



Red Hat OpenShift Serverless 1.29

함수

OpenShift Serverless Functions 설정 및 사용

법적 공지

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[®] 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.

초록

이 문서에서는 OpenShift Serverless Functions를 시작하고 Quarkus, Node.js, TypeScript 및 Python을 사용하여 함수를 개발하고 배포하는 방법에 대한 정보를 제공합니다.

차례

1장. OPENSIFT SERVERLESS FUNCTIONS 설정	4
1.1. 사전 요구 사항	4
1.2. PODMAN 설정	4
1.3. MACOS에서 PODMAN 설정	5
1.4. 다음 단계	6
2장. 함수 시작하기	7
2.1. 사전 요구 사항	7
2.2. 함수 생성	7
2.3. 로컬에서 함수 실행	8
2.4. 함수 빌드	8
2.5. 함수 배포	9
2.6. 테스트 이벤트를 사용하여 배포된 함수 호출	10
2.7. 함수 삭제	11
2.8. OPENSIFT CONTAINER PLATFORM의 추가 리소스	11
2.9. 다음 단계	11
3장. 클러스터 내 기능 빌드 및 배포	12
3.1. 클러스터에서 함수 빌드 및 배포	12
3.2. 함수 버전 지정	13
4장. QUARKUS 함수 개발	15
4.1. 사전 요구 사항	15
4.2. QUARKUS 함수 템플릿 구조	15
4.3. QUARKUS 함수 호출 정보	16
4.4. CLOUDEVENT 속성	18
4.5. QUARKUS 함수 반환 값	19
4.6. QUARKUS 함수 테스트	20
4.7. 다음 단계	20
5장. NODE.JS 함수 개발	21
5.1. 사전 요구 사항	21
5.2. NODE.JS 함수 템플릿 구조	21
5.3. NODE.JS 함수 호출 정보	21
5.4. NODE.JS 함수 반환 값	23
5.5. NODE.JS 함수 테스트	24
5.6. 다음 단계	24
6장. TYPESCRIPT 함수 개발	25
6.1. 사전 요구 사항	25
6.2. TYPESCRIPT 함수 템플릿 구조	25
6.3. TYPESCRIPT 함수 호출 정보	25
6.4. TYPESCRIPT 함수 반환 값	27
6.5. TYPESCRIPT 함수 테스트	29
6.6. 다음 단계	29
7장. PYTHON 함수 개발	30
7.1. 사전 요구 사항	30
7.2. PYTHON 함수 템플릿 구조	30
7.3. PYTHON 함수 호출 정보	30
7.4. PYTHON 함수 반환 값	31
7.5. PYTHON 함수 테스트	32
7.6. 다음 단계	32

8장. KNATIVE EVENTING에서 함수 사용	33
8.1. 개발자 화면을 사용하여 이벤트 소스를 함수에 연결	33
9장. FUNC.YAML의 함수 프로젝트 구성	34
9.1. FUNC.YAML의 구성 가능한 필드	34
9.2. FUNC.YAML 필드의 로컬 환경 변수 참조	37
9.3. 추가 리소스	38
10장. 함수에서 보안 및 구성 맵에 액세스	39
10.1. 시크릿 및 구성 맵에 대한 함수 액세스의 상호 작용 변경	39
10.2. 특수 명령을 사용하여 시크릿 및 구성 맵에 대한 함수 액세스 수정	40
10.3. 시크릿 및 구성 맵에 수동으로 함수 액세스 추가	40
11장. 함수에 주석 추가	46
11.1. 함수에 주석 추가	46
12장. 함수 개발 참조 가이드	47
12.1. NODE.JS 컨텍스트 오브젝트 참조	47
12.2. TYPESCRIPT 컨텍스트 오브젝트 참조	49

1장. OPENSIFT SERVERLESS FUNCTIONS 설정

애플리케이션 코드 배포 프로세스를 개선하기 위해 OpenShift Serverless를 사용하여 OpenShift Container Platform에 상태 비저장 이벤트 중심 기능을 Knative 서비스로 배포할 수 있습니다. 함수를 개발하려면 설정 단계를 완료해야 합니다.

1.1. 사전 요구 사항

클러스터에서 OpenShift Serverless Functions을 사용하려면 다음 단계를 완료해야 합니다.

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.



참고

함수는 Knative 서비스로 배포됩니다. 이벤트 중심 아키텍처를 기능과 함께 사용하려면 Knative Eventing도 설치해야 합니다.

- **oc CLI**가 설치되어 있어야 합니다.
- **Knative(kn) CLI**가 설치되어 있어야 합니다. Knative CLI를 설치하면 함수를 생성하고 관리하는데 사용할 수 있는 **kn func** 명령을 사용할 수 있습니다.
- Docker Container Engine 또는 Podman 버전 3.4.7 이상을 설치했습니다.
- 사용 가능한 이미지 레지스트리(예: OpenShift Container Registry)에 액세스할 수 있습니다.
- **Quay.io**를 이미지 레지스트리로 사용하는 경우 리포지토리가 비공개가 아닌지 확인하거나 **pod**가 다른 보안 레지스트리의 이미지를 참조하도록 허용하는 **OpenShift Container Platform 설명서**를 따라야 합니다.
- OpenShift Container Registry를 사용하는 경우 클러스터 관리자가 **레지스트리를 공개해야** 합니다.

1.2. PODMAN 설정

고급 컨테이너 관리 기능을 사용하려면 OpenShift Serverless Functions에서 Podman을 사용할 수 있습니다. 이렇게 하려면 Podman 서비스를 시작하고 연결하도록 Knative(kn) CLI를 구성해야 합니다.

절차

1. **`\${XDG_RUNTIME_DIR}/podman/podman.sock`**의 UNIX 소켓에서 Docker API를 제공하는 Podman 서비스를 시작합니다.

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



참고

대부분의 시스템에서 이 소켓은 **/run/user/\$(id -u)/podman/podman.sock**에 있습니다.

2. 기능을 구축하는 데 사용되는 환경 변수를 설정합니다.

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


- 3. 함수 프로젝트 디렉터리 내에서 **-v** 플래그를 사용하여 빌드 명령을 실행하여 자세한 출력을 확인합니다. 로컬 UNIX 소켓에 대한 연결이 표시됩니다.

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

1.3. MACOS에서 PODMAN 설정

고급 컨테이너 관리 기능을 사용하려면 OpenShift Serverless Functions에서 Podman을 사용할 수 있습니다. macOS에서 이를 수행하려면 Podman 시스템을 시작하고 연결하도록 Knative(**kn**) CLI를 구성해야 합니다.

절차

1. Podman 시스템을 생성합니다.

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

2. UNIX 소켓에서 Docker API를 제공하는 Podman 시스템을 시작합니다.

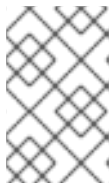
```
$ podman machine start
Starting machine "podman-machine-default"
Waiting for VM ...
Mounting volume... /Users/myuser:/Users/user
```

[...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
```



참고

대부분의 macOS 시스템에서 이 소켓은 **/Users/myuser/.local/share/containers/podman/podman-machine-machine-default/podman.sock**에 있습니다.

3. 기능을 구축하는 데 사용되는 환경 변수를 설정합니다.

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

4. 함수 프로젝트 디렉터리 내에서 **-v** 플래그를 사용하여 빌드 명령을 실행하여 자세한 출력을 확인합니다. 로컬 UNIX 소켓에 대한 연결이 표시됩니다.

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

1.4. 다음 단계

- Docker Container Engine 또는 Podman에 대한 자세한 내용은 컨테이너 [빌드 툴 옵션](#)을 참조하십시오.
- [함수 시작하기](#)를 참조하십시오.

2장. 함수 시작하기

기능 라이프사이클 관리에는 함수 생성, 빌드 및 배포가 포함됩니다. 선택적으로 호출하여 배포된 함수를 테스트할 수도 있습니다. **kn func** 툴을 사용하여 OpenShift Serverless에서 이러한 모든 작업을 수행할 수 있습니다.

2.1. 사전 요구 사항

다음 절차를 완료하려면 먼저 [OpenShift Serverless Functions](#) 설정에서 모든 사전 요구 사항 작업을 완료해야 합니다.

2.2. 함수 생성

함수를 빌드하고 배포하려면 먼저 Knative(**kn**) CLI를 사용하여 함수를 생성해야 합니다. 명령줄에서 경로, 런타임, 템플릿 및 이미지 레지스트리를 플래그로 지정하거나 **-c** 플래그를 사용하여 터미널에서 대화형 환경을 시작할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.

절차

- 함수 프로젝트를 생성합니다.

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

- 허용되는 런타임 값에는 **quarkus,node,typescript,go,python,springboot,rust** 가 포함됩니다.
- 허용되는 템플릿 값에는 **http** 및 **cloudevents** 가 포함됩니다.

명령 예

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

출력 예

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

- 또는 사용자 지정 템플릿이 포함된 리포지토리를 지정할 수 있습니다.

명령 예

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

출력 예

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

2.3. 로컬에서 함수 실행

kn func run 명령을 사용하여 현재 디렉터리 또는 **--path** 플래그로 지정된 디렉터리에서 함수를 로컬로 실행할 수 있습니다. 실행 중인 함수가 이전에 빌드되지 않았거나 프로젝트 파일이 마지막으로 빌드되었으므로 수정된 경우 **kn func run** 명령은 기본적으로 함수를 실행하기 전에 함수를 빌드합니다.

현재 디렉터리에서 함수를 실행하는 명령의 예

```
$ kn func run
```

경로로 지정된 디렉터리에서 함수를 실행하는 명령의 예

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

--build 플래그를 사용하여 프로젝트 파일을 변경하지 않아도 함수를 실행하기 전에 기존 이미지를 강제로 다시 빌드할 수도 있습니다.

빌드 플래그를 사용하는 run 명령 예

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

빌드 플래그를 false로 설정하면 이미지 빌드가 비활성화되고 이전에 빌드된 이미지를 사용하여 이 함수를 실행합니다.

빌드 플래그를 사용하는 run 명령 예

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

help 명령을 사용하여 **kn func run** 명령 옵션에 대해 자세히 알아볼 수 있습니다.

빌드 도움말 명령

```
$ kn func help run
```

2.4. 함수 빌드

함수를 실행하려면 함수 프로젝트를 빌드해야 합니다. **kn func run** 명령을 사용하는 경우 함수가 자동으로 빌드됩니다. 그러나 **kn func build** 명령을 사용하여 실행하지 않고 함수를 빌드할 수 있으며 고급 사용자 또는 디버깅 시나리오에 유용할 수 있습니다.

kn func build 명령은 컴퓨터 또는 OpenShift Container Platform 클러스터에서 로컬로 실행할 수 있는 OCI 컨테이너 이미지를 생성합니다. 이 명령은 함수 프로젝트 이름과 이미지 레지스트리 이름을 사용하여 함수의 정규화된 이미지 이름을 구성합니다.

2.4.1. 이미지 컨테이너 유형

기본적으로 **kn func build** 는 Red Hat S2I(Source-to-Image) 기술을 사용하여 컨테이너 이미지를 생성합니다.

Red Hat S2I(Source-to-Image)를 사용하는 빌드 명령 예

```
$ kn func build
```

2.4.2. 이미지 레지스트리 유형

OpenShift Container Registry는 기본적으로 함수 이미지를 저장하기 위한 이미지 레지스트리로 사용됩니다.

OpenShift Container Registry를 사용하는 빌드 명령 예

```
$ kn func build
```

출력 예

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

--registry 플래그를 사용하여 OpenShift Container Registry를 기본 이미지 레지스트리로 덮어쓸 수 있습니다.

quay.io를 사용하도록 OpenShift Container Registry 덮어쓰기 빌드 명령 예

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

출력 예

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

2.4.3. push 플래그

--push 플래그를 **kn func build** 명령에 추가하여 함수 이미지를 성공적으로 빌드한 후 자동으로 푸시할 수 있습니다.

OpenShift Container Registry를 사용하는 빌드 명령 예

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

2.4.4. 도움말 명령

help 명령을 사용하여 **kn func build** 명령 옵션에 대해 자세히 알아볼 수 있습니다.

빌드 도움말 명령

```
$ kn func help build
```

2.5. 함수 배포

kn func deploy 명령을 사용하여 Knative 서비스로 클러스터에 함수를 배포할 수 있습니다. 대상 함수가 이미 배포된 경우 컨테이너 이미지 레지스트리로 푸시된 새 컨테이너 이미지로 업데이트되고 Knative 서비스가 업데이트됩니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 프로젝트를 생성했거나 OpenShift Container Platform에서 애플리케이션 및 기타 워크로드를 생성하는 데 적절한 역할 및 권한이 있는 프로젝트에 액세스할 수 있습니다.
- 배포하려는 함수를 이미 생성하고 초기화해야 합니다.

절차

- 함수를 배포합니다.

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

출력 예

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

- **namespace**를 지정하지 않으면 함수가 현재 네임스페이스에 배포됩니다.
- 이 함수는 **path**를 지정하지 않는 한 현재 디렉터리에서 배포됩니다.
- Knative 서비스 이름은 프로젝트 이름에서 파생되며 이 명령을 사용하여 변경할 수 없습니다.



참고

개발자 화면의 +추가 보기에서 Git 에서 가져오기 또는 서버리스 생성 함수를 사용하여 Git 리포지토리 URL을 사용하여 서버리스 함수를 생성할 수 있습니다.

2.6. 테스트 이벤트를 사용하여 배포된 함수 호출

kn func invoke CLI 명령을 사용하여 로컬 또는 OpenShift Container Platform 클러스터에서 함수를 호출하도록 테스트 요청을 보낼 수 있습니다. 이 명령을 사용하여 함수가 작동하고 이벤트를 올바르게 수신할 수 있는지 테스트할 수 있습니다. 함수를 로컬로 호출하는 것은 함수 개발 중 빠른 테스트에 유용합니다. 클러스터에서 함수를 호출하면 프로덕션 환경에 더 가까운 테스트에 유용합니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 프로젝트를 생성했거나 OpenShift Container Platform에서 애플리케이션 및 기타 워크로드를 생성하는 데 적절한 역할 및 권한이 있는 프로젝트에 액세스할 수 있습니다.
- 호출하려는 함수를 이미 배포해야 합니다.

절차

- 함수를 호출합니다.

```
$ kn func invoke
```

- **kn func invoke** 명령은 현재 실행 중인 로컬 컨테이너 이미지가 있거나 클러스터에 배포된 함수가 있는 경우에만 작동합니다.
- **kn func invoke** 명령은 기본적으로 로컬 디렉터리에서 실행되며 이 디렉터리는 함수 프로젝트라고 가정합니다.

2.7. 함수 삭제

kn func delete 명령을 사용하여 함수를 삭제할 수 있습니다. 이 기능은 더 이상 기능이 필요하지 않은 경우 유용하며 클러스터에 리소스를 저장하는 데 도움이 될 수 있습니다.

절차

- 함수를 삭제합니다.

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

- 삭제할 함수의 이름 또는 경로가 지정되지 않은 경우 현재 디렉터리에서 **func.yaml** 파일을 검색하고 삭제할 함수를 결정합니다.
- 네임스페이스를 지정하지 않으면 기본값은 **func.yaml** 파일의 **namespace** 값으로 설정됩니다.

2.8. OPENSIFT CONTAINER PLATFORM의 추가 리소스

- [기본 레지스트리를 수동으로 공개](#)
- [IntelliJ Knative 플러그인의 마켓플레이스 페이지](#)
- [Visual Studio Code Knative 플러그인의 마켓플레이스 페이지](#)
- [개발자 화면을 사용하여 애플리케이션 생성](#)

2.9. 다음 단계

- [Knative Eventing에서 함수 사용을 참조하십시오.](#)

3장. 클러스터 내 기능 빌드 및 배포

함수를 로컬로 빌드하는 대신 클러스터에서 직접 함수를 빌드할 수 있습니다. 로컬 개발 시스템에서 이 워크플로를 사용하는 경우 함수 소스 코드만 사용해야 합니다. 예를 들어 docker 또는 podman과 같은 클러스터 함수 빌드 툴을 설치할 수 없는 경우 유용합니다.

3.1. 클러스터에서 함수 빌드 및 배포

Knative(**kn**) CLI를 사용하여 함수 프로젝트 빌드를 시작한 다음 클러스터에 함수를 직접 배포할 수 있습니다. 이러한 방식으로 함수 프로젝트를 빌드하려면 클러스터가 액세스할 수 있는 Git 리포지토리 분기에 함수 프로젝트의 소스 코드가 있어야 합니다.

사전 요구 사항

- Red Hat OpenShift Pipelines가 클러스터에 설치되어 있어야 합니다.
- OpenShift CLI(**oc**)가 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.

절차

1. OpenShift Pipelines를 실행하고 함수를 배포하려는 각 네임스페이스에서 다음 리소스를 생성해야 합니다.

- a. 파이프라인에서 Source-to-Image를 사용할 수 있도록 **s2i** Tekton 작업을 생성합니다.

```
$ 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. 파이프라인에 함수를 배포할 수 있도록 **kn func** deploy Tekton 작업을 생성합니다.

```
$ 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. 함수를 생성합니다.

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

3. 새 함수 프로젝트를 생성한 후에는 프로젝트를 Git 리포지토리에 추가하고 클러스터에서 리포지토리를 사용할 수 있는지 확인해야 합니다. 이 Git 리포지토리에 대한 정보는 다음 단계에서 **func.yaml** 파일을 업데이트하는 데 사용됩니다.

4. 함수 프로젝트의 **func.yaml** 파일에서 Git 리포지토리에 대한 클러스터 내 빌드를 활성화하도록 구성을 업데이트합니다.

```
...
git:
  url: <git_repository_url> 1
  revision: main 2
  contextDir: <directory_path> 3
...
```


- 1 필수 항목입니다. 함수의 소스 코드가 포함된 Git 리포지토리를 지정합니다.
 - 2 선택 사항: 사용할 Git 리포지토리 버전을 지정합니다. 이는 분기, 태그 또는 커밋일 수 있습니다.
 - 3 선택 사항: 함수가 Git 리포지토리 루트 폴더에 없는 경우 함수의 디렉터리 경로를 지정합니다.
5. 함수의 비즈니스 논리를 구현합니다. 그런 다음 Git을 사용하여 변경 사항을 커밋하고 내보냅니다.
 6. 함수를 배포합니다.

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

함수 구성에서 참조되는 컨테이너 레지스트리에 로그인하지 않은 경우 함수 이미지를 호스팅하는 원격 컨테이너 레지스트리에 대한 인증 정보를 제공하라는 메시지가 표시됩니다.

출력 및 프롬프트 예

```
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. 함수를 업데이트하려면 Git을 사용하여 새 변경 사항을 커밋하고 내보낸 다음 **kn func deploy --remote** 명령을 다시 실행합니다.

3.2. 함수 버전 지정

클러스터에서 함수를 빌드하고 배포할 때 리포지토리 내에 Git 리포지토리, 분기, 하위 디렉터를 지정하여 함수 코드의 위치를 지정해야 합니다. 기본 분기를 사용하는 경우 분기를 지정할 필요가 없습니다. 마찬가지로 함수가 리포지토리의 루트에 있는 경우 하위 디렉터를 지정할 필요가 없습니다. **func.yaml** 구성 파일에서 이러한 매개변수를 지정하거나 **kn func deploy** 명령과 함께 플래그를 사용할 수 있습니다.

사전 요구 사항

- Red Hat OpenShift Pipelines가 클러스터에 설치되어 있어야 합니다.
- OpenShift(**oc**) CLI를 설치했습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.

절차

- 함수를 배포합니다.

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

- 1 **--remote** 플래그를 사용하면 빌드가 원격으로 실행됩니다.
- 2 **<repo-url>**을 Git 리포지토리의 URL로 바꿉니다.
- 3 **<branch>**를 Git 분기, 태그 또는 커밋으로 바꿉니다. 기본 분기에 최신 커밋을 사용하는 경우 이 플래그를 건너뛸 수 있습니다.
- 4 리포지토리 루트 디렉터리와 다른 경우 **<function-dir>**을 함수를 포함하는 디렉터리로 바꿉니다.

예를 들면 다음과 같습니다.

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

4장. QUARKUS 함수 개발

Quarkus 함수 프로젝트를 생성한 후에는 제공된 템플릿 파일을 수정하여 비즈니스 로직을 함수에 추가할 수 있습니다. 여기에는 함수 호출 및 반환된 헤더 및 상태 코드가 포함됩니다.

4.1. 사전 요구 사항

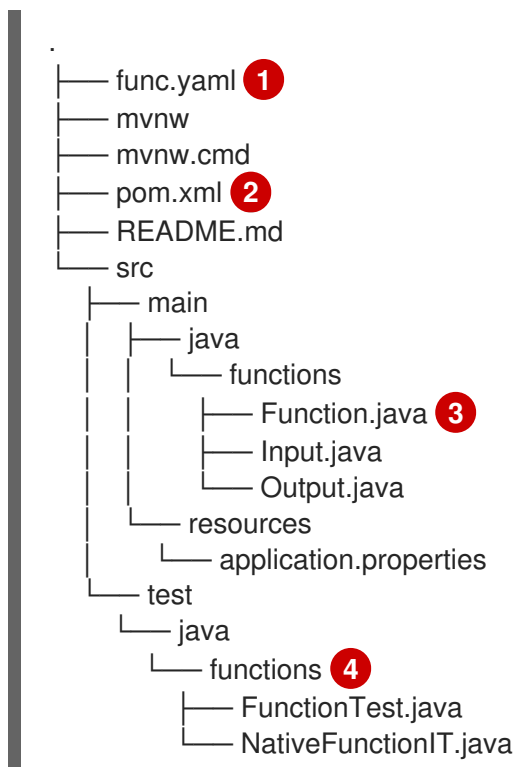
- 함수를 개발하려면 먼저 [OpenShift Serverless Functions 설정에서 설정](#) 단계를 완료해야 합니다.

4.2. QUARKUS 함수 템플릿 구조

Knative(**kn**) CLI를 사용하여 Quarkus 함수를 생성할 때 프로젝트 디렉터리는 일반적인 Maven 프로젝트와 유사합니다. 또한 프로젝트에는 함수 구성에 사용되는 **func.yaml** 파일이 포함되어 있습니다.

http 및 **event** 트리거 함수 모두 동일한 템플릿 구조를 갖습니다.

템플릿 구조



- 1 이미지 이름과 레지스트리를 결정하는 데 사용됩니다.
- 2 POM(Project Object Model) 파일에는 종속성에 대한 정보와 같은 프로젝트 구성이 포함되어 있습니다. 이 파일을 수정하여 다른 종속 항목을 추가할 수 있습니다.

추가 종속 항목 예

```

...
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>test</scope>
  </dependency>
</dependencies>
  
```

```

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

```

종속성은 첫 번째 컴파일 중에 다운로드됩니다.

- 3 함수 프로젝트에는 **@Funq** 주석이 추가된 Java 메서드가 포함되어야 합니다. 이 메서드를 **Function.java** 클래스에 배치할 수 있습니다.
- 4 함수의 로컬 테스트에 사용할 수 있는 간단한 테스트 케이스가 포함되어 있습니다.

4.3. QUARKUS 함수 호출 정보

클라우드 이벤트에 응답하는 Quarkus 프로젝트 또는 간단한 HTTP 요청에 응답하는 Quarkus 프로젝트를 생성할 수 있습니다. Knative의 클라우드 이벤트는 HTTP를 통해 POST 요청으로 전송되므로 두 기능 유형 모두 들어오는 HTTP 요청을 수신하고 응답할 수 있습니다.

들어오는 요청이 수신되면 Quarkus 함수가 허용된 유형의 인스턴스와 함께 호출됩니다.

표 4.1. 함수 호출 옵션

호출 메소드	인스턴스에 포함된 데이터 유형	데이터 예
HTTP POST 요청	요청 본문에 있는 JSON 오브젝트	<code>{ "customerId": "0123456", "productId": "6543210" }</code>
HTTP GET 요청	쿼리 문자열의 데이터	<code>? customerId=0123456&productId=6543210</code>
CloudEvent	data 속성의 JSON 개체	<code>{ "customerId": "0123456", "productId": "6543210" }</code>

다음 예제에서는 이전 표에 나열된 **customerId** 및 **productId** 구매 데이터를 수신하고 처리하는 함수를 보여줍니다.

Quarkus 함수의 예

```

public class Functions {
  @Funq
  public void processPurchase(Purchase purchase) {
    // process the purchase
  }
}

```

구매 데이터를 포함하는 **Purchase** JavaBean 클래스는 다음과 같습니다.

클래스 예

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

4.3.1. 호출 예

다음 예제 코드는 **withBeans**, **withCloudEvent**, **withBinary**라는 세 가지 함수를 정의합니다.

예제

```
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 클래스의 **withBeans** 함수는 다음을 통해 호출할 수 있습니다.

- JSON 본문이 있는 HTTP POST 요청:

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

- 쿼리 매개변수가 있는 HTTP GET 요청:

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

- 바이너리 인코딩의 **CloudEvent** 오브젝트:

```
$ 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."}'
```

- 구조화된 인코딩의 **CloudEvent** 오브젝트:

```
$ 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 클래스의 **withCloudEvent** 함수는 **withBeans** 함수와 유사하게 **CloudEvent** 오브젝트를 사용하여 호출할 수 있습니다. 그러나 **withBeans**와 달리 **withCloudEvent**는 일반 HTTP 요청으로 호출할 수 없습니다.

Functions 클래스의 **withBinary** 함수는 다음을 통해 호출할 수 있습니다.

- 바이너리 인코딩의 **CloudEvent** 오브젝트:

```
$ 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'
```

- 구조화된 인코딩의 **CloudEvent** 오브젝트:

```
$ 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 속성

type 또는 **subject**와 같은 CloudEvent의 속성을 읽거나 작성해야 하는 경우 **CloudEvent<T>** 일반 인터페이스와 **CloudEventBuilder** 빌더를 사용할 수 있습니다. **<T>** 유형 매개변수는 허용된 유형 중 하나여야 합니다.

다음 예에서 **CloudEventBuilder**는 구매 처리 성공 또는 실패를 반환하는 데 사용됩니다.

```
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 함수 반환 값

함수는 허용된 유형 목록에서 모든 유형의 인스턴스를 반환할 수 있습니다. 또는 **Uni<T>** 유형을 반환할 수 있습니다. 여기서 **<T>** 유형 매개변수는 허용된 유형의 모든 유형일 수 있습니다.

Uni<T> 유형은 반환된 오브젝트가 수신된 오브젝트와 동일한 형식으로 직렬화되기 때문에 함수가 비동기 API를 호출할 때 유용합니다. 예를 들면 다음과 같습니다.

- 함수가 HTTP 요청을 수신하면 반환된 오브젝트가 HTTP 응답 본문에 전송됩니다.
- 함수가 바이너리 인코딩으로 **CloudEvent** 오브젝트를 수신하는 경우 반환된 오브젝트는 바이너리 인코딩 **CloudEvent** 오브젝트의 데이터 속성으로 전송됩니다.

다음 예제에서는 구매 목록을 가져오는 함수를 보여줍니다.

명령 예

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

- HTTP 요청을 통해 이 함수를 호출하면 응답 본문에서 구매한 목록을 포함하는 HTTP 응답이 생성됩니다.
- 들어오는 **CloudEvent** 오브젝트를 통해 이 함수를 호출하면 **data** 속성에서 구매 목록이 포함된 **CloudEvent** 응답이 생성됩니다.

4.5.1. 허용된 유형

함수의 입력 및 출력은 **빈, 문자열** 또는 **byte[]** 유형 중 하나일 수 있습니다. 또한 기본 유형과 래퍼(예: **int** 및 **Integer**)일 수 있습니다. 또한 Javaovns, 맵, 목록, 배열 및 특수 **CloudEvents<T>** 유형 등 복잡한 오브젝트일 수도 있습니다.

map, lists, arrays, < T > type 매개 변수 of the **CloudEvents<T>** 유형의 CloudEvents<T> 및 JavaECDHEs의 속성은 여기에 나열된 유형일 수 있습니다.

예제

```
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 함수 테스트

Quarkus 함수는 컴퓨터에서 로컬로 테스트할 수 있습니다. **kn func create** 를 사용하여 함수를 생성할 때 생성되는 기본 프로젝트에는 기본 Maven 테스트가 포함된 CloudEvent **/test/** 디렉터리가 있습니다. 이러한 테스트는 필요에 따라 확장할 수 있습니다.

사전 요구 사항

- Quarkus 함수를 생성했습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.

절차

1. 함수의 프로젝트 폴더로 이동합니다.
2. Maven 테스트를 실행합니다.

```
$ ./mvnw test
```

4.7. 다음 단계

- 함수를 빌드 및 배포합니다.

5장. NODE.JS 함수 개발

Node.js 함수 프로젝트를 생성한 후에는 제공된 템플릿 파일을 수정하여 비즈니스 로직을 기능에 추가할 수 있습니다. 여기에는 함수 호출 및 반환된 헤더 및 상태 코드가 포함됩니다.

5.1. 사전 요구 사항

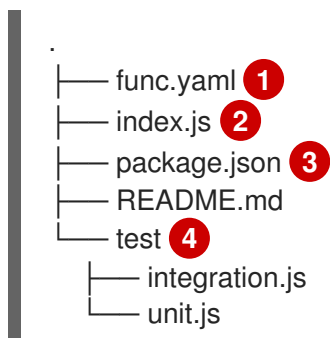
- 함수를 개발하려면 먼저 [OpenShift Serverless Functions 설정](#) 단계를 완료해야 합니다.

5.2. NODE.JS 함수 템플릿 구조

Knative(**kn**) CLI를 사용하여 Node.js 함수를 생성할 때 프로젝트 디렉터리는 일반적인 Node.js 프로젝트와 같습니다. 유일한 예외는 추가 **func.yaml** 파일입니다. 이 파일은 함수를 구성하는 데 사용됩니다.

http 및 **event** 트리거 함수 모두 동일한 템플릿 구조를 갖습니다.

템플릿 구조



- 1 **func.yaml** 구성 파일은 이미지 이름과 레지스트리를 결정하는 데 사용됩니다.
- 2 프로젝트에는 단일 함수를 내보내는 **index.js** 파일이 포함되어야 합니다.
- 3 템플릿 **package.json** 파일에 제공된 종속성으로 제한되지 않습니다. 다른 Node.js 프로젝트에서와 마찬가지로 추가 종속 항목을 추가할 수 있습니다.

npm 종속성 추가 예

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

프로젝트가 배포용으로 빌드되면 이러한 종속 항목은 생성된 런타임 컨테이너 이미지에 포함됩니다.

- 4 통합 및 테스트 스크립트는 함수 템플릿의 일부로 제공됩니다.

5.3. NODE.JS 함수 호출 정보

Knative(**kn**) CLI를 사용하여 함수 프로젝트를 생성하는 경우 CloudEvents에 응답하는 프로젝트 또는 간단한 HTTP 요청에 응답하는 프로젝트를 생성할 수 있습니다. Knative의 CloudEvents는 HTTP를 통해 POST 요청으로 전송되므로 함수 유형 모두 수신되는 HTTP 이벤트를 수신하고 응답합니다.

Node.js 함수는 간단한 HTTP 요청을 사용하여 호출할 수 있습니다. 들어오는 요청이 수신되면 **context** 오브젝트를 첫 번째 매개 변수로 사용하여 함수가 호출됩니다.

5.3.1. Node.js 컨텍스트 오브젝트

함수는 **context** 오브젝트를 첫 번째 매개 변수로 제공하여 호출됩니다. 이 오브젝트는 들어오는 HTTP 요청 정보에 대한 액세스를 제공합니다.

컨텍스트 오브젝트의 예

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

이 정보에는 HTTP 요청 메서드, 요청, HTTP 버전, 요청 본문으로 전송된 모든 쿼리 문자열 또는 헤더가 포함됩니다. **CloudEvent** 가 포함된 들어오는 요청은 **context.cloudevent** 를 사용하여 액세스할 수 있도록 CloudEvent의 들어오는 인스턴스를 컨텍스트 오브젝트에 연결합니다.

5.3.1.1. 컨텍스트 오브젝트 메서드

context 오브젝트에는 데이터 값을 수락하고 CloudEvent를 반환하는 단일 메서드 **cloudEventResponse()**가 있습니다.

Knative 시스템에서 서비스로 배포된 함수가 CloudEvent를 보내는 이벤트 브로커에 의해 호출되는 경우 브로커는 응답을 확인합니다. 응답이 CloudEvent인 경우 브로커가 이 이벤트를 처리합니다.

컨텍스트 오브젝트 메서드 예

```
// 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 데이터

들어오는 요청이 CloudEvent인 경우 CloudEvent와 관련된 모든 데이터가 이벤트에서 추출되며 두 번째 매개 변수로 제공됩니다. 예를 들어 데이터 속성에 다음과 유사한 JSON 문자열이 포함된 CloudEvent가 수신되는 경우 다음과 같이 됩니다.

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

호출될 때 **context** 오브젝트 다음에 함수에 대한 두 번째 매개 변수는 **customerId** 및 **productId** 속성이 있는 JavaScript 오브젝트가 됩니다.

서명 예

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

이 예제의 **data** 매개 변수는 **customerId** 및 **productId** 속성을 포함하는 JavaScript 오브젝트입니다.

5.4. NODE.JS 함수 반환 값

함수는 유효한 JavaScript 유형을 반환하거나 반환 값이 없을 수 있습니다. 함수에 반환 값이 지정되지 않고 실패가 표시되지 않으면 호출자는 **204 No Content** 응답을 받습니다.

또한 함수는 이벤트를 Knative Eventing 시스템으로 푸시하기 위해 **CloudEvent** 또는 **Message** 오브젝트를 반환할 수 있습니다. 이 경우 개발자는 CloudEvent 메시징 사양을 이해하고 구현할 필요가 없습니다. 반환된 값의 헤더 및 기타 관련 정보는 추출된 응답으로 전송됩니다.

예

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

5.4.1. 헤더 반환

return 오브젝트에 **headers** 속성을 추가하여 응답 헤더를 설정할 수 있습니다. 이러한 헤더는 추출된 호출에 대한 응답으로 전송됩니다.

응답 헤더의 예

```
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. 상태 코드 반환

statusCode 속성을 **return** 오브젝트에 추가하여 호출자에게 반환된 상태 코드를 설정할 수 있습니다.

상태 코드 예

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

상태 코드는 함수에서 생성되어 발생하는 오류에 대해 설정할 수 있습니다.

오류 상태 코드의 예

```
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 함수 테스트

Node.js 함수는 컴퓨터에서 로컬로 테스트할 수 있습니다. **kn func create** 를 사용하여 함수를 생성할 때 생성되는 기본 프로젝트에는 몇 가지 간단한 단위 및 통합 **테스트가 포함된 테스트** 폴더가 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- **kn func create** 를 사용하여 함수를 생성했습니다.

절차

1. 함수의 **테스트** 폴더로 이동합니다.
2. 테스트를 실행합니다.

```
$ npm test
```

5.6. 다음 단계

- [Node.js 컨텍스트 오브젝트 참조](#) 설명서를 참조하십시오.
- 함수를 **빌드** 및 **배포**합니다.

6장. TYPESCRIPT 함수 개발

TypeScript 함수 프로젝트를 생성한 후에는 제공된 템플릿 파일을 수정하여 비즈니스 로직을 함수에 추가할 수 있습니다. 여기에는 함수 호출 및 반환된 헤더 및 상태 코드가 포함됩니다.

6.1. 사전 요구 사항

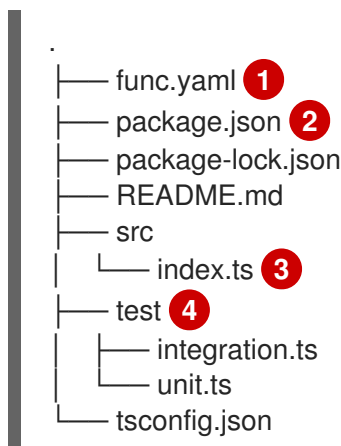
- 함수를 개발하려면 먼저 [OpenShift Serverless Functions 설정](#) 단계를 완료해야 합니다.

6.2. TYPESCRIPT 함수 템플릿 구조

Knative(**kn**) CLI를 사용하여 TypeScript 함수를 생성할 때 프로젝트 디렉터리는 일반적인 TypeScript 프로젝트와 같습니다. 유일한 예외는 함수를 구성하는 데 사용되는 추가 **func.yaml** 파일입니다.

http 및 **event** 트리거 함수 모두 동일한 템플릿 구조를 갖습니다.

템플릿 구조



- 1 **func.yaml** 구성 파일은 이미지 이름과 레지스트리를 결정하는 데 사용됩니다.
- 2 템플릿 **package.json** 파일에 제공된 종속성으로 제한되지 않습니다. 다른 TypeScript 프로젝트에서와 마찬가지로 종속 항목을 추가할 수 있습니다.

npm 종속성 추가 예

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

프로젝트가 배포용으로 빌드되면 이러한 종속 항목은 생성된 런타임 컨테이너 이미지에 포함됩니다.

- 3 프로젝트에는 **handle**라는 함수를 내보내는 **src/index.js** 파일이 포함되어야 합니다.
- 4 통합 및 테스트 스크립트는 함수 템플릿의 일부로 제공됩니다.

6.3. TYPESCRIPT 함수 호출 정보

Knative(**kn**) CLI를 사용하여 함수 프로젝트를 생성할 때 CloudEvents에 응답하는 프로젝트 또는 간단한 HTTP 요청에 응답하는 프로젝트를 생성할 수 있습니다. Knative의 CloudEvents는 HTTP를 통해 POST 요청으로 전송되므로 함수 유형 모두 수신되는 HTTP 이벤트를 수신하고 응답합니다.

간단한 HTTP 요청을 사용하여 TypeScript 함수를 호출할 수 있습니다. 들어오는 요청이 수신되면 **context** 오브젝트를 첫 번째 매개 변수로 사용하여 함수가 호출됩니다.

6.3.1. TypeScript 컨텍스트 오브젝트

함수를 호출하려면 **context** 오브젝트를 첫 번째 매개 변수로 제공합니다. **컨텍스트** 오브젝트의 속성에 액세스하면 들어오는 HTTP 요청에 대한 정보를 제공할 수 있습니다.

컨텍스트 오브젝트의 예

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

이 정보에는 HTTP 요청 메서드, 요청, HTTP 버전, 요청 본문으로 전송된 모든 쿼리 문자열 또는 헤더가 포함됩니다. **CloudEvent** 가 포함된 들어오는 요청은 **context.cloudevent** 를 사용하여 액세스할 수 있도록 CloudEvent의 들어오는 인스턴스를 컨텍스트 오브젝트에 연결합니다.

6.3.1.1. 컨텍스트 오브젝트 메서드

context 오브젝트에는 데이터 값을 수락하고 CloudEvent를 반환하는 단일 메서드 **cloudEventResponse()**가 있습니다.

Knative 시스템에서 서비스로 배포된 함수가 CloudEvent를 보내는 이벤트 브로커에 의해 호출되는 경우 브로커는 응답을 확인합니다. 응답이 CloudEvent인 경우 브로커가 이 이벤트를 처리합니다.

컨텍스트 오브젝트 메서드 예

```
// 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. 컨텍스트 유형

TypeScript 유형 정의 파일은 함수에 사용하기 위해 다음 유형을 내보냅니다.

내보낸 유형 정의

```
// Invokable is the expected 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 데이터

들어오는 요청이 CloudEvent인 경우 CloudEvent와 관련된 모든 데이터가 이벤트에서 추출되며 두 번째 매개변수로 제공됩니다. 예를 들어 데이터 속성에 다음과 유사한 JSON 문자열이 포함된 CloudEvent가 수신되는 경우 다음과 같이 됩니다.

```

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

```

호출될 때 **context** 오브젝트 다음에 함수에 대한 두 번째 매개 변수는 **customerId** 및 **productId** 속성이 있는 JavaScript 오브젝트가 됩니다.

서명 예

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

이 예제의 **cloudevent** 매개 변수는 **customerId** 및 **productId** 속성이 포함된 JavaScript 오브젝트입니다.

6.4. TYPESCRIPT 함수 반환 값

함수는 유효한 JavaScript 유형을 반환하거나 반환 값이 없을 수 있습니다. 함수에 반환 값이 지정되지 않고 실패가 표시되지 않으면 호출자는 **204 No Content** 응답을 받습니다.

또한 함수는 이벤트를 Knative Eventing 시스템으로 푸시하기 위해 CloudEvent 또는 **Message** 오브젝트를 반환할 수 있습니다. 이 경우 개발자는 CloudEvent 메시징 사양을 이해하고 구현할 필요가 없습니다. 반환된 값의 헤더 및 기타 관련 정보는 추출된 응답으로 전송됩니다.

예

```
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. 헤더 반환

return 오브젝트에 **headers** 속성을 추가하여 응답 헤더를 설정할 수 있습니다. 이러한 헤더는 추출된 호출에 대한 응답으로 전송됩니다.

응답 헤더의 예

```
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. 상태 코드 반환

statusCode 속성을 **return** 오브젝트에 추가하여 호출자에게 반환된 상태 코드를 설정할 수 있습니다.

상태 코드 예

```
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
  }
}
```

상태 코드는 함수에서 생성되어 발생하는 오류에 대해 설정할 수 있습니다.

오류 상태 코드의 예

```
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 함수 테스트

TypeScript 함수는 컴퓨터에서 로컬에서 테스트할 수 있습니다. **kn func create**를 사용하여 함수를 만들 때 생성되는 기본 프로젝트에는 몇 가지 간단한 단위 및 통합 테스트가 포함된 테스트 폴더가 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- **kn func create**를 사용하여 함수를 생성했습니다.

절차

1. 이전에 테스트를 실행하지 않은 경우 먼저 종속성을 설치합니다.

```
$ npm install
```

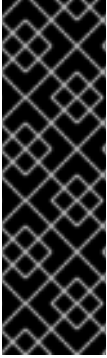
2. 함수의 테스트 폴더로 이동합니다.
3. 테스트를 실행합니다.

```
$ npm test
```

6.6. 다음 단계

- [TypeScript 컨텍스트 오브젝트 참조](#) 설명서를 참조하십시오.
- 함수를 빌드 및 배포합니다.
- 함수 로깅에 대한 자세한 내용은 [Pino API 설명서](#)를 참조하십시오.

7장. PYTHON 함수 개발



중요

OpenShift Serverless Functions with Python은 기술 프리뷰 기능 전용입니다. 기술 프리뷰 기능은 Red Hat 프로덕션 서비스 수준 계약(SLA)에서 지원되지 않으며 기능적으로 완전하지 않을 수 있습니다. 따라서 프로덕션 환경에서 사용하는 것은 권장하지 않습니다. 이러한 기능을 사용하면 향후 제품 기능을 조기에 이용할 수 있어 개발 과정에서 고객이 기능을 테스트하고 피드백을 제공할 수 있습니다.

Red Hat 기술 프리뷰 기능의 지원 범위에 대한 자세한 내용은 [기술 프리뷰 기능 지원 범위](#)를 참조하십시오.

Python 함수 프로젝트를 생성한 후에는 제공된 템플릿 파일을 수정하여 비즈니스 로직을 함수에 추가할 수 있습니다. 여기에는 함수 호출 및 반환된 헤더 및 상태 코드가 포함됩니다.

7.1. 사전 요구 사항

- 함수를 개발하려면 먼저 [OpenShift Serverless Functions 설정](#) 단계를 완료해야 합니다.

7.2. PYTHON 함수 템플릿 구조

Knative(kn) CLI를 사용하여 Python 함수를 생성할 때 프로젝트 디렉터리는 일반적인 Python 프로젝트와 유사합니다. Python 함수는 약간의 제한 사항이 있습니다. 유일한 요구 사항은 프로젝트에 **main()** 함수와 **func.yaml** 구성 파일이 포함된 **func.py** 파일이 포함되어 있다는 것입니다.

개발자는 템플릿 **requirements.txt** 파일에 제공된 종속성으로 제한되지 않습니다. 추가 종속 항목은 다른 Python 프로젝트에서 추가될 수 있습니다. 프로젝트가 배포용으로 빌드되면 이러한 종속성이 생성된 런타임 컨테이너 이미지에 포함됩니다.

http 및 **event** 트리거 함수 모두 동일한 템플릿 구조를 갖습니다.

템플릿 구조

```

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

```

- 1 **main()** 함수를 포함합니다.
- 2 이미지 이름과 레지스트리를 결정하는 데 사용됩니다.
- 3 다른 Python 프로젝트에 있는 것처럼 **requirements.txt** 파일에 기타 종속 항목을 추가할 수 있습니다.
- 4 함수의 로컬 테스트에 사용할 수 있는 간단한 단위 테스트가 포함되어 있습니다.

7.3. PYTHON 함수 호출 정보

Python 함수는 간단한 HTTP 요청으로 호출할 수 있습니다. 들어오는 요청이 수신되면 **context** 오브젝트를 첫 번째 매개 변수로 사용하여 함수가 호출됩니다.

context 오브젝트는 두 개의 속성이 있는 Python 클래스입니다.

- **request** 속성은 항상 존재하며 Flask **request** 오브젝트를 포함합니다.
- 들어오는 요청이 **CloudEvent** 오브젝트인 경우 두 번째 속성 **cloud_event**가 채워집니다.

개발자는 컨텍스트 오브젝트에서 모든 **CloudEvent** 데이터에 액세스할 수 있습니다.

컨텍스트 오브젝트의 예

```
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 함수 반환 값

함수는 Flask에서 지원하는 모든 값을 반환할 수 있습니다. 호출 프레임워크가 이러한 값을 Flask 서버에 직접 프록시하기 때문입니다.

예제

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

함수는 함수 호출에서 헤더와 응답 코드를 모두 2차 및 3차 응답 값으로 설정할 수 있습니다.

7.4.1. CloudEvents 반환

개발자는 **@event** 데코레이터를 사용하여 응답을 보내기 전에 함수 반환 값을 CloudEvent로 변환해야 함을 호출자에게 알릴 수 있습니다.

예

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

이 예제에서는 **"my.type"** 유형과 **"/my/function"** 소스를 사용하여 CloudEvent를 응답 값으로 보냅니다. CloudEvent **data** 속성은 반환된 **data** 변수로 설정됩니다. **event_source** 및 **event_type** 데코레이터 속성은 선택 사항입니다.

7.5. PYTHON 함수 테스트

컴퓨터에서 Python 함수를 로컬로 테스트할 수 있습니다. 기본 프로젝트에는 기능에 대한 간단한 단위 테스트를 제공하는 **test_proxyc.py** 파일이 포함되어 있습니다.



참고

Python 함수의 기본 테스트 프레임워크는 **unittest**입니다. 필요에 따라 다른 테스트 프레임워크를 사용할 수 있습니다.

사전 요구 사항

- Python 함수 테스트를 로컬에서 실행하려면 필요한 종속 항목을 설치해야 합니다.

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

절차

1. **test_func.py** 파일이 포함된 함수의 폴더로 이동합니다.
2. 테스트를 실행합니다.

```
$ python3 test_func.py
```

7.6. 다음 단계

- 함수를 [빌드](#) 및 [배포](#)합니다.

8장. KNATIVE EVENTING에서 함수 사용

함수는 OpenShift Container Platform 클러스터에 Knative 서비스로 배포됩니다. 들어오는 이벤트를 수신할 수 있도록 함수를 Knative Eventing 구성 요소에 연결할 수 있습니다.

8.1. 개발자 화면을 사용하여 이벤트 소스를 함수에 연결

함수는 OpenShift Container Platform 클러스터에 Knative 서비스로 배포됩니다. OpenShift Container Platform 웹 콘솔을 사용하여 이벤트 소스를 생성할 때 해당 소스에서 이벤트가 전송되는 배포된 함수를 지정할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator, Knative Serving, Knative Eventing이 OpenShift Container Platform 클러스터에 설치되어 있습니다.
- 웹 콘솔에 로그인한 후 **개발자** 화면으로 갑니다.
- 프로젝트를 생성했거나 OpenShift Container Platform에서 애플리케이션 및 기타 워크로드를 생성하는 데 적절한 역할 및 권한이 있는 프로젝트에 액세스할 수 있습니다.
- 함수를 생성하고 배포했습니다.

절차

1. **+추가** → 이벤트 소스로 이동하여 생성할 **이벤트 소스** 유형을 선택하여 모든 유형의 이벤트 소스를 생성합니다.
2. **이벤트 소스 생성** 양식 보기의 **싱크** 섹션에서 **리소스** 목록에서 함수를 선택합니다.
3. **생성**을 클릭합니다.

검증

토폴로지 페이지를 확인하여 이벤트 소스가 생성되었고 기능에 연결되어 있는지 확인할 수 있습니다.

1. **개발자** 화면에서 **토폴로지**로 이동합니다.
2. 이벤트 소스를 보고 연결된 기능을 클릭하여 오른쪽 패널에서 기능 세부 정보를 확인합니다.

9장. FUNC.YAML의 함수 프로젝트 구성

func.yaml 파일에는 함수 프로젝트의 구성이 포함되어 있습니다. **func.yaml**에 지정된 값은 **kn func** 명령을 실행할 때 사용됩니다. 예를 들어 **kn func build** 명령을 실행하면 **build** 필드의 값이 사용됩니다. 경우에 따라 명령줄 플래그 또는 환경 변수로 이러한 값을 재정의할 수 있습니다.

9.1. FUNC.YAML의 구성 가능한 필드

func.yaml의 대부분의 필드는 함수를 생성, 빌드 및 배포할 때 자동으로 생성됩니다. 그러나 함수 이름 또는 이미지 이름과 같은 사항을 변경하기 위해 수동으로 변경하는 필드도 있습니다.

9.1.1. buildEnvs

buildEnvs 필드를 사용하면 함수를 빌드하는 환경에서 사용 가능한 환경 변수를 설정할 수 있습니다. **envs**를 사용하여 설정한 변수와 달리 **buildEnv**를 사용하여 설정한 변수는 함수 런타임 중에 사용할 수 없습니다.

값에서 직접 **buildEnv** 변수를 설정할 수 있습니다. 다음 예제에서 **EXAMPLE1**이라는 **buildEnv** 변수에는 하나의 값이 직접 할당됩니다.

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

로컬 환경 변수에서 **buildEnv** 변수를 설정할 수도 있습니다. 다음 예에서 **EXAMPLE2**라는 **buildEnv** 변수에는 **LOCAL_ENV_VAR** 로컬 환경 변수의 값이 할당됩니다.

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

9.1.2. envs

envs 필드를 사용하면 런타임 시 사용할 수 있는 환경 변수를 설정할 수 있습니다. 환경 변수를 여러 가지 방법으로 설정할 수 있습니다.

1. 값을 직접 설정할 수 있습니다.
2. 로컬 환경 변수에 할당된 값에서 설정합니다. 자세한 내용은 **func.yaml** 필드에서 로컬 환경 변수 참조 섹션을 참조하십시오.
3. 시크릿 또는 구성 맵에 저장된 키-값 쌍에서 설정합니다.
4. 생성된 환경 변수의 이름으로 사용되는 키를 사용하여 시크릿 또는 구성 맵에 저장된 모든 키-값 쌍을 가져올 수도 있습니다.

이 예제에서는 환경 변수를 설정하는 다양한 방법을 보여줍니다.

```
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 값에서 직접 설정된 환경 변수.
- ❷ 로컬 환경 변수에 할당된 값에서 설정된 환경 변수.
- ❸ 시크릿에 저장된 키-값 쌍에서 할당된 환경 변수.
- ❹ 구성 맵에 저장된 키-값 쌍에서 할당된 환경 변수.
- ❺ 시크릿의 키-값 쌍에서 가져온 환경 변수 세트.
- ❻ 구성 맵의 키-값 쌍에서 가져온 환경 변수 세트.

9.1.3. builder

builder 필드는 함수에서 이미지를 빌드하는 데 사용하는 전략을 지정합니다. **pack** 또는 **s2i** 값을 허용합니다.

9.1.4. BUILD

build 필드는 함수를 빌드하는 방법을 나타냅니다. 값 **local** 은 함수가 시스템에 로컬로 빌드되었음을 나타냅니다. 값 **git** 은 **git** 필드에 지정된 값을 사용하여 함수가 클러스터에 빌드되었음을 나타냅니다.

9.1.5. volumes

volumes 필드를 사용하면 다음 예와 같이 지정된 경로에서 기능에 액세스할 수 있는 볼륨으로 시크릿 및 구성 맵을 마운트할 수 있습니다.

```

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

```

- ❶ **mysecret** 시크릿은 **/workspace/secret**에 있는 볼륨으로 마운트됩니다.
- ❷ **myconfigmap** 구성 맵은 **/workspace/configmap**에 있는 볼륨으로 마운트됩니다.

9.1.6. options

options 필드를 사용하면 자동 스케일링과 같이 배포된 함수에 대한 Knative Service 속성을 수정할 수 있습니다. 이러한 옵션이 설정되어 있지 않으면 기본 옵션이 사용됩니다.

다음 옵션을 사용할 수 있습니다.

- **scale**
 - **min**: 최소 복제 수입니다. 음수가 아닌 정수여야 합니다. 기본값은 0입니다.
 - **max**: 최대 복제본 수입니다. 음수가 아닌 정수여야 합니다. 기본값은 0이며 이는 제한이 없음을 의미합니다.
 - **metric**: Autoscaler에서 감시하는 메트릭 유형을 정의합니다. 기본값인 **concurrency** 또는 **rps**로 설정할 수 있습니다.
 - **target**: 동시에 수신되는 요청 수에 따라 버전을 확장할 시기에 대한 권장 사항을 제공합니다. **target** 옵션은 0.01보다 큰 부동 소수점 값을 지정할 수 있습니다. **options.resources.limits.concurrency**가 설정되지 않는 한 기본값은 100입니다. 이 경우 **target**은 기본값으로 설정됩니다.
 - **utilization**: 스케일 업하기 전에 허용된 동시 요청 사용률(백분율)입니다. 1에서 100까지의 부동 소수점 값을 지정할 수 있습니다. 기본값은 **70**입니다.
- **resources**
 - **requests**
 - **cpu**: 배포된 함수가 있는 컨테이너에 대한 CPU 리소스 요청입니다.
 - **memory**: 배포된 함수가 있는 컨테이너에 대한 메모리 리소스 요청입니다.
 - **limits**
 - **cpu**: 배포된 함수가 있는 컨테이너의 CPU 리소스 제한입니다.
 - **memory**: 배포된 함수가 있는 컨테이너의 메모리 리소스 제한입니다.
 - **concurrency**: 단일 복제본에 의해 처리되는 동시 요청 수에 대한 하드 제한입니다. 0보다 크거나 같은 정수 값이 될 수 있습니다. 기본값은 0이며 이는 제한이 없음을 의미합니다.

다음은 **scale** 옵션 구성의 예입니다.

```
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 필드는 해당 함수의 이미지 이름을 빌드한 후 설정합니다. 이 필드는 필요에 따라 수정할 수 있습니다. 이 경우 다음에 **kn func build** 또는 **kn func deploy**를 실행하면 함수 이미지가 새 이름으로 생성됩니다.

9.1.8. imageDigest

imageDigest 필드에는 함수가 배포될 때 이미지 매니페스트의 SHA256 해시가 포함됩니다. 이 값은 변경하지 마십시오.

9.1.9. labels

labels 필드를 사용하면 배포된 함수에 라벨을 설정할 수 있습니다.

값에서 직접 라벨을 설정할 수 있습니다. 다음 예에서 **role** 키가 있는 레이블에는 **backend** 값이 직접 할당됩니다.

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

로컬 환경 변수에서 레이블을 설정할 수도 있습니다. 다음 예에서 **작성자** 키가 있는 레이블에는 **USER** 로컬 환경 변수의 값이 할당됩니다.

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

9.1.10. name

name 필드는 함수의 이름을 정의합니다. 이 값은 배포 시 Knative 서비스의 이름으로 사용됩니다. 이 필드를 변경하여 후속 배포의 함수 이름을 변경할 수 있습니다.

9.1.11. namespace

namespace 필드는 함수가 배포되는 네임스페이스를 지정합니다.

9.1.12. runtime

runtime 필드는 기능에 대한 언어 런타임(예: **python**)을 지정합니다.

9.2. FUNC.YAML 필드의 로컬 환경 변수 참조

함수 구성에 API 키와 같은 중요한 정보를 저장하지 않으려면 로컬 환경에서 사용 가능한 환경 변수에 대한 참조를 추가할 수 있습니다. **func.yaml** 파일의 **envs** 필드를 수정하여 이 작업을 수행할 수 있습니다.

사전 요구 사항

- 함수 프로젝트를 생성해야 합니다.
- 로컬 환경에는 참조하려는 변수가 포함되어야 합니다.

절차

- 로컬 환경 변수를 참조하려면 다음 구문을 사용합니다.

```

{{ env:ENV_VAR }}

```

ENV_VAR을 사용하려는 로컬 환경의 변수 이름으로 바꿉니다.

예를 들어 로컬 환경에서 사용할 수 있는 **API_KEY** 변수가 있을 수 있습니다. **MY_API_KEY** 변수에 해당 값을 할당하면 함수 내에서 직접 사용할 수 있습니다.

함수 예

```

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

```

9.3. 추가 리소스

- [함수 시작하기](#)
- [Serverless 함수에서 시크릿 및 구성 맵에 액세스](#)
- [자동 확장에 대한 Knative 문서](#)
- [컨테이너 리소스 관리에 대한 Kubernetes 문서](#)
- [동시성 구성에 대한 Knative 문서](#)

10장. 함수에서 보안 및 구성 맵에 액세스

기능이 클러스터에 배포되면 시크릿 및 구성 맵에 저장된 데이터에 액세스할 수 있습니다. 이 데이터는 볼륨으로 마운트하거나 환경 변수에 할당할 수 있습니다. Knative CLI를 사용하거나 함수 구성 YAML 파일을 편집하여 수동으로 이 액세스를 대화식으로 구성할 수 있습니다.



중요

시크릿 및 구성 맵에 액세스하려면 함수를 클러스터에 배포해야 합니다. 이 기능은 로컬에서 실행되는 함수에 사용할 수 없습니다.

시크릿 또는 구성 맵 값에 액세스할 수 없는 경우 액세스할 수 없는 값을 지정하는 오류 메시지와 함께 배포가 실패합니다.

10.1. 시크릿 및 구성 맵에 대한 함수 액세스의 상호 작용 변경

kn func config 대화형 유틸리티를 사용하여 함수에서 액세스하는 시크릿 및 구성 맵을 관리할 수 있습니다. 사용 가능한 작업에는 구성 맵과 시크릿에 저장된 값 나열, 추가, 제거, 볼륨 나열, 추가, 제거 등이 포함됩니다. 이 기능을 사용하면 함수에서 액세스할 수 있는 클러스터에 저장된 데이터를 관리할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수 프로젝트에서 다음 명령을 실행합니다.

```
$ kn func config
```

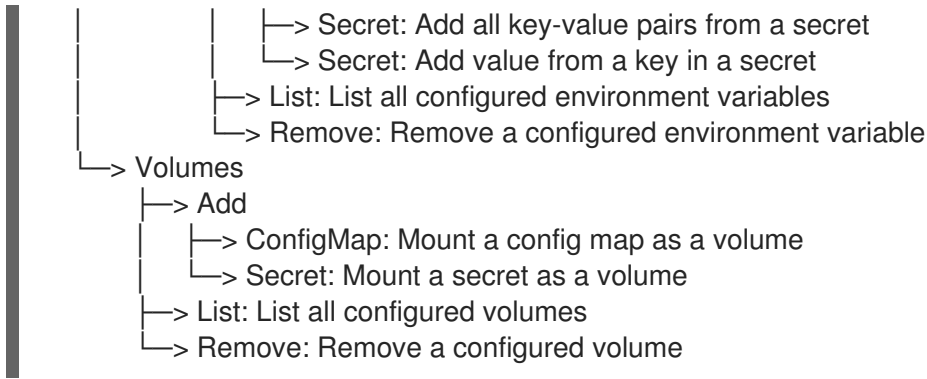
또는 **--path** 또는 **-p** 옵션을 사용하여 함수 프로젝트 디렉터리를 지정할 수 있습니다.

2. 대화형 인터페이스를 사용하여 필요한 작업을 수행합니다. 예를 들어 유틸리티를 사용하여 구성된 볼륨을 나열하면 다음과 유사한 출력이 생성됩니다.

```
$ 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"
```

이 스키마는 대화형 유틸리티에서 사용할 수 있는 모든 작업과 해당 유틸리티로 이동하는 방법을 보여줍니다.

```
kn func config
├─> Environment variables
│   └─> Add
│       ├──> ConfigMap: Add all key-value pairs from a config map
│       └─> ConfigMap: Add value from a key in a config map
```



3. 선택 사항: 함수를 배포하여 변경 사항을 적용합니다.

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

10.2. 특수 명령을 사용하여 시크릿 및 구성 맵에 대한 함수 액세스 수정

kn func config 유틸리티를 실행할 때마다 전체 대화 상자를 탐색하여 이전 섹션에서와 같이 필요한 작업을 선택해야 합니다. 단계를 저장하려면 **kn func config** 명령 보다 구체적인 양식을 실행하여 특정 작업을 직접 수행합니다.

- 구성된 환경 변수를 나열하려면 다음을 수행합니다.

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

- 함수 구성에 환경 변수를 추가하려면 다음을 수행합니다.

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

- 함수 구성에서 환경 변수를 제거하려면 다음을 수행합니다.

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

- 구성된 볼륨을 나열하려면 다음을 수행합니다.

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

- 함수 구성에 볼륨을 추가하려면 다음을 수행합니다.

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

- 함수 구성에서 볼륨을 제거하려면 다음을 수행합니다.

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

10.3. 시크릿 및 구성 맵에 수동으로 함수 액세스 추가

시크릿 및 구성 맵에 액세스하는 구성을 함수에 수동으로 추가할 수 있습니다. 기존 구성 스니펫이 있는 경우 **kn func config** 대화형 유틸리티 및 명령을 사용하는 것이 좋습니다.

10.3.1. 시크릿을 볼륨으로 마운트

시크릿을 볼륨으로 마운트할 수 있습니다. 시크릿이 마운트되면 일반 파일로 함수에서 액세스할 수 있습니다. 이를 통해 함수에 필요한 클러스터 데이터(예: 함수에서 액세스해야 하는 URI 목록)에 저장할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수에 사용할 **func.yaml** 파일을 엽니다.
2. 볼륨으로 마운트하려는 각 시크릿에 대해 **volumes** 섹션에 다음 YAML을 추가합니다.

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

- **mysecret**을 대상 시크릿의 이름으로 대체합니다.
- 시크릿을 마운트하려는 경로로 **/workspace/secret**을 대체합니다. 예를 들어 주소 시크릿을 마운트하려면 다음 YAML을 사용합니다.

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

3. 설정을 저장합니다.

10.3.2. 구성 맵을 볼륨으로 마운트

구성 맵을 볼륨으로 마운트할 수 있습니다. 구성 맵이 마운트되면 일반 파일로 함수에서 액세스할 수 있습니다. 이를 통해 함수에 필요한 클러스터 데이터(예: 함수에서 액세스해야 하는 URI 목록)에 저장할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수에 사용할 **func.yaml** 파일을 엽니다.
2. 볼륨으로 마운트하려는 각 구성 맵에 대해 **volumes** 섹션에 다음 YAML을 추가합니다.

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

- **myconfigmap**을 대상 구성 맵의 이름으로 대체합니다.
- **/workspace/configmap**을 구성 맵을 마운트하려는 경로로 바꿉니다.
예를 들어 주소 구성 맵을 마운트하려면 다음 YAML을 사용합니다.

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

3. 설정을 저장합니다.

10.3.3. 시크릿에 정의된 키 값에서 환경 변수 설정

시크릿으로 정의된 키 값에서 환경 변수를 설정할 수 있습니다. 그런 다음 시크릿에 이전에 저장된 값은 런타임 시 함수에서 환경 변수로 액세스할 수 있습니다. 이 기능은 사용자 ID와 같이 시크릿에 저장된 값에 액세스하는 데 유용할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수에 사용할 **func.yaml** 파일을 엽니다.
2. 환경 변수에 할당할 시크릿 키-값 쌍의 각 값에 대해 **envs** 섹션에 다음 YAML을 추가합니다.

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

```

envs:
- name: EXAMPLE
  value: '{{ secret:mysecret:key }}'

```

- **EXAMPLE**을 환경 변수 이름으로 대체합니다.
- **mysecret**을 대상 시크릿의 이름으로 대체합니다.
- **key**를 대상 값에 매핑된 키로 대체합니다.
예를 들어 **userdetailssecret**에 저장된 사용자 ID에 액세스하려면 다음 YAML을 사용합니다.

```

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

```

3. 설정을 저장합니다.

10.3.4. 구성 맵에 정의된 키 값에서 환경 변수 설정

구성 맵으로 정의된 키 값에서 환경 변수를 설정할 수 있습니다. 그런 다음 이전에 구성 맵에 저장된 값은 런타임 시 함수에 의해 환경 변수로 액세스할 수 있습니다. 이 기능은 사용자 ID와 같이 구성 맵에 저장된 값에 액세스하는 데 유용할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수에 사용할 **func.yaml** 파일을 엽니다.
2. 환경 변수에 할당할 구성 맵 키-값 쌍의 각 값에 대해 **envs** 섹션에 다음 YAML을 추가합니다.

```

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

```

- **EXAMPLE**을 환경 변수 이름으로 대체합니다.
- **myconfigmap**을 대상 구성 맵의 이름으로 대체합니다.
- **key**를 대상 값에 매핑된 키로 대체합니다.
예를 들어 **usercontentmap**에 저장된 사용자 ID에 액세스하려면 다음 YAML을 사용합니다.

■

```

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

```

3. 설정을 저장합니다.

10.3.5. 시크릿에 정의된 모든 값에서 환경 변수 설정

시크릿에 정의된 모든 값에서 환경 변수를 설정할 수 있습니다. 그런 다음 이전에 시크릿에 저장된 값은 런타임 시 함수에서 환경 변수로 액세스할 수 있습니다. 이 기능은 시크릿에 저장된 값 모음(예: 사용자와 관련된 데이터 집합)에 대한 액세스 권한을 동시에 얻는 데 유용할 수 있습니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수에 사용할 **func.yaml** 파일을 엽니다.
2. 모든 키-값 쌍을 환경 변수로 가져오려는 모든 시크릿에 다음 YAML을 **envs** 섹션에 추가합니다.

```

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

```

1 **mysecret**을 대상 시크릿의 이름으로 대체합니다.

예를 들어 **userdetailssecret**에 저장된 모든 사용자 데이터에 액세스하려면 다음 YAML을 사용합니다.

```

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

```

3. 설정을 저장합니다.

10.3.6. 구성 맵에 정의된 모든 값에서 환경 변수 설정

구성 맵에 정의된 모든 값에서 환경 변수를 설정할 수 있습니다. 구성 맵에 이전에 저장된 값은 런타임 시 함수에서 환경 변수로 액세스할 수 있습니다. 이 기능은 구성 맵에 저장된 값 컬렉션에 동시에 액세스하는데 유용할 수 있습니다(예: 사용자와 관련된 데이터 세트).

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수에 사용할 **func.yaml** 파일을 엽니다.
2. 모든 키-값 쌍을 환경 변수로 가져오려는 모든 구성 맵에 대해 **envs** 섹션에 다음 YAML을 추가합니다.

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

- 1 **myconfigmap**을 대상 구성 맵의 이름으로 대체합니다.

예를 들어 **userdetailsmap**에 저장된 모든 사용자 데이터에 액세스하려면 다음 YAML을 사용합니다.

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

3. 파일을 저장합니다.

11장. 함수에 주석 추가

배포된 Serverless 함수에 Kubernetes 주석을 추가할 수 있습니다. 주석을 사용하면 함수의 목적에 대한 참고와 같이 임의의 메타데이터를 함수에 연결할 수 있습니다. 주석은 **func.yaml** 구성 파일의 **annotations** 섹션에 추가됩니다.

함수 주석 기능에는 다음 두 가지 제한 사항이 있습니다.

- 함수 주석이 클러스터의 해당 Knative 서비스로 전파되면 **func.yaml** 파일에서 삭제하여 서비스에서 제거할 수 없습니다. 서비스의 YAML 파일을 직접 수정하거나 OpenShift Container Platform 웹 콘솔을 사용하여 Knative 서비스에서 주석을 제거해야 합니다.
- Knative에서 설정한 주석(예: **autoscaling** 주석)을 설정할 수 없습니다.

11.1. 함수에 주석 추가

함수에 주석을 추가할 수 있습니다. 레이블과 유사하게 주석은 키-값 맵으로 정의됩니다. 주석은 예를 들어 함수 작성자와 같은 함수에 대한 메타데이터를 제공하는 데 유용합니다.

사전 요구 사항

- OpenShift Serverless Operator 및 Knative Serving이 클러스터에 설치되어 있습니다.
- Knative(**kn**) CLI가 설치되어 있습니다.
- 함수를 생성했습니다.

절차

1. 함수에 사용할 **func.yaml** 파일을 엽니다.
2. 추가할 모든 주석에 대해 **annotations** 섹션에 다음 YAML을 추가합니다.

```
name: test
namespace: ""
runtime: go
...
annotations:
  <annotation_name>: "<annotation_value>" 1
```

- 1 **<annotation_name>: "<annotation_value>"**를 주석으로 바꿉니다.

예를 들어 Alice에서 함수가 생성되었음을 나타내기 위해 다음 주석을 포함할 수 있습니다.

```
name: test
namespace: ""
runtime: go
...
annotations:
  author: "alice@example.com"
```

3. 설정을 저장합니다.

다음에 함수를 클러스터에 배포할 때 해당 Knative 서비스에 주석이 추가됩니다.

12장. 함수 개발 참조 가이드

OpenShift Serverless Functions는 기본 기능을 생성하는 데 사용할 수 있는 템플릿을 제공합니다. 템플릿은 함수 프로젝트 상용구를 시작하고 **kn func** 툴과 함께 사용할 수 있도록 준비합니다. 각 함수 템플릿은 특정 런타임에 맞게 조정되며 해당 규칙을 따릅니다. 템플릿을 사용하면 함수 프로젝트를 자동으로 시작할 수 있습니다.

다음 런타임 템플릿을 사용할 수 있습니다.

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

12.1. NODE.JS 컨텍스트 오브젝트 참조

context 오브젝트에는 함수 개발자가 액세스할 수 있는 여러 속성이 있습니다. 이러한 속성에 액세스하면 HTTP 요청에 대한 정보를 제공하고 클러스터 로그에 대한 출력을 작성할 수 있습니다.

12.1.1. log

클러스터 로그에 출력을 작성하는 데 사용할 수 있는 로깅 오브젝트를 제공합니다. 로그는 [Pino 로깅 API](#)를 따릅니다.

로그 예

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

kn func invoke 명령을 사용하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

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

로그 수준을 **fatal,error,warn,info,debug,trace** 또는 **silent** 중 하나로 변경할 수 있습니다. 이렇게 하려면 **config** 명령을 사용하여 해당 값 중 하나를 환경 변수 **FujiNC_LOG_LEVEL**에 할당하여 **logLevel** 값을 변경합니다.

12.1.2. query

요청에 대한 쿼리 문자열을 키-값 쌍으로 반환합니다. 이러한 속성은 컨텍스트 오브젝트 자체에서도 확인할 수 있습니다.

예제 쿼리

```
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 명령을 사용하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

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

12.1.3. body

필요한 경우 요청 본문을 반환합니다. 요청 본문에 JSON 코드가 포함된 경우 속성을 직접 사용할 수 있도록 구문 분석됩니다.

본문의 예

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

curl 명령을 사용하여 이를 호출하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

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

12.1.4. headers

HTTP 요청 헤더를 오브젝트로 반환합니다.

헤더 예

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

kn func invoke 명령을 사용하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

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

12.1.5. HTTP 요청

method

HTTP 요청 메서드를 문자열로 반환합니다.

httpVersion

HTTP 버전을 문자열로 반환합니다.

httpVersionMajor

HTTP 주요 버전 번호를 문자열로 반환합니다.

httpVersionMinor

HTTP 마이너 버전 번호를 문자열로 반환합니다.

12.2. TYPESCRIPT 컨텍스트 오브젝트 참조

context 오브젝트에는 함수 개발자가 액세스할 수 있는 여러 속성이 있습니다. 이러한 속성에 액세스하면 들어오는 HTTP 요청에 대한 정보를 제공하고 클러스터 로그에 출력을 쓸 수 있습니다.

12.2.1. log

클러스터 로그에 출력을 작성하는 데 사용할 수 있는 로깅 오브젝트를 제공합니다. 로그는 [Pino 로깅 API](#)를 따릅니다.

로그 예

```
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 명령을 사용하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

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

로그 수준을 **fatal,error,warn,info,debug,trace** 또는 **silent** 중 하나로 변경할 수 있습니다. 이렇게 하려면 **config** 명령을 사용하여 해당 값 중 하나를 환경 변수 **FujiNC_LOG_LEVEL**에 할당하여 **logLevel** 값을 변경합니다.

12.2.2. query

요청에 대한 쿼리 문자열을 키-값 쌍으로 반환합니다. 이러한 속성은 컨텍스트 오브젝트 자체에서도 확인할 수 있습니다.

예제 쿼리

```
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 명령을 사용하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

```
{"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. body

요청 본문(있는 경우)을 반환합니다. 요청 본문에 JSON 코드가 포함된 경우 속성을 직접 사용할 수 있도록 구문 분석됩니다.

본문의 예

```
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 명령을 사용하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

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

12.2.4. headers

HTTP 요청 헤더를 오브젝트로 반환합니다.

헤더 예

```

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 명령을 사용하여 이를 호출하여 함수에 액세스할 수 있습니다.

명령 예

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

출력 예

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

12.2.5. HTTP 요청

method

HTTP 요청 메서드를 문자열로 반환합니다.

httpVersion

HTTP 버전을 문자열로 반환합니다.

httpVersionMajor

HTTP 주요 버전 번호를 문자열로 반환합니다.

httpVersionMinor

HTTP 마이너 버전 번호를 문자열로 반환합니다.