Red Hat OpenShift Serverless 1.30

가시성

관리자 및 개발자 메트릭, 클러스터 로깅, 추적 등 관찰 기능.

Last Updated: 2023-09-20

Red Hat OpenShift Serverless 1.30 가시성

관리자 및 개발자 메트릭, 클러스터 로깅, 추적 등 관찰 기능.

법적 공지

초록

이 문서에서는 Knative 서비스의 성능을 모니터링하는 방법에 대해 자세히 설명합니다. 또한 OpenShift Serverless에서 OpenShift Logging 및 OpenShift distributed tracing을 사용하는 방법을 자세히 설명합니다.

차례

1장. 관리자 지표

1.1. 서버리스 관리자 메트릭

클러스터 관리자는 메트릭을 사용하여 OpenShift Serverless 클러스터 구성 요소 및 워크로드를 수행하는 방법을 모니터링할 수 있습니다.

웹 콘솔 **관리자** 화면에서 **대시보드** 로 이동하여 OpenShift Serverless의 다양한 메트릭을 볼 수 있습니다.

1.1.1. 사전 요구 사항

- 클러스터 메트릭 활성화에 대한 정보는 메트릭 관리에 대한 OpenShift Container Platform 설명서를 참조하십시오.
- 클러스터 관리자 액세스 권한이 있는 계정(또는 AWS의 OpenShift Dedicated 또는 Red Hat OpenShift Service의 전용 관리자 액세스)에 액세스할 수 있습니다.
- 웹 콘솔의 **관리자** 화면에 액세스할 수 있습니다.

주의

Service Mesh가 mTLS를 사용하여 사용하도록 설정된 경우 Service Mesh가 Prometheus의 메트릭 스크랩을 허용하지 않기 때문에 기본적으로 Knative Serviceing에 대한 메트릭이 사용되지 않도록 설정됩니다.

이 문제를 해결하는 방법에 대한 자세한 내용은 mTLS와 Service Mesh를 사용할 때 Knative Serving 메트릭 활성화를 참조하십시오.

메트릭을 스크랩하는 작업은 스크랩 요청이 활성화를 통과하지 않기 때문에 Knative 서비스의 자동 확장에 영향을 미치지 않습니다. 결과적으로 실행 중인 Pod가 없는 경우 스크랩이 수행되지 않습니다.

1.2. 서버리스 컨트롤러 메트릭

다음 메트릭은 컨트롤러 로직을 구현하는 구성 요소에서 출력됩니다. 이러한 메트릭은 조정 작업 및 조정 요청이 작업 대기열에 추가되는 작업 대기열 동작에 대한 세부 정보를 표시합니다.

메트릭 이름	설명	유형	태그	단위
work_queue_depth	작업 대기열의 깊이	게이지	**조정기**	정수(단위 없음)
reconcile_count	조정 작업 수	카운터	**reconciler**, **success**	정수(단위 없음)
reconcile_latency	조정 작업 대기 시간	히스토그램	**reconciler**, **success**	밀리초

메트릭 이름	설명	유형	태그	단위
workqueue_adds_total	작업 대기열에서 처리하는 총 추가 작업 수	카운터	**name**	정수(단위 없음)
workqueue_queue_latency_seconds	항목이 요청되기 전에 작업 대기열에 남아 있는 시간	히스토그램	**name**	초
workqueue_retries_total	작업 대기열에서 처리한 총 재시도 횟수	카운터	**name**	정수(단위 없음)
workqueue_work_duration_seconds	작업 대기열에서 처리하는 데 걸리는 시간 및 항목	히스토그램	**name**	초
workqueue_unfinished_work_seconds	처리되지 않은 작업 대기열 항목이 진행 중인 시간	히스토그램	**name**	초
workqueue_longest_running_processor_seconds	가장 긴 미결 작업 대기열 항목이 진행 중인 시간	히스토그램	**name**	초

1.3. WEBHOOK 메트릭

Webhook 메트릭은 작업에 대한 유용한 정보를 표시합니다. 예를 들어, 많은 작업이 실패하면 사용자가 생성한 리소스에 문제가 있음을 나타내는 것일 수 있습니다.

메트릭 이름	설명	유형	태그	단위
request_count	webhook로 라우팅되는 요청 수입니다.	카운터	**admission_allowed**, **kind_group**, **kind_kind**, **kind_version**, **request_operation**, **resource_group**, **resource_namespace**, **resource_resource**, **resource_version**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
request_latencies	webhook 요청에 대한 응답 시간입니다.	히스토그램	**admission_allowed**, **kind_group**, **kind_kind**, **kind_version**, **request_operation**, **resource_group**, **resource_namespace**, **resource_resource**, **resource_version**	밀리초

1.4. KNATIVE EVENTING 메트릭

클러스터 관리자는 Knative Eventing 구성 요소에 대한 다음 메트릭을 볼 수 있습니다.

HTTP 코드에서 메트릭을 집계하여 정상적인 이벤트 (2xx) 및 실패한 이벤트(5xx)의 두 가지 범주로 구분할 수 있습니다.

1.4.1. 브로커 Ingress 메트릭

다음 메트릭을 사용하여 브로커 Ingress를 디버그하고, 수행 방법을 확인하고, Ingress 구성 요소에서 전달되는 이벤트를 확인할 수 있습니다.

메트릭 이름	설명	유형	태그	단위
event_count	브로커가 수신한 이벤트 수.	카운터	**broker_name**, **event_type**, **namespace_name**, **response_code**, **response_code_class**, **unique_name**	정수(단위 없음)
event_dispatch_latencies	이벤트를 채널로 전달하는 데 걸린 시간입니다.	히스토그램	**broker_name**, **event_type**, **namespace_name**, **response_code**, **response_code_class**, **unique_name**	밀리초

1.4.2. 브로커 필터 메트릭

다음 메트릭을 사용하여 브로커 필터를 디버그하고, 수행 방법을 확인하고, 필터에서 전달되는 이벤트를 확인할 수 있습니다. 이벤트에서 필터링 작업의 대기 시간을 측정할 수도 있습니다.

메트릭 이름	설명	유형	태그	단위
event_count	브로커가 수신한 이벤트 수.	카운터	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	정수(단위 없음)
event_dispatch_latencies	이벤트를 채널로 전달하는 데 걸린 시간입니다.	히스토그램	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	밀리초
event_processing_latencies	트리거 구독자에게 전달되기 전에 이벤트를 처리하는 데 걸리는 시간입니다.	히스토그램	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **trigger_name**, **unique_name**	밀리초

1.4.3. InMemoryChannel 디스패처 메트릭

다음 메트릭을 사용하여 **InMemoryChannel** 채널을 디버그하고, 어떻게 수행하는지, 채널에서 디스패치 중인 이벤트를 확인할 수 있습니다.

메트릭 이름	설명	유형	태그	단위

메트릭 이름	설명	유형	태그	단위
event_count	**InMemoryChannel** 채널에서 디스패치하는 이벤트 수.	카운터	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	정수(단위 없음)
event_dispatch_latencies	**InMemoryChannel** 채널에서 이벤트를 디스패치하는 데 걸린 시간.	히스토그램	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	밀리초

1.4.4. 이벤트 소스 메트릭

다음 메트릭을 사용하여 이벤트 소스에서 연결된 이벤트 싱크로 이벤트가 전달되었는지 확인할 수 있습니다.

메트릭 이름	설명	유형	태그	단위
event_count	이벤트 소스에서 보낸 이벤트 수입니다.	카운터	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
retry_event_count	처음에 이벤트 소스가 전송되지 못한 후 이벤트 소스에 의해 전송되는 재시도 이벤트 수입니다.	카운터	**event_source**, **event_type**, **name**, **namespace_name**, **resource_group**, **response_code**, **response_code_class**, **response_error**, **response_timeout**	정수(단위 없음)

1.5. KNATIVE SERVING 메트릭

클러스터 관리자는 Knative Serving 구성 요소에 대한 다음 메트릭을 볼 수 있습니다.

1.5.1. 활성화기 메트릭

다음 메트릭을 사용하여 트래픽이 활성화기를 통해 전달될 때 애플리케이션이 응답하는 방식을 파악할 수 있습니다.

메트릭 이름	설명	유형	태그	단위
request_concurrency	활성화기 또는 보고 기간에 평균 동시성으로 라우팅되는 동시 요청 수입니다.	게이지	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **revision_name**, **service_name**	정수(단위 없음)
request_count	활성화기로 라우팅되는 요청 수입니다. 이러한 요청은 활성화기 처리기에서 실행된 요청입니다.	카운터	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**,	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
request_latencies	실행된 라우팅된 요청에 대한 응답 시간(밀리초)입니다.	히스토그램	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	밀리초

1.5.2. 자동 스케일러 메트릭

자동 스케일러 구성 요소는 각 버전의 자동 스케일러 동작과 관련된 여러 메트릭을 표시합니다. 예를 들어, 자동 스케일러가 서비스에 할당하려는 대상 Pod 수, 안정적인 기간 동안 초당 평균 요청 수 또는 Knative Pod 자동 스케일러(KPA)를 사용하는 경우 자동 스케일러가 패닉 모드인지를 모니터링할 수 있습니다.

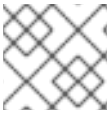
메트릭 이름	설명	유형	태그	단위
desired_pods	자동 스케일러가 서비스에 할당하려는 Pod 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
excess_burst_capacity	stable 창에서 제공되는 추가 버스트 용량입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
stable_request_concurrency	stable 창에서 모니터링되는 각 pod의 평균 요청 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
panic_request_concurrency	panic 창에서 모니터링되는 각 Pod의 평균 요청 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
target_concurrency_per_pod	자동 스케일러가 각 pod에 보내려는 동시 요청 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
stable_requests_per_second	stable 창에서 모니터링되는 각 pod의 평균 요청 수(초당)입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
panic_requests_per_second	panic 창에서 모니터링되는 각 Pod의 평균 요청 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
target_requests_per_second	각 pod에 대해 자동 스케일러가 대상으로 하는 초당 요청 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
panic_mode	자동 스케일러가 패닉 모드이면 이 값은 **1**이며 자동 스케일러가 패닉 모드가 아닌 경우 **0**입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
requested_pods	Kubernetes 클러스터에서 자동 스케일러가 요청한 Pod 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
actual_pods	할당되어 있고 현재 준비 상태인 Pod 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
not_ready_pods	준비되지 않은 상태의 Pod 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
pending_pods	현재 보류 중인 pod 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)
terminating_pods	현재 종료 중인 Pod 수입니다.	게이지	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	정수(단위 없음)

1.5.3. Go 런타임 메트릭

각 Knative Serving 컨트롤 플레인 프로세스는 수많은 Go 런타임 메모리 통계(MemStats)를 출력합니다.



참고

각 메트릭의 **name** 태그는 빈 태그입니다.

메트릭 이름	설명	유형	태그	단위
go_alloc	할당된 힙 오브젝트의 바이트 수입니다. 이 메트릭은 **heap_alloc**과 동일합니다.	게이지	**name**	정수(단위 없음)
go_total_alloc	힙 오브젝트에 할당된 누적 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_sys	운영 체제에서 얻은 총 메모리 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_lookups	런타임에서 수행하는 포인터 검색 수입니다.	게이지	**name**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
go_mallocs	할당된 힙 오브젝트의 누적 개수입니다.	게이지	**name**	정수(단위 없음)
go_frees	해제된 힙 오브젝트의 누적 개수입니다.	게이지	**name**	정수(단위 없음)
go_heap_alloc	할당된 힙 오브젝트의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_heap_sys	운영 체제에서 얻은 힙 메모리의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_heap_idle	유휴 상태의 사용되지 않는 범위의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_heap_in_use	현재 사용 중인 범위의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_heap_released	운영 체제로 반환되는 실제 메모리의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_heap_objects	할당된 힙 오브젝트 수입니다.	게이지	**name**	정수(단위 없음)
go_stack_in_use	현재 사용 중인 스택 범위의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_stack_sys	운영 체제에서 얻은 스택 메모리의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_mspan_in_use	할당된 **mspan** 구조의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_mspan_sys	**mspan** 구조의 운영 체제에서 얻은 메모리 바이트 수입니다.	게이지	**name**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
go_mcache_in_use	할당된 **mcache** 구조의 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_mcache_sys	**mcache** 구조의 운영 체제에서 얻은 메모리 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_bucket_hash_sys	버킷 해시 테이블 프로파일에서의 메모리 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_gc_sys	가비지 컬렉션 메타데이터의 메모리 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_other_sys	기타, 오프-힙 런타임 할당의 메모리 바이트 수입니다.	게이지	**name**	정수(단위 없음)
go_next_gc	다음 가비지 컬렉션 사이클의 대상 힙 크기입니다.	게이지	**name**	정수(단위 없음)
go_last_gc	마지막 가비지 컬렉션이*Epoch* 또는 Unix 시간으로 완료된 시간입니다.	게이지	**name**	나노초
go_total_gc_pause_ns	프로그램이 시작된 이후 가비지 컬렉션 *stop-the-world* 일시 중지의 누적 시간입니다	게이지	**name**	나노초
go_num_gc	완료된 가비지 컬렉션 사이클 수입니다.	게이지	**name**	정수(단위 없음)
go_num_forced_gc	가비지 컬렉션 기능을 호출하는 애플리케이션으로 인해 강제된 가비지 컬렉션 사이클의 수입니다.	게이지	**name**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
go_gc_cpu_fraction	프로그램이 시작된 이후 가비지 컬렉터에서 사용한 프로그램의 사용 가능한 일부 CPU 시간입니다.	게이지	**name**	정수(단위 없음)

2장. 개발자 메트릭

2.1. 서버리스 개발자 메트릭 개요

지표를 사용하면 개발자가 Knative 서비스 성능을 모니터링할 수 있습니다. OpenShift Container Platform 모니터링 스택을 사용하여 Knative 서비스에 대한 상태 점검 및 메트릭을 기록하고 확인할 수 있습니다.

웹 콘솔 **개발자** 화면에서 **대시보드** 로 이동하여 OpenShift Serverless의 다양한 메트릭을 볼 수 있습니다.

주의

Service Mesh가 mTLS를 사용하여 사용하도록 설정된 경우 Service Mesh가 Prometheus의 메트릭 스크랩을 허용하지 않기 때문에 기본적으로 Knative Serviceing에 대한 메트릭이 사용되지 않도록 설정됩니다.

이 문제를 해결하는 방법에 대한 자세한 내용은 mTLS와 Service Mesh를 사용할 때 Knative Serving 메트릭 활성화를 참조하십시오.

메트릭을 스크랩하는 작업은 스크랩 요청이 활성화를 통과하지 않기 때문에 Knative 서비스의 자동 확장에 영향을 미치지 않습니다. 결과적으로 실행 중인 Pod가 없는 경우 스크랩이 수행되지 않습니다.

2.1.1. OpenShift Container Platform에 대한 추가 리소스

- 모니터링 개요
- 사용자 정의 프로젝트 모니터링 활성화
- 서비스 모니터링 방법 지정

2.2. 기본적으로 노출되는 KNATIVE 서비스 메트릭

표 2.1. 포트 9090의 각 Knative 서비스에 대해 기본적으로 노출되는 메트릭

메트릭 이름, 단위 및 유형	설명	메트릭 태그
queue_requests_per_second 메트릭 단위: 무차원 단위 메트릭 유형: 게이지	큐 프록시에 도달하는 초당 요청 수입니다. 공식: **stats.RequestCount / r.reportingPeriodSeconds** **stats.RequestCount**는 지정된 보고 기간 동안 네트워킹 **pkg** 통계에서 직접 계산됩니다.	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

메트릭 이름, 단위 및 유형	설명	메트릭 태그
queue_proxied_operations_per_second 메트릭 단위: 무차원 단위 메트릭 유형: 게이지	초당 프록시된 요청 수입니다. 공식: **stats.ProxiedRequestCount / r.reportingPeriodSeconds** **stats.ProxiedRequestCount** 는 지정된 보고 기간 동안 네트워킹 **pkg** 통계에서 직접 계산됩니다.	
queue_average_concurrent_requests 메트릭 단위: 무차원 단위 메트릭 유형: 게이지	이 Pod에서 현재 처리 중인 요청 수입니다. 평균 동시성은 다음과 같이 네트워킹 **pkg** 측에서 계산됩니다. ● **req** 변경이 발생하면 변경 사이의 시간이 계산됩니다. 결과에 따라 delta를 초과하는 현재 동시성이 계산되어 현재 계산된 동시성에 추가됩니다. 또한 delta의 합계가 유지됩니다. delta를 통한 현재 동시성은 다음과 같이 계산됩니다. **global_concurrency** × delta ● 보고가 완료되면 합계와 현재 계산된 동시성이 재설정됩니다. ● 평균 동시성을 보고할 때 현재 계산된 동시성은 deltas의 합계로 나뉩니다. ● 새 요청이 도착하면 글로벌 동시성 카운터가 증가합니다. 요청이 완료되면 카운터가 줄어듭니다.	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"
queue_average_proxied_concurrent_requests 메트릭 단위: 무차원 단위 메트릭 유형: 게이지	이 Pod에서 현재 처리하는 프록시된 요청 수입니다. **stats.AverageProxiedConcurrency**	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

메트릭 이름, 단위 및 유형	설명	메트릭 태그
process_uptime 메트릭 단위: 초 메트릭 유형: 게이지	프로세스가 작동된 시간(초)입니다.	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

표 2.2. 포트 9091의 각 Knative 서비스에 대해 기본적으로 노출되는 메트릭

메트릭 이름, 단위 및 유형	설명	메트릭 태그
request_count 메트릭 단위: 무차원 단위 메트릭 유형: 카운터	**queue-proxy**로 라우팅되는 요청 수입니다.	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
request_latencies 메트릭 단위: 밀리초 메트릭 유형: 히스토그램	응답 시간(밀리초)입니다.	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
app_request_count 메트릭 단위: 무차원 단위 메트릭 유형: 카운터	**user-container**로 라우팅되는 요청 수입니다.	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

메트릭 이름, 단위 및 유형	설명	메트릭 태그
app_request_latencies 메트릭 단위: 밀리초 메트릭 유형: 히스토그램	응답 시간(밀리초)입니다.	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
queue_depth 메트릭 단위: 무차원 단위 메트릭 유형: 게이지	서비스 및 대기열에 있는 현재 항목 수 또는 무제한 동시 실행인 경우 보고되지 않은 항목 수입니다. **breaker.inFlight**가 사용됩니다.	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

2.3. 사용자 정의 애플리케이션 메트릭이 있는 KNATIVE 서비스

Knative 서비스에서 내보낸 메트릭 집합을 확장할 수 있습니다. 정확한 구현은 애플리케이션 및 사용된 언어에 따라 다릅니다.

다음 목록에서는 처리된 이벤트 사용자 지정 메트릭을 내보내는 샘플 Go 애플리케이션을 구현합니다.

```
package main

import (
  "fmt"
  "log"
  "net/http"
  "os"

  "github.com/prometheus/client_golang/prometheus" 1
  "github.com/prometheus/client_golang/prometheus/promauto"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
  opsProcessed = promauto.NewCounter(prometheus.CounterOpts{ 2
    Name: "myapp_processed_ops_total",
    Help: "The total number of processed events",
  })
```

```
)


func handler(w http.ResponseWriter, r *http.Request) {
  log.Print("helloworld: received a request")
  target := os.Getenv("TARGET")
  if target == "" {
    target = "World"
  }
  fmt.Fprintf(w, "Hello %s!\n", target)
  opsProcessed.Inc() 3
}

func main() {
  log.Print("helloworld: starting server...")

  port := os.Getenv("PORT")
  if port == "" {
    port = "8080"
  }

  http.HandleFunc("/", handler)

  // Separate server for metrics requests
  go func() { 4
    mux := http.NewServeMux()
    server := &http.Server{
      Addr: fmt.Sprintf(":%s", "9095"),
      Handler: mux,
    }
    mux.Handle("/metrics", promhttp.Handler())
    log.Printf("prometheus: listening on port %s", 9095)
    log.Fatal(server.ListenAndServe())
  }()

   // Use same port as normal requests for metrics
  //http.Handle("/metrics", promhttp.Handler()) 5
  log.Printf("helloworld: listening on port %s", port)
  log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```

1 Prometheus 패키지 포함.

2 **opsProcessed** 메트릭 정의.

3 **opsProcessed** 메트릭 증가.

4 메트릭 요청에 대해 별도의 서버를 사용하도록 구성.

5 메트릭 및 **metrics** 하위 경로에 대한 일반 요청과 동일한 포트를 사용하도록 구성.

2.4. 사용자 정의 메트릭 스크랩 구성

사용자 정의 메트릭 스크랩은 사용자 워크로드 모니터링을 위해 Prometheus의 인스턴스에서 수행합니다. 사용자 워크로드 모니터링을 활성화하고 애플리케이션을 생성한 후에는 모니터링 스택에서 메트릭을 스크랩하는 방법을 정의하는 구성이 필요합니다.

다음 샘플 구성은 애플리케이션에 대한 **ksvc**를 정의하고 서비스 모니터를 구성합니다. 정확한 구성은 애플리케이션과 메트릭을 내보내는 방법에 따라 다릅니다.

```
apiVersion: serving.knative.dev/v1 (1)
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
      annotations:
    spec:
      containers:
      - image: docker.io/skonto/helloworld-go:metrics
        resources:
          requests:
            cpu: "200m"
        env:
        - name: TARGET
          value: "Go Sample v1"
---
apiVersion: monitoring.coreos.com/v1 (2)
kind: ServiceMonitor
metadata:
  labels:
  name: helloworld-go-sm
spec:
  endpoints:
  - port: queue-proxy-metrics
    scheme: http
  - port: app-metrics
    scheme: http
  namespaceSelector: {}
  selector:
    matchLabels:
       name:  helloworld-go-sm
---
apiVersion: v1 (3)
kind: Service
metadata:
  labels:
    name:  helloworld-go-sm
  name:  helloworld-go-sm
spec:
  ports:
  - name: queue-proxy-metrics
    port: 9091
    protocol: TCP
    targetPort: 9091
  - name: app-metrics
```

```
    port: 9095
    protocol: TCP
    targetPort: 9095
  selector:
    serving.knative.dev/service: helloworld-go
  type: ClusterIP
```

1. 애플리케이션 사양.
2. 스크랩되는 애플리케이션의 메트릭 구성.
3. 메트릭을 스크랩하는 방식의 구성.

2.5. 서비스의 메트릭 검사

메트릭을 내보내도록 애플리케이션을 구성하고 모니터링 스택을 스크랩하면 웹 콘솔에서 메트릭을 검사할 수 있습니다.

사전 요구 사항

- OpenShift Container Platform 웹 콘솔에 로그인했습니다.
- OpenShift Serverless Operator 및 Knative Serving이 설치되어 있습니다.

프로세스

1. 선택 사항: 메트릭에서 볼 수 있는 애플리케이션에 대한 요청을 실행합니다.

```
$ hello_route=$(oc get ksvc helloworld-go -n ns1 -o jsonpath='{.status.url}') && \
    curl $hello_route
```

출력 예

```
Hello Go Sample v1!
```

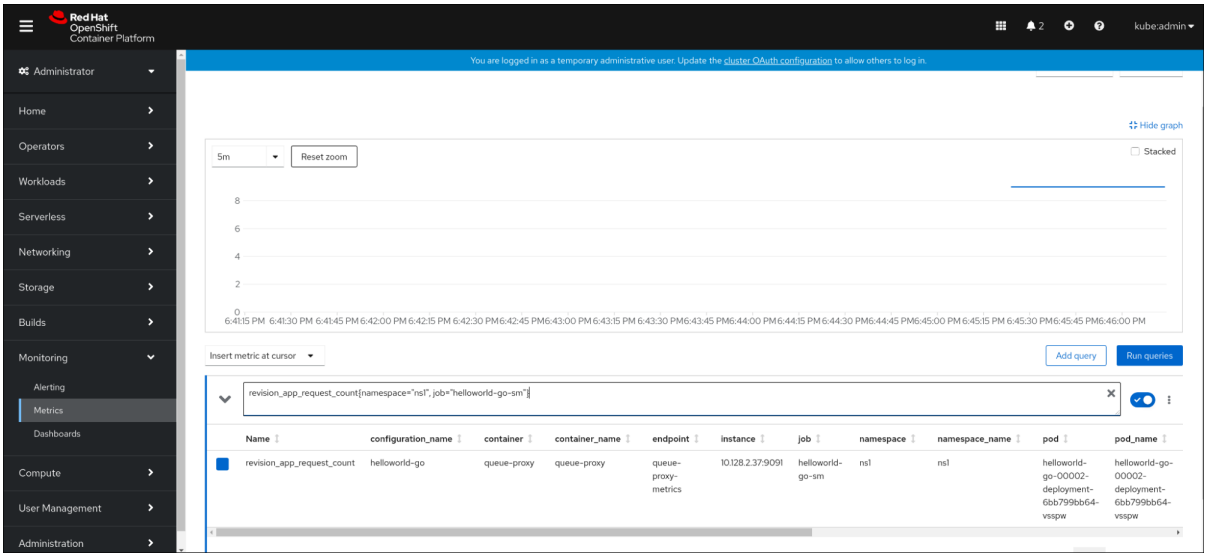
2. 웹 콘솔에서 **모니터링 → 메트릭** 인터페이스로 이동합니다.
3. 입력 필드에 모니터링할 메트릭의 쿼리를 입력합니다. 예를 들면 다음과 같습니다.

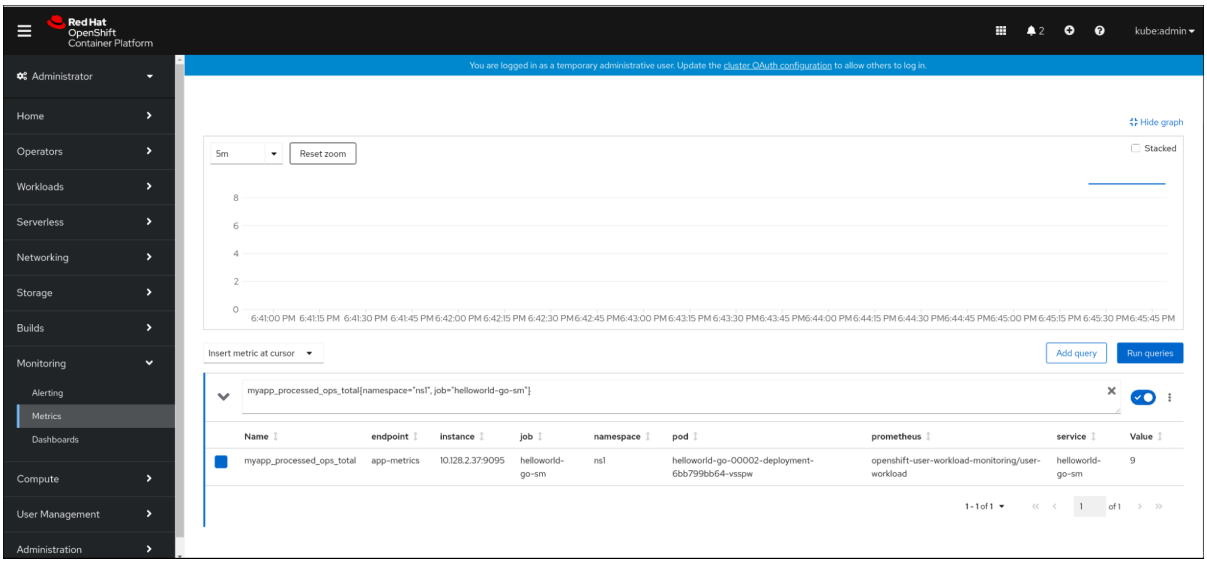
```
revision_app_request_count{namespace="ns1", job="helloworld-go-sm"}
```

다른 예:

```
myapp_processed_ops_total{namespace="ns1", job="helloworld-go-sm"}
```

4. 시각화된 메트릭을 모니터링합니다.





2.5.1. 대기열 프록시 메트릭

각 Knative 서비스에는 애플리케이션 컨테이너에 대한 연결을 프록시하는 프록시 컨테이너가 있습니다. 큐 프록시 성능에 대해 여러 메트릭이 보고됩니다.

다음 메트릭을 사용하여 요청이 프록시 측에 대기되었는지 및 애플리케이션에서 요청을 처리할 때의 실제 지연을 측정할 수 있습니다.

메트릭 이름	설명	유형	태그	단위
revision_request_count	**queue-proxy** Pod로 라우팅되는 요청 수입니다.	카운터	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	정수(단위 없음)

메트릭 이름	설명	유형	태그	단위
revision_request_latencies	수정 버전 요청의 응답 시간입니다.	히스토그램	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	밀리초
revision_app_request_count	**user-container** pod로 라우팅되는 요청 수입니다.	카운터	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	정수(단위 없음)
revision_app_request_latencies	수정 버전 앱 요청의 응답 시간입니다.	히스토그램	**configuration_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	밀리초
revision_queue_depth	**serving** 및 **waiting** 대기열의 현재 항목 수입니다. 무제한 동시 실행이 구성된 경우 이 메트릭은 보고되지 않습니다.	게이지	**configuration_name**, **event-display**, **container_name**, **namespace_name**, **pod_name**, **response_code_class**, **revision_name**, **service_name**	정수(단위 없음)

2.6. 서비스 메트릭 대시보드

네임스페이스별로 큐 프록시 메트릭을 집계하는 전용 대시보드를 사용하여 메트릭을 검사할 수 있습니다.

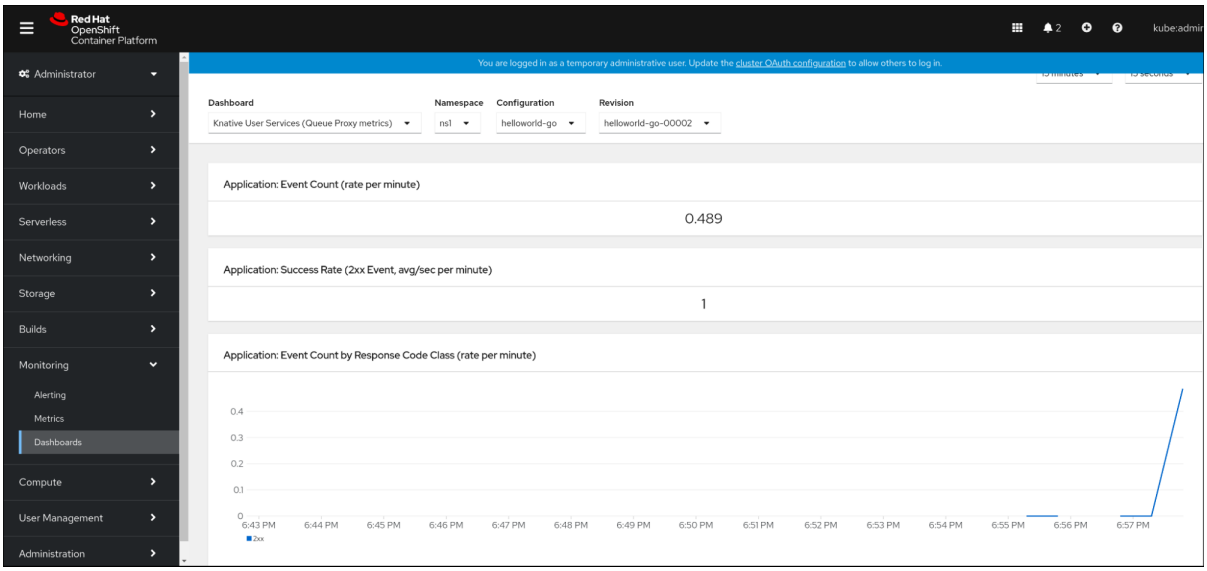
2.6.1. 대시보드에서 서비스의 메트릭 검사

사전 요구 사항

- OpenShift Container Platform 웹 콘솔에 로그인했습니다.
- OpenShift Serverless Operator 및 Knative Serving이 설치되어 있습니다.

프로세스

1. 웹 콘솔에서 **모니터링 → 메트릭** 인터페이스로 이동합니다.
2. **Knative 사용자 서비스(Queue 프록시 메트릭)** 대시보드를 선택합니다.
3. 애플리케이션에 해당하는 **네임스페이스,구성** 및 **개정 버전**을 선택합니다.
4. 시각화된 메트릭을 모니터링합니다.



3장. 클러스터 로깅

3.1. OPENSHIFT SERVERLESS에서 OPENSHIFT LOGGING 사용

3.1.1. Red Hat OpenShift의 로깅 하위 시스템 배포 정보

OpenShift Container Platform 클러스터 관리자는 OpenShift Container Platform 웹 콘솔 또는 CLI를 사용하여 로깅 하위 시스템을 배포하여 OpenShift Elasticsearch Operator 및 Red Hat OpenShift Logging Operator를 설치할 수 있습니다. Operator가 설치되면 **ClusterLogging** 사용자 정의 리소스(CR)를 생성하여 로깅 하위 시스템 Pod 및 로깅 하위 시스템을 지원하는 데 필요한 기타 리소스를 예약합니다. Operator는 로깅 하위 시스템을 배포, 업그레이드 및 유지보수해야 합니다.

ClusterLogging CR은 로그를 수집, 저장 및 시각화하기 위해 로깅 스택의 모든 구성 요소를 포함하는 전체 로깅 하위 시스템 환경을 정의합니다. Red Hat OpenShift Logging Operator는 로깅 하위 시스템 CR을 감시하고 그에 따라 로깅 배포를 조정합니다.

관리자와 애플리케이션 개발자는 보기 권한이 있는 프로젝트의 로그를 볼 수 있습니다.

3.1.2. Red Hat OpenShift의 로깅 하위 시스템 배포 및 구성 정보

로깅 하위 시스템은 중소 규모의 OpenShift Container Platform 클러스터에 맞게 조정되는 기본 구성과 함께 사용하도록 설계되었습니다.

다음 설치 지침에는 로깅 하위 시스템 인스턴스를 생성하고 로깅 하위 시스템 환경을 구성하는 데 사용할 수 있는 샘플 **ClusterLogging** 사용자 정의 리소스(CR)가 포함되어 있습니다.

기본 로깅 하위 시스템 설치를 사용하려면 샘플 CR을 직접 사용할 수 있습니다.

배포를 사용자 정의하려면 필요에 따라 샘플 CR을 변경합니다. 이어지는 내용에서는 OpenShift Logging 인스턴스를 설치할 때 수행하거나 설치 후 수정할 수 있는 구성에 대해 설명합니다. **ClusterLogging** 사용자 정의 리소스 외부에서 수행할 수 있는 수정을 포함하여 각 구성 요소의 작업에 대한 자세한 내용은 구성 섹션을 참조하십시오.

3.1.2.1. 로깅 하위 시스템 구성 및 튜닝

openshift-logging 프로젝트에 배포된 **ClusterLogging** 사용자 정의 리소스를 수정하여 로깅 하위 시스템을 구성할 수 있습니다.

설치 시 또는 설치 후 다음 구성 요소를 수정할 수 있습니다.

메모리 및 CPU

유효한 메모리 및 CPU 값으로 **resources** 블록을 수정하여 각 구성 요소의 CPU 및 메모리 제한을 모두 조정할 수 있습니다.

```
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu:
          memory: 16Gi
        requests:
          cpu: 500m
          memory: 16Gi
```

```
    type: "elasticsearch"
  collection:
    logs:
      fluentd:
        resources:
          limits:
            cpu:
            memory:
          requests:
            cpu:
            memory:
        type: "fluentd"
  visualization:
    kibana:
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu:
          memory:
      type: kibana
```

Elasticsearch 스토리지

storageClass name 및 **size** 매개변수를 사용하여 Elasticsearch 클러스터의 영구 스토리지 클래스 및 크기를 구성할 수 있습니다. Red Hat OpenShift Logging Operator는 이러한 매개변수를 기반으로 Elasticsearch 클러스터의 각 데이터 노드에 대한 PVC(영구 볼륨 클레임)를 생성합니다.

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

이 예제에서는 클러스터의 각 데이터 노드가 "gp2" 스토리지의 "200G"를 요청하는 PVC에 바인딩되도록 지정합니다. 각 기본 분할은 단일 복제본에서 지원합니다.

참고

storage 블록을 생략하면 임시 스토리지만 포함하는 배포가 생성됩니다.

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

Elasticsearch 복제 정책

Elasticsearch 분할이 클러스터의 여러 데이터 노드에 복제되는 방법을 정의하는 정책을 설정할 수 있습니다.

- **FullRedundancy**. 각 인덱스의 분할이 모든 데이터 노드에 완전히 복제됩니다.
- **MultipleRedundancy**. 각 인덱스의 분할이 데이터 노드의 1/2에 걸쳐 있습니다.
- **SingleRedundancy**. 각 분할의 단일 사본입니다. 두 개 이상의 데이터 노드가 존재하는 한 항상 로그를 사용할 수 있고 복구할 수 있습니다.
- **ZeroRedundancy**. 분할 복사본이 없습니다. 노드가 다운되거나 실패하는 경우 로그를 사용할 수 없거나 로그가 손실될 수 있습니다.

3.1.2.2. 수정된 ClusterLogging 사용자 정의 리소스 샘플

다음은 이전에 설명한 옵션을 사용하여 수정한 **ClusterLogging** 사용자 정의 리소스의 예입니다.

수정된 ClusterLogging 사용자 정의 리소스 샘플

```
apiVersion: "logging.openshift.io/v1"
kind: "ClusterLogging"
metadata:
  name: "instance"
  namespace: "openshift-logging"
spec:
  managementState: "Managed"
  logStore:
    type: "elasticsearch"
    retentionPolicy:
      application:
        maxAge: 1d
      infra:
        maxAge: 7d
      audit:
        maxAge: 7d
    elasticsearch:
      nodeCount: 3
      resources:
        limits:
          cpu: 200m
          memory: 16Gi
        requests:
          cpu: 200m
          memory: 16Gi
        storage:
          storageClassName: "gp2"
          size: "200G"
      redundancyPolicy: "SingleRedundancy"
  visualization:
    type: "kibana"
    kibana:
      resources:
        limits:
          memory: 1Gi
        requests:
```

```
        cpu: 500m
        memory: 1Gi
    replicas: 1
collection:
  logs:
    type: "fluentd"
    fluentd:
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: 200m
          memory: 1Gi
```

3.2. KNATIVE SERVING 구성 요소의 로그 찾기

다음 절차를 사용하여 Knative Serving 구성 요소의 로그를 찾을 수 있습니다.

3.2.1. OpenShift 로깅을 사용하여 Knative Serving 구성 요소 로그 찾기

사전 요구 사항

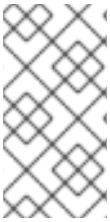
- OpenShift CLI(**oc**)를 설치합니다.

프로세스

1. Kibana 경로를 가져옵니다.

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

2. 경로의 URL을 사용하여 Kibana 대시보드로 이동한 후 로그인합니다.
3. 인덱스가 **.all**로 설정되어 있는지 확인합니다. 인덱스가 **.all**로 설정되어 있지 않으면 OpenShift Container Platform 시스템 로그만 나열됩니다.
4. **knative-serving** 네임스페이스를 사용하여 로그를 필터링합니다. 검색 상자에 **kubernetes.namespace_name:knative-serving**을 입력하여 결과를 필터링합니다.



참고

Knative Serving에서는 기본적으로 구조화된 로깅을 사용합니다. OpenShift Logging Fluentd 설정을 사용자 정의하여 이러한 로그의 구문 분석을 활성화할 수 있습니다. 이렇게 하면 로그를 더 쉽게 검색할 수 있고 로그 수준에서 필터링하여 문제를 빠르게 확인할 수 있습니다.

3.3. KNATIVE SERVING 서비스의 로그 검색

다음 절차를 사용하여 Knative Serving 서비스의 로그를 찾을 수 있습니다.

3.3.1. OpenShift Logging을 사용하여 Knative Serving으로 배포한 서비스의 로그 찾기

OpenShift Logging을 사용하면 애플리케이션에서 콘솔에 쓰는 로그가 Elasticsearch에 수집됩니다. 다음 절차에서는 Knative Serving을 사용하여 배포한 애플리케이션에 이러한 기능을 적용하는 방법을 간략하게 설명합니다.

사전 요구 사항

- OpenShift CLI(**oc**)를 설치합니다.

프로세스

1. Kibana 경로를 가져옵니다.

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

2. 경로의 URL을 사용하여 Kibana 대시보드로 이동한 후 로그인합니다.

3. 인덱스가 **.all**로 설정되어 있는지 확인합니다. 인덱스가 **.all**로 설정되어 있지 않으면 OpenShift 시스템 로그만 나열됩니다.

4. **knative-serving** 네임스페이스를 사용하여 로그를 필터링합니다. 검색 상자에 서비스 필터를 입력하여 결과를 필터링합니다.

   필터 예제

```
kubernetes.namespace_name:default AND kubernetes.labels.serving_knative_dev\/service:{service_name}
```

   /configuration 또는 **/revision**을 사용하여 필터링할 수도 있습니다.

5. 애플리케이션에서 생성한 로그만 표시하려면 **kubernetes.container_name:<user_container>**를 사용하여 검색 범위를 좁힙니다. 그러지 않으면 큐 프록시의 로그가 표시됩니다.

참고

애플리케이션에서 JSON 기반 구조의 로깅을 사용하면 프로덕션 환경에서 이러한 로그를 빠르게 필터링할 수 있습니다.

4장. 추적

4.1. 요청 추적

분산 추적은 애플리케이션을 구성하는 다양한 서비스를 통해 요청 경로를 기록합니다. 이 기능은 분산 트랜잭션의 전체 이벤트 체인을 이해하기 위해 서로 다른 작업 단위에 대한 정보를 함께 연결하는 데 사용됩니다. 작업 단위는 여러 프로세스 또는 호스트에서 실행될 수 있습니다.

4.1.1. 분산 추적 개요

서비스 소유자로서 분산 추적을 사용하여 서비스 아키텍처에 대한 정보를 수집하도록 서비스를 조정할 수 있습니다. 최신 클라우드 네이티브, 마이크로서비스 기반 애플리케이션의 구성 요소 간 상호 작용, 모니터링, 네트워크 프로파일링 및 문제 해결에 분산 추적을 사용할 수 있습니다.

분산 추적을 사용하면 다음 기능을 수행할 수 있습니다.

- 분산 트랜잭션 모니터링
- 성능 및 대기 시간 최적화
- 근본 원인 분석 수행

Red Hat OpenShift 분산 추적은 다음 두 가지 주요 구성 요소로 구성됩니다.

- **Red Hat OpenShift 분산 추적 platform**- 이 구성 요소는 오픈 소스 Jaeger 프로젝트를 기반으로 합니다.
- **Red Hat OpenShift 분산 추적 data collection**- 이 구성 요소는 오픈 소스 OpenTelemetry 프로젝트를 기반으로 합니다.

이러한 두 구성 요소는 벤더 중립 OpenTracing API 및 계측을 기반으로 합니다.

4.1.2. OpenShift Container Platform에 대한 추가 리소스

- Red Hat OpenShift 분산 추적 아키텍처
- 분산 추적 설치 중

4.2. RED HAT OPENSHIFT DISTRIBUTED TRACING 사용

Red Hat OpenShift distributed tracing을 OpenShift Serverless와 함께 사용하여 서버리스 애플리케이션을 모니터링하고 문제를 해결할 수 있습니다.

4.2.1. Red Hat OpenShift distributed tracing을 사용하여 분산 추적 활성화

Red Hat OpenShift 분석 추적은 추적 데이터를 수집, 저장 및 표시하기 위해 함께 작동하는 여러 구성 요소로 구성됩니다.

사전 요구 사항

- 클러스터 관리자 액세스 권한이 있는 OpenShift Container Platform 계정에 액세스할 수 있습니다.

- OpenShift Serverless Operator, Knative Serving 및 Knative Eventing을 아직 설치하지 않았습니다. 이는 Red Hat OpenShift distributed tracing 설치 후 설치해야 합니다.
- OpenShift Container Platform "Distributed tracing 설치" 문서에 따라 Red Hat OpenShift distributed tracing을 설치했습니다.
- OpenShift CLI(**oc**)가 설치되어 있습니다.
- 프로젝트를 생성했거나 OpenShift Container Platform에서 애플리케이션 및 기타 워크로드를 생성하는 데 적절한 역할 및 권한이 있는 프로젝트에 액세스할 수 있습니다.

프로세스

1. **OpenTelemetryCollector** CR(사용자 정의 리소스)을 생성합니다.

   OpenTelemetryCollector CR의 예

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: cluster-collector
  namespace: <namespace>
spec:
  mode: deployment
  config: |
    receivers:
      zipkin:
    processors:
    exporters:
      jaeger:
        endpoint: jaeger-all-in-one-inmemory-collector-headless.tracing-system.svc:14250
        tls:
          ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
      logging:
    service:
      pipelines:
        traces:
          receivers: [zipkin]
          processors: []
          exporters: [jaeger, logging]
```

2. Red Hat OpenShift distributed tracing이 설치된 네임스페이스에서 두 개의 포드가 실행되고 있는지 확인합니다.

```
$ oc get pods -n <namespace>
```

   출력 예

```
NAME                                         READY   STATUS    RESTARTS   AGE
cluster-collector-collector-85c766b5c-b5g99   1/1     Running   0          5m56s
jaeger-all-in-one-inmemory-ccbc9df4b-ndkl5    2/2     Running   0          15m
```

3. 다음 헤드리스 서비스가 생성되었는지 확인합니다.

```
$ oc get svc -n <namespace> | grep headless
```

-

출력 예

```
cluster-collector-collector-headless          ClusterIP   None          <none>        9411/TCP
7m28s
jaeger-all-in-one-inmemory-collector-headless   ClusterIP   None          <none>
9411/TCP,14250/TCP,14267/TCP,14268/TCP   16m
```

이러한 서비스는 Jaeger, Knative Serving 및 Knative Eventing을 구성하는 데 사용됩니다. Jaeger 서비스의 이름은 다를 수 있습니다.

4. "OpenShift Serverless Operator 설치" 설명서에 따라 OpenShift Serverless Operator를 설치합니다.

5. 다음 KnativeServing CR을 생성하여 **Knative Serving** 을 설치합니다.

KnativeServing CR의 예

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
    name: knative-serving
    namespace: knative-serving
spec:
  config:
    tracing:
      backend: "zipkin"
      zipkin-endpoint: "http://cluster-collector-collector-headless.tracing-
system.svc:9411/api/v2/spans"
      debug: "false"
      sample-rate: "0.1" 1
```

1 **sample-rate**는 샘플링 가능성을 정의합니다. **sample-rate: "0.1"** 을 사용하면 10개의 추적 중 1개가 샘플링됩니다.

6. 다음 **KnativeEventing** CR을 생성하여 Knative Eventing을 설치합니다.

KnativeEventing CR의 예

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
    name: knative-eventing
    namespace: knative-eventing
spec:
  config:
    tracing:
      backend: "zipkin"
      zipkin-endpoint: "http://cluster-collector-collector-headless.tracing-
system.svc:9411/api/v2/spans"
      debug: "false"
      sample-rate: "0.1" 1
```

1 **sample-rate**는 샘플링 가능성을 정의합니다. **sample-rate: "0.1"** 을 사용하면 10개의 추적 중 1개가 샘플링됩니다.

7. Knative 서비스를 생성합니다.

서비스의 예

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
      annotations:
        autoscaling.knative.dev/minScale: "1"
        autoscaling.knative.dev/target: "1"
    spec:
      containers:
      - image: quay.io/openshift-knative/helloworld:v1.2
        imagePullPolicy: Always
        resources:
          requests:
            cpu: "200m"
        env:
        - name: TARGET
          value: "Go Sample v1"
```

8. 서비스에 대한 일부 요청을 수행합니다.

HTTPS 요청의 예

```
$ curl https://helloworld-go.example.com
```

9. Jaeger 웹 콘솔의 URL을 가져옵니다.

명령 예

```
$ oc get route jaeger-all-in-one-inmemory  -o jsonpath='{.spec.host}' -n <namespace>
```

이제 Jaeger 콘솔을 사용하여 추적을 검사할 수 있습니다.

4.3. JAEGER 분산 추적 사용

Red Hat OpenShift distributed tracing의 모든 구성 요소를 설치하지 않으려면 OpenShift Serverless와 함께 OpenShift Container Platform에서 분산 추적을 계속 사용할 수 있습니다.

4.3.1. 분산 추적을 활성화하도록 Jaeger 구성

Jaeger를 사용하여 분산 추적을 활성화하려면 Jaeger를 독립 실행형 통합으로 설치하고 구성해야 합니다.

사전 요구 사항

- OpenShift Container Platform에 대한 클러스터 관리자 권한이 있거나 AWS 또는 OpenShift Dedicated의 Red Hat OpenShift Service에 대한 클러스터 또는 전용 관리자 권한이 있습니다.
- OpenShift Serverless Operator, Knative Serving 및 Knative Eventing을 설치했습니다.
- Red Hat OpenShift distributed tracing Platform Operator를 설치했습니다.
- OpenShift CLI(**oc**)가 설치되어 있습니다.
- 프로젝트를 생성하거나 애플리케이션 및 기타 워크로드를 생성할 수 있는 적절한 역할 및 권한이 있는 프로젝트에 액세스할 수 있습니다.

프로세스

1. 다음을 포함하는 **Jaeger** 사용자 정의 리소스 (CR) 파일을 생성하고 적용합니다.

   Jaeger CR

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

2. **KnativeServing** CR을 편집하고 추적에 필요한 YAML 구성을 추가하여 Knative Serving에 대한 추적을 활성화합니다.

   Serving의 YAML 추적 예

   ```
   apiVersion: operator.knative.dev/v1beta1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     config:
       tracing:
         sample-rate: "0.1" 1
         backend: zipkin 2
         zipkin-endpoint: "http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans" 3
         debug: "false" 4
   ```

   1. **sample-rate**는 샘플링 가능성을 정의합니다. **sample-rate: "0.1"** 을 사용하면 10개의 추적 중 1개가 샘플링됩니다.
   2. **backend**를 **zipkin**으로 설정해야 합니다.
   3. **zipkin-endpoint**는 **jaeger-collector** 서비스 끝점을 가리켜야 합니다. 이 끝점을 가져오려면 Jaeger CR이 적용되는 네임스페이스를 대체합니다.
   4. 디버깅을 **false**로 설정해야 합니다. **debug: "true"**를 설정하여 디버그 모드를 활성화하면 모든 범위가 서버에 전송되어 샘플링 단계를 건너뜁니다.

3. **KnativeEventing** CR을 편집하여 Knative Eventing의 추적을 활성화합니다.

Eventing의 YAML 추적 예

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config:
    tracing:
      sample-rate: "0.1" 1
      backend: zipkin 2
      zipkin-endpoint: "http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans" 3
      debug: "false" 4
```

1. **sample-rate**는 샘플링 가능성을 정의합니다. **sample-rate: "0.1"** 을 사용하면 10개의 추적 중 1개가 샘플링됩니다.
2. **backend** 를 **zipkin** 으로 설정합니다.
3. **zipkin-endpoint** 를 **jaeger-collector** 서비스 끝점을 지정합니다. 이 끝점을 가져오려면 Jaeger CR이 적용되는 네임스페이스를 대체합니다.
4. 디버깅을 **false**로 설정해야 합니다. **debug: "true"**를 설정하여 디버그 모드를 활성화하면 모든 범위가 서버에 전송되어 샘플링 단계를 건너뜁니다.

검증

jaeger 경로를 사용하여 Jaeger 웹 콘솔에 액세스하여 추적 데이터를 볼 수 있습니다.

1. 다음 명령을 입력하여 **jaeger** 경로의 호스트 이름을 가져옵니다.

```
$ oc get route jaeger -n default
```

출력 예

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

2. 콘솔을 보려면 브라우저에서 끝점 주소를 엽니다.