

OpenShift Serverless Deep Dive

Ayush Garg

OpenShift Container Platform Team



What we'll
discuss today

What is serverless?

Knative Components

Knative Serving

Why is serverless
important

Installation

Autoscale, HPA, HA

OpenShift Serverless

Serverless Flow

Traffic Splitting and
Blue-Green Deployment

Knative

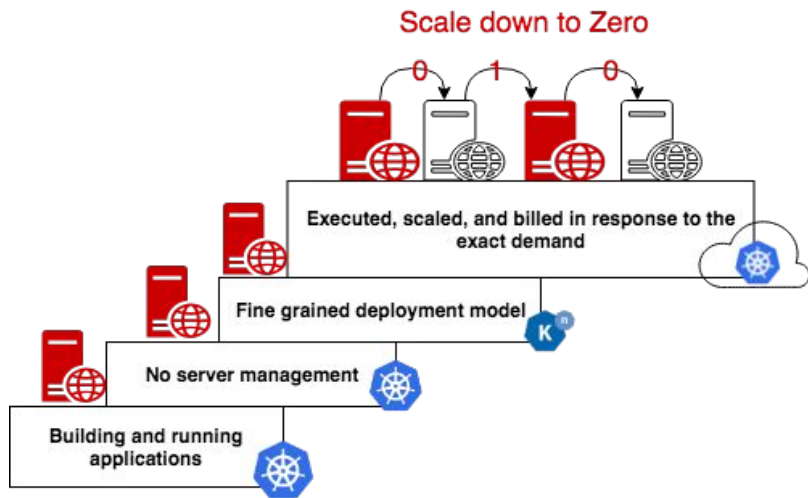
Kourier (Ingress)

Knative Eventing

What is serverless?

Serverless is a deployment model that allows you to build and run applications without requiring deep insight into the underlying infrastructure. The idea is that the platform is ubiquitous and simply works. Developers can focus on writing code and determining where it needs to run without worrying about the infrastructure.

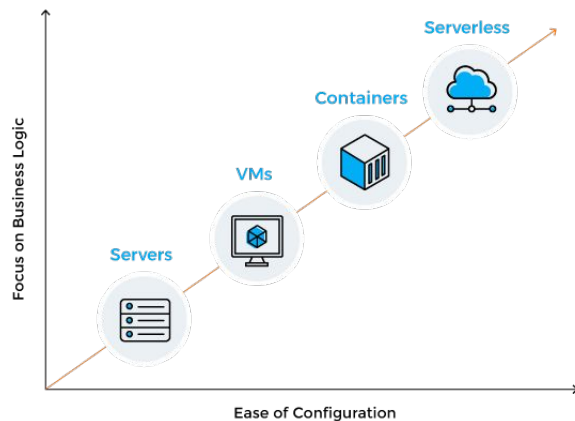
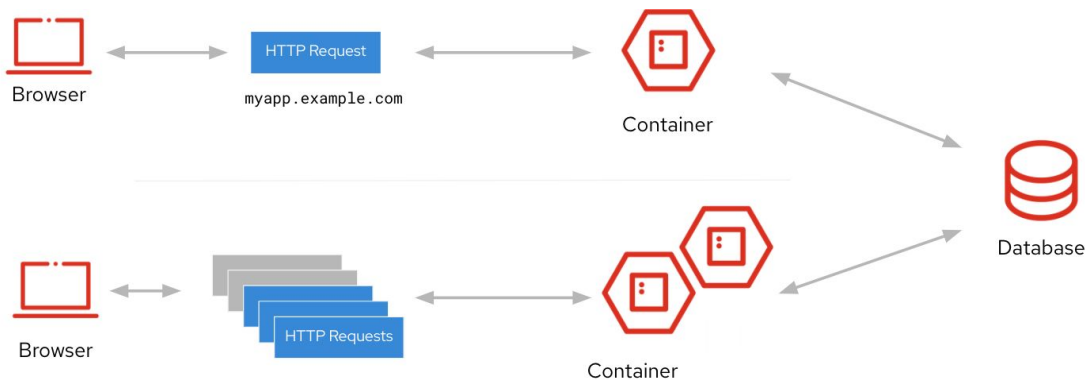
An event-driven serverless deployment makes it possible to run code and provision infrastructure only when necessary. That allows the application to be idle when it isn't needed. A serverless application will automatically scale up based on event-triggers in response to incoming demand, and is then able to scale to zero after use.



Why is serverless important?

The serverless model further unlocks the innovative power of Red Hat OpenShift. Serverless helps organizations innovate faster because the application is abstracted from the underlying infrastructure. Applications are packaged as OCI compliant containers that can be run anywhere, regardless of how they are written.

You can use any of these event triggers to run the application on demand. This structure makes it possible to deconstruct your monolithic application into individual containers, and let the application logic trigger each container, using incoming events to determine when to launch your application.



V0000000

OpenShift Serverless

Developers can use Red Hat OpenShift Serverless to build, deploy and run event-driven applications that will start based on an event trigger, scale up resources as needed, then scale to zero after resource burst. With the power of Knative, Red Hat OpenShift Serverless applications can run anywhere Red Hat OpenShift is installed - on-premises, across multiple public cloud locations, or at the edge - all using the same interface.

OpenShift Serverless is based on the Knative project and supports almost any containerized application as it is designed to utilize many of the baseline features of OpenShift. Beyond auto-scaling for HTTP requests, you can trigger serverless containers from a variety of event sources and receive events such as Kafka messages, file upload to storage, timers for recurring jobs, and 100+ event sources like Salesforce, ServiceNow, email, etc, and is powered by Camel-K.

- deploying new application features or revisions
- performing canary
- A/B or blue-green testing with gradual traffic rollout

Knative

Knative is a Kubernetes-based platform to deploy and manage modern serverless workloads.

- Started as an **Open Source** Project mid-2018 by Google
- Community driven with a lot of vendor backing
 - <https://knative.dev/>
 - <https://github.com/knative>
 - <https://slack.knative.dev/>
 - Backed by Google, **Red Hat**, IBM and more
- Use Knative on **OpenShift**

<https://docs.openshift.com/container-platform/4.6/serverless/serverless-getting-started.html>

Knative Components

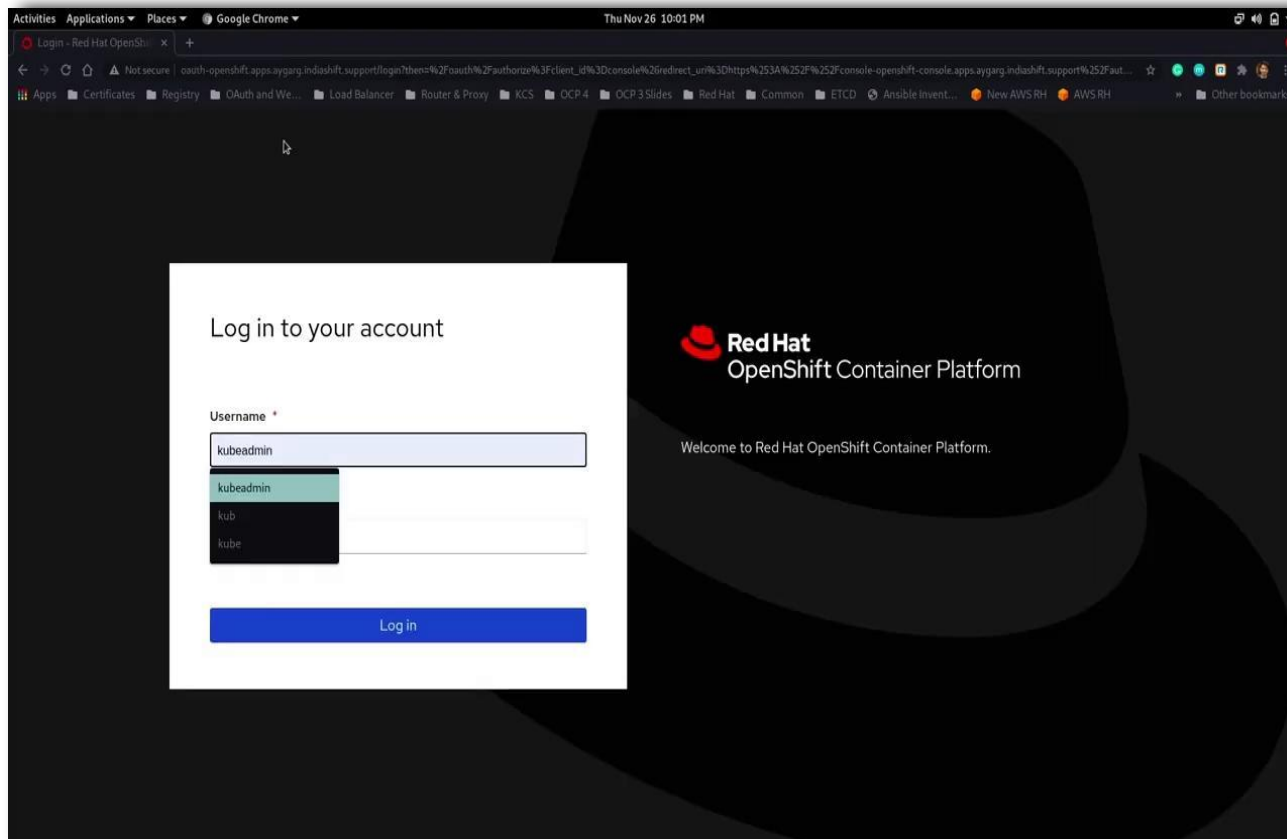
Serving

Run serverless containers on Kubernetes with ease, Knative takes care of the details of networking, autoscaling (even to zero), and revision tracking. You just have to focus on your core logic.

Eventing

Universal subscription, delivery, and management of events. Build modern apps by attaching compute to a data stream with declarative event connectivity and developer-friendly object model.

Installing OpenShift Serverless

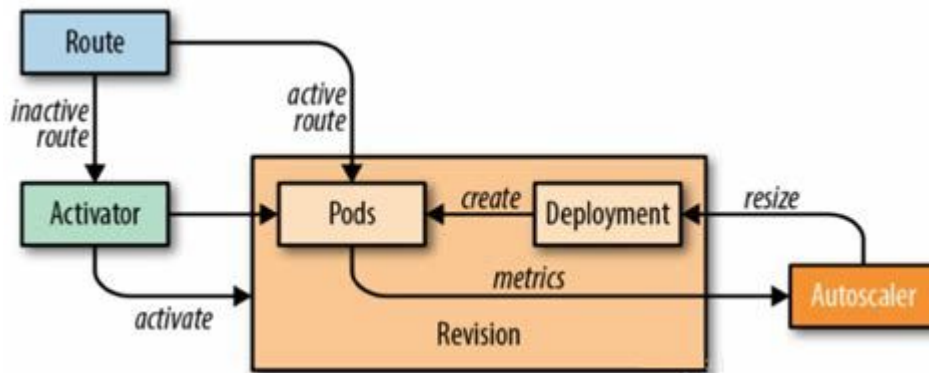


OpenShift Serverless Flow

After a defined time of idleness (called the stable-window) a revision is considered inactive, which causes a few things to happen. First off, all routes pointing to the now inactive revision will be pointed to the **activator**.

Its primary responsibilities are to receive and buffer requests for revisions that are inactive as well as report metrics to the autoscaler.

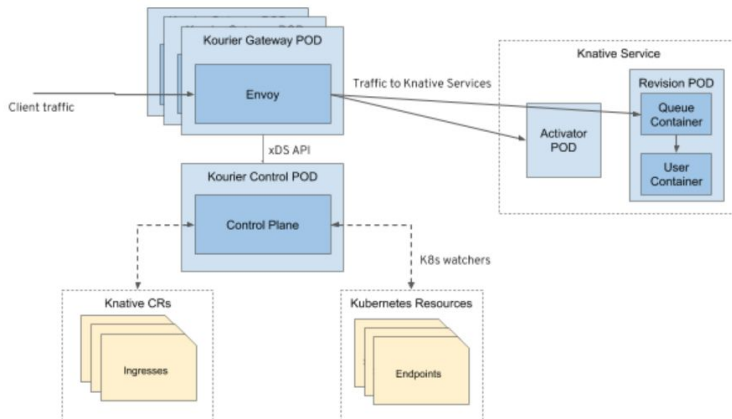
If we try to access the service while it is scaled to zero the activator will pick up the request(s) and buffer them until the Autoscaler is able to quickly create pods for the given revision.



Kourier: A lightweight Knative Serving ingress

Until recently, Knative Serving used Istio as its default networking component for handling external cluster traffic and service-to-service communication. Istio is a great service mesh solution, but it can add unwanted complexity and resource use to your cluster if you don't need it. That's why the Kourier: To simplify the ingress side of Knative Serving.

- **The Kourier gateway** is Envoy running with a base bootstrap configuration that connects back to the Kourier control plane.
- **The Kourier control plane** handles Knative ingress objects and keeps the Envoy configuration up to date.



When a new service is deployed in Knative Serving, it creates an Ingress object that contains information about how the service should be exposed.

Kourier subscribes to changes in ingresses that are managed by Knative Serving. Kourier is notified every time an ingress is created, deleted, or modified. When that happens, Kourier analyzes the information in the ingress and transforms the information into objects in an Envoy configuration.

```
bash
aygarg @ ayush-garg in ~ [06:34 PM]
$ oc get pod -n knative-serving-ingress
NAME                                READY   STATUS    RESTARTS   AGE
3scale-kourier-control-685df54668-8vhs2 1/1     Running   0           41m
3scale-kourier-control-685df54668-h9tcq 1/1     Running   0           41m
3scale-kourier-gateway-7d49fc7d88-5rtk8 1/1     Running   0           3d20h
3scale-kourier-gateway-7d49fc7d88-jjqdw 1/1     Running   0           3d20h
aygarg @ ayush-garg in ~ [06:34 PM]
$
```

```
bash
aygarg @ ayush-garg in ~ [06:42 PM]
$ oc get cm/config-network -o yaml -n knative-serving | grep "ingress.class: kourier"
  ingress.class: kourier.ingress.networking.knative.dev
aygarg @ ayush-garg in ~ [06:42 PM]
$
```

Knative Serving Architecture

Knative Serving on OpenShift Container Platform enables developers to write [cloud-native applications](#) using [serverless architecture](#). Serverless is a cloud computing model where application developers don't need to provision servers or manage scaling for their applications. These routine tasks are abstracted away by the platform, allowing developers to push code to production much faster than in traditional models.

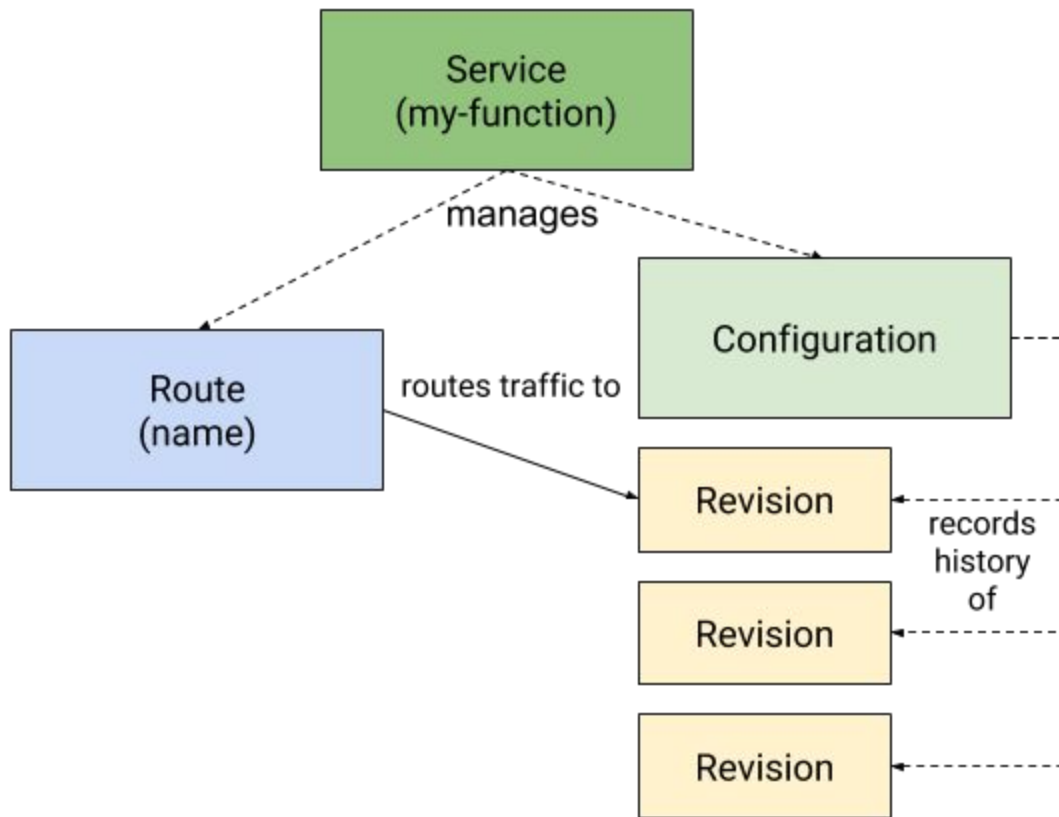
Knative Serving supports deploying and managing cloud-native applications by providing a set of objects as Kubernetes Custom Resource Definitions (CRDs) that define and control the behavior of serverless workloads on an OpenShift Container Platform cluster. For more information about CRDs, see [Extending the Kubernetes API with Custom Resource Definitions](#).

- Rapid deployment of serverless containers
- [Automatic scaling up and down to zero](#)
- Routing and network programming for Istio components
- Point-in-time snapshots of deployed code and configurations

Knative Serving CRDs

- **Service:** The `service.serving.knative.dev` CRD automatically manages the life cycle of your workload to ensure that the application is deployed and reachable through the network. It creates a Route, a Configuration, and a new Revision for each change to a user created Service, or custom resource. Most developer interactions in Knative are carried out by modifying Services.
- **Revision:** The `revision.serving.knative.dev` CRD is a point-in-time snapshot of the code and configuration for each modification made to the workload. Revisions are immutable objects and can be retained for as long as necessary.
- **Route:** The `route.serving.knative.dev` CRD maps a network endpoint to one or more Revisions. You can manage the traffic in several ways, including fractional traffic and named routes.
- **Configuration:** The `configuration.serving.knative.dev` CRD maintains the desired state for your deployment. It provides a clean separation between code and configuration. Modifying a configuration creates a new Revision.

```
bash
aygarg @ ayush-garg in ~ [08:09 AM]
$ oc api-resources --api-group serving.knative.dev
NAME             SHORTNAMES  APIGROUP          NAMESPACED  KIND
configurations   config,cfg  serving.knative.dev true          Configuration
revisions        rev        serving.knative.dev true          Revision
routes           rt         serving.knative.dev true          Route
services         kservice,ksvc serving.knative.dev true          Service
aygarg @ ayush-garg in ~ [08:10 AM]
$
```



Hello-Serverless App

```

aygarg @ ayush-garg in ~ [09:40 PM]
$ oc get all
NAME                                     READY   STATUS    RESTARTS   AGE
pod/hello-22xt4-deployment-f7d6ff884-hq2xf 2/2     Terminating    0           71s

NAME                                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)
service/hello                       AGE          ExternalName    <none>            kourier-internal.knative-serving-ingress.svc.cluster.local <none>
service/hello-22xt4                 67s         ClusterIP       172.30.104.116   <none>            80/TCP
service/hello-22xt4-private         74s         ClusterIP       172.30.91.56    <none>            80/TCP,9091/TCP,8022/TCP
deployment.apps/hello-22xt4-deployment 0/0         0               0               76s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/hello-22xt4-deployment-f7d6ff884 0         0         0       79s

NAME                                URL                                         LATESTCREATED   LATESTREADY   READY   REASON
service.serving.knative.dev/hello  http://hello-test.apps.aygarg.indiashift.support hello-22xt4      hello-22xt4    True

NAME                                URL                                         READY   REASON
route.serving.knative.dev/hello  http://hello-test.apps.aygarg.indiashift.support True

NAME                                CONFIG NAME   K8S SERVICE NAME   GENERATION   READY   REASON
revision.serving.knative.dev/hello-22xt4 hello         hello-22xt4        1            True

NAME                                LATESTCREATED   LATESTREADY   READY   REASON
configuration.serving.knative.dev/hello hello-22xt4      hello-22xt4    True
aygarg @ ayush-garg in ~ [09:41 PM]
$

```

Knative Service Ingress Object

```
Activities Applications Places Terminal Mon Nov 30 6:49 PM
Terminal
  Knative-Serving-Namespace: test
  Knative-Serving-Revision: hello-5tfgf
  percent: 100
  serviceName: hello-5tfgf
  serviceNamespace: test
  servicePort: 80
  timeout: 48h0m0s
  visibility: ClusterLocal
- hosts:
- hello-test.apps.aygarg.indiashift.support
  http:
    paths:
      - splits:
        - appendHeaders:
            Knative-Serving-Namespace: test
            Knative-Serving-Revision: hello-5tfgf
            percent: 100
            serviceName: hello-5tfgf
            serviceNamespace: test
            servicePort: 80
            timeout: 48h0m0s
            visibility: ExternalIP
            visibility: ExternalIP
status:
  conditions:
    - lastTransitionTime: "2020-11-30T05:36:55Z"
      status: "True"
      type: LoadBalancerReady
    - lastTransitionTime: "2020-11-30T05:36:55Z"
      status: "True"
      type: NetworkConfigured
    - lastTransitionTime: "2020-11-30T05:36:55Z"
      status: "True"
      type: Ready
  loadBalancer:
```


Autoscaling

One of the main features of Knative is automatic scaling of replicas for an application to closely match incoming demand, including scaling applications to zero if no traffic is being received. Knative Serving enables this by default, using the Knative Pod Autoscaler (KPA). The Autoscaler component watches traffic flow to the application, and scales replicas up or down based on configured metrics.

Knative Pod Autoscaler (KPA)

- Part of the Knative Serving core and enabled by default once Knative Serving is installed.
- Supports scale to zero functionality.
- Does not support CPU-based autoscaling.

Horizontal Pod Autoscaler (HPA)

- Does not support scale to zero functionality.
- Supports CPU-based autoscaling.

Name ↑	Status ↓	Ready ↓	Restarts ↓	Owner ↓	Memory ↓	CPU ↓	Created ↓	
 autoscaler-6c64cc4b98-6xhs2	 Running	1/1	0	 autoscaler-6c64cc4b98	39.9 MiB	0.001 cores	 Nov 26, 10:03 pm	
 autoscaler-hpa-89698bf4d-nrfk8	 Running	1/1	0	 autoscaler-hpa-89698bf4d	26.3 MiB	0.000 cores	 Nov 26, 10:03 pm	
 autoscaler-hpa-89698bf4d-tm9mg	 Running	1/1	0	 autoscaler-hpa-89698bf4d	29.2 MiB	0.001 cores	 Nov 26, 10:03 pm	

V0000000

Configuring the Autoscaler implementation

The type of Autoscaler implementation (KPA or HPA) can be configured by using the class annotation.

- **Global settings key:** `pod-autoscaler-class`
- **Per-revision annotation key:** `autoscaling.knative.dev/class`
- **Possible values:** `"kpa.autoscaling.knative.dev"` or `"hpa.autoscaling.knative.dev"`
- **Default:** `"kpa.autoscaling.knative.dev"`

Global (ConfigMaP)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-autoscaler
  namespace: knative-serving
data:
  pod-autoscaler-class: "kpa.autoscaling.knative.dev"
```

Per Revision

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/class: "kpa.autoscaling.knative.dev"
    spec:
      containers:
        - image: gcr.io/knative-samples/helloworld-go
```

Metrics

The metric configuration defines which metric type is watched by the Autoscaler. For per-revision configuration, this is determined using the `autoscaling.knative.dev/metric` annotation. The possible metric types that can be configured per revision depend on the type of Autoscaler implementation you are using:

- The default KPA Autoscaler supports the `concurrency` and `rps` metrics.
- The HPA Autoscaler supports the `concurrency`, `rps` and `cpu` metrics.
- **Per-revision annotation key:** `autoscaling.knative.dev/metric`
- **Possible values:** `"concurrency"`, `"rps"` or `"cpu"`, depending on your Autoscaler type. The `cpu` metric is only supported on revisions with the HPA class.
- **Default:** `"concurrency"`

Per-revision concurrency configuration

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/metric: "concurrency"
```

Per-revision cpu

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/metric: "cpu"
```

Autoscaler ConfigMap, Min-Max Pod, HPA and HA

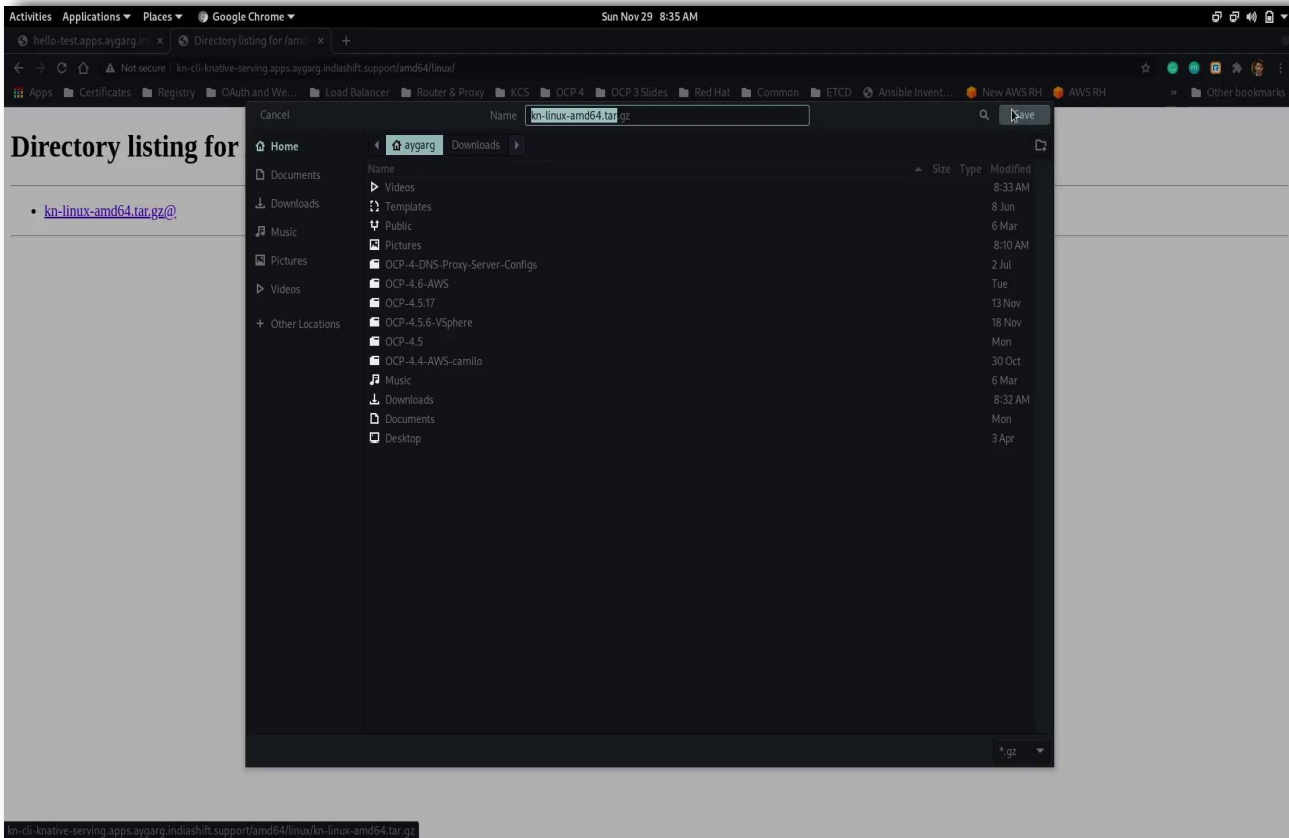
```

Activities Applications Places Terminal
Fri Nov 27 3:49 PM
root@ayush-garg-var/lib/snapd/snap/hyef1/bin

tion).\n# For legacy and backwards compatibility reasons, this value also accepts\n# fraction values in 10, 1 interval (i.e. 0.1 = 10 %).\n# Thus minimal percentage value must be greater than 1.0, or it will be\n# treated as a fraction.\n# NOTE: that this value does not affect actual number of concurrent requests\n# ..... the user container may receive, but only the average number of requests\n# ..... that the revision pods will receive.\ncontainer-concurrency-target-percentage: \"70\"\n\n# The container concurrency target default is what the Autoscaler will\n# try to maintain when concurrency is used as the scaling metric for the\n# Revision and the Revision specifies unlimited concurrency.\n# When revision explicitly specifies container concurrency, that value\n# will be used as a scaling target for a autoscaler.\n# When specifying unlimited concurrency, the autoscaler will\n# horizontally scale the application based on this target concurrency.\n# This is what we call \"soft limit\" in the documentation, i.e. it only\n# affects number of pods and does not affect the number of requests\n# individual pod processes.\n# The value must be a positive number such that the value multiplied\n# by container-concurrency-target-percentage is greater than 0.01.\n# NOTE: that this value will be adjusted by application of\n# ..... container-concurrency-target-percentage, i.e. by default\n# ..... the system will target on average 70 concurrent requests\n# ..... per revision pod.\n# NOTE: Only one metric can be used for autoscaling a Revision.\ncontainer-concurrency-target-default: \"100\"\n\n# The requests per second (RPS) target default is what the Autoscaler will\n# try to maintain when RPS is used as the scaling metric for a Revision and\n# the Revision specifies unlimited RPS. Even when specifying unlimited RPS,\n# the autoscaler will horizontally scale the application based on this\n# target RPS.\n# Must be greater than 1.0.\n# NOTE: Only one metric can be used for autoscaling a Revision.\nrequests-per-second-target-default: \"200\"\n\n# The target burst capacity specifies the size of burst in concurrent\n# requests that the system operator expects the system will receive.\n# Autoscaler will try to protect the system from queuing by introducing\n# Activator in the request path if the current spare capacity of the\n# service is less than this setting.\n# If this setting is 0, then Activator will be in the request path only\n# when the revision is scaled to 0.\n# If this setting is 0.003e-0 and container-concurrency-target-percentage is\n# 100% or 1.0, then activator will always be in the request path.\n# -1 denotes unlimited target-burst-capacity and activator will always\n# be in the request path.\n# Other negative values are invalid.\ntarget-burst-capacity: \"200\"\n\n# When operating in a stable mode, the autoscaler operates on the\n# average concurrency over the stable window.\n# Stable window must be in whole seconds.\nstable-window: \"60s\"\n\n# When observed average concurrency during the panic window reaches\n# panic-threshold-percentage the target concurrency, the autoscaler\n# enters panic mode. When operating in panic mode, the autoscaler\n# scales on the average concurrency over the panic window which is\n# panic-window-percentage of the stable-window.\n# Must be in the [1, 100] range.\n# When computing the panic window it will be rounded to the closest\n# whole second, at least 1s.\npanic-window-percentage: \"10.0\"\n\n# The percentage of the container concurrency target at which to\n# enter panic mode when reached within the panic window.\npanic-threshold-percentage: \"200.0\"\n\n# Max scale up rate limits the rate at which the autoscaler will\n# increase pod count. It is the maximum ratio of desired pods versus\n# observed pods.\n# Cannot be less or equal to 1,\n# i.e. with value of 2.0 the number of pods can at most go N to 2N\n# over single Autoscaler period (2s), but at least N to N+1,\n# if Autoscaler needs to scale up.\nmax-scale-up-rate: \"1000.0\"\n\n# Max scale down rate limits the rate at which the autoscaler will\n# decrease pod count. It is the maximum ratio of observed pods versus\n# desired pods.\n# Cannot be less or equal to 1,\n# i.e. with value of 2.0 the number of pods can at most go N to N/2\n# over single Autoscaler evaluation period (2s), but at least N to N-1,\n# if Autoscaler needs to scale down.\nmax-scale-down-rate: \"2.0\"\n\n# Scale to zero feature flag.\nenable-scale-to-zero: \"true\"\n\n# Scale to zero grace period is the time an inactive revision is left\n# running before it is scaled to zero (min: 6s).\n# This is the upper limit and is provided not to enforce timeout after\n# the revision stopped receiving requests for stable window, but

```

Assigning Tag Revisions and Traffic Splitting



Setting up a custom domain

By default, Knative services have a fixed domain format:

→ <application_name>-<namespace>.<openshift_cluster_domain>

To change the default domain or add a new one, the “config-domain” CM needs to be modified.

→ \$ oc edit cm config-domain --namespace knative-serving

```
apiVersion: v1
data:
  apps.aygarg.indiashift.support: ""
  custom.aygarg.indiashift.support: |
    selector:
      custom: domain
```

```
1  apiVersion: serving.knative.dev/v1
2  kind: Service
3  metadata:
4    name: hello
5    namespace: test
6    labels:
7      custom: domain
8  spec:
9    template:
10     spec:
11       containers:
12         - image: docker.io/openshift/hello-openshift
13           env:
14             - name: RESPONSE
15               value: "Hello Serverless!"
```


Kourier Ingress LB

An ingress LB (loadbalancer type service) gets created for kourier inside the “knative-serving-ingress” namespace when the OpenShift Serverless operator is installed. However, the DNS records are not created for it and only “http” based traffic is accepted over it as no TLS certificates are there by default. Since the DNS records are not created for this LB, all the routes for serverless applications are served by the default OpenShift IngressController which sends the request to HAProxy pods first then to “kourier-gateway” pod.

Already deployed knative services can be accessed through that LB using the following curl command:

```
bash
aygarg @ ayush-garg in ~ [06:01 PM]
$ curl -H "Host: hello-test.apps.aygarg.indiashift.support" http://a090e7d10ea58438abd702454b5b7a09-383542004.us-east-1.elb.amazonaws.com
Hello Serverless!
aygarg @ ayush-garg in ~ [06:06 PM]
$
```

```
bash
$ oc get svc -n knative-serving-ingress
NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kourier                            LoadBalancer        172.30.160.126   a090e7d10ea58438abd702454b5b7a09-383542004.us-east-1.elb.amazonaws.com  80:30689/TCP,443:30210/TCP  18h
kourier-control                    ClusterIP            172.30.18.104    <none>            18000/TCP        18h
kourier-internal                   ClusterIP            172.30.40.51     <none>            80/TCP           18h
aygarg @ ayush-garg in ~ [06:01 PM]
$
```

Since the DNS records are not created for the kourier ingress LB and routes are served by default IngressController then the default HAProxy router pods become critical otherwise if the default OpenShift ingress goes down then the serverless routes won't be accessible.

As per the requirement, the DNS records can be created pointing to the new kourier ingress LB with respect to the cloud provider on which cluster is deployed. A wild-card DNS record can also be created pointing to the kourier ingress LB as the requests made by the routes to the LB contains the proper hostname in the host-header of the packet.

If the DNS record is created then the routes will be accessible over only the “http” protocol as there is no TLS certificate configured for the kourier ingress LB.

Configure TLS Certificate for Kourier Ingress

```
Activities Applications Places Terminal
Sun Nov 29 9:31 PM
Terminal
$ openssl x509 -in ingress-cert/ayush.crt -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      46:3e:f3:40:33:04:9a:d1:d2:db:3a:46:b6:91:32:d1:93:29:14:56
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: CN = customCA
    Validity
      Not Before: Nov 29 13:59:12 2020 GMT
      Not After : Nov 29 13:59:12 2021 GMT
    Subject: CN = *.apps.aygarg.indiashift.support
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (2048 bit)
      Modulus:
        00:df:81:dd:8d:75:da:58:36:d6:bb:38:1f:e6:55:
        3c:61:0a:a0:03:f0:80:61:96:3c:7a:07:df:29:06:
        3a:6a:b1:68:74:98:af:94:d5:3b:5f:e2:f6:d0:f4:
        86:84:14:1b:d0:3a:88:1f:b7:96:4c:d2:5d:4e:a5:
        6d:47:c3:cc:31:c8:d4:c0:93:85:82:f6:98:8e:de:
        ff:94:2d:a3:15:d4:82:42:51:d8:aa:43:f9:a5:79:
        17:ca:d4:0b:7c:5d:fb:73:fe:63:cb:a3:30:31:49:
        95:3e:77:1c:89:00:85:89:1e:3a:80:98:85:80:83:
        3f:0d:c1:77:e1:c4:e9:ae:fd:8d:81:32:78:30:cd:
        70:f2:dd:56:1f:bc:c5:e4:ff:17:60:34:39:02:8f:
        ba:3f:01:26:9e:4c:8f:bb:e2:7b:cc:ee:cd:ac:34:
        85:13:3e:1d:79:25:b5:58:6a:45:92:99:3f:5c:6c:
        8f:a6:5d:9c:a4:0a:c7:eb:75:44:63:ff:12:b8:6d:
        8c:4a:b6:d0:ae:4a:7e:81:ad:d3:7a:9d:a9:2c:f5:
        91:bc:3b:19:f9:b0:4b:bd:c3:b2:5e:34:01:76:54:
        d9:4f:a6:2e:bf:20:34:64:2b:72:c9:cb:2b:8a:80:
        b5:76:41:05:84:e5:c3:27:ec:cc:68:ee:74:5f:5d:
```

Knative Eventing

Knative Eventing on OpenShift Container Platform enables developers to use an event-driven architecture with serverless applications. An event-driven architecture is based on the concept of decoupled relationships between event producers that create events, and event sinks, or consumers, that receive them.

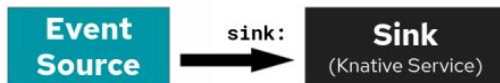
Knative Eventing uses standard HTTP POST requests to send and receive events between event producers and consumers. These events conform to the CloudEvents specifications, which enables creating, parsing, sending, and receiving events in any programming language.

OpenShift Serverless provides several mechanisms for building event-driven applications:

- Direct connections
- Channels and subscriptions
- Event filtering with brokers and triggers

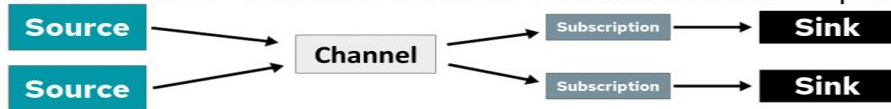
Direct connections

Event Source → Knative Service : Direct Connection



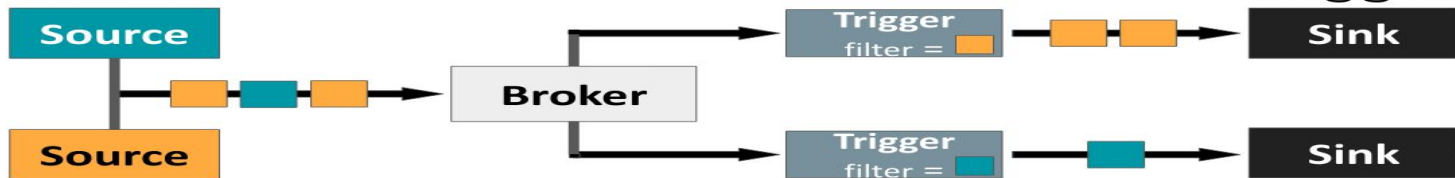
Channels and subscriptions

Event Source → Knative Service : Channel & Subscription



Event brokers and triggers

Event Source → Knative Service: Broker & Trigger



Event Sources

An event source is an object that links an event producer with an event sink, or consumer. A sink can be a Knative service, channel, or broker that receives events from an event source.

- **ApiServerSource**
Connects a sink to the Kubernetes API server.
- **PingSource**
Periodically sends ping events with a constant payload. It can be used as a timer.
- **SinkBinding**
Allows you to connect core Kubernetes resource objects such as a Deployment, Job, or StatefulSet with a sink.
- **KafkaSource**
Connect a Kafka cluster to a sink as an event source.

PingSource Eventing Example

```

Activities Applications Places Terminal Mon Nov 30 11:46 PM
Terminal
NAME                                READY  STATUS   RESTARTS  AGE
event-display-d2tsv-deployment-67d84ff688-jsnqm 2/2    Running   0          22s
aygarg @ ayush-garg in ~ [11:44 PM]
$ oc get PingSource
NAME      SINK                                AGE  READY  REASON
test-ping-source  http://event-display.test.svc.cluster.local 11s  True
aygarg @ ayush-garg in ~ [11:44 PM]
$ oc logs event-display-d2tsv-deployment-67d84ff688-jsnqm -c user-container
aygarg @ ayush-garg in ~ [11:45 PM]
$ oc logs -f event-display-d2tsv-deployment-67d84ff688-jsnqm -c user-container
rpc error: code = NotFound desc = could not find container "e587181335f003f30fefe862a3592bb34d0fea6b50c59d73b671e2f7d75d457a": container
with ID starting with e587181335f003f30fefe862a3592bb34d0fea6b50c59d73b671e2f7d75d457a not found: ID does not existaygarg @ ayush-garg
in ~ [11:46 PM]
$ oc get pod
NAME                                READY  STATUS   RESTARTS  AGE
event-display-d2tsv-deployment-67d84ff688-c564v 1/2    Running   0          5s
event-display-d2tsv-deployment-67d84ff688-jsnqm 0/2    Terminating 0         2m2s
aygarg @ ayush-garg in ~ [11:46 PM]
$ oc logs event-display-d2tsv-deployment-67d84ff688-c564v -c user-container
cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/test/pingsources/test-ping-source
  id: 33a6a500-14ee-4cd3-907c-4e34f6b72e0a
  time: 2020-11-30T18:16:00.000092805Z
  datacontenttype: application/json
Data,
{
  "message": "Hello world!"
}
aygarg @ ayush-garg in ~ [11:46 PM]
$

```

Thank you

Red Hat is the world's leading provider of
enterprise open source software solutions.
Award-winning support, training, and consulting
services make
Red Hat a trusted adviser to the Fortune 500.

 linkedin.com/company/red-hat

 youtube.com/user/RedHatVideos

 facebook.com/redhatinc

 twitter.com/RedHat