



Red Hat OpenShift Serverless 1.33

Serving

Knative Serving の使用開始および設定サービス

Knative Serving の使用開始および設定サービス

法律上の通知

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

このドキュメントでは、Knative Serving の使用を開始する方法を説明します。このドキュメントでは、アプリケーションの設定方法を示し、自動スケーリング、トラフィック分割、外部および受信ルーティングなどの機能を説明します。

目次

第1章 KNATIVE SERVING を使い始める	4
1.1. SERVERLESS アプリケーションの作成	4
第2章 自動スケーリング	13
2.1. 自動スケーリング	13
2.2. スケーリング限度	13
2.3. 並行処理性	15
2.4. SCALE-TO-ZERO	17
第3章 OPENSIFT SERVERLESS アプリケーションの設定	20
3.1. SERVING のマルチコンテナサポート	20
3.2. EMPTYDIR ボリューム	20
3.3. 配信のための永続ボリューム要求	21
3.4. INIT コンテナ	23
3.5. イメージタグのダイジェストへの解決	24
3.6. TLS 認証の設定	25
3.7. KOURIER の設定	26
3.8. 制限のあるネットワークポリシー	27
第4章 SERVERLESS アプリケーションのデバッグ	30
4.1. ターミナル出力の確認	30
4.2. POD ステータスの確認	30
4.3. リビジョンステータスの確認	31
4.4. INGRESS のステータスの確認	31
4.5. ルートステータスの確認	32
4.6. INGRESS と ISTIO ルーティングの確認	32
第5章 KOURIER と ISTIO INGRESS	34
5.1. KOURIER および ISTIO INGRESS ソリューション	34
第6章 トラフィック分割	36
6.1. トラフィック分割の概要	36
6.2. トラフィックスペックの例	36
6.3. KNATIVE CLI を使用したトラフィック分割	38
6.4. トラフィック分割の CLI フラグ	39
6.5. リビジョン間でのトラフィックの分割	40
6.6. ブルーグリーン戦略を使用したトラフィックの再ルーティング	42
第7章 外部およびイングレスルーティング	45
7.1. ルーティングの概要	45
7.2. ラベルとアノテーションのカスタマイズ	45
7.3. KNATIVE サービスのルートの設定	46
7.4. 外部ルートの URL スキーム	49
7.5. クラスターローカルの可用性	49
7.6. KOURIER GATEWAY サービスタイプ	51
7.7. HTTP2 と GRPC の使用	52
7.8. OPENSIFT INGRESS シャーディングでの SERVING の使用	53
第8章 HTTP の設定	57
8.1. グローバル HTTPS リダイレクト	57
8.2. サービスごとの HTTPS リダイレクト	57
8.3. HTTP/1 のフルデュプレックスサポート	57

第9章 KNATIVE サービスへのアクセスの設定	59
9.1. KNATIVE サービスの JSON WEB TOKEN 認証の設定	59
9.2. SERVICE MESH 2.X での JSON WEB トークン認証の使用	59
9.3. SERVICE MESH 1.X での JSON WEB トークン認証の使用	62
第10章 SERVING の KUBE-RBAC-PROXY の設定	65
10.1. SERVING の KUBE-RBAC-PROXY リソースの設定	65
第11章 NET-KOURIER のバーストと QPS の設定	66
11.1. NET-KOURIER のバースト値と QPS 値の設定	66
第12章 KNATIVE サービスのカスタムドメインの設定	67
12.1. KNATIVE サービスのカスタムドメインの設定	67
12.2. カスタムドメインマッピング	67
12.3. KNATIVE CLI を使用した KNATIVE サービスのカスタムドメイン	68
12.4. 開発者パースペクティブを使用したドメインマッピング	69
12.5. ADMINISTRATOR パースペクティブを使用したドメインマッピング	71
12.6. TLS 証明書を使用してマッピングされたサービスを保護する	73
第13章 KNATIVE SERVING の高可用性設定	77
13.1. KNATIVE サービスの高可用性	77
13.2. KNATIVE デプロイメントの高可用性	77
第14章 サービス提供設定の調整	79
14.1. KNATIVE SERVING システムのデプロイメント設定のオーバーライド	79
第15章 QUEUE PROXY リソースの設定	81
15.1. KNATIVE サービスの QUEUE PROXY リソースの設定	81

第1章 KNATIVE SERVING を使い始める

1.1. SERVERLESS 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: showcase 1
  namespace: default 2
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase 3
          env:
            - name: GREET 4
              value: Ciao
```

- 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
OpenShift Container Platform の場合は、[開発者パースペクティブを使用したアプリケーションの作成](#) 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。

1.1.1. 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。

- Knative (**kn**) CLI 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

- Knative サービスを作成します。

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

ここでは、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 showcase \  
  --image quay.io/openshift-knative/showcase
```

出力例

```
Creating service 'showcase' in namespace 'default':  
  
0.271s The Route is still working to reflect the latest desired specification.  
0.580s Configuration "showcase" is waiting for a Revision to become ready.  
3.857s ...  
3.861s Ingress has not yet been reconciled.  
4.270s Ready to serve.  
  
Service 'showcase' created with latest revision 'showcase-00001' and URL:  
http://showcase-default.apps-crc.testing
```

1.1.2. 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は、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。

- 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. 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: showcase
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase
          env:
            - name: GREET
              value: Bonjour

```

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>
```

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 Platform Web コンソールの **Administrator** 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。

1.1.3. 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: showcase 1
  namespace: default 2
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase 3

```

```
env:
  - name: GREET 4
    value: Ciao
```

- 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** リストで、**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。

1.1.4. 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 製品サポート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場合があります。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 がインストールされている。

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 showcase \
  --image quay.io/openshift-knative/showcase \
  --target ./\
  --namespace test
```

出力例

```
Service 'showcase' 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ツリーを保存するディレクトリとして指定します。

既存のディレクトリを指定せずに、**--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 Container Platform クラスターにログインしていると、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構造を確認します。

```
$ tree ./
```

出力例

```
./
├── test
│   └── ksvc
│       └── showcase.yaml
```

```
2 directories, 1 file
```

- **--target** で指定する現在の ./ ディレクトリには新しい **test/** ディレクトリが含まれます。このディレクトリの名前は、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名前が付けられる記述子ファイル **showcase.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/showcase.yaml
```

出力例

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

4. 新しいサービスに関する情報をリスト表示します。

```
$ kn service describe showcase --target ./ --namespace test
```

出力例

```
Name:      showcase
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 Container Platform クラスターにログインしている場合、**kn** は現在の namespace にちなんで名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/showcase.yaml
```

出力例

```
Creating service 'showcase' in namespace 'test':

0.058s The Route is still working to reflect the latest desired specification.
0.098s ...
0.168s Configuration "showcase" 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 'showcase' created to latest revision 'showcase-00001' is available at URL:
http://showcase-test.apps.example.com
```

1.1.5. サーバーレスアプリケーションのデプロイメントの確認

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

手順

1. アプリケーション URL を検索します。

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

出力例

```
NAME      URL                                LATESTCREATED  LATESTREADY  READY
REASON
showcase  http://showcase-default.example.com  showcase-00001  showcase-00001
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 リクエストの例 (HTTPie ツールを使用)

```
$ http showcase-default.example.com
```

HTTPS 要求の例

```
$ https showcase-default.example.com
```

出力例

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Quarkus/2.13.7.Final-redhat-00003 Java/17.0.7
X-Config: {"sink":"http://localhost:31111","greet":"Ciao","delay":0}
X-Version: v0.7.0-4-g23d460f
content-length: 49

{
  "artifact": "knative-showcase",
  "greeting": "Ciao"
}
```

- オプション: システムに HTTPie ツールがインストールされていない場合は、代わりに curl ツールを使用できる可能性があります。

HTTPS 要求の例

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

出力例

```
{"artifact":"knative-showcase","greeting":"Ciao"}
```

- 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、HTTPie コマンドに **--verify=no** フラグを追加して、エラーを無視できます。

```
$ https --verify=no showcase-default.example.com
```

出力例

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Quarkus/2.13.7.Final-redhat-00003 Java/17.0.7
X-Config: {"sink":"http://localhost:31111","greet":"Ciao","delay":0}
X-Version: v0.7.0-4-g23d460f
content-length: 49

{
  "artifact": "knative-showcase",
  "greeting": "Ciao"
}
```



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。

- 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://showcase-default.example.com --cacert <file>
```

出力例

```
{"artifact":"knative-showcase","greeting":"Ciao"}
```

1.1.6. 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](#)

第2章 自動スケーリング

2.1. 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 (または Red Hat OpenShift Service on AWS および OpenShift Dedicated の専用管理者) によって設定されるグローバル設定、または個々のサービスに対して設定されるリビジョンごとに設定できます。

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。

2.2. 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。

2.2.1. 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と、**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** のアノテーション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: showcase
  namespace: default
spec:
  template:
```

```

metadata:
  annotations:
    autoscaling.knative.dev/min-scale: "0"
...

```

2.2.1.1. Knative CLI を使用した最小スケールアノテーションの設定

min-scale 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 がインストールされている。

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 showcase --image quay.io/openshift-knative/showcase --scale-min 2
```

2.2.2. 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: showcase
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/max-scale: "10"
...

```

2.2.2.1. Knative CLI を使用した最大スケールアノテーションの設定

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** フラグを指定した **kn service** コマンドを使用して、サービスの **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 がインストールされている。

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 showcase --image quay.io/openshift-knative/showcase --scale-max 10
```

2.3. 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。

2.3.1. 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: showcase
  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 showcase --image quay.io/openshift-knative/showcase --concurrency-target 50
```

2.3.2. 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: showcase
  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 showcase --image quay.io/openshift-knative/showcase --concurrency-limit 50
```

2.3.3. 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: showcase
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target-utilization-percentage: "70"
  ...
```

2.4. SCALE-TO-ZERO

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。

2.4.1. scale-to-zero の有効化

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。
- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。

- 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、ゼロにスケール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/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      enable-scale-to-zero: "false" ❶
```

- ❶ **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ル下限にスケールダウンされます。デフォルト値は **"true"** です。

2.4.2. 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。
- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- 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、ゼロにスケール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/v1beta1
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。

第3章 OPENSIFT SERVERLESS アプリケーションの設定

3.1. SERVING のマルチコンテナサポート

単一の Knative サービスを使用してマルチコンテナ Pod をデプロイできます。この方法は、アプリケーションの役割を小さく特化した部分に分離する場合に便利です。

3.1.1. マルチコンテナサービスの設定

マルチコンテナのサポートはデフォルトで有効になっています。サービス内の複数のコンテナを指定してマルチコンテナ Pod を作成できます。

手順

1. サービスを変更して、追加のコンテナを追加します。リクエストを処理できるコンテナは1つだけであるため、**ポート** は1つのコンテナにのみ指定してください。以下は、2つのコンテナの設定例です。

複数のコンテナ設定

```
apiVersion: serving.knative.dev/v1
kind: Service
...
spec:
  template:
    spec:
      containers:
        - name: first-container 1
          image: gcr.io/knative-samples/helloworld-go
          ports:
            - containerPort: 8080 2
        - name: second-container 3
          image: gcr.io/knative-samples/helloworld-java
```

- 1** 最初のコンテナ設定。
- 2** 最初のコンテナのポート仕様。
- 3** 2つ目のコンテナ設定。

3.2. EMPTYDIR ボリューム

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。

3.2.1. EmptyDir 拡張機能の設定

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。

KnativeService CR の例

```

apiVersion: operator.knative.dev/v1beta1
kind: KnativeService
metadata:
  name: knative-serving
spec:
  config:
    features:
      kubernetes.podspec-volumes-emptydir: enabled
  ...

```

3.3. 配信のための永続ボリューム要求

Serverless アプリケーションの中には、永続的なデータストレージを必要とするものもあります。さまざまなボリュームタイプを設定することで、Knative サービスにデータストレージを提供できます。Serving は、**secret**、**configMap**、**projected**、**emptyDir** などのボリュームタイプのマウントをサポートしています。

Knative サービスの永続ボリューム要求 (PVC) を設定できます。永続ボリュームタイプはプラグインとして実装されます。使用可能な永続ボリュームタイプがあるかどうかを確認するには、クラスター内で使用可能なストレージクラスまたはインストールされているストレージクラスを確認します。永続ボリュームはサポートされていますが、機能フラグを有効にする必要があります。



警告

大量のボリュームをマウントすると、アプリケーションの起動時間が大幅に遅れる可能性があります。

3.3.1. PVC サポートの有効化

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、**KnativeService** 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** ストレージクラスを使用できます。

ocs-storagecluster-cephfs ストレージクラスがサポートされており、[Red Hat OpenShift Data Foundation](#) から提供されています。

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 コンテナユーザーのユーザー権限などの追加の設定が必要です。

3.3.2. OpenShift Container Platform の関連情報

- [永続ストレージについて](#)

3.4. INIT コンテナ

Init コンテナ は、Pod 内のアプリケーションコンテナの前に実行される特殊なコンテナです。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 コンテナの使用を有効にできます。



注記

Init コンテナを使用すると、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。

3.4.1. init コンテナの有効化

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 に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。

手順

- **KnativeServing** CR に **kubernetes.podspec-init-containers** フラグを追加して、init コンテナの使用を有効にします。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
```

```

config:
  features:
    kubernetes.podspec-init-containers: enabled
...

```

3.5. イメージタグのダイジェストへの解決

Knative Serving コントローラーがコンテナ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。

3.5.1. タグからダイジェストへの解決

コントローラーに OpenShift Container Platform のコンテナ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** config map を使用して Knative Serving 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** config map にクラスター全体の証明書を設定し、config map をボリュームとしてコントローラーにマウントします。

3.5.1.1. 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 に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- 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>.cert=<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/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  controller-custom-certs:
    name: custom-secret
    type: Secret

```

3.6. 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、Apache Kafka リソースの Knative ブローカーに SASL と TLS の両方を併用することを推奨します。



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。

OpenShift Container Platform および Red Hat OpenShift Service on AWS の場合は、[mTLS で Service Mesh を使用する場合の Knative Serving メトリクスの有効化](#) のドキュメントを参照してください。

3.6.1. 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 製品サポート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場合があります。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。
- OpenShift (**oc**) CLI がインストールされている。

手順

1. **KnativeServing** リソースを作成または更新し、仕様に **internal-encryption: "true"** フィールドが含まれていることを確認します。

```
...
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
```

関連情報

- [Apache Kafka の Knative ブローカーの TLS 認証設定](#)
- [Apache Kafka のチャネルの TLS 認証設定](#)
- [mTLS で Service Mesh を使用する場合の Knative Serving メトリックの有効化](#)

3.7. KOURIER の設定

Kourier は、Knative Serving 用の軽量の Kubernetes ネイティブ Ingress です。Kourier は Knative のゲートウェイとして機能し、HTTP トラフィックを Knative サービスにルーティングします。

3.7.1. Kourier ゲートウェイ用の **kourier-bootstrap** のカスタマイズ

Kourier の Envoy プロキシコンポーネントは、Knative サービスの受信および送信 HTTP トラフィックを処理します。デフォルトでは、Kourier には、**knative-serving-ingress** namespace の **kourier-bootstrap** 設定マップに Envoy ブートストラップ設定が含まれています。この設定は変更できます。

前提条件

- OpenShift Serverless Operator および Knative Serving がインストールされている。
- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。

手順

- **KnativeServing** カスタムリソース (CR) の **spec.ingress.kourier.bootstrap-configmap** フィールドを変更して、カスタムブートストラップ設定マップを指定します。

KnativeServing CR の例

■

```

apiVersion: operator.knative.dev/v1beta1
kind: KnativeService
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    network:
      ingress-class: kourier.ingress.networking.knative.dev
  ingress:
    kourier:
      bootstrap-configmap: my-configmap
      enabled: true
# ...

```

3.8. 制限のあるネットワークポリシー

3.8.1. 制限のあるネットワークポリシーを持つクラスター

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

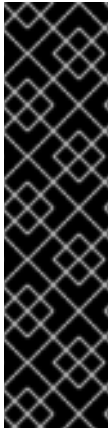
```

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

```

3.8.2. 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

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**) がインストールされている。
- 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。

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。

第4章 SERVERLESS アプリケーションのデバッグ

さまざまな方法を使用して、Serverless アプリケーションのトラブルシューティングを行うことができます。

4.1. ターミナル出力の確認

デプロイコマンドの出力を確認して、デプロイメントが成功したか確認できます。デプロイメントプロセスが終了した場合は、デプロイメントが失敗した理由を説明するエラーメッセージが出力に表示されます。この種の失敗は、マニフェストの設定が間違っているか、無効なコマンドが原因である可能性が高いです。

手順

- アプリケーションをデプロイおよび管理するクライアントでコマンド出力を開きます。次の例は、**oc apply** コマンドが失敗した後に表示される可能性のあるエラーです。

```

Error from server (InternalError): error when applying patch:
{"metadata":{"annotations":{"kubectl.kubernetes.io/last-applied-configuration":
{"apiVersion":"serving.knative.dev/v1","kind":"Route","metadata":{"annotations":
{},"name":"route-example","namespace":"default"},"spec":{"traffic":
[{"configurationName":"configuration-example","percent":50}]}}, "spec":{"traffic":
[{"configurationName":"configuration-example","percent":50}]}}
to:
&{0xc421d98240 0xc421e77490 default route-example STDIN 0xc421db0488 264682 false}
for: "STDIN": Internal error occurred: admission webhook "webhook.knative.dev" denied the
request: mutation failed: The route must have traffic percent sum equal to 100.
ERROR: Non-zero return code '1' from command: Process exited with status 1
  
```

この出力は、ルートトラフィックのパーセントを 100 に等しくなるように設定する必要があることを示しています。

4.2. POD ステータスの確認

Serverless アプリケーションの問題を特定するには、**Pod** オブジェクトのステータスを確認することを推奨します。

手順

- 次のコマンドを実行して、デプロイメントのすべての Pod をリスト表示します。

```
$ oc get pods
```

出力例

NAME	READY	STATUS	RESTARTS	AGE
configuration-example-00001-deployment-659747ff99-9bvr4	2/2	Running	0	3h
configuration-example-00002-deployment-5f475b7849-gxcht	1/2	CrashLoopBackOff	2	36s

出力では、すべての Pod とそのステータスに関する一部のデータが表示されます。

2. 次のコマンドを実行して、Pod のステータスに関する詳細情報を表示します。

出力例

```
$ oc get pod <pod_name> --output yaml
```

出力の **conditions** フィールドと **containerStatuses** フィールドが、デバッグに特に役立つ場合があります。

4.3. リビジョンステータスの確認

Serverless アプリケーションの問題を特定するには、リビジョンのステータスを確認することを推奨します。

手順

1. **Configuration** オブジェクトを使用してルートを設定する場合は、次のコマンドを実行して、デプロイメント用に作成された **Revision** オブジェクトの名前を取得します。

```
$ oc get configuration <configuration_name> --output jsonpath="{.status.latestCreatedRevisionName}"
```

設定名は **Route.yaml** ファイルで確認できます。このファイルは、OpenShift **Route** リソースを定義することでルーティング設定を指定します。

リビジョンを使用してルートを直接設定する場合は、**Route.yaml** ファイルでリビジョン名を検索します。

2. 次のコマンドを実行して、リビジョンのステータスをクエリーします。

```
$ oc get revision <revision-name> --output yaml
```

準備完了リビジョンには、ステータスに **reason: ServiceReady**、**status: "True"**、および **type: Ready** 条件が含まれている必要があります。これらの状況が存在する場合は、Pod のステータスまたは Istio ルーティングを確認することを推奨します。それ以外の場合は、リソースのステータスにエラーメッセージが含まれます。

4.3.1. 関連情報

- [ルート設定](#)

4.4. INGRESS のステータスの確認

Serverless アプリケーションの問題を特定するには、Ingress のステータスを確認することを推奨します。

手順

- 次のコマンドを実行して、Ingress の IP アドレスを確認します。

```
$ oc get svc -n istio-system istio-ingressgateway
```

istio-ingressgateway サービスは、Knative が使用する **LoadBalancer** サービスです。

外部 IP アドレスがない場合は、次のコマンドを実行します。

```
$ oc describe svc istio-ingressgateway -n istio-system
```

このコマンドは、IP アドレスがプロビジョニングされなかった理由を出力します。クォータの問題が原因である可能性が高いです。

4.5. ルートステータスの確認

場合によっては、**Route** オブジェクトに問題があることがあります。OpenShift CLI (**oc**) を使用してステータスを確認できます。

手順

- 次のコマンドを実行して、アプリケーションのデプロイに使用した **Route** オブジェクトのステータスを表示します。

```
$ oc get route <route_name> --output yaml
```

<route_name> を **Route** オブジェクトの名前に置き換えます。

status オブジェクトの **conditions** オブジェクトには、失敗した場合の理由が示されます。

4.6. INGRESS と ISTIO ルーティングの確認

Istio が Ingress レイヤーとして使用されている場合、Ingress と Istio のルーティングに問題が発生することがあります。OpenShift CLI (**oc**) を使用すると、それらの詳細を確認できます。

手順

- 次のコマンドを実行して、すべての Ingress リソースとそれに対応するラベルをリスト表示します。

```
$ oc get ingresses.networking.internal.knative.dev -o=custom-columns='NAME:.metadata.name,LABELS:.metadata.labels'
```

出力例

```
NAME          LABELS
helloworld-go map[serving.knative.dev/route:helloworld-go
serving.knative.dev/routeNamespace:default serving.knative.dev/service:helloworld-go]
```

この出力では、**serving.knative.dev/route** ラベルおよび **serving.knative.dev/routeNamespace** ラベルは、Ingress リソースが存在する **Route** を示しています。**Route** と Ingress がリスト表示されているはずですが。

Ingress が存在しない場合、ルートコントローラーは、**Route** オブジェクトまたは **Service** オブジェクトの対象である **Revision** オブジェクトの準備ができていないと想定します。他のデバッグ手順に進み、**Revision** の readiness ステータスを診断します。

- Ingress がリスト表示されている場合は、次のコマンドを実行して、ルートに対して作成された **ClusterIngress** オブジェクトを調べます。

```
$ oc get ingresses.networking.internal.knative.dev <ingress_name> --output yaml
```

出力のステータスセクションで、**type=Ready** の条件のステータスが **True** であれば、Ingress は正常に機能しています。それ以外の場合は、出力にエラーメッセージが表示されます。

3. Ingress のステータスが **Ready** の場合、対応する **VirtualService** オブジェクトがあります。次のコマンドを実行して、**VirtualService** オブジェクトの設定を確認します。

```
$ oc get virtualservice -l networking.internal.knative.dev/ingress=<ingress_name> -n <ingress_namespace> --output yaml
```

VirtualService オブジェクトのネットワーク設定は、**Ingress** オブジェクトおよび **Route** オブジェクトのネットワーク設定と一致する必要があります。**VirtualService** オブジェクトは **Status** フィールドを公開しないため、その設定が反映されるまで待つことを推奨します。

4.6.1. 関連情報

- [Maistra Service Mesh のドキュメント](#)

第5章 KOURIER と ISTIO INGRESS

OpenShift Serverless は、次の 2 つの Ingress ソリューションをサポートします。

- Kourier
- Red Hat OpenShift Service Mesh を使用した Istio

デフォルトは Kourier です。

5.1. KOURIER および ISTIO INGRESS ソリューション

5.1.1. Kourier

Kourier は、OpenShift Serverless のデフォルトの Ingress ソリューションです。次のような特性があります。

- Envoy プロキシをベースとしています。
- シンプルで軽量です。
- Serverless が一連の機能を提供するために必要な基本的なルーティング機能を提供します。
- 基本的な可観測性とメトリクスをサポートします。
- Knative Service ルーティングの基本的な TLS Termination をサポートします。
- 限られた設定および拡張オプションのみが提供されます。

5.1.2. OpenShift Service Mesh を使用した Istio

OpenShift Serverless の Ingress ソリューションとして Istio を使用すると、Red Hat OpenShift Service Mesh が提供するものに基づく追加の機能セットが有効になります。

- すべての接続間のネイティブ mTLS
- サービスメッシュの一部である Serverless コンポーネント
- 追加の可観測性とメトリクス
- 認可と認証のサポート
- Red Hat OpenShift Service Mesh でサポートされるカスタムルールと設定

ただし、追加機能には、より高いオーバーヘッドとリソース消費が伴います。詳細は、Red Hat OpenShift Service Mesh のドキュメントを参照してください。

Istio の要件とインストール手順は、Serverless ドキュメントの「Service Mesh と OpenShift Serverless の統合」セクションを参照してください。

5.1.3. トラフィックの設定とルーティング

Kourier と Istio のどちらを使用するかに関係なく、Knative Service のトラフィックは、それぞれ **net-kourier-controller** または **net-istio-controller** によって **knative-serving** namespace に設定されます。

コントローラーは **KnativeService** とその子カスタムリソースを読み取り、Ingress ソリューションを設定します。どちらの Ingress ソリューションも、トラフィックパスの一部となる Ingress ゲートウェイ Pod を提供します。どちらの Ingress ソリューションも Envoy をベースとしています。デフォルトでは、Serverless には各 **KnativeService** オブジェクトに対して 2 つのルートがあります。

- OpenShift ルーターによって転送される **cluster-external ルート** (例: **myapp-namespace.example.com**)。
- クラスタドメインを含む **cluster-local ルート** (例: **myapp.namespace.svc.cluster.local**)。このドメインは、Knative または他のユーザーワークロードから Knative サービスを呼び出すために使用できるため、使用する必要があります。

Ingress ゲートウェイは、serve モードまたはプロキシモードのいずれかでリクエストを転送できません。

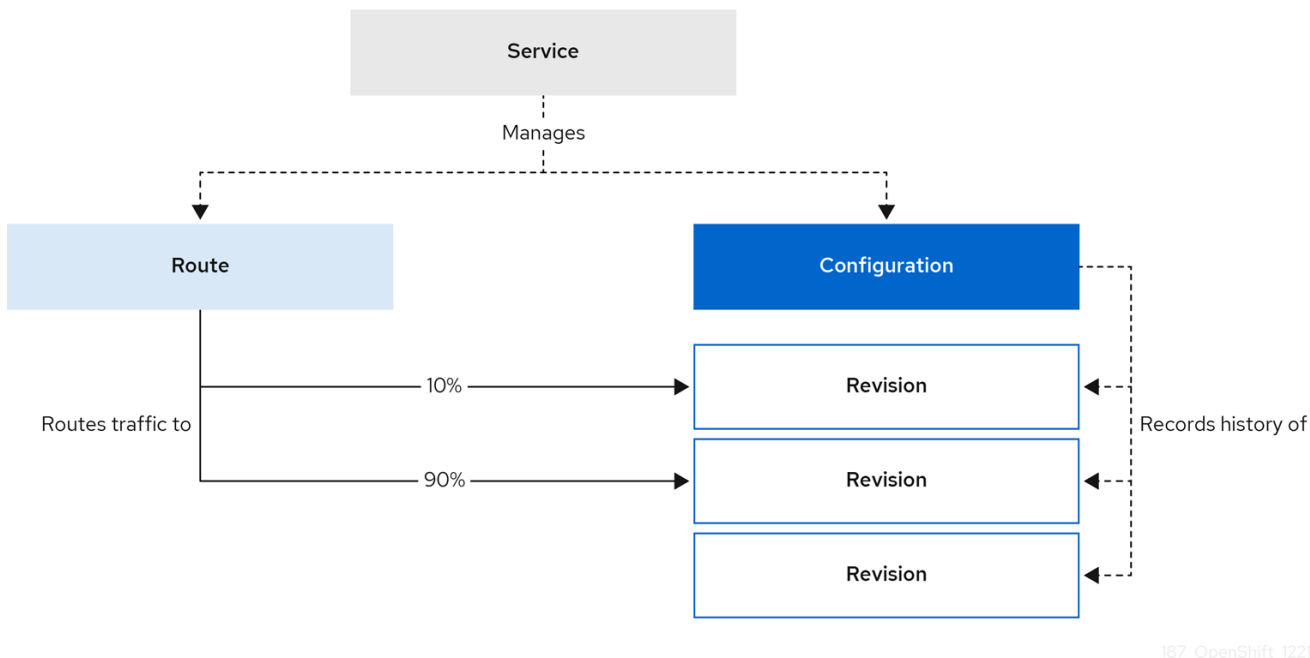
- serve モードでは、リクエストは Knative サービスの Queue-Proxy サイドカーコンテナに直接送信されます。
- プロキシモードでは、リクエストはまず **knative-serving** namespace の Activator コンポーネントを通過します。

モードの選択は、Knative の設定、Knative サービス、および現在のトラフィックによって異なります。たとえば、Knative サービスがゼロにスケールされた場合、リクエストは Activator コンポーネントに送信され、新しい Knative サービス Pod が開始されるまでバッファーとして機能します。

第6章 トラフィック分割

6.1. トラフィック分割の概要

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。



187_OpenShift_1221

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構成され、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。

6.2. 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
```

6.3. KNATIVE CLI を使用したトラフィック分割

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。

6.3.1. Knative CLI を使用してトラフィック分割を作成する

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 が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

- 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>
```

ここでは、以下ようになります。

- **<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 showcase --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 showcase --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。

6.4. トラフィック分割の CLI 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。

6.4.1. Knative CLI トラフィック分割フラグ

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=Perce nt	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

6.4.1.1. 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せん。
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。

6.4.1.2. 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。

6.4.1.2.1. 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
```

6.4.1.2.2. 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
```

6.5. リビジョン間でのトラフィックの分割

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。

6.5.1. 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

前提条件

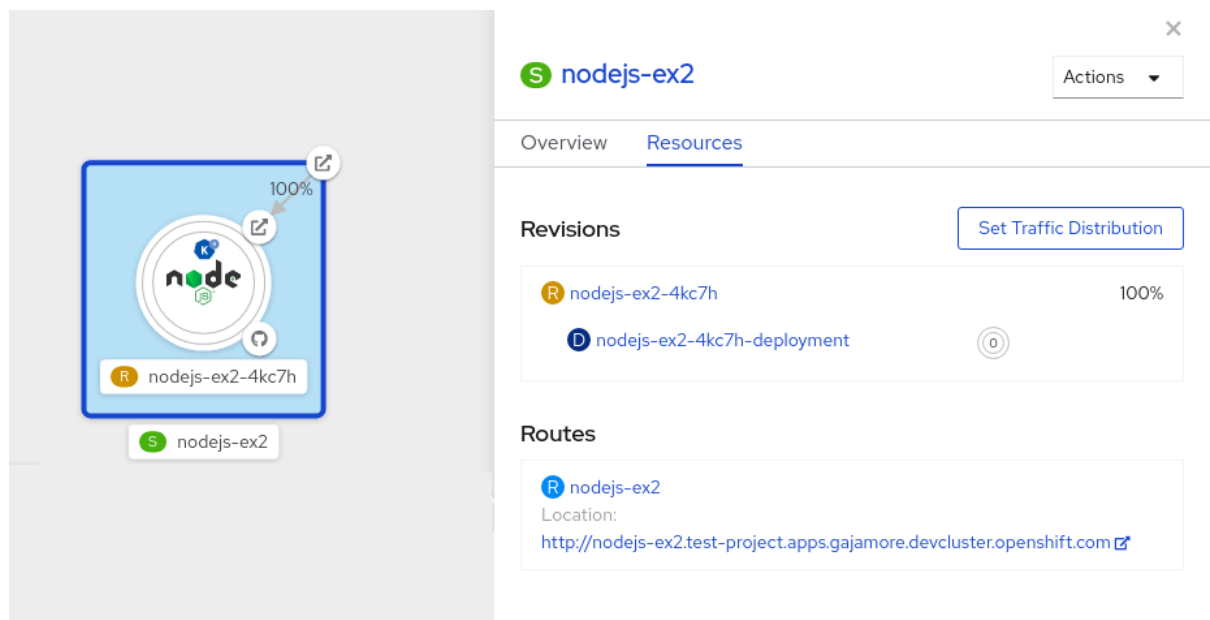
- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- OpenShift Container Platform Web コンソールにログインしている。

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。

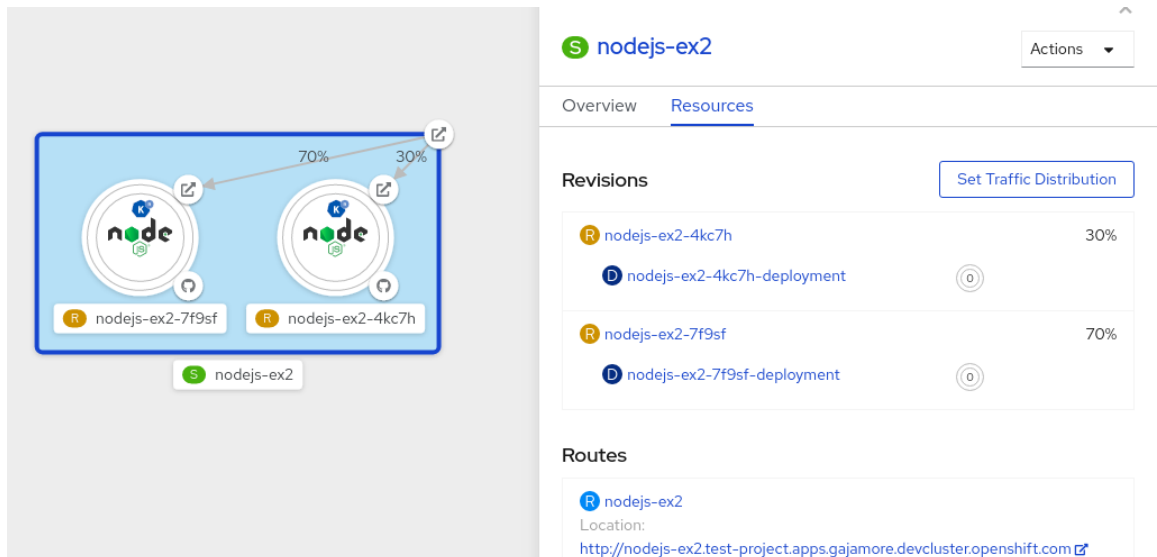
1. Knative サービスをクリックし、サイドパネルの概要を表示します。
2. **Resources** タブをクリックして、サービスの **Revisions** および **Routes** の一覧を表示します。

図6.1 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6.2 Serverless アプリケーションのリビジョン



6.6. ブルーグリーン戦略を使用したトラフィックの再ルーティング

[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。

6.6.1. 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

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**) がインストールされている。

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 showcase -o=jsonpath='{.status.latestCreatedRevisionName}'
```

出力例

```
$ showcase-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2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。

第7章 外部およびイングレスルーティング

7.1. ルーティングの概要

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。

7.1.1. OpenShift Container Platform の関連情報

- [ルート固有のアノテーション](#)

7.2. ラベルと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。

7.2.1. 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

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**) がインストールされている。

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> \ ❶
-l serving.knative.openshift.io/ingressNamespace=<service_namespace> \ ❷
-n knative-serving-ingress -o yaml \
  | grep -e "<label_name>: \"<label_value>\"" -e "<annotation_name>:"
<annotation_value>" ❸

```

- ❶ サービスの名前を使用します。
- ❷ サービスが作成された namespace を使用します。
- ❸ 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。

7.3. KNATIVE サービスのルートの設定

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。

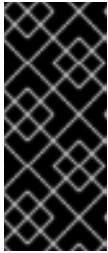
7.3.1. OpenShift Container Platform ルートでの Knative サービスの設定

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**) がインストールされている。

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が。

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>
```

7.4. 外部ルート URL スキーム

セキュリティを強化するために、外部ルート 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。

7.4.1. 外部ルート URL スキームの設定

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"
...
```

7.5. クラスターローカルの可用性

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、**servicing.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。

7.5.1. クラスターローカルへのクラスター可用性の設定

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

```

7.5.2. 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。
- 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. Kourier 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 Gateway 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。

7.6. KOURIER GATEWAY サービスタイプ

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
...

```

7.6.1. Kourier Gateway サービスタイプの設定

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

```

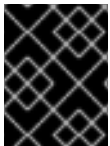
...
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...

```

7.7. HTTP2 と gRPC の使用

OpenShift Serverless はセキュアでないルートまたは edge-terminated 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-terminated 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。

7.7.1. 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



重要

この方法は、OpenShift Container Platform 4.10 以降が対象です。古いバージョンについては、以下のセクションを参照してください。

前提条件

- OpenShift Serverless Operator と Knative Serving をクラスターに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。
- Knative サービスを作成する。
- OpenShift Container Platform 4.10 以降をアップグレードする。
- OpenShift Ingress コントローラーで HTTP/2 を有効にする。

手順

1. **serverless.openshift.io/default-enable-http2=true** アノテーションを **KnativeServing** カスタムリソースに追加します。


```
$ oc annotate knativeserving <your_knative_CR> -n knative-serving
serverless.openshift.io/default-enable-http2=true
```

2. アノテーションが追加されたら、Kourier サービスの **appProtocol** 値が **h2c** であることを確認できます。

```
$ oc get svc -n knative-serving-ingress kourier -o jsonpath="{.spec.ports[0].appProtocol}"
```

出力例

```
h2c
```

3. 以下のように、外部トラフィックに HTTP/2 プロトコルで gRPC フレームワークを使用できるようになりました。

```
import "google.golang.org/grpc"

grpc.Dial(
  YOUR_URL, ①
  grpc.WithTransportCredentials(insecure.NewCredentials()), ②
)
```

① **ksvc** URL。

② 証明書。

関連情報

- [HTTP/2 Ingress 接続の有効化](#)

7.8. OPENSIFT INGRESS シャーディングでの SERVING の使用

Knative Serving と OpenShift Ingress シャーディングを使用して、ドメインに基づいて Ingress トラフィックを分割できます。これにより、ネットワークトラフィックをより効率的に管理し、クラスターのさまざまな部分にルーティングできるようになります。



注記

OpenShift Ingress シャーディングが導入されていても、OpenShift Serverless トラフィックは、単一の Knative Ingress Gateway と **knative-serving** プロジェクトのアクティベーターコンポーネントを介してルーティングされます。

ネットワークトラフィックの分離に関する詳細は、[Service Mesh を使用して OpenShift Serverless でネットワークトラフィックを分離する](#) 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。
- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。

7.8.1. OpenShift Ingress シャードの設定

Knative Serving を設定する前に、OpenShift Ingress シャードを設定する必要があります。

手順

- **IngressController** CR のラベルセレクターを使用して、異なるドメインの特定の Ingress シャードと一致させるように OpenShift Serverless を設定します。

IngressController CR の例

```

apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: ingress-dev ❶
  namespace: openshift-ingress-operator
spec:
  routeSelector:
    matchLabels:
      router: dev ❷
  domain: "dev.serverless.cluster.example.com" ❸
  # ...
---
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: ingress-prod ❹
  namespace: openshift-ingress-operator
spec:
  routeSelector:
    matchLabels:
      router: prod ❺
  domain: "prod.serverless.cluster.example.com" ❻
  # ...

```

- ❶ 最初の Ingress シャードの名前。
- ❷ **ingress-dev** シャードに一致するラベルセレクター。
- ❸ **ingress-dev** シャードのカスタムドメイン。
- ❹ 2 番目の Ingress シャードの名前。
- ❺ **ingress-prod** シャードと一致するラベルセレクター。
- ❻ **ingress-prod** シャードのカスタムドメイン。

7.8.2. KnativeServing CR でのカスタムドメインの設定

OpenShift Ingress シャードを設定した後、それらに一致するように Knative Serving を設定する必要があります。

手順

- **KnativeService** CR で、**spec.config.domain** フィールドを追加して、Ingress シャードと同じドメインとラベルを使用するように Serving を設定します。

KnativeService CR の例

```
spec:
  config:
    domain: ❶
    dev.serverless.cluster.example.com: |
      selector:
        router: dev
    prod.serverless.cluster.example.com: |
      selector:
        router: prod
# ...
```

- ❶ これらの値は、Ingress シャード設定の値と一致させる必要があります。

7.8.3. Knative サービスの特定の Ingress シャードをターゲットにする

Ingress シャーディングと Knative Serving を設定した後、ラベルを使用して Knative Service リソース内の特定の Ingress シャードをターゲットにすることができます。

手順

- **Service** CR で、特定のシャードに一致するラベルセレクターを追加します。

サービス CR の例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello-dev
labels:
  router: dev ❶
spec:
  template:
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello-prod
labels:
  router: prod ❷
spec:
  template:
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift
# ...
```

1 2 ラベルは **KnativeServing** CR の設定と一致させる必要があります。

7.8.4. OpenShift Ingress シャーディング設定を使用したサービスの検証

Ingress シャーディング、Knative Serving、サービスを設定した後、サービスが正しいルートと選択した Ingress シャードを使用していることを確認できます。

手順

1. 次のコマンドを実行して、クラスター内のサービスに関する情報を出力します。

```
$ oc get ksvc
```

出力例

```
NAME      URL                                                                 LATESTCREATED
LATESTREADY  READY REASON
hello-dev  https://hello-dev-default.dev.serverless.cluster.example.com  hello-dev-00001
hello-dev-00001  True
hello-prod  https://hello-prod-default.prod.serverless.cluster.example.com  hello-prod-00001
hello-prod-00001  True
```

2. 次のコマンドを実行して、サービスが正しいルートと選択した Ingress シャードを使用していることを確認します。

```
$ oc get route -n knative-serving-ingress -o jsonpath='{range .items[*]}{@.metadata.name}{"\n"}{@.spec.host}{"\n"}{@.status.ingress[*].routerName}{"\n"}{"\n"}{"end"}
```

出力例

```
route-19e6628b-77af-4da0-9b4c-1224934b2250-323461616533 hello-prod-
default.prod.serverless.cluster.example.com ingress-prod
route-cb5085d9-b7da-4741-9a56-96c88c6adaaa-373065343266 hello-dev-
default.dev.serverless.cluster.example.com ingress-dev
```

第8章 HTTP の設定

8.1. グローバル HTTPS リダイレクト

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。**KnativeService** 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。

8.1.1. HTTPS リダイレクトのグローバル設定

HTTPS リダイレクトを有効にする KnativeService CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeService
metadata:
  name: knative-serving
spec:
  config:
    network:
      httpProtocol: "redirected"
  ...
```

8.2. 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。

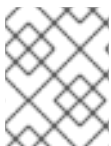
8.2.1. サービスの HTTPS のリダイレクト

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-protocol: "redirected"
spec:
  ...
```

8.3. HTTP/1 のフルデュプレックスサポート

features.knative.dev/http-full-duplex アノテーションを設定することで、サービスに対して HTTP/1 フルデュプレックスサポートを有効にできます。



注記

以前のバージョンのクライアントは HTTP/1 フルデュプレックスをサポートしていない可能性があるため、有効にする前に HTTP クライアントを確認してください。

次の例は、リビジョン仕様レベルの 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。

HTTP/1のフルデュプレックスサポートを提供する KnativeService CR の例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    spec:
      annotations:
        features.knative.dev/http-full-duplex: "Enabled"
  ...
```

第9章 KNATIVE サービスへのアクセスの設定

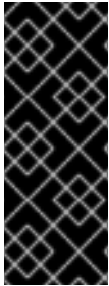
9.1. KNATIVE サービスの JSON WEB TOKEN 認証の設定

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。

9.2. SERVICE MESH 2.X 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。

9.2.1. Service Mesh 2.x および OpenShift Serverless の JSON Web トークン認証の設定



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。

OpenShift Container Platform では、これらの namespace の Pod にサイドカーの挿入が必要な場合は、**サービスマッシュと 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**) 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. **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" ❶
sidecar.istio.io/rewriteAppHTTPProbers: "true" ❷
```

...

- ❶ **sidecar.istio.io/inject="true"** アノテーションを追加します。
- ❷ 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 ❶
            - /healthz ❷
```

- ❶ 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。
- ❷ 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、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!
```

9.3. SERVICE MESH 1.X 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。

9.3.1. Service Mesh 1.x および OpenShift Serverless の JSON Web トークン認証の設定



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。

OpenShift Container Platform では、これらの namespace の Pod にサイドカーの挿入が必要な場合は、**サービスマッシュと 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**) 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. **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

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、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

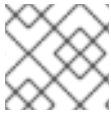
```
Hello OpenShift!
```

第10章 SERVING の KUBE-RBAC-PROXY の設定

kube-rbac-proxy コンポーネントは、Knative Serving の内部認証および認可機能を提供します。

10.1. SERVING の KUBE-RBAC-PROXY リソースの設定

OpenShift Serverless Operator CR を使用して、**kube-rbac-proxy** コンテナのリソース割り当てをグローバルにオーバーライドできます。



注記

特定のデプロイメントのリソース割り当てをオーバーライドすることもできます。

次の設定では、Knative Serving **kube-rbac-proxy** の最小および最大の CPU およびメモリー割り当てを設定します。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    deployment:
      "kube-rbac-proxy-cpu-request": "10m" ①
      "kube-rbac-proxy-memory-request": "20Mi" ②
      "kube-rbac-proxy-cpu-limit": "100m" ③
      "kube-rbac-proxy-memory-limit": "100Mi" ④
```

- ① 最小 CPU 割り当てを設定します。
- ② 最小 RAM 割り当てを設定します。
- ③ 最大 CPU 割り当てを設定します。
- ④ 最大 RAM 割り当てを設定します。

第11章 NET-KOURIER のバーストと QPS の設定

1秒あたりのクエリー数 (QPS) とバースト値によって、API サーバーへのリクエストまたは API 呼び出しの頻度が決まります。

11.1. NET-KOURIER のバースト値と QPS 値の設定

1秒あたりのクエリー数 (QPS) の値によって、API サーバーに送信されるクライアントリクエストまたは API 呼び出しの数が決まります。

バースト値によって、処理のために保存できるクライアントからのリクエストの数が決まります。このバッファを超えるリクエストはドロップされます。これは、バースト性が高く、要求が時間内に均一に分散されないコントローラーに役立ちます。

net-kourier-controller が再起動すると、クラスターにデプロイされたすべての **ingress** リソースが解析されるため、かなりの数の API 呼び出しが発生します。このため、**net-kourier-controller** の起動に時間がかかることがあります。

KnativeServing CR で **net-kourier-controller** の QPS およびバースト値を調整できます。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  workloads:
  - name: net-kourier-controller
    env:
    - container: controller
      envVars:
      - name: KUBE_API_BURST
        value: "200" ①
      - name: KUBE_API_QPS
        value: "200" ②
```

- ① コントローラーと API サーバー間の通信の QPS レート。デフォルト値は 200 です。
- ② Kubelet と API サーバー間の通信のバースト容量。デフォルト値は 200 です。

第12章 KNATIVE サービスのカスタムドメインの設定

12.1. 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。

12.2. カスタムドメインマッピング

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。

12.2.1. 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**) 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。
- 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/v1beta1
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/v1beta1
kind: DomainMapping
metadata:
  name: example.com
  namespace: default
spec:
  ref:
    name: showcase
    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/v1beta1
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>
```

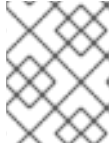
12.3. KNATIVE CLI を使用した KNATIVE サービスのカスタムドメイン

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。

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

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 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

- 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 showcase
```

--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は、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:showcase: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
```

12.4. 開発者パースペクティブを使用したドメインマッピング

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。OpenShift Container Platform Web コンソールの **Developer** パースペクティブを使用して、**DomainMapping** カスタムリソース (CR) を Knative サービスにマッピングできます。

12.4.1. 開発者パースペクティブを使用したカスタムドメインのサービスへのマッピング

前提条件

- Web コンソールにログインしている。
- **Developer** パースペクティブを使用している。
- OpenShift 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。これはクラスター管理者が完了する必要があります。
- 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。
- 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. **Topology** ページに移動します。
2. ドメインにマッピングするサービスを右クリックし、サービス名が含まれる **Edit** オプションを選択します。たとえば、サービスの名前が **showcase** の場合は、**Edit showcase** オプションを選択します。
3. **Advanced options** セクションで、**Show advanced Routing options** をクリックします。
 - a. サービスにマッピングするドメインマッピング CR がすでに存在する場合は、**ドメインマッピング** リストで選択できます。
 - b. 新規ドメインマッピング CR を作成する場合は、ドメイン名をボックスに入力し、**Create** オプションを選択します。たとえば、**example.com** と入力すると、**Create** オプションは **Create "example.com"** になります。
4. **Save** をクリックしてサービスへの変更を保存します。

検証

1. **Topology** ページに移動します。
2. 作成したサービスをクリックします。
3. サービス情報ウィンドウの **Resources** タブで、**Domain mappings** セクションにサービスにマッピングしたドメインが表示されます。

12.5. ADMINISTRATOR パースペクティブを使用したドメインマッピング

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り、Knative (**kn**) CLI または YAML ファイルを使用しない場合は、OpenShift Container Platform Web コンソールの **Administrator** パースペクティブを使用できます。

12.5.1. Administrator パースペクティブを使用したカスタムドメインのサービスへのマッピング

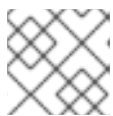
Knative サービスには、クラスターの設定に基づいてデフォルトのドメイン名が自動的に割り当てられます。例: `<service_name>-<namespace>.example.com`。所有するカスタムドメイン名を Knative サービスにマッピングすることで、Knative サービスのドメインをカスタマイズできます。

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。

OpenShift Container Platform のクラスター管理者権限 (または OpenShift Dedicated または Red Hat OpenShift Service on AWS のクラスターまたは専用管理者権限) が割り当てられている場合は、Web コンソールの **管理者** パースペクティブを使用して **DomainMapping** カスタムリソース (CR) を作成できます。

前提条件

- Web コンソールにログインしている。
- **Administrator** パースペクティブに切り替えられている。
- OpenShift Serverless Operator がインストールされている。
- Knative Serving がインストールされています。
- 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が割り当てられた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

カスタムドメインがクラスターの IP アドレスを参照する。

手順

1. **CustomResourceDefinitions** に移動し、検索ボックスを使用して **DomainMapping** カスタムリソース定義 (CRD) を検索します。
2. **DomainMapping** CRD をクリックしてから **Instances** タブに移動します。
3. **Create DomainMapping** をクリックします。
4. インスタンスの以下の情報が含まれるように **DomainMapping** CR の YAML を変更します。

```
apiVersion: serving.knative.dev/v1beta1
kind: DomainMapping
metadata:
  name: <domain_name> 1
```

```

namespace: <namespace> ❷
spec:
  ref:
    name: <target_name> ❸
    kind: <target_type> ❹
    apiVersion: serving.knative.dev/v1

```

- ❶ ターゲット CR にマップするカスタムドメイン名。
- ❷ **DomainMapping** CR とターゲット CR の両方の namespace。
- ❸ カスタムドメインにマップするサービス名。
- ❹ カスタムドメインにマップされる CR のタイプ。

Knative サービスへのドメインマッピングの例

```

apiVersion: serving.knative.dev/v1beta1
kind: DomainMapping
metadata:
  name: custom-ksvc-domain.example.com
  namespace: default
spec:
  ref:
    name: showcase
    kind: Service
    apiVersion: serving.knative.dev/v1

```

検証

- **curl** リクエストを使用してカスタムドメインにアクセスします。以下に例を示します。

コマンドの例

```
$ curl custom-ksvc-domain.example.com
```

出力例

```
{"artifact":"knative-showcase","greeting":"Welcome"}
```

12.5.2. 管理者パースペクティブを使用した暗号スイートの制限

Ingress に **net-kourier** を指定し、**DomainMapping** を使用すると、OpenShift ルーティングの TLS がパススルーに設定され、TLS は Kourier Gateway によって処理されます。このような場合、ユーザーに許可される Kourier の TLS 暗号スイートを限定することを推奨します。

前提条件

- Web コンソールにログインしている。
- **Administrator** パースペクティブに切り替えられている。

- OpenShift Serverless Operator がインストールされている。
- Knative Serving がインストールされています。
- 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が割り当てられたプロジェクトにアクセスできる。



注記

カスタムドメインがクラスターの IP アドレスを参照する。

手順

- **KnativeServing** CR で、**cipher-suites** 値を使用して、有効化する暗号スイートを指定します。

KnativeServing CR の例

```
spec:
  config:
    courier:
      cipher-suites: ECDHE-ECDSA-AES128-GCM-SHA256,ECDHE-ECDSA-CHACHA20-
        POLY1305
```

他の暗号スイートは無効化されます。コンマで区切って複数のスイートを指定できます。



注記

Kourier Gateway のコンテナイメージは Envoy プロキシイメージを利用し、デフォルトで有効化されている暗号スイートは Envoy プロキシのバージョンによって異なります。

12.6. TLS 証明書を使用してマッピングされたサービスを保護する

12.6.1. 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**) がインストールされている。

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. **networking.internal.knative.dev/certificate-uid: <id>** ラベルを Kubernetes TLS シークレットに追加します。

```
$ oc label secret <tls_secret_name> networking.internal.knative.dev/certificate-uid="<id>"
```

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

Red Hat OpenShift の cert-manager Operator は、テクノロジープレビューの機能です。詳細は、[Red Hat OpenShift ドキュメントの cert-manager Operator のインストール](#) 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/v1beta1
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。

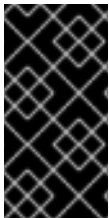
```
$ curl https://<domain_name>
```

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。

12.6.2. シークレットフィルタリングを使用した **net-kourier** のメモリー使用量の改善

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-kourier** Ingress 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-kourier** Ingress コントローラーではフィルタ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。

シークレットのフィルタリングは、OpenShift Serverless Operator 側でデフォルトで有効化されています。環境変数 **ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID=true** は、デフォルトで **net-kourier** コントローラー Pod に追加されます。



重要

シークレットフィルタリングを有効にする場合は、すべてのシークレットに **networking.internal.knative.dev/certificate-uid: "<id>"** というラベルを付ける必要があります。そうしないと、Knative Serving がシークレットを検出しないため、障害が発生します。新規および既存のシークレットの両方にラベルを付ける必要があります。

前提条件

- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- 自分で作成したプロジェクトまたは、アプリケーションや他のワークロードを作成するパーミッションおよびロールがあるプロジェクト。
- OpenShift Serverless Operator および Knative Serving 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。

KnativeServing カスタムリソース (CR) の **workloads** フィールドを使用して、**ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID** 変数を **false** に設定することで、シークレットフィルターを無効化できます。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  ...
  workloads:
    - env:
      - container: controller
        envVars:
```

┆ - name: ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID
┆ value: 'false'
name: net-kourier-controller

第13章 KNATIVE SERVING の高可用性設定

13.1. KNATIVE サービスの高可用性

高可用性 (HA) は Kubernetes API の標準的な機能で、中断が生じる場合に API が稼働を継続するのに役立ちます。HA デプロイメントでは、アクティブなコントローラーがクラッシュまたは削除されると、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は、必要時に備えてコントローラーのインスタンスがスケジュールされ、クラスター内で実行されます。このコントローラーインスタンスは、リーダー選出ロックと呼ばれる共有リソースを使用するために競合します。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。

13.2. KNATIVE デプロイメントの高可用性

高可用性 (HA) は、それぞれ 2 つのレプリカを持つように設定された Knative Serving **activator**、**autoscaler**、**autoscaler-hpa**、**controller**、**webhook**、**domain-mapping**、**domainmapping-webhook**、**kourier-control**、および **kourier-gateway** コンポーネントでデフォルトで利用可能です。**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。

13.2.1. Knative Serving の高可用性レプリカの設定

適格なデプロイメントリソースに最小 3 つのレプリカを指定するには、カスタムリソースのフィールド **spec.high-availability.replicas** の値を **3** に設定します。

前提条件

- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- 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. 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。

You are logged in as a temporary administrative user. Update the [cluster OAuth configuration](#) to allow others to log in.

Project: knative-serving

Installed Operators > serverless-operator.v1.16.0 > KnativeServing details

KS knative-serving Actions

Details **YAML** Resources Events

```

88 deployment:
89   queueSidecarImage: >-
90     registry.redhat.io/openshift-serverless-1/
91   domain:
92     apps.ci-ln-nt5xzit-f76d1.origin-ci-int-gce.d
93   controller-custom-certs:
94     name: config-service-ca
95     type: ConfigMap
96   high-availability:
97     replicas: 2
98   knative-ingress-gateway: {}
99   registry:
100  override:
101    imc-controller/controller: >-
102      registry.redhat.io/openshift-serverless-1/
103    mt-broker-filter/filter: >-
104      registry.redhat.io/openshift-serverless-1/

```

Save Reload Cancel

5. **KnativeServing** CR のレプリカ数を変更します。

サンプル YAML

```

apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  high-availability:
    replicas: 3

```

第14章 サービス提供設定の調整

14.1. 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 **workloads** 仕様を変更することで、一部の特定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。

14.1.1. システムのデプロイメント設定の上書き

現在、**resources**、**replicas**、**labels**、**annotations**、**nodeSelector** フィールド、およびプローブの **readiness** と **liveness** フィールドで、デフォルトの構成設定のオーバーライドが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。

- **net-kourier-controller** の **readiness** プロブのタイムアウトは 10 秒に設定されています。
- 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: label** ラベルが追加されました。
- **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/v1beta1
kind: KnativeServing
metadata:
  name: ks
  namespace: knative-serving
spec:
  high-availability:
    replicas: 2
  workloads:
    - name: net-kourier-controller
      readinessProbes: ①
        - container: controller
          timeoutSeconds: 10
    - 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
```

- 1 **readiness** および **liveness** プロブオーバーライドを使用して、プロブハンドラーに関連するフィールド (**exec**、**grpc**、**httpGet**、および **tcpSocket**) を除き、Kubernetes API で指定されているデプロイメントのコンテナ内のプロブのすべてのフィールドをオーバーライドできます。

関連情報

- [Kubernetes API ドキュメントのプロブ設定セクション](#)

第15章 QUEUE PROXY リソースの設定

Queue Proxy は、サービス内の各アプリケーションコンテナに対するサイドカーコンテナです。これにより、Serverless ワークロードの管理が向上し、リソースを効率的に使用できます。Queue Proxy を設定できます。

15.1. KNATIVE サービスの QUEUE PROXY リソースの設定

デプロイメント configmap で Queue Proxy リソースのリクエストと制限をグローバルに設定することとは別に、CPU、メモリー、および一時ストレージリソースタイプを対象とした対応するアノテーションを使用して、それらをサービスレベルで設定できます。

前提条件

- Red Hat OpenShift パイプラインがクラスターにインストールされている。
- OpenShift (**oc**) 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。

手順

- リソースのリクエストと制限を使用して、サービスの configmap を変更します。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        queue.sidecar.serving.knative.dev/cpu-resource-request: "1"
        queue.sidecar.serving.knative.dev/cpu-resource-limit: "2"
        queue.sidecar.serving.knative.dev/memory-resource-request: "1Gi"
        queue.sidecar.serving.knative.dev/memory-resource-limit: "2Gi"
        queue.sidecar.serving.knative.dev/ephemeral-storage-resource-request: "400Mi"
        queue.sidecar.serving.knative.dev/ephemeral-storage-resource-limit: "450Mi"
```

あるいは、特別な **queue.sidecar.serving.knative.dev/resource-percentage** アノテーションを使用することもできます。これにより、アプリケーションコンテナのパーセンテージとして Queue Proxy リソースが計算されます。CPU およびメモリーのリソース要件がアプリケーションコンテナの要件から計算され、それらが以下の境界外にある場合、値は境界内に収まるように調整されます。この場合、CPU とメモリーのリソース要件には、次の最小値と最大値の境界が適用されます。

表15.1 リソース要件の境界

リソース要件	Min	Max
CPU request	25m	100 m

リソース要件	Min	Max
CPU limit	40m	500 m
メモリー要求	50 Mi	200Mi
メモリー制限	200Mi	500 Mi



注記

対応するリソースアノテーションを使用して、パーセンテージアノテーションと特定のリソース値を同時に設定した場合は、後者が優先されます。



警告

`queue.sidecar.serving.knative.dev/resource-percentage` アノテーションは現在非推奨であり、OpenShift Serverless の今後のバージョンでは削除される予定です。