



# Red Hat OpenShift Serverless 1.29

## 関数

OpenShift Serverless Functions のセットアップおよび使用



# Red Hat OpenShift Serverless 1.29 関数

---

OpenShift Serverless Functions のセットアップおよび使用

## 法律上の通知

Copyright © 2023 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<sup>®</sup> is the registered trademark of Linus Torvalds in the United States and other countries.

Java<sup>®</sup> is a registered trademark of Oracle and/or its affiliates.

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

MySQL<sup>®</sup> is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js<sup>®</sup> 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<sup>®</sup> 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、OpenShift Serverless Functions の使用を開始する方法や、Quarkus、Node.js、TypeScript、および Python を使用した関数の開発およびデプロイに関する情報を提供します。

## 目次

<b>第1章 OPENSIFT SERVERLESS FUNCTIONS の設定</b> .....	<b>4</b>
1.1. 前提条件	4
1.2. PODMAN の設定	4
1.3. MACOS での PODMAN のセットアップ	5
1.4. 次のステップ	6
<b>第2章 関数を使い始める</b> .....	<b>7</b>
2.1. 前提条件	7
2.2. 関数の作成	7
2.3. 機能をローカルで実行する	8
2.4. 関数のビルド	8
2.5. 関数のデプロイ	9
2.6. テストイベントでのデプロイされた関数の呼び出し	10
2.7. 関数の削除	11
2.8. OPENSIFT CONTAINER PLATFORM の関連情報	11
2.9. 次のステップ	11
<b>第3章 クラスター上での機能の構築とデプロイ</b> .....	<b>12</b>
3.1. クラスターでの関数のビルドとデプロイ	12
3.2. 関数リビジョンの指定	13
<b>第4章 QUARKUS 関数の開発</b> .....	<b>15</b>
4.1. 前提条件	15
4.2. QUARKUS 関数テンプレートの構造	15
4.3. QUARKUS 関数の呼び出しについて	16
4.4. CLOUDEVENT 属性	19
4.5. QUARKUS 関数の戻り値	19
4.6. QUARKUS 関数のテスト	20
4.7. 次のステップ	20
<b>第5章 NODE.JS 関数の開発</b> .....	<b>21</b>
5.1. 前提条件	21
5.2. NODE.JS 関数テンプレート構造	21
5.3. NODE.JS 関数の呼び出しについて	21
5.4. NODE.JS 関数の戻り値	23
5.5. NODE.JS 関数のテスト	24
5.6. 次のステップ	24
<b>第6章 TYPESCRIPT 関数の開発</b> .....	<b>25</b>
6.1. 前提条件	25
6.2. TYPESCRIPT 関数テンプレートの構造	25
6.3. TYPESCRIPT 関数の呼び出しについて	25
6.4. TYPESCRIPT 関数の戻り値	28
6.5. TYPESCRIPT 関数のテスト	29
6.6. 次のステップ	29
<b>第7章 PYTHON 関数の開発</b> .....	<b>31</b>
7.1. 前提条件	31
7.2. PYTHON 関数テンプレート構造	31
7.3. PYTHON 関数の呼び出しについて	32
7.4. PYTHON 関数の戻り値	32
7.5. PYTHON 関数のテスト	33
7.6. 次のステップ	33

---

<b>第8章 KNATIVE EVENTING での関数の使用</b> .....	<b>34</b>
8.1. 開発者パースペクティブを使用してイベントソースを機能に接続する	34
<b>第9章 FUNC.YAML の関数プロジェクト設定</b> .....	<b>35</b>
9.1. FUNC.YAML の設定可能なフィールド	35
9.2. FUNC.YAML フィールドからのローカル環境変数の参照	38
9.3. 関連情報	39
<b>第10章 関数からのシークレットおよび CONFIG MAP へのアクセス</b> .....	<b>40</b>
10.1. シークレットおよび CONFIG MAP への関数アクセスの対話的な変更	40
10.2. 特殊なコマンドを使用したシークレットおよび CONFIG MAP への関数アクセスの対話的な変更	41
10.3. シークレットおよび CONFIG MAP への関数アクセスの手動による追加	42
<b>第11章 アノテーションの関数への追加</b> .....	<b>48</b>
11.1. 関数へのアノテーションの追加	48
<b>第12章 関数開発リファレンスガイド</b> .....	<b>50</b>
12.1. NODE.JS コンテキストオブジェクトのリファレンス	50
12.2. TYPESCRIPT コンテキストオブジェクトの参照	52



## 第1章 OPENSIFT SERVERLESS FUNCTIONS の設定

アプリケーションコードのデプロイプロセスを改善するために、OpenShift Serverless を使用して、ステートレスでイベント駆動型の関数を Knative サービスとして OpenShift Container Platform にデプロイできます。関数を開発する場合は、セットアップ手順を完了する必要があります。

### 1.1. 前提条件

クラスターで OpenShift Serverless Functions の使用を有効にするには、以下の手順を実行する必要があります。

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



#### 注記

関数は Knative サービスとしてデプロイされます。関数でイベント駆動型のアーキテクチャを使用する必要がある場合は、Knative Eventing もインストールする必要があります。

- **oc CLI** がインストールされている。
- **Knative (kn) CLI** がインストールされている。Knative CLI をインストールすると、関数の作成および管理に使用できる **kn func** コマンドを使用できます。
- Docker Container Engine または Podman バージョン 3.4.7 以降がインストールされている。
- OpenShift Container Registry などの利用可能なイメージレジストリーにアクセスできる。
- [Quay.io](#) をイメージレジストリーとして使用する場合は、リポジトリーがプライベートではないか確認するか、OpenShift Container Platform ドキュメント [Pod が他のセキュアなレジストリーからイメージを参照できるようにする設定](#) に従っていることを確認する必要があります。
- OpenShift Container レジストリーを使用している場合は、クラスター管理者が [レジストリーを公開する](#) 必要があります。

### 1.2. PODMAN の設定

高度なコンテナ管理機能を使用するには、OpenShift Serverless Functions で Podman を使用することが推奨されます。そのためには、Podman サービスを開始し、それに接続するように Knative (**kn**) CLI を設定する必要があります。

#### 手順

1. `${XDG_RUNTIME_DIR}/podman/podman.sock` で、UNIX ソケットで Docker API を提供する Podman サービスを起動します。

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



#### 注記

多くのシステムでは、このソケットは `/run/user/$(id -u)/podman/podman.sock` にあります。

- 関数のビルドに使用する環境変数を確立します。

```
$ export DOCKER_HOST="unix://${XDG_RUNTIME_DIR}/podman/podman.sock"
```

- v フラグを指定して、関数プロジェクトディレクトリー内で build コマンドを実行し、詳細な出力を表示します。ローカルの UNIX ソケットへの接続が表示されるはずですが。

```
$ kn func build -v
```

### 1.3. MACOS での PODMAN のセットアップ

高度なコンテナ管理機能を使用するには、OpenShift Serverless Functions で Podman を使用することが推奨されます。macOS でこれを行うには、Podman マシンを起動し、それに接続するように Knative (**kn**) CLI を設定する必要があります。

#### 手順

- Podman マシンを作成します。

```
$ podman machine init --memory=8192 --cpus=2 --disk-size=20
```

- UNIX ソケットで Docker API を提供する Podman マシンを開始します。

```
$ podman machine start
Starting machine "podman-machine-default"
Waiting for VM ...
Mounting volume... /Users/myuser:/Users/user
```

[...truncated output...]

You can still connect Docker API clients by setting DOCKER\_HOST using the following command in your terminal session:

```
export
DOCKER_HOST='unix:///Users/myuser/.local/share/containers/podman/machine/podman-machine-default/podman.sock'
```

Machine "podman-machine-default" started successfully



#### 注記

ほとんどの macOS システムでは、このソケットは **/Users/myuser/.local/share/containers/podman/machine/podman-machine-default/podman.sock** にあります。

- 関数のビルドに使用する環境変数を確立します。

```
$ export
DOCKER_HOST='unix:///Users/myuser/.local/share/containers/podman/machine/podman-machine-default/podman.sock'
```

4. `-v` フラグを指定して、関数プロジェクトディレクトリー内で `build` コマンドを実行し、詳細な出力を表示します。ローカルの UNIX ソケットへの接続が表示されるはずですが。

```
$ kn func build -v
```

## 1.4. 次のステップ

- Docker Container Engine または Podman の詳細は、[コンテナービルドツールのオプション](#) を参照してください。
- [関数を使い始める](#) を参照してください。

## 第2章 関数を使い始める

関数のライフサイクル管理には、関数の作成、構築、デプロイが含まれます。必要に応じて、デプロイされた関数を呼び出してテストすることもできます。これらの操作はすべて、**kn func** ツールを使用して OpenShift Serverless で実行できます。

### 2.1. 前提条件

以下の手順を実行する前に、[OpenShift Serverless 機能の設定](#)の要件タスクをすべて完了している必要があります。

### 2.2. 関数の作成

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

#### 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

- 関数プロジェクトを作成します。

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

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

#### コマンドの例

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

#### 出力例

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

- または、カスタムテンプレートを含むリポジトリを指定することもできます。

#### コマンドの例

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

#### 出力例

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

## 2.3. 機能をローカルで実行する

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

### 現在のディレクトリーで機能を実行するコマンドの例

```
$ kn func run
```

### パスとして指定されたディレクトリーで機能を実行するコマンドの例

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

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

### ビルドフラグを使用した実行コマンドの例

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

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

### ビルドフラグを使用した実行コマンドの例

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

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

### help コマンドの構築

```
$ kn func help run
```

## 2.4. 関数のビルド

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

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

### 2.4.1. イメージコンテナの種類

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

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

```
$ kn func build
```

### 2.4.2. イメージレジストリーの種類

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

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

```
$ kn func build
```

### 出力例

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

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

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

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

### 出力例

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

### 2.4.3. Push フラグ

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

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

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

### 2.4.4. Help コマンド

**kn func build** コマンドオプションの詳細は、**help** コマンドを使用できます。

### help コマンドの構築

```
$ kn func help build
```

## 2.5. 関数のデプロイ

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

### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI 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。
- デプロイする関数を作成し、初期化している必要がある。

### 手順

- 関数をデプロイします。

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

### 出力例

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

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



### 注記

Developer パースペクティブの **+Add** ビューで **Import from Git** または **Create Serverless Function** を使用して、Git リポジトリー URL を使用してサーバーレス関数を作成できます。

## 2.6. テストイベントでのデプロイされた関数の呼び出し

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

### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI 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。
- 呼び出す関数をすでにデプロイしている。

### 手順

- 関数を呼び出します。

```
$ kn func invoke
```

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

## 2.7. 関数の削除

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

### 手順

- 関数を削除します。

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

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

## 2.8. OPENSIFT CONTAINER PLATFORM の関連情報

- [Exposing a default registry manually](#)
- [Marketplace page for the IntelliJ Knative plugin](#)
- [Marketplace page for the Visual Studio Code Knative plugin](#)
- [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](#)

## 2.9. 次のステップ

- [Knative Eventing でのチャンネルの使用](#) を参照してください。

## 第3章 クラスター上での機能の構築とデプロイ

関数をローカルでビルドする代わりに、クラスターで直接関数をビルドできます。このワークフローをローカル開発マシンで使用する場合は、関数のソースコードのみを操作する必要があります。これは、たとえば、docker や podman などのクラスター上の関数構築ツールをインストールできない場合に役立ちます。

### 3.1. クラスターでの関数のビルドとデプロイ

Knative (**kn**) CLI を使用して、関数プロジェクトのビルドを開始し、関数をクラスターに直接デプロイできます。この方法で関数プロジェクトをビルドするには、関数プロジェクトのソースコードが、クラスターにアクセスできる Git リポジトリブランチに存在する必要があります。

#### 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 CLI (**oc**) 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

1. OpenShift パイプラインを実行して関数をデプロイする各 namespace で、次のリソースを作成する必要があります。

- a. パイプラインで Source-to-Image を使用できるように **s2i** Tekton タスクを作成します。

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.29.0/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
```

- b. **kn func** deploy Tekton タスクを作成して、パイプラインに関数をデプロイできるようにします。

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.29.0/pkg/pipelines/resources/tekton/task/func-deploy/0.1/func-deploy.yaml
```

2. 関数を作成します。

```
$ kn func create <function_name> -l <runtime>
```

3. 新しい関数プロジェクトを作成したら、プロジェクトを Git リポジトリに追加し、リポジトリがクラスターで使用可能であることを確認する必要があります。この Git リポジトリに関する情報は、次のステップで **func.yaml** ファイルを更新するために使用されます。

4. 関数プロジェクトの **func.yaml** ファイルの設定を更新して、Git リポジトリのクラスター上ビルドを有効にします。

```
...
git:
  url: <git_repository_url> 1
```

```
revision: main ②
contextDir: <directory_path> ③
...
```

- ① 必須。関数のソースコードを含む Git リポジトリを指定します。
  - ② オプション: 使用する Git リポジトリのリビジョンを指定します。これは、ブランチ、タグ、またはコミットにすることができます。
  - ③ オプション: 関数が Git リポジトリのルートフォルダーにない場合は、関数のディレクトリパスを指定します。
5. 関数のビジネスロジックを実装します。次に、Git を使用して変更をコミットしてプッシュします。
  6. 関数をデプロイします。

```
$ kn func deploy --remote
```

関数設定で参照されているコンテナレジストリーにログインしていない場合は、関数イメージをホストするリモートコンテナレジストリーの認証情報を入力するように求められます。

### 出力例とプロンプト

```
Creating Pipeline resources
Please provide credentials for image registry used by Pipeline.
? Server: https://index.docker.io/v1/
? Username: my-repo
? Password: *****
Function deployed at URL: http://test-function.default.svc.cluster.local
```

7. 関数を更新するには、Git を使用して新しい変更をコミットしてプッシュしてから、**kn func deploy --remote** コマンドを再度実行します。

## 3.2. 関数リビジョンの指定

関数をビルドしてクラスターにデプロイするときは、リポジトリ内の Git リポジトリ、ブランチ、およびサブディレクトリーを指定して、関数コードの場所を指定する必要があります。**main** ブランチを使用する場合は、ブランチを指定する必要はありません。同様に、関数がリポジトリのルートにある場合は、サブディレクトリーを指定する必要はありません。これらのパラメーターは、**func.yaml** 設定ファイルで指定するか、**kn func deploy** コマンドでフラグを使用して指定できます。

### 前提条件

- Red Hat OpenShift パイプライン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

- 関数をデプロイします。

```
$ kn func deploy --remote \ 1  
  --git-url <repo-url> \ 2  
  [--git-branch <branch>] \ 3  
  [--git-dir <function-dir>] 4
```

- 1 **--remote** フラグを使用すると、ビルドはリモートで実行されます。
- 2 **<repo-url>** を Git リポジトリの URL に置き換えます。
- 3 **<branch>** を Git ブランチ、タグ、またはコミットに置き換えます。 **main** ブランチで最新のコミットを使用している場合は、このフラグをスキップできます。
- 4 **<function-dir>** がリポジトリのルートディレクトリと異なる場合は、関数を含むディレクトリに置き換えます。

以下に例を示します。

```
$ kn func deploy --remote \  
  --git-url https://example.com/alice/myfunc.git \  
  --git-branch my-feature \  
  --git-dir functions/example-func/
```

## 第4章 QUARKUS 関数の開発

Quarkus 関数プロジェクトを作成したら、指定のテンプレートを変更して、関数にビジネスロジックを追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

### 4.1. 前提条件

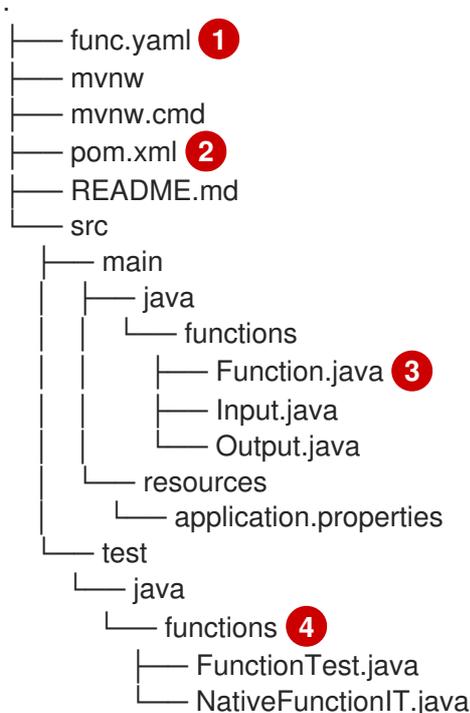
- 関数を開発する前に、[OpenShift Serverless 機能の設定](#) の設定手順を完了している。

### 4.2. QUARKUS 関数テンプレートの構造

Knative (kn) CLI を使用して Quarkus 関数を作成すると、プロジェクトディレクトリーは一般的な Maven プロジェクトと同様になります。さらに、プロジェクトには、関数の設定に使用される `func.yaml` ファイルが含まれています。

`http` および `event` トリガー関数のテンプレート構造はいずれも同じです。

#### テンプレート構造



- ① イメージ名とレジストリーを決定するために使用されます。
- ② プロジェクトオブジェクトモデル (POM) ファイルには、依存関係に関する情報などのプロジェクト設定が含まれています。このファイルを変更して、別の依存関係を追加できます。

#### 追加の依存関係の例

```

...
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>

```

```

    <scope>test</scope>
  </dependency>
</dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>3.8.0</version>
  <scope>test</scope>
</dependency>
</dependencies>
...

```

依存関係は、最初のコンパイル時にダウンロードされます。

- 3 関数プロジェクトには、**@Funq** アノテーションが付けられた Java メソッドが含まれている必要があります。このメソッドは **Function.java** クラスに配置できます。
- 4 関数のローカルでのテストに使用できる単純なテストケースが含まれます。

### 4.3. QUARKUS 関数の呼び出しについて

CloudEvents に応答する Quarkus プロジェクトや、簡単な HTTP 要求に応答する Quarkus プロジェクトを作成できます。Knative の CloudEvents は HTTP 経由で POST 要求として転送されるため、いずれかの関数タイプは受信 HTTP 要求をリスンして応答します。

受信要求が受信されると、Quarkus 関数は使用可能なタイプのインスタンスと合わせて呼び出されま

表4.1 関数呼び出しオプション

呼び出しメソッド	インスタンスに含まれるデータタイプ	データの例
HTTP POST 要求	要求のボディに含まれる JSON オブジェクト	<code>{ "customerId": "0123456", "productId": "6543210" }</code>
HTTP GET 要求	クエリー文字列のデータ	<code>? customerId=0123456&amp;productId=6543210</code>
<b>CloudEvent</b>	<b>data</b> プロパティの JSON オブジェクト	<code>{ "customerId": "0123456", "productId": "6543210" }</code>

以下の例は、以前の表に記載されている **customerId** および **productId** の購入データを受信して処理する関数です。

#### Quarkus 関数の例

```

public class Functions {
  @Funq
  public void processPurchase(Purchase purchase) {

```

```

    // process the purchase
  }
}

```

購入データが含まれる、該当の **Purchase** JavaBean クラスは以下のようになります。

### クラスの例

```

public class Purchase {
    private long customerId;
    private long productId;
    // getters and setters
}

```

#### 4.3.1. StorageLocation の例

以下のコード例は、**withBeans**、**withCloudEvent**、および **withBinary** の3つの関数を定義します。

#### 例

```

import io.quarkus.funqy.Funq;
import io.quarkus.funqy.knative.events.CloudEvent;

public class Input {
    private String message;

    // getters and setters
}

public class Output {
    private String message;

    // getters and setters
}

public class Functions {
    @Funq
    public Output withBeans(Input in) {
        // function body
    }

    @Funq
    public CloudEvent<Output> withCloudEvent(CloudEvent<Input> in) {
        // function body
    }

    @Funq
    public void withBinary(byte[] in) {
        // function body
    }
}

```

**Functions** クラスの **withBeans** 機能は、以下の方法で呼び出すことができます。

- JSON ボディが含まれる HTTP POST 要求:

```
$ curl "http://localhost:8080/withBeans" -X POST \
-H "Content-Type: application/json" \
-d '{"message": "Hello there."}'
```

- クエリーパラメーターが含まれる HTTP GET 要求:

```
$ curl "http://localhost:8080/withBeans?message=Hello%20there." -X GET
```

- バイナリーエンコーディングの **CloudEvent** オブジェクト:

```
$ curl "http://localhost:8080/" -X POST \
-H "Content-Type: application/json" \
-H "Ce-SpecVersion: 1.0" \
-H "Ce-Type: withBeans" \
-H "Ce-Source: cURL" \
-H "Ce-Id: 42" \
-d '{"message": "Hello there."}'
```

- 構造化されたエンコーディングでの **CloudEvent** オブジェクト:

```
$ curl http://localhost:8080/ \
-H "Content-Type: application/cloudevents+json" \
-d '{"data": {"message": "Hello there."},
  "datacontenttype": "application/json",
  "id": "42",
  "source": "curl",
  "type": "withBeans",
  "specversion": "1.0"}'
```

**Functions** クラスの **withCloudEvent** 機能は、**withBeans** 関数と同様に **CloudEvent** オブジェクトを使用して呼び出すことができます。ただし、**withBeans** とは異なり、**withCloudEvent** はプレーン HTTP 要求で呼び出すことはできません。

**Functions** クラスの **withBinary** 関数は、以下に呼び出すことができます。

- バイナリーエンコーディングの **CloudEvent** オブジェクト:

```
$ curl "http://localhost:8080/" -X POST \
-H "Content-Type: application/octet-stream" \
-H "Ce-SpecVersion: 1.0" \
-H "Ce-Type: withBinary" \
-H "Ce-Source: cURL" \
-H "Ce-Id: 42" \
--data-binary '@img.jpg'
```

- 構造化されたエンコーディングでの **CloudEvent** オブジェクト:

```
$ curl http://localhost:8080/ \
-H "Content-Type: application/cloudevents+json" \
-d '{"data_base64": "$(base64 --wrap=0 img.jpg)",
  "datacontenttype": "application/octet-stream",
  "id": "42",
```

```
"source": "curl",
"type": "withBinary",
"specversion": "1.0"}
```

## 4.4. CLOUDEVENT 属性

CloudEvent の属性 (**type**、**subject** など) を読み取るか、書き込む必要がある場合は、**CloudEvent<T>** 汎用インターフェイスおよび **CloudEventBuilder** ビルダーを使用できます。<T> タイプパラメーターは使用可能なタイプのいずれかでなければなりません。

以下の例では、**CloudEventBuilder** を使用して、購入処理の成功または失敗を返します。

```
public class Functions {

    private boolean _processPurchase(Purchase purchase) {
        // do stuff
    }

    public CloudEvent<Void> processPurchase(CloudEvent<Purchase> purchaseEvent) {
        System.out.println("subject is: " + purchaseEvent.subject());

        if (!_processPurchase(purchaseEvent.data())) {
            return CloudEventBuilder.create()
                .type("purchase.error")
                .build();
        }
        return CloudEventBuilder.create()
            .type("purchase.success")
            .build();
    }
}
```

## 4.5. QUARKUS 関数の戻り値

関数は、許可された型のリストから任意の型のインスタンスを返すことができます。または、**Uni<T>** 型を返すこともできます。ここで、<T> 型パラメーターは、許可されている型の任意の型にすることができます。

**Uni<T>** タイプは、返されるオブジェクトが受信したオブジェクトと同じ形式でシリアライズされるため、関数が非同期 API を呼び出す場合に便利です。以下に例を示します。

- 関数が HTTP 要求を受信すると、返されるオブジェクトが HTTP 応答のボディに送信されません。
- 関数がバイナリーエンコーディングで **CloudEvent** オブジェクトを受信する場合に、返されるオブジェクトはバイナリーエンコードされた **CloudEvent** オブジェクトの data プロパティで送信されます。

以下の例は、購入リストを取得する関数を示しています。

### コマンドの例

```
public class Functions {
    @Funq
```

```
public List<Purchase> getPurchasesByName(String name) {
    // logic to retrieve purchases
}
}
```

- HTTP 要求経由でこの関数を呼び出すと、応答のボディに購入された一覧が含まれる HTTP 応答が生成されます。
- 受信 **CloudEvent** オブジェクト経由でこの関数を呼び出すと、**data** プロパティの購入リストが含まれる **CloudEvent** 応答が生成されます。

#### 4.5.1. 使用可能なタイプ

関数の入力と出力は、**void**、**String**、または **byte[]** 型のいずれかです。さらに、プリミティブ型とそのラッパー (**int** や **Integer** など) にすることもできます。これらは、JavaBean、マップ、リスト、配列、および特殊な **CloudEvents<T>** タイプの複合オブジェクトにすることもできます。

マップ、リスト、配列、**CloudEvents<T>** 型の **<T>** 型パラメーター、および JavaBeans の属性は、ここにリストされている型のみに行うことができます。

#### 例

```
public class Functions {
    public List<Integer> getIds();
    public Purchase[] getPurchasesByName(String name);
    public String getNameById(int id);
    public Map<String,Integer> getNamedMapping();
    public void processImage(byte[] img);
}
```

## 4.6. QUARKUS 関数のテスト

Quarkus 関数は、コンピューターに対してローカルでテストできます。**kn func create** を使用して関数を作成するときに作成されるデフォルトプロジェクトには、基本的な Maven テストを含む **src/test/** ディレクトリーがあります。これらのテストは、必要に応じて拡張できます。

#### 前提条件

- Quarkus 関数を作成している。
- Knative (**kn**) CLI がインストールされている。

#### 手順

1. 関数のプロジェクトフォルダーに移動します。
2. Maven テストを実行します。

```
$ ./mvnw test
```

## 4.7. 次のステップ

- 関数を **構築** して **デプロイ** します。

## 第5章 NODE.JS 関数の開発

Node.js 関数プロジェクトを作成したら、指定のテンプレートを変更して、関数にビジネスロジックを追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

### 5.1. 前提条件

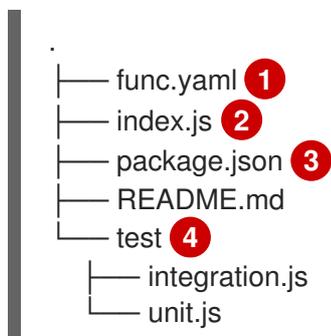
- 関数を開発する前に、[OpenShift Serverless 機能の設定](#) の設定手順を完了している。

### 5.2. NODE.JS 関数テンプレート構造

Knative (**kn**) CLI を使用して Node.js 関数を作成すると、プロジェクトディレクトリーは典型的な Node.js プロジェクトのようになります。唯一の例外は、関数の設定に使用される追加の **func.yaml** ファイルです。

**http** および **event** トリガー関数のテンプレート構造はいずれも同じです。

#### テンプレート構造



- func.yaml** 設定ファイルは、イメージ名とレジストリーを判断するために使用されます。
- プロジェクトに関数を1つエクスポートする **index.js** ファイルを追加する必要があります。
- テンプレート **package.json** ファイルにある依存関係に限定されるわけではありません。他の Node.js プロジェクトと同様に、別の依存関係を追加できます。

#### npm 依存関係の追加例

```
npm install --save opossum
```

デプロイメント用にプロジェクトをビルドすると、これらの依存関係は作成したランタイムコンテナイメージに含まれます。

- 統合およびテストスクリプトは、関数テンプレートに含まれます。

### 5.3. NODE.JS 関数の呼び出しについて

Knative (**kn**) CLI を使用して関数プロジェクトを作成する場合に、CloudEvents に応答するプロジェクト、または単純な HTTP 要求に応答するプロジェクトを生成できます。Knative の CloudEvents は HTTP 経由で POST 要求として転送されるため、関数タイプはいずれも受信 HTTP イベントをリッスンして応答します。

Node.js 関数は、単純な HTTP 要求で呼び出すことができます。受信要求を受け取ると、関数は **context** オブジェクトで最初のパラメーターとして呼び出されます。

### 5.3.1. Node.js コンテキストオブジェクト

関数は、**context** オブジェクトを最初のパラメーターとして渡して呼び出されます。このオブジェクトは、受信 HTTP 要求情報へのアクセスを提供します。

#### コンテキストオブジェクトの例

```
function handle(context, data)
```

この情報には、HTTP リクエストメソッド、リクエストと共に送信されたクエリー文字列またはヘッダー、HTTP バージョン、およびリクエスト本文が含まれます。**CloudEvent** の受信インスタンスが含まれる受信要求はコンテキストオブジェクトにアタッチし、**context.cloudevent** を使用してアクセスできるようにします。

#### 5.3.1.1. コンテキストオブジェクトメソッド

**context** オブジェクトには、データの値を受け入れ、CloudEvent を返す **cloudEventResponse()** メソッドが1つあります。

Knative システムでは、サービスとしてデプロイされた関数が CloudEvent を送信するイベントブローカーによって呼び出される場合に、ブローカーが応答を確認します。応答が CloudEvent の場合、このイベントはブローカーによって処理されます。

#### コンテキストオブジェクトメソッドの例

```
// Expects to receive a CloudEvent with customer data
function handle(context, customer) {
  // process the customer
  const processed = handle(customer);
  return context.cloudEventResponse(customer)
    .source('/handle')
    .type('fn.process.customer')
    .response();
}
```

#### 5.3.1.2. CloudEvent data

受信要求が CloudEvent の場合は、CloudEvent に関連付けられたデータがすべてイベントから抽出され、2 番目のパラメーターとして提供されます。たとえば、以下のように data プロパティに JSON 文字列が含まれる CloudEvent が受信されると、以下ようになります。

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

呼び出されると、**context** オブジェクトの後の関数の 2 番目のパラメーターは、**customerId** プロパティと **productId** プロパティを持つ JavaScript オブジェクトになります。

#### 署名の例

```
function handle(context, data)
```

この例の **data** パラメーターは、**customerId** および **productId** プロパティーが含まれる JavaScript オブジェクトです。

## 5.4. NODE.JS 関数の戻り値

関数は、有効な JavaScript タイプを返すことも、戻り値を持たないこともできます。関数に戻り値が指定されておらず、失敗を指定しないと、呼び出し元は **204 No Content** 応答を受け取ります。

関数は、CloudEvent または **Message** オブジェクトを返してイベントを Knative Eventing システムにプッシュすることもできます。この場合、開発者は CloudEvent メッセージング仕様を理解したり実装したりする必要はありません。返された値からのヘッダーおよびその他の関連情報は抽出され、応答で送信されます。

### 例

```
function handle(context, customer) {  
  // process customer and return a new CloudEvent  
  return new CloudEvent({  
    source: 'customer.processor',  
    type: 'customer.processed'  
  })  
}
```

### 5.4.1. 返されるヘッダー

**headers** プロパティーを **return** オブジェクトに追加して応答ヘッダーを設定できます。これらのヘッダーは抽出され、呼び出し元に応答して送信されます。

#### 応答ヘッダーの例

```
function handle(context, customer) {  
  // process customer and return custom headers  
  // the response will be '204 No content'  
  return { headers: { customerId: customer.id } };  
}
```

### 5.4.2. 返されるステータスコード

**statusCode** プロパティーを **return** オブジェクトに追加して、呼び出し元に返されるステータスコードを設定できます。

#### ステータスコード

```
function handle(context, customer) {  
  // process customer  
  if (customer.restricted) {  
    return { statusCode: 451 }  
  }  
}
```

ステータスコードは、関数で作成および出力されるエラーに対して設定することもできます。

### エラーステータスコードの例

```
function handle(context, customer) {  
  // process customer  
  if (customer.restricted) {  
    const err = new Error('Unavailable for legal reasons');  
    err.statusCode = 451;  
    throw err;  
  }  
}
```

## 5.5. NODE.JS 関数のテスト

Node.js 関数は、コンピューターに対してローカルでテストできます。**kn func create** を使用して関数を作成する際に作成されるデフォルトプロジェクトには、簡単なユニットテストおよびインテグレーションテストが含まれる **test** フォルダがあります。

### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- **kn func create** を使用して関数を作成している。

### 手順

1. 関数の **test** フォルダに移動します。
2. テストを実行します。

```
$ npm test
```

## 5.6. 次のステップ

- [Node.js コンテキストオブジェクトの参照ドキュメント](#)を参照してください。
- 関数を **構築** して **デプロイ** します。

## 第6章 TYPESCRIPT 関数の開発

TypeScript 関数プロジェクトを作成したら、指定のテンプレートを変更して、関数にビジネスロジックを追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

### 6.1. 前提条件

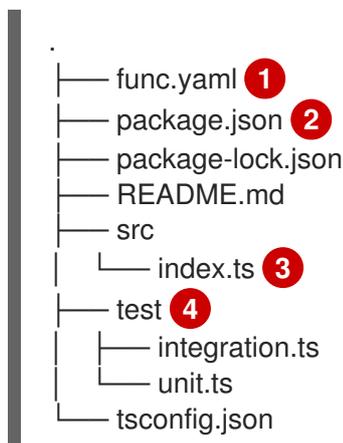
- 関数を開発する前に、[OpenShift Serverless 機能の設定](#) の設定手順を完了している。

### 6.2. TYPESCRIPT 関数テンプレートの構造

Knative (**kn**) CLI を使用して TypeScript 関数を作成すると、プロジェクトディレクトリーは典型的な TypeScript プロジェクトのようになります。唯一の例外は、関数の設定に使用される追加の **func.yaml** ファイルです。

**http** および **event** トリガー関数のテンプレート構造はいずれも同じです。

#### テンプレート構造



- func.yaml** 設定ファイルは、イメージ名とレジストリーを判断するために使用されます。
- テンプレート **package.json** ファイルにある依存関係に限定されるわけではありません。他の TypeScript プロジェクトと同様に、別の依存関係を追加できます。

#### npm 依存関係の追加例

```
npm install --save opossum
```

デプロイメント用にプロジェクトをビルドすると、これらの依存関係は作成したランタイムコンテナイメージに含まれます。

- プロジェクトには、**handle** という名前の関数をエクスポートする **src/index.js** ファイルが含まれている必要があります。
- 統合およびテストスクリプトは、関数テンプレートに含まれます。

### 6.3. TYPESCRIPT 関数の呼び出しについて

Knative (**kn**) CLI を使用して関数プロジェクトを作成する場合に、CloudEvents に応答するプロジェク

ト、または単純な HTTP 要求に応答するプロジェクトを生成できます。Knative の CloudEvents は HTTP 経由で POST 要求として転送されるため、関数タイプはいずれも受信 HTTP イベントをリッスンして応答します。

Typescript 関数は、単純な HTTP 要求で呼び出すことができます。受信要求を受け取ると、関数は **context** オブジェクトで最初のパラメーターとして呼び出されます。

### 6.3.1. Typescript コンテキストオブジェクト

関数を呼び出すには、**context** オブジェクトを最初のパラメーターとして指定します。**context** オブジェクトのプロパティにアクセスすると、着信 HTTP 要求に関する情報を提供できます。

#### コンテキストオブジェクトの例

```
function handle(context:Context): string
```

この情報には、HTTP リクエストメソッド、リクエストと共に送信されたクエリー文字列またはヘッダー、HTTP バージョン、およびリクエスト本文が含まれます。**CloudEvent** の受信インスタンスが含まれる受信要求はコンテキストオブジェクトにアタッチし、**context.cloudevent** を使用してアクセスできるようにします。

#### 6.3.1.1. コンテキストオブジェクトメソッド

**context** オブジェクトには、データの値を受け入れ、CloudEvent を返す **cloudEventResponse()** メソッドが1つあります。

Knative システムでは、サービスとしてデプロイされた関数が CloudEvent を送信するイベントブローカーによって呼び出される場合に、ブローカーが応答を確認します。応答が CloudEvent の場合、このイベントはブローカーによって処理されます。

#### コンテキストオブジェクトメソッドの例

```
// Expects to receive a CloudEvent with customer data
export function handle(context: Context, cloudevent?: CloudEvent): CloudEvent {
  // process the customer
  const customer = cloudevent.data;
  const processed = processCustomer(customer);
  return context.cloudEventResponse(customer)
    .source('/customer/process')
    .type('customer.processed')
    .response();
}
```

#### 6.3.1.2. コンテキストタイプ

TypeScript タイプの定義ファイルは、関数で使用する以下のタイプをエクスポートします。

#### エクスポートタイプの定義

```
// Invokable is the expeted Function signature for user functions
export interface Invokable {
  (context: Context, cloudevent?: CloudEvent): any
}
```

```

// Logger can be used for structural logging to the console
export interface Logger {
  debug: (msg: any) => void,
  info: (msg: any) => void,
  warn: (msg: any) => void,
  error: (msg: any) => void,
  fatal: (msg: any) => void,
  trace: (msg: any) => void,
}

// Context represents the function invocation context, and provides
// access to the event itself as well as raw HTTP objects.
export interface Context {
  log: Logger;
  req: IncomingMessage;
  query?: Record<string, any>;
  body?: Record<string, any>|string;
  method: string;
  headers: IncomingHttpHeaders;
  httpVersion: string;
  httpVersionMajor: number;
  httpVersionMinor: number;
  cloudevent: CloudEvent;
  cloudEventResponse(data: string|object): CloudEventResponse;
}

// CloudEventResponse is a convenience class used to create
// CloudEvents on function returns
export interface CloudEventResponse {
  id(id: string): CloudEventResponse;
  source(source: string): CloudEventResponse;
  type(type: string): CloudEventResponse;
  version(version: string): CloudEventResponse;
  response(): CloudEvent;
}

```

### 6.3.1.3. CloudEvent data

受信要求が CloudEvent の場合は、CloudEvent に関連付けられたデータがすべてイベントから抽出され、2 番目のパラメーターとして提供されます。たとえば、以下のように data プロパティーに JSON 文字列が含まれる CloudEvent が受信されると、以下のようになります。

```

{
  "customerId": "0123456",
  "productId": "6543210"
}

```

呼び出されると、**context** オブジェクトの後の関数の 2 番目のパラメーターは、**customerId** プロパティーと **productId** プロパティーを持つ JavaScript オブジェクトになります。

#### 署名の例

```

function handle(context: Context, cloudevent?: CloudEvent): CloudEvent

```

この例の **cloudevent** パラメーターは、**customerId** および **productId** プロパティが含まれる JavaScript オブジェクトです。

## 6.4. TYPESCRIPT 関数の戻り値

関数は、有効な JavaScript タイプを返すことも、戻り値を持たないこともできます。関数に戻り値が指定されておらず、失敗を指定しないと、呼び出し元は **204 No Content** 応答を受け取ります。

関数は、CloudEvent または **Message** オブジェクトを返してイベントを Knative Eventing システムにプッシュすることもできます。この場合、開発者は CloudEvent メッセージング仕様を理解したり実装したりする必要はありません。返された値からのヘッダーおよびその他の関連情報は抽出され、応答で送信されます。

### 例

```
export const handle: Invokable = function (
  context: Context,
  cloudevent?: CloudEvent
): Message {
  // process customer and return a new CloudEvent
  const customer = cloudevent.data;
  return HTTP.binary(
    new CloudEvent({
      source: 'customer.processor',
      type: 'customer.processed'
    })
  );
};
```

### 6.4.1. 返されるヘッダー

**headers** プロパティを **return** オブジェクトに追加して応答ヘッダーを設定できます。これらのヘッダーは抽出され、呼び出し元に応答して送信されます。

#### 応答ヘッダーの例

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer and return custom headers
  const customer = cloudevent.data as Record<string, any>;
  return { headers: { 'customer-id': customer.id } };
}
```

### 6.4.2. 返されるステータスコード

**statusCode** プロパティを **return** オブジェクトに追加して、呼び出し元に返されるステータスコードを設定できます。

#### ステータスコード

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
```

```

return {
  statusCode: 451
}
}
// business logic, then
return {
  statusCode: 240
}
}
}

```

ステータスコードは、関数で作成および出力されるエラーに対して設定することもできます。

### エラーステータスコードの例

```

export function handle(context: Context, cloudevent?: CloudEvent): Record<string, string> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
    err.statusCode = 451;
    throw err;
  }
}
}

```

## 6.5. TYPESCRIPT 関数のテスト

Typescript 機能は、お使いのコンピューターでローカルでテストできます。**kn func create** を使用して関数を作成するときに作成されるデフォルトのプロジェクトには、いくつかの単純な単体テストと統合テストを含む `test` フォルダがあります。

### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- **kn func create** を使用して関数を作成している。

### 手順

1. テストを実行していない場合は、最初に依存関係をインストールします。

```
$ npm install
```

2. 関数の `test` フォルダに移動します。
3. テストを実行します。

```
$ npm test
```

## 6.6. 次のステップ

- [TypeScript コンテキストオブジェクトの参照](#) ドキュメントを参照してください。
- 関数を [構築](#) して [デプロイ](#) します。
- 関数に関するロギングの詳細は、[Pino API のドキュメント](#) を参照してください。

## 第7章 PYTHON 関数の開発



### 重要

Python を使用した OpenShift Serverless Functions は、テクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、[テクノロジープレビュー機能のサポート範囲](#) を参照してください。

[Python 関数プロジェクトを作成](#) したら、指定したテンプレートファイルを変更して、ビジネスロジックを機能に追加できます。これには、関数呼び出しと返されるヘッダーとステータスコードの設定が含まれます。

### 7.1. 前提条件

- 関数を開発する前に、[OpenShift Serverless 機能の設定](#) の設定手順を完了している。

### 7.2. PYTHON 関数テンプレート構造

Knative (**kn**) CLI を使用して Python 関数を作成する場合、プロジェクトディレクトリーは一般的な Python プロジェクトと似ています。Python 関数にはいくつかの制限があります。プロジェクトの要件として唯一、**main()** 関数と **func.yaml** 設定ファイルで設定される **func.py** が含まれることが挙げられます。

開発者は、テンプレート **requirements.txt** ファイルにある依存関係しか使用できないわけではありません。その他の依存関係は、他の Python プロジェクトに配置されるように追加できます。デプロイメント用にプロジェクトをビルドすると、これらの依存関係は作成したランタイムコンテナイメージに含まれます。

**http** および **event** トリガー関数のテンプレート構造はいずれも同じです。

#### テンプレート構造

```
fn
├── func.py ①
├── func.yaml ②
├── requirements.txt ③
└── test_func.py ④
```

- 1** **main()** 関数が含まれます。
- 2** イメージ名とレジストリーを決定するために使用されます。
- 3** その他の依存関係は、他の Python プロジェクトにあるため、**requirements.txt** ファイルに追加できます。
- 4** 関数のローカルでのテストに使用できる単純なユニットテストが含まれます。

## 7.3. PYTHON 関数の呼び出しについて

Python 関数は、単純な HTTP 要求で呼び出すことができます。受信要求を受け取ると、関数は **context** オブジェクトで最初のパラメーターとして呼び出されます。

**context** オブジェクトは、2つの属性を持つ Python クラスです。

- **request** 属性は常に存在し、Flask リクエスト オブジェクトが含まれます。
- 2番目の属性 **cloud\_event** は、受信リクエストが **CloudEvent** オブジェクトの場合に設定されます。

開発者は、コンテキストオブジェクトからすべての **CloudEvent** データにアクセスできます。

### コンテキストオブジェクトの例

```
def main(context: Context):
    """
    The context parameter contains the Flask request object and any
    CloudEvent received with the request.
    """
    print(f"Method: {context.request.method}")
    print(f"Event data {context.cloud_event.data}")
    # ... business logic here
```

## 7.4. PYTHON 関数の戻り値

関数は、Flask でサポートされている任意の値を返すことができます。これは、呼び出しフレームワークがこれらの値を Flask サーバーに直接プロキシするためです。

### 例

```
def main(context: Context):
    body = { "message": "Howdy!" }
    headers = { "content-type": "application/json" }
    return body, 200, headers
```

関数は、関数呼び出しの 2 番目および 3 番目の応答値として、ヘッダーと応答コードの両方を設定できます。

### 7.4.1. Returning CloudEvents

開発者は **@event** デコレーターを使用して、呼び出し元に対して、応答を送信する前に関数の戻り値を **CloudEvent** に変換する必要があることを指示できます。

### 例

```
@event("event_source"="/my/function", "event_type"="my.type")
def main(context):
    # business logic here
    data = do_something()
    # more data processing
    return data
```

この例では、タイプが **"my.type"**、ソースが **"/my/function"** の応答値として CloudEvent を送信します。CloudEvent **data プロパティ** は、返された **data** 変数に設定されます。**event\_source** および **event\_type** デコレーター属性は任意です。

## 7.5. PYTHON 関数のテスト

Python 機能は、お使いのコンピューターのローカルにテストできます。デフォルトのプロジェクトには、関数の単純な単体テストを提供する **test\_func.py** ファイルが含まれています。



### 注記

Python 関数のデフォルトのテストフレームワークは **unittest** です。必要に応じて、別のテストフレームワークを使用できます。

### 前提条件

- Python 関数テストをローカルで実行するには、必要な依存関係をインストールする必要があります。

```
$ pip install -r requirements.txt
```

### 手順

1. **test\_func.py** ファイルが含まれる関数のフォルダーに移動します。
2. テストを実行します。

```
$ python3 test_func.py
```

## 7.6. 次のステップ

- 関数を **構築** して **デプロイ** します。

## 第8章 KNATIVE EVENTING での関数の使用

関数は、OpenShift Container Platform クラスターに Knative サービスとしてデプロイされます。機能を Knative Eventing コンポーネントに接続して、受信イベントを受信できるようにすることができます。

### 8.1. 開発者パースペクティブを使用してイベントソースを機能に接続する

関数は、OpenShift Container Platform クラスターに Knative サービスとしてデプロイされます。OpenShift Container Platform Web コンソールを使用してイベントソースを作成すると、そのソースからイベントが送信されるデプロイ済み機能を指定できます。

#### 前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing が OpenShift Container Platform クラスターにインストールされている。
- Web コンソールにログインしており、**Developer** パースペクティブを使用している。
- OpenShift Container Platform でアプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションを持つプロジェクトにアクセスできる。
- 機能を作成してデプロイしている。

#### 手順

1. **+Add** → **Event Source** に移動して任意のタイプのイベントソースを作成し、作成するイベントソースを選択します。
2. **Create Event Source** フォームビューの **Sink** セクションで、**Resource** リストで機能を選択します。
3. **Create** をクリックします。

#### 検証

**Topology** ページを表示して、イベントソースが作成され、機能に接続されていることを確認できます。

1. **Developer** パースペクティブで、**Topology** に移動します。
2. イベントソースを表示し、接続された機能をクリックして、右側のパネルに機能の詳細を表示します。

## 第9章 FUNC.YAML の関数プロジェクト設定

**func.yaml** ファイルには、関数プロジェクトの設定が含まれます。**kn func** コマンドを実行すると、**func.yaml** に指定された値が使用されます。たとえば、**kn func build** コマンドを実行すると、**build** フィールドの値が使用されます。一部のケースでは、この値はコマンドラインフラグまたは環境変数で上書きできます。

### 9.1. FUNC.YAML の設定可能なフィールド

**func.yaml** のフィールドの多くは、関数の作成、ビルド、およびデプロイ時に自動的に生成されます。ただし、関数名またはイメージ名などの変更用に手動で変更するフィールドもあります。

#### 9.1.1. buildEnvs

**buildEnvs** フィールドを使用すると、関数をビルドする環境で利用できる環境変数を設定できます。**envs** を使用して設定する変数とは異なり、**buildEnv** を使用して設定する変数は、関数の実行時には使用できません。

**buildEnv** 変数を値から直接設定できます。以下の例では、**EXAMPLE1** という名前の **buildEnv** 変数に値 **one** が直接割り当てられます。

```
buildEnvs:
- name: EXAMPLE1
  value: one
```

また、ローカルの環境変数から **buildEnv** 変数を設定することもできます。以下の例では、**EXAMPLE2** という名前の **buildEnv** 変数にローカル環境変数 **LOCAL\_ENV\_VAR** の値が割り当てられます。

```
buildEnvs:
- name: EXAMPLE1
  value: '{{ env:LOCAL_ENV_VAR }}'
```

#### 9.1.2. envs

**envs** フィールドを使用すると、ランタイム時に関数でできるように環境変数を設定できます。環境変数は、複数の異なる方法で設定できます。

1. 値から直接設定します。
2. ローカル環境変数に割り当てられた値から設定します。詳細は、**func.yaml** フィールドからのローカル環境変数の参照のセクションを参照してください。
3. シークレットまたは Config Map に格納されているキーと値のペアから設定します。
4. 作成された環境変数の名前として使用されるキーを使用して、シークレットまたは Config Map に格納されているすべてのキーと値のペアをインポートすることもできます。

以下の例は、環境変数を設定するさまざまな方法を示しています。

```
name: test
namespace: ""
runtime: go
...
envs:
```

```

- name: EXAMPLE1 ❶
  value: value
- name: EXAMPLE2 ❷
  value: '{{ env:LOCAL_ENV_VALUE }}'
- name: EXAMPLE3 ❸
  value: '{{ secret:mysecret:key }}'
- name: EXAMPLE4 ❹
  value: '{{ configMap:myconfigmap:key }}'
- value: '{{ secret:mysecret2 }}' ❺
- value: '{{ configMap:myconfigmap2 }}' ❻

```

- ❶ 値から直接設定された環境変数。
- ❷ ローカル環境変数に割り当てられた値から設定された環境変数。
- ❸ シークレットに格納されているキーと値のペアから割り当てられた環境変数。
- ❹ Config Map に保存されるキーと値のペアから割り当てられる環境変数。
- ❺ シークレットのキーと値のペアからインポートされた環境変数のセット。
- ❻ Config Map のキーと値のペアからインポートされた環境変数のセット。

### 9.1.3. builder

**builder** フィールドは、機能がイメージを構築するために使用する戦略を指定します。 **pack** または **s2i** の値を受け入れます。

### 9.1.4. build

**build** フィールドは、機能を構築する方法を示します。値 **local** は、機能がマシン上でローカルに構築されていることを示します。値 **git** は、関数が **git** フィールドで指定された値を使用してクラスター上に構築されることを示します。

### 9.1.5. volumes

以下の例のように、**volumes** フィールドを使用すると、指定したパスで関数にアクセスできるボリュームとしてシークレットと Config Map をマウントできます。

```

name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret ❶
  path: /workspace/secret
- configMap: myconfigmap ❷
  path: /workspace/configmap

```

- ❶ **mysecret** シークレットは、 **/workspace/secret** にあるボリュームとしてマウントされます。
- ❷ **myconfigmap** Config Map は、 **/workspace/configmap** にあるボリュームとしてマウントされます。

## 9.1.6. オプション

**options** フィールドを使用すると、自動スケーリングなど、デプロイされた関数の Knative Service プロパティを変更できます。これらのオプションが設定されていない場合は、デフォルトのオプションが使用されます。

これらのオプションを利用できます。

- **scale**
  - **min**: レプリカの最小数。負ではない整数でなければなりません。デフォルトは 0 です。
  - **max**: レプリカの最大数。負ではない整数でなければなりません。デフォルトは 0 で、これは制限がないことを意味します。
  - **metric**: Autoscaler によって監視されるメトリクスタイプを定義します。これは、デフォルトの **concurrency**、または **rps** に設定できます。
  - **target**: 同時に受信する要求の数に基づくスケールアップのタイミングの推奨。 **target** オプションは、0.01 より大きい浮動小数点値を指定できます。 **options.resources.limits.concurrency** が設定されていない限り、デフォルトは 100 になります。この場合、 **target** はデフォルトでその値になります。
  - **utilization**: スケールアップする前に許可された同時リクエスト使用率のパーセンテージ。1 から 100 までの浮動小数点値を指定できます。デフォルトは 70 です。
- **resources**
  - **requests**
    - **cpu**: デプロイされた関数を持つコンテナの CPU リソース要求。
    - **memory**: デプロイされた関数を持つコンテナのメモリーリソース要求。
  - **limits**
    - **cpu**: デプロイされた関数を持つコンテナの CPU リソース制限。
    - **memory**: デプロイされた関数を持つコンテナのメモリーリソース制限。
    - **concurrency**: 単一レプリカによって処理される同時要求のハード制限。0 以上の整数値を指定できます。デフォルトは 0 です (制限なしを意味します)。

これは、 **scale** オプションの設定例です。

```
name: test
namespace: ""
runtime: go
...
options:
  scale:
    min: 0
    max: 10
    metric: concurrency
    target: 75
    utilization: 75
  resources:
    requests:
```

```
cpu: 100m
memory: 128Mi
limits:
  cpu: 1000m
  memory: 256Mi
concurrency: 100
```

### 9.1.7. image

**image** フィールドは、関数がビルドされた後の関数のイメージ名を設定します。このフィールドは必要に応じて変更できます。変更する場合は、次に **kn func build** または **kn func deploy** を実行すると、関数イメージが新しい名前で作成されます。

### 9.1.8. imageDigest

**imageDigest** フィールドには、関数のデプロイ時のイメージマニフェストの SHA256 ハッシュが含まれます。この値は変更しないでください。

### 9.1.9. labels

**labels** フィールドを使用すると、デプロイされた関数にラベルを設定できます。

値から直接ラベルを設定できます。以下の例では、**role** キーを持つラベルに **backend** の値が直接割り当てられます。

```
labels:
- key: role
  value: backend
```

ローカル環境変数からラベルを設定することもできます。以下の例では、**author** キーの付いたラベルに **USER** ローカル環境変数の値が割り当てられます。

```
labels:
- key: author
  value: '{{ env:USER }}'
```

### 9.1.10. name

**name** フィールドは、関数の名前を定義します。この値は、デプロイ時に Knative サービスの名前として使用されます。このフィールドを変更して、後続のデプロイメントで関数の名前を変更できます。

### 9.1.11. namespace

**namespace** フィールドは、関数がデプロイされる namespace を指定します。

### 9.1.12. runtime

**runtime** フィールドは、関数の言語ランタイムを指定します (例: **python**)。

## 9.2. FUNC.YAML フィールドからのローカル環境変数の参照

API キーなどの機密情報を関数設定に保存したくない場合は、ローカル環境で使用可能な環境変数への参照を追加できます。これを行うには、**func.yaml** ファイルの **envs** フィールドを変更します。

### 前提条件

- 関数プロジェクトを作成する必要があります。
- ローカル環境には、参照する変数が含まれている必要があります。

### 手順

- ローカル環境変数を参照するには、以下の構文を使用します。

```
  {{ env:ENV_VAR }}
```

**ENV\_VAR** を、使用するローカル環境の変数の名前に置き換えます。

たとえば、ローカル環境で **API\_KEY** 変数が利用可能な場合があります。その値を **MY\_API\_KEY** 変数に割り当てることができます。これにより、関数内で直接使用できます。

### 関数の例

```
name: test
namespace: ""
runtime: go
...
envs:
- name: MY_API_KEY
  value: '{{ env:API_KEY }}'
...
```

## 9.3. 関連情報

- [関数を使い始める](#)
- [Serverless 関数からのシークレットおよび Config Map へのアクセス](#)
- [自動スケーリングに関する Knative ドキュメント](#)
- [コンテナのリソースの管理に関する Kubernetes のドキュメント](#)
- [並行性の設定に関する Knative ドキュメント](#)

## 第10章 関数からのシークレットおよび CONFIG MAP へのアクセス

関数がクラスターにデプロイされた後に、それらはシークレットおよび Config Map に保存されているデータにアクセスできます。このデータはボリュームとしてマウントすることも、環境変数に割り当てることもできます。Knative CLI を使用して、このアクセスを対話的に設定するか、関数設定 YAML ファイルを編集して手動で設定できます。



### 重要

シークレットおよび Config Map にアクセスするには、関数をクラスターにデプロイする必要があります。この機能は、ローカルで実行している関数では利用できません。

シークレットまたは Config Map の値にアクセスできない場合、デプロイメントは失敗し、アクセスできない値を指定するエラーメッセージが表示されます。

### 10.1. シークレットおよび CONFIG MAP への関数アクセスの対話的な変更

**kn func config** 対話型ユーティリティーを使用して、関数がアクセスするシークレットおよび Config Map を管理できます。使用可能な操作には、config map とシークレットに環境変数として保存されている値の一覧表示、追加、および削除、およびボリュームの一覧表示、追加、および削除が含まれます。この機能を使用すると、クラスターに保存されているどのデータを関数からアクセスできるかを管理できます。

#### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- 関数を作成している。

#### 手順

1. 関数プロジェクトディレクトリーで以下のコマンドを実行します。

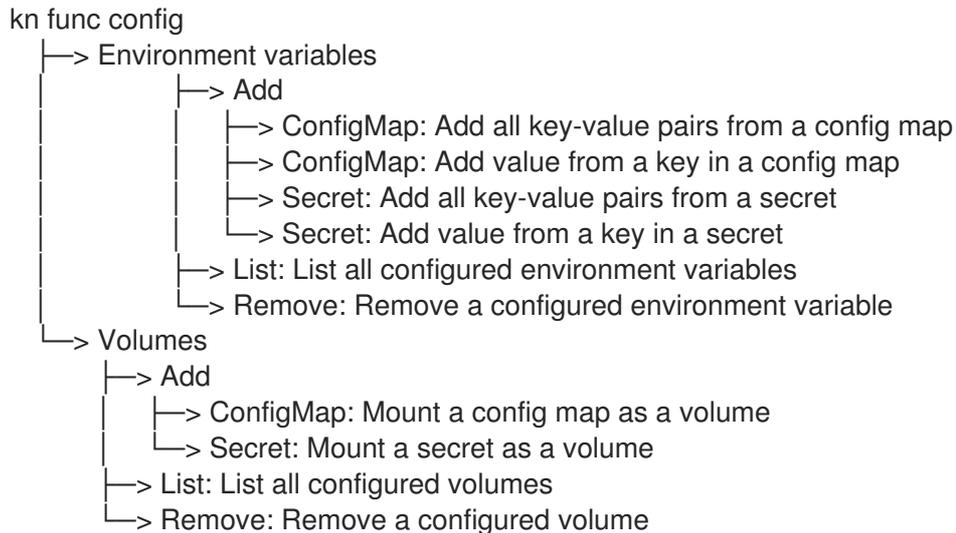
```
$ kn func config
```

あるいは、**--path** または **-p** オプションを使用して、関数プロジェクトディレクトリーを指定できます。

2. 対話型インターフェイスを使用して必要な操作を実行します。たとえば、ユーティリティーを使用して設定したボリュームの一覧を表示すると、以下のような出力が生成されます。

```
$ kn func config
? What do you want to configure? Volumes
? What operation do you want to perform? List
Configured Volumes mounts:
- Secret "mysecret" mounted at path: "/workspace/secret"
- Secret "mysecret2" mounted at path: "/workspace/secret2"
```

このスキームは、対話型ユーティリティーで利用可能なすべての操作と、それらに移動する方法を示しています。



3. オプション: 変更を反映させるため、関数をデプロイします。

```
$ kn func deploy -p test
```

## 10.2. 特殊なコマンドを使用したシークレットおよび CONFIG MAP への関数アクセスの対話的な変更

**kn func config** ユーティリティを実行するたびにダイアログ全体を移動して、直前のセクションで示されているように、必要な操作を選択する必要があります。ステップを保存するには、**kn func config** コマンドのより具体的なフォームを実行することで、特定の操作を直接実行します。

- 設定した環境変数を一覧表示するには、以下を実行します。

```
$ kn func config envs [-p <function-project-path>]
```

- 関数設定に環境変数を追加するには、以下を実行します。

```
$ kn func config envs add [-p <function-project-path>]
```

- 関数設定から環境変数を削除するには、以下を実行します。

```
$ kn func config envs remove [-p <function-project-path>]
```

- 設定したボリュームを一覧表示するには、以下を実行します。

```
$ kn func config volumes [-p <function-project-path>]
```

- 関数設定にボリュームを追加するには、以下を実行します。

```
$ kn func config volumes add [-p <function-project-path>]
```

- 関数設定からボリュームを削除するには、以下を実行します。

```
$ kn func config volumes remove [-p <function-project-path>]
```

## 10.3. シークレットおよび CONFIG MAP への関数アクセスの手動による追加

シークレットおよび Config Map にアクセスするための設定を手動で関数に追加できます。これは、既存の設定スニペットがある場合などに、**kn func config** 対話型ユーティリティーとコマンドを使用するよりも望ましい場合があります。

### 10.3.1. シークレットのボリュームとしてのマウント

シークレットをボリュームとしてマウントできます。シークレットがマウントされると、関数から通常のファイルとしてアクセスできます。これにより、関数がアクセスする必要がある URI のリストなど、関数が必要とするデータをクラスターに格納できます。

#### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- 関数を作成している。

#### 手順

1. 関数の **func.yaml** ファイルを開きます。
2. ボリュームとしてマウントするシークレットごとに、以下の YAML を **volumes** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret
  path: /workspace/secret
```

- **mysecret** をターゲットシークレットの名前に置き換えます。
- **/workspace/secret** は、シークレットをマウントするパスに置き換えます。たとえば、**addresses** シークレットをマウントするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/secret-addresses
```

3. 設定を保存します。

### 10.3.2. Config Map のボリュームとしてのマウント

Config Map をボリュームとしてマウントできます。Config Map がマウントされると、関数から通常のファイルとしてアクセスできます。これにより、関数がアクセスする必要がある URI のリストなど、関数が必要とするデータをクラスターに格納できます。

### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- 関数を作成している。

### 手順

1. 関数の **func.yaml** ファイルを開きます。
2. ボリュームとしてマウントする Config Map ごとに、以下の YAML を **volumes** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: myconfigmap
  path: /workspace/configmap
```

- **myconfigmap** をターゲット Config Map の名前に置き換えます。
- **/workspace/configmap** は、Config Map をマウントするパスに置き換えます。たとえば、**addresses** config map をマウントするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/configmap-addresses
```

3. 設定を保存します。

### 10.3.3. シークレットで定義されるキー値からの環境変数の設定

シークレットとして定義されたキー値から環境変数を設定できます。以前にシークレットに保存されていた値は、実行時に環境変数として関数がアクセスできます。これは、ユーザーの ID など、シークレットに格納されている値にアクセスする場合に役立ちます。

### 前提条件

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

- 関数を作成している。

## 手順

1. 関数の **func.yaml** ファイルを開きます。
2. 環境変数に割り当てる秘密鍵と値のペアからの値ごとに、以下の YAML を **envs** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ secret:mysecret:key }}'
```

- **EXAMPLE** を環境変数の名前に置き換えます。
- **mysecret** をターゲットシークレットの名前に置き換えます。
- **key** をターゲット値にマッピングしたキーに置き換えます。  
たとえば、**userdetailssecret** に保存されているユーザー ID にアクセスするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret:userid }}'
```

3. 設定を保存します。

### 10.3.4. Config Map で定義されるキー値からの環境変数の設定

config map として定義されたキー値から環境変数を設定できます。以前に config map に格納されていた値は、実行時に環境変数として関数がアクセスできます。これは、ユーザーの ID など、config map に格納されている値にアクセスするのに役立ちます。

#### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- 関数を作成している。

## 手順

1. 関数の **func.yaml** ファイルを開きます。
2. 環境変数に割り当てる Config Map のキーと値のペアからの値ごとに、以下の YAML を **envs** セクションに追加します。

```

name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ configMap:myconfigmap:key }}'

```

- **EXAMPLE** を環境変数の名前に置き換えます。
- **myconfigmap** をターゲット Config Map の名前に置き換えます。
- **key** をターゲット値にマッピングしたキーに置き換えます。  
たとえば、**userdetailsmap** に格納されているユーザー ID にアクセスするには、次の YAML を使用します。

```

name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap:userid }}'

```

3. 設定を保存します。

### 10.3.5. シークレットで定義されたすべての値からの環境変数の設定

シークレットで定義されているすべての値から環境変数を設定できます。以前にシークレットに保存されていた値は、実行時に環境変数として関数がアクセスできます。これは、シークレットに格納されている値のコレクション (ユーザーに関する一連のデータなど) に同時にアクセスする場合に役立ちます。

#### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- 関数を作成している。

#### 手順

1. 関数の **func.yaml** ファイルを開きます。
2. すべてのキーと値のペアを環境変数としてインポートするすべてのシークレットについて、以下の YAML を **envs** セクションに追加します。

```

name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ secret:mysecret }}' 1

```

- 1 **mysecret** をターゲットシークレットの名前に置き換えます。

たとえば、**userdetailssecret** に保存されているすべてのユーザー データにアクセスするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret }}'
```

3. 設定を保存します。

### 10.3.6. Config Map で定義されたすべての値からの環境変数の設定

config map で定義されたすべての値から環境変数を設定できます。以前に config map に格納されていた値は、実行時に環境変数として関数がアクセスできます。これは、config map に格納されている値のコレクション (ユーザーに関する一連のデータなど) に同時にアクセスする場合に役立ちます。

#### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- 関数を作成している。

#### 手順

1. 関数の **func.yaml** ファイルを開きます。
2. すべてのキーと値のペアを環境変数としてインポートするすべての Config Map について、以下の YAML を **envs** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:myconfigmap }}' 1
```

- 1 **myconfigmap** をターゲット Config Map の名前に置き換えます。

たとえば、**userdetailsmap** に保存されているすべてのユーザー データにアクセスするには、次の YAML を使用します。

```
name: test
namespace: ""
runtime: go
```

```
...  
envs:  
- value: '{{ configMap:userdetailsmap }}'
```

3. ファイルを保存します。

## 第11章 アノテーションの関数への追加

デプロイされたサーバーレス機能に Kubernetes アノテーションを追加できます。注釈を使用すると、関数の目的に関するメモなど、任意のメタデータを関数に添付できます。注釈は、**func.yaml** 設定ファイルの **annotations** セクションに追加されます。

関数アノテーション機能には、以下の2つの制限があります。

- 関数アノテーションがクラスター上の対応する Knative サービスに伝播されてからは、**func.yaml** ファイルから削除してもサービスから削除することができません。サービスの YAML ファイルを直接変更するか、OpenShift Container Platform Web コンソールを使用して、Knative サービスからアノテーションを削除する必要があります。
- **autoscaling** アノテーションなど、Knative によって設定されるアノテーションを設定することはできません。

### 11.1. 関数へのアノテーションの追加

関数にアノテーションを追加できます。ラベルと同様に、アノテーションはキーと値のマッピングとして定義されます。アノテーションは、関数の作成者など、関数に関するメタデータを提供する場合などに役立ちます。

#### 前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- 関数を作成している。

#### 手順

1. 関数の **func.yaml** ファイルを開きます。
2. 追加するすべてのアノテーションについて、以下の YAML を **annotations** セクションに追加します。

```
name: test
namespace: ""
runtime: go
...
annotations:
  <annotation_name>: "<annotation_value>" ❶
```

- ❶ **<annotation\_name>: "<annotation\_value>"** をお使いのアノテーションに置き換えます。

たとえば、関数が Alice によって作成者されたことを示すには、以下のアノテーションを含めることができます。

```
name: test
namespace: ""
runtime: go
```

```
...  
annotations:  
  author: "alice@example.com"
```

3. 設定を保存します。

次に関数をクラスターにデプロイすると、アノテーションが対応する Knative サービスに追加されます。

## 第12章 関数開発リファレンスガイド

OpenShift Serverless Functions は、基本的な関数を作成するために使用できるテンプレートを提供します。テンプレートは、関数プロジェクトのボイラープレートを開始し、**kn func** ツールで使用できるように準備します。各関数テンプレートは特定のランタイム用に調整されており、その規則に従います。テンプレートを使用すると、関数プロジェクトを自動的に開始できます。

次のランタイムのテンプレートが利用可能です。

- [Node.js](#)
- [Quarkus](#)
- [TypeScript](#)

### 12.1. NODE.JS コンテキストオブジェクトのリファレンス

**context** オブジェクトには、関数開発者が利用可能なプロパティが複数あります。これらのプロパティにアクセスすると、HTTP 要求に関する情報が提供され、出力がクラスターログに書き込まれます。

#### 12.1.1. log

出力をクラスターロギングに書き込むために使用可能なロギングオブジェクトを提供します。ログは [Pino logging API](#) に準拠します。

##### ログの例

```
function handle(context) {
  context.log.info("Processing customer");
}
```

**kn func invoke** コマンドを使用して、この関数にアクセスできます。

##### コマンドの例

```
$ kn func invoke --target 'http://example.function.com'
```

##### 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Processing customer"}
```

ログレベルは、**fatal**、**error**、**warn**、**info**、**debug**、**trace**、または **silent** のいずれかに設定できます。これを実行するには、**config** コマンドを使用してこれらの値のいずれかを環境変数 **FUNC\_LOG\_LEVEL** に割り当てて、**logLevel** の値を変更します。

#### 12.1.2. query

要求のクエリー文字列 (ある場合) をキーと値のペアとして返します。これらの属性はコンテキストオブジェクト自体にも表示されます。

##### サンプルクエリー

```
function handle(context) {  
  // Log the 'name' query parameter  
  context.log.info(context.query.name);  
  // Query parameters are also attached to the context  
  context.log.info(context.name);  
}
```

**kn func invoke** コマンドを使用して、この関数にアクセスできます。

#### コマンドの例

```
$ kn func invoke --target 'http://example.com?name=tiger'
```

#### 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

### 12.1.3. ボディ

要求ボディ (ある場合) を返します。要求ボディに JSON コードが含まれている場合は、属性が直接利用できるように解析されます。

#### ボディの例

```
function handle(context) {  
  // log the incoming request body's 'hello' parameter  
  context.log.info(context.body.hello);  
}
```

**curl** コマンドを使用してこの関数を呼び出すことができます。

#### コマンドの例

```
$ kn func invoke -d '{"Hello": "world"}'
```

#### 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"world"}
```

### 12.1.4. ヘッダー

HTTP 要求ヘッダーをオブジェクトとして返します。

#### ヘッダーの例

```
function handle(context) {  
  context.log.info(context.headers["custom-header"]);  
}
```

**kn func invoke** コマンドを使用して、この関数にアクセスできます。

### コマンドの例

```
$ kn func invoke --target 'http://example.function.com'
```

### 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"some-value"}
```

## 12.1.5. HTTP 要求

### 方法

HTTP 要求メソッドを文字列として返します。

### httpVersion

HTTP バージョンを文字列として返します。

### httpVersionMajor

HTTP メジャーバージョン番号を文字列として返します。

### httpVersionMinor

HTTP マイナーバージョン番号を文字列として返します。

## 12.2. TYPESCRIPT コンテキストオブジェクトの参照

**context** オブジェクトには、関数開発者が利用可能なプロパティが複数あります。これらのプロパティにアクセスすると、着信 HTTP 要求に関する情報が提供され、出力がクラスターログに書き込まれます。

### 12.2.1. log

出力をクラスターロギングに書き込むために使用可能なロギングオブジェクトを提供します。ログは [Pino logging API](#) に準拠します。

### ログの例

```
export function handle(context: Context): string {  
  // log the incoming request body's 'hello' parameter  
  if (context.body) {  
    context.log.info((context.body as Record<string, string>).hello);  
  } else {  
    context.log.info('No data received');  
  }  
  return 'OK';  
}
```

**kn func invoke** コマンドを使用して、この関数にアクセスできます。

### コマンドの例

```
$ kn func invoke --target 'http://example.function.com'
```

## 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Processing customer"}
```

ログレベルは、**fatal**、**error**、**warn**、**info**、**debug**、**trace**、または **silent** のいずれかに設定できます。これを実行するには、**config** コマンドを使用してこれらの値のいずれかを環境変数 **FUNC\_LOG\_LEVEL** に割り当てて、**logLevel** の値を変更します。

### 12.2.2. query

要求のクエリー文字列 (ある場合) をキーと値のペアとして返します。これらの属性はコンテキストオブジェクト自体にも表示されます。

#### サンプルクエリー

```
export function handle(context: Context): string {
  // log the 'name' query parameter
  if (context.query) {
    context.log.info((context.query as Record<string, string>).name);
  } else {
    context.log.info("No data received");
  }
  return 'OK';
}
```

**kn func invoke** コマンドを使用して、この関数にアクセスできます。

#### コマンドの例

```
$ kn func invoke --target 'http://example.function.com' --data '{"name": "tiger"}
```

## 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

### 12.2.3. ボディ

要求ボディ (ある場合) を返します。要求ボディに JSON コードが含まれている場合は、属性が直接利用できるように解析されます。

#### ボディの例

```
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.body as Record<string, string>).hello);
  } else {
    context.log.info("No data received");
  }
}
```

```
}  
  return 'OK';  
}
```

**kn func invoke** コマンドを使用して、この関数にアクセスできます。

#### コマンドの例

```
$ kn func invoke --target 'http://example.function.com' --data '{"hello": "world"}'
```

#### 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"world"}
```

### 12.2.4. ヘッダー

HTTP 要求ヘッダーをオブジェクトとして返します。

#### ヘッダーの例

```
export function handle(context: Context): string {  
  // log the incoming request body's 'hello' parameter  
  if (context.body) {  
    context.log.info((context.headers as Record<string, string>)['custom-header']);  
  } else {  
    context.log.info('No data received');  
  }  
  return 'OK';  
}
```

**curl** コマンドを使用してこの関数を呼び出すことができます。

#### コマンドの例

```
$ curl -H'x-custom-header: some-value' http://example.function.com
```

#### 出力例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"some-value"}
```

### 12.2.5. HTTP 要求

#### 方法

HTTP 要求メソッドを文字列として返します。

#### httpVersion

HTTP バージョンを文字列として返します。

#### httpVersionMajor

HTTP メジャーバージョン番号を文字列として返します。

**httpVersionMinor**

HTTP マイナーバージョン番号を文字列として返します。