、**addresses** config map をマウントするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/configmap-addresses
```

3. 設定を保存します。

11.10.3.3. シークレットで定義されるキー値からの環境変数の設定

シークレットとして定義されたキー値から環境変数を設定できます。以前にシークレットに保存された値は、実行時に関数によって環境変数としてアクセスできます。これは、ユーザーの ID など、シークレットに格納されている値にアクセスする場合に役立ちます。

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。
- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数の **func.yaml** ファイルを開きます。
2. 環境変数に割り当てる秘密鍵と値のペアからの値ごとに、以下の YAML を **envs** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ secret:mysecret:key }}'
```

   - **EXAMPLE** を環境変数の名前に置き換えます。
   - **mysecret** をターゲットシークレットの名前に置き換えます。
   - **key** をターゲット値にマッピングしたキーに置き換えます。
   たとえば、**userdetailssecret** に保存されているユーザー ID にアクセスするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret:userid }}'
```

3. 設定を保存します。

11.10.3.4. 設定マップで定義されるキー値からの環境変数の設定

config map として定義されたキー値から環境変数を設定できます。以前に config map に格納された値は、実行時に関数によって環境変数としてアクセスできます。これは、ユーザーの ID など、config map に格納されている値にアクセスするのに役立ちます。

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。

- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数の **func.yaml** ファイルを開きます。
2. 環境変数に割り当てる設定マップのキーと値のペアからの値ごとに、以下の YAML を **envs** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ configMap:myconfigmap:key }}'
```

   - **EXAMPLE** を環境変数の名前に置き換えます。
   - **myconfigmap** をターゲット設定マップの名前に置き換えます。
   - **key** をターゲット値にマッピングしたキーに置き換えます。
   たとえば、**userdetailsmap** に格納されているユーザー ID にアクセスするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap:userid }}'
```

3. 設定を保存します。

11.10.3.5. シークレットで定義されたすべての値からの環境変数の設定

シークレットで定義されているすべての値から環境変数を設定できます。以前にシークレットに保存された値は、実行時に関数によって環境変数としてアクセスできます。これは、シークレットに格納されている値のコレクション (ユーザーに関する一連のデータなど) に同時にアクセスする場合に役立ちます。

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。
- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数の **func.yaml** ファイルを開きます。

2. すべてのキーと値のペアを環境変数としてインポートするすべてのシークレットについて、以下の YAML を **envs** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ secret:mysecret }}' 1
```

1 **mysecret** をターゲットシークレットの名前に置き換えます。

たとえば、**userdetailssecret** に保存されているすべてのユーザー データにアクセスするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret }}'
```

3. 設定を保存します。

11.10.3.6. 設定マップで定義されたすべての値からの環境変数の設定

config map で定義されたすべての値から環境変数を設定できます。以前に config map に格納された値は、実行時に関数によって環境変数としてアクセスできます。これは、config map に格納されている値のコレクション (ユーザーに関する一連のデータなど) に同時にアクセスする場合に役立ちます。

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。
- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数の **func.yaml** ファイルを開きます。
2. すべてのキーと値のペアを環境変数としてインポートするすべての設定マップについて、以下の YAML を **envs** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:myconfigmap }}' 1
```

1 **myconfigmap** をターゲット設定マップの名前に置き換えます。

たとえば、**userdetailsmap** に保存されているすべてのユーザー データにアクセスするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap }}'
```

3. ファイルを保存します。

11.11. アノテーションの関数への追加

デプロイされたサーバーレス機能に Kubernetes アノテーションを追加できます。注釈を使用すると、関数の目的に関するメモなど、任意のメタデータを関数に添付できます。注釈は、**func.yaml** 設定ファイルの **annotations** セクションに追加されます。

関数アノテーション機能には、以下の 2 つの制限があります。

- 関数アノテーションがクラスター上の対応する Knative サービスに伝播されると、**func.yaml** ファイルから削除することでサービスから削除することはできません。サービスの YAML ファイルを直接変更するか、または OpenShift Container Platform Web コンソールを使用して、Knative サービスからアノテーションを削除する必要があります。
- **autoscaling** アノテーションなど、Knative によって設定されるアノテーションを設定することはできません。

11.11.1. 関数へのアノテーションの追加

関数にアノテーションを追加できます。ラベルと同様に、アノテーションはキーと値のマップとして定義されます。アノテーションは、関数の作成者など、関数に関するメタデータを提供する場合などに役立ちます。

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。
- Knative (**kn**) CLI をインストールしている。
- 関数を作成している。

手順

1. 関数の **func.yaml** ファイルを開きます。
2. 追加するすべてのアノテーションについて、以下の YAML を **annotations** セクションに追加します。

```
name: test
namespace: ""
```

```
runtime: go
...
annotations:
  <annotation_name>: "<annotation_value>" (1)
```

(1) **<annotation_name>: "<annotation_value>"** をお使いのアノテーションに置き換えます。

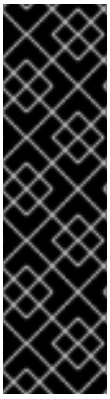
たとえば、関数が Alice によって作成者されたことを示すには、以下のアノテーションを含めることができます。

```
name: test
namespace: ""
runtime: go
...
annotations:
  author: "alice@example.com"
```

3. 設定を保存します。

次に関数をクラスターにデプロイすると、アノテーションが対応する Knative サービスに追加されます。

11.12. 関数開発リファレンスガイド



重要

OpenShift Serverless Functions 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 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。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関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

OpenShift Serverless Functions は、基本的な関数を作成するために使用できるテンプレートを提供します。テンプレートは、関数プロジェクトのボイラープレートを開始し、**kn func** ツールで使用できるように準備します。各関数テンプレートは特定のランタイム用に調整されており、その規則に従います。テンプレートを使用すると、関数プロジェクトを自動的に開始できます。

次のランタイムのテンプレートが利用可能です。

- Node.js
- Python
- Go
- Quarkus
- TypeScript

11.12.1. Node.js コンテキストオブジェクトのリファレンス

context オブジェクトには、関数開発者が利用可能なプロパティーが複数あります。これらのプロパティーにアクセスすると、HTTP 要求に関する情報が提供され、出力がクラスターログに書き込まれます。

11.12.1.1. log

出力をクラスターロギングに書き込むために使用可能なロギングオブジェクトを提供します。ログは Pino logging API に準拠します。

ログの例

```
function handle(context) {
  context.log.info(“Processing customer”);
}
```

kn func invoke コマンドを使用して、この関数にアクセスできます。

コマンドの例

```
$ kn func invoke --target 'http://example.function.com'
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Pr
ocessing customer"}
```

ログレベルは、**fatal**、**error**、**warn**、**info**、**debug**、**trace**、または **silent** のいずれかに設定できます。これを実行するには、**config** コマンドを使用してこれらの値のいずれかを環境変数 **FUNC_LOG_LEVEL** に割り当てて、**logLevel** の値を変更します。

11.12.1.2. query

要求のクエリー文字列 (ある場合) をキーと値のペアとして返します。これらの属性はコンテキストオブジェクト自体にも表示されます。

サンプルクエリー

```
function handle(context) {
  // Log the 'name' query parameter
  context.log.info(context.query.name);
  // Query parameters are also attached to the context
  context.log.info(context.name);
}
```

kn func invoke コマンドを使用して、この関数にアクセスできます。

コマンドの例

```
$ kn func invoke --target 'http://example.com?name=tiger'
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tig
er"}
```

11.12.1.3. ボディー

要求ボディー (ある場合) を返します。要求ボディーに JSON コードが含まれる場合には、属性は直接利用できるように解析されます。

ボディーの例

```
function handle(context) {
  // log the incoming request body's 'hello' parameter
  context.log.info(context.body.hello);
}
```

curl コマンドを使用してこの関数を呼び出すことができます。

コマンドの例

```
$ kn func invoke -d '{"Hello": "world"}'
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"w
orld"}
```

11.12.1.4. ヘッダー

HTTP 要求ヘッダーをオブジェクトとして返します。

ヘッダーの例

```
function handle(context) {
  context.log.info(context.headers["custom-header"]);
}
```

kn func invoke コマンドを使用して、この関数にアクセスできます。

コマンドの例

```
$ kn func invoke --target 'http://example.function.com'
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"so
me-value"}
```

11.12.1.5. HTTP 要求

方法

HTTP 要求メソッドを文字列として返します。

httpVersion

HTTP バージョンを文字列として返します。

httpVersionMajor

HTTP メジャーバージョン番号を文字列として返します。

httpVersionMinor

HTTP マイナーバージョン番号を文字列として返します。

11.12.2. Typescript コンテキストオブジェクトの参照

context オブジェクトには、関数開発者が利用可能なプロパティーが複数あります。これらのプロパティーにアクセスすると、着信 HTTP 要求に関する情報が提供され、出力がクラスターログに書き込まれます。

11.12.2.1. log

出力をクラスターロギングに書き込むために使用可能なロギングオブジェクトを提供します。ログは Pino logging API に準拠します。

ログの例

```
export function handle(context: Context): string {
    // log the incoming request body's 'hello' parameter
    if (context.body) {
      context.log.info((context.body as Record<string, string>).hello);
    } else {
      context.log.info('No data received');
    }
    return 'OK';
}
```

kn func invoke コマンドを使用して、この関数にアクセスできます。

コマンドの例

```
$ kn func invoke --target 'http://example.function.com'
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Processing customer"}
```

ログレベルは、**fatal**、**error**、**warn**、**info**、**debug**、**trace**、または **silent** のいずれかに設定できます。これを実行するには、**config** コマンドを使用してこれらの値のいずれかを環境変数 **FUNC_LOG_LEVEL** に割り当てて、**logLevel** の値を変更します。

11.12.2.2. query

要求のクエリー文字列 (ある場合) をキーと値のペアとして返します。これらの属性はコンテキストオブジェクト自体にも表示されます。

サンプルクエリー

```
```

```
export function handle(context: Context): string {
    // log the 'name' query parameter
  if (context.query) {
    context.log.info((context.query as Record<string, string>).name);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

kn func invoke コマンドを使用して、この関数にアクセスできます。

コマンドの例

```
$ kn func invoke --target 'http://example.function.com' --data '{"name": "tiger"}'
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

11.12.2.3. ボディー

要求ボディー (ある場合) を返します。要求ボディーに JSON コードが含まれる場合には、属性は直接利用できるように解析されます。

ボディーの例

```
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.body as Record<string, string>).hello);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

kn func invoke コマンドを使用して、この関数にアクセスできます。

コマンドの例

```
$ kn func invoke --target 'http://example.function.com' --data '{"hello": "world"}'
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"world"}
```

11.12.2.4. ヘッダー

HTTP 要求ヘッダーをオブジェクトとして返します。

ヘッダーの例

```
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.headers as Record<string, string>)['custom-header']);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

curl コマンドを使用してこの関数を呼び出すことができます。

コマンドの例

```
$ curl -H'x-custom-header: some-value" http://example.function.com
```

出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"some-value"}
```

11.12.2.5. HTTP 要求

方法

HTTP 要求メソッドを文字列として返します。

httpVersion

HTTP バージョンを文字列として返します。

httpVersionMajor

HTTP メジャーバージョン番号を文字列として返します。

httpVersionMinor

HTTP マイナーバージョン番号を文字列として返します。

第12章 統合

12.1. サーバーレスと COST MANAGEMENT SERVICE の統合

Cost Management は OpenShift Container Platform のサービスで、クラウドおよびコンテナーのコストをより正確に把握し、追跡することができます。これは、オープンソースの Koku プロジェクトに基づいています。

12.1.1. 前提条件

- クラスター管理者パーミッションがある。
- コスト管理を設定し、OpenShift Container Platform source を追加しました。

12.1.2. コスト管理クエリーにラベルを使用する

コスト管理では **タグ** とも呼ばれるラベルは、ノード、namespace、または Pod に適用できます。各ラベルはキーと値のペアです。複数のラベルを組み合わせてレポートを生成できます。Red Hat ハイブリッドコンソール を使用して、コストに関するレポートにアクセスできます。

ラベルは、ノードから namespace に、namespace から Pod に継承されます。ただし、ラベルがリソースに既に存在する場合、ラベルはオーバーライドされません。たとえば、Knative サービスにはデフォルトの **app=<revision_name>** ラベルがあります。

例 Knative サービスのデフォルトラベル

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
spec:
...
      labels:
        app: <revision_name>
...
```

app=my-domain の ように namespace のラベルを定義した場合、**app=my-domain** タグを使用してアプリケーションに問い合わせたときに、**app=<revision_name>** タグの Knative サービスから生じるコストは Cost Management Service では考慮されません。このタグを持つ Knative サービスのコストは、**app=<revision_name>** タグの下で照会する必要があります。

12.1.3. 関連情報

- ソースへのタグ付けの設定
- Cost Explorer を使用したコストの可視化

12.2. 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

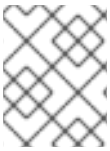
Nvidia は、OpenShift Container Platform での GPU リソースの実験的な使用をサポートします。OpenShift Container Platform での GPU リソースの設定に関する詳細は、OpenShift Container Platform on NVIDIA GPU accelerated clusters を参照してください。

12.2.1. サービスの GPU 要件の指定

OpenShift Container Platform クラスターの GPU リソースが有効化された後に、Knative (**kn**) CLI を使用して Knative サービスの GPU 要件を指定できます。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がクラスターにインストールされている。
- Knative (**kn**) CLI をインストールしている。
- GPU リソースが OpenShift Container Platform クラスターで有効にされている。
- 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。



注記

NVIDIA GPU リソースの使用は IBM Z および IBM Power Systems ではサポートされません。

手順

1. Knative サービスを作成し、**--limit nvidia.com/gpu=1** フラグを使用して、GPU リソース要件の制限を **1** に設定します。

```
$ kn service create hello --image <service-image> --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。

2. オプション。既存のサービスについては、**--limit nvidia.com/gpu=3** フラグを使用して、GPU リソース要件の制限を **3** に変更できます。

```
$ kn service update hello --limit nvidia.com/gpu=3
```

12.2.2. 関連情報

- 拡張リソースのリソースクォータの設定