Cloud Native Runtimes for VMware Tanzu 1.3

Cloud Native Runtimes for VMware Tanzu 1.3

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

Cloud Native Runtimes Overview

This topic gives you an overview of Cloud Native Runtimes, commonly known as CNR.

Overview

Cloud Native Runtimes (CNR) is enterprise supported Knative, with the Carvel tools suite for deployment and Contour for networking. CNR offers everything Knative does and some extras that make it ideal for cloud native application development. Cloud Native Runtimes gives developers environmental simplicity and administrators deployment control. Cloud Native Runtimes works on any single Kubernetes cluster running Kubernetes v1.21 and later.

CNR utilizes Knative's main features of Serving and Eventing to provide:

- Automatic pod scaling.
- Traffic splitting by code release version.
- Event-triggered workloads.

Cloud Native Runtimes simplifies the Developer experience.

Kubernetes Developers need to know:	Cloud Native Runtimes Developers need to know:
Pods	Pods
Deployment & Rollout Progress	Knative Service
Service (networking model)	
Ingress	
Labels and selectors	

Cloud Native Runtimes increases Administrator control and support.

Administrators can:
Manage infrastructure costs with request driven autoscaling.
Test deployments with traffic splitting by code version
Use Carvel command tools to simplify deployment.
Enterprise Support when you need it.

Cloud Native Runtimes works well with these use cases:

- Batch Jobs Processing
- AI/ML

- Application or Network Monitoring
- IOT
- Event driven and serverless application architectures.

For more information on the software that makes Cloud Native Runtimes see:

- Knative Documentation Home - Knative
- Carvel Tools Suite Documentation Carvel - Home
- Contour Networking Documentation Contour

Cloud Native Runtimes release notes

This topic contains release notes for Cloud Native Runtimes (CNR) for Tanzu v1.3.

v1.3.0

Release Date: July 12, 2022

New features

New features in this release:

- **Documentation:** The Cloud Native Runtimes documentation is refactored to align with Tanzu Application Platform.
- **Removed config option `local_dns`:** The config options for `local_dns` were removed. They were added when CNR was a standalone offering, but now that Cloud Native Runtimes is part of Tanzu Application Platform, the config options don't work as intended.
- **Knative 1.3:** See Knative Release Notes for serving and eventing.
- **Contour 1.19.1** Included as part of new Knative version
- **Golang Bump:** Built with Golang 1.17.11.

Resolved issues

This release has the following fixes:

- The `rabbitmq-controller-manager` pod is no longer repeatedly restarted.
- Kapp-controller updates `default_tls_secret` in the `config-contour` ConfigMap when needed.
- The **lite** config option in the Cloud Native Runtimes package definition is now on by default when the **provider** config value is set to "local".

Known issues

This release has the following issues:

- Cloud Native Runtimes fails to install when configured with `reuse_crds:true` and no internal and/or external Contour namespace provided on a cluster with Contour installed at a version other than 1.19.1. Tanzu Application Platform 1.2.0 includes Contour 1.18.2. Therefore, it is succeptible to this issue when configured as described. See Installing Cloud Native Runtimes with an Existing Contour Installation for more details.

Components

Cloud Native Runtimes v1.3.0 uses the following component versions:

Release	Details
Version	v1.3.0
Release date	July 12, 2022
Component	**Version**
Knative Serving	1.3.2
Knative Eventing	1.3.2
Knative Eventing RabbitMQ Integration	1.3.1
Knative cert-manager Integration	1.3.0
Knative Serving Contour Integration	1.3.0
VMware Tanzu Sources for Knative	1.3.0
TriggerMesh Sources from Amazon Web Services (SAWS)	1.6.0
vSphere Event Sources	1.3.0

Integrations you can use with Cloud Native Runtimes

This topic tells you the supported integrations for Cloud Native Runtimes. For more information regarding these integrations, see the CNR Administrator Guide.

CNR Integrations

CNR Integration	Version	Documentation
VMware Tanzu Observability	Supported	Configuring Observability for CNR
Avi Vantage	Supported	Configuring CNR with Avi Vantage
Rabbit MQ	Supported	Configuring CNR with RabbitMQ
Tanzu Service Mesh	Supported	Configuring CNR with Tanzu Service Mesh

For tools and software compatibility, please refer to Tanzu Application Platform's requirements.

Install Cloud Native Runtimes

This document describes how you can install Cloud Native Runtimes, commonly known as CNR, from the Tanzu Application Platform package repository.

Note: Use the instructions on this page if you do not want to use a profile to install packages. Both the full and light profiles include Cloud Native Runtimes. For more information about profiles, see Installing the Tanzu Application Platform Package and Profiles.

Prerequisites

Before installing Cloud Native Runtimes:

- Complete all prerequisites to install Tanzu Application Platform. For more information, see Prerequisites.
- Contour is installed in the cluster. Contour can be installed from the Tanzu Application package repository. If you have have an existing Contour installation, see Installing Cloud Native Runtimes with an Existing Contour Installation.

Install

To install Cloud Native Runtimes:

1. List version information for the package by running:

   ```
   tanzu package available list cnrs.tanzu.vmware.com --namespace tap-install
   ```

   For example:

   ```
   $ tanzu package available list cnrs.tanzu.vmware.com --namespace tap-install
   - Retrieving package versions for cnrs.tanzu.vmware.com...
     NAME                   VERSION  RELEASED-AT
     cnrs.tanzu.vmware.com  1.3.0    2022-07-12 00:00:00 -0800 PST
   ```

2. (Optional) Make changes to the default installation settings:

   1. Gather values schema.

      ```
      tanzu package available get cnrs.tanzu.vmware.com/1.3.0 --values-schema -
      n tap-install
      ```

      For example:

      ```
      $ tanzu package available get cnrs.tanzu.vmware.com/1.3.0 --values-schema
       -n tap-install
      ```

```
  Retrieving package details for cnrs.tanzu.vmware.com/1.3.0...
  KEY                           DEFAULT  TYPE             DESCRIPTION
  nodeport.enable               false                               boole
an  Optional: Set to "true" to change envoy Service in contour-external f
rom Type LoadBalancer to Type NodePort. On by default when "provider" is
set to "local".
  provider                      <nil>                               strin
g   Optional: Kubernetes cluster provider. To be specified if deploying C
NR on TKGs or on a local Kubernetes cluster provider.
  default_tls_secret            <nil>                               strin
g   Optional: Overrides the config-contour configmap in namespace knative
-serving.
  domain_template               {{.Name}}.{{.Namespace}}.{{.Domain}}  strin
g   Optional: specifies the golang text template string to use when const
ructing the Knative service's DNS name.
  lite.enable                   false                               boole
an  Optional: Not recommended for production. Set to "true" to reduce CPU
 and Memory resource requests for all CNR Deployments, Daemonsets, and St
atefulsets by half. On by default when "provider" is set to "local".
  pdb.enable                    true                                boole
an  Optional: Set to true to enable Pod Disruption Budget. If provider lo
cal is set to "local", the PDB will be disabled automatically.
  domain_config                 <nil>                               objec
t   Optional: Overrides the config-domain configmap in namespace knative-
serving. Must be valid YAML.
  domain_name                   <nil>                               strin
g   Optional: Default domain name for Knative Services.
  ingress.external.namespace    <nil>                               strin
g   Optional: Only valid if a Contour instance already present in the clu
ster. Specify a namespace where an existing Contour is installed on your
cluster (for external services) if you want CNR to use your Contour insta
nce.
  ingress.internal.namespace    <nil>                               strin
g   Optional: Only valid if a Contour instance already present in the clu
ster. Specify a namespace where an existing Contour is installed on your
cluster (for internal services) if you want CNR to use your Contour insta
nce.
  ingress.reuse_crds            false                               boole
an  Optional: Only valid if a Contour instance already present in the clu
ster. Set to "true" if you want CNR to re-use the cluster's existing Cont
our CRDs.
```

2. Create a `cnr-values.yaml` file by using the following sample as a guide to configure Cloud Native Runtimes:

   Note: For most installations, you can leave the `cnr-values.yaml` empty, and use the default values.

   ```
   ---
   # Configures the domain that Knative Services will use
   domain_name: "mydomain.com"
   ```

   Configuration Notes: * If you are running on a single-node cluster, such as minikube, set the `provider: local` option. This option reduces resource requirements by using a NodePort service instead of a LoadBalancer and reduces the number of replicas.

- If you are running on TKGs, set the `provider: tkgs` option. This option applies TKGs specific configurations.
- Cloud Native Runtimes reuses the existing `tanzu-system-ingress` Contour installation for external and internal access when installed in the `light` or `full` profile. If you want to use a separate Contour installation for system-internal traffic, set `cnrs.ingress.internal.namespace` to the empty string (`""`).
- If you are running on a multinode cluster, do not set `provider`.
- If your environment already has Contour installed, the installed Contour might conflict with the Cloud Native Runtimes installation. For information about how to prevent conflicts, see Installing Cloud Native Runtimes with an Existing Contour Installation. Specify values for `ingress.reuse_crds`, `ingress.external.namespace`, and `ingress.internal.namespace` in the `cnr-values.yaml` file.

3. Install the package by running:

```
tanzu package install cloud-native-runtimes -p cnrs.tanzu.vmware.com -v 1.3.0 -
n tap-install -f cnr-values.yaml --poll-timeout 30m
```

For example:

```
$ tanzu package install cloud-native-runtimes -p cnrs.tanzu.vmware.com -v 1.3.0
 -n tap-install -f cnr-values.yaml --poll-timeout 30m
- Installing package 'cnrs.tanzu.vmware.com'
| Getting package metadata for 'cnrs.tanzu.vmware.com'
| Creating service account 'cloud-native-runtimes-tap-install-sa'
| Creating cluster admin role 'cloud-native-runtimes-tap-install-cluster-role'
| Creating cluster role binding 'cloud-native-runtimes-tap-install-cluster-role
binding'
- Creating package resource
- Package install status: Reconciling

 Added installed package 'cloud-native-runtimes' in namespace 'tap-install'
```

4. Verify the package install by running:

```
tanzu package installed get cloud-native-runtimes -n tap-install
```

For example:

```
tanzu package installed get cloud-native-runtimes -n tap-install
| Retrieving installation details for cc...
NAME:                    cloud-native-runtimes
PACKAGE-NAME:            cnrs.tanzu.vmware.com
PACKAGE-VERSION:         1.3.0
STATUS:                  Reconcile succeeded
CONDITIONS:              [{ReconcileSucceeded True  }]
USEFUL-ERROR-MESSAGE:
```

Verify that `STATUS` is `Reconcile succeeded`.

Administrator Guide for Cloud Native Runtimes

The next several pages show you how to use, troubleshoot, integrate, upgrade and uninstall CNR.

Configure your External DNS with Cloud Native Runtimes

This topic describes how you can configure your external DNS with Cloud Native Runtimes, commonly known as CNR.

Overview

Knative uses `example.com` as the default domain.

Note: If you are setting up Cloud Native Runtimes for development or testing, you do not have to set up an external DNS. However, if you want to access your workloads (apps) over the internet, then you do need to set up a custom domain and an external DNS.

Configure custom domain

To set up the custom domain and its external DNS record:

1. Configure your custom domain:

   When your workloads are created, Knative will automatically create URLs for each workload based on the configuration in the domain ConfigMap.

   - To set a default custom domain, edit your cnr-values.yml file to contain the following:

     ```
     ---
     domain_name: "mydomain.com"
     ```

     This will modify the Knative domain ConfigMap to use `domain_name` as the default domain.

     Note: `domain_name` must be a valid DNS subdomain.

   - **Advanced:** To overwrite the domain ConfigMap entirely, edit your cnr-values.yml file to contain your desired config-domain options, similar to the following:

     ```
     ---
     domain_config: |
      ---
      mydomain.com: |
     ```

```
mydomain.org: |
  selector:
    app: nonprofit
```

This will replace the body of the Knative domain ConfigMap with `domain_config`. This will allow you to configure multiple custom domains, and configure a custom domain for a service depending on its labels.

See Changing the default domain for more information about the structure of the domain ConfigMap.

Note: `domain_config` must be valid YAML and a valid domain ConfigMap.

Note: You can only use one of `domain_config` or `domain_name` at a time. You may not use both.

2. Get the address of the cluster load balancer:

```
kubectl get service envoy -n EXTERNAL-CONTOUR-NS --output 'jsonpath={.status.lo
adBalancer.ingress}'
```

Where `EXTERNAL-CONTOUR-NS` is the namespace where a Contour serving external traffic is installed. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-ingress`.

If this command returns a URL instead of an IP address, then `ping` the URL to get the load balancer IP address.

3. Create a wildcard DNS `A` record that assigns the custom domain to the load balancer IP. Follow the instructions provided by your domain name registrar for creating records.

The record created looks like:

```
*.DOMAIN IN A TTL LOADBALANCER-IP
```

Where:

- `DOMAIN` is the custom domain.
- `TTL` is the time-to-live.
- `LOADBALANCER-IP` is the load balancer IP.

For example:

```
*.mydomain.com IN A 3600 198.51.100.6
```

If you chose to configure multiple custom domains above, you will need to create a wildcard DNS record for each domain.

Configure Knative Service Domain Template

Knative uses domain template which specifies the golang text template string to use when constructing the Knative service's DNS name. The default value is `{{.Name}}.{{.Namespace}}.{{.Domain}}`. Valid variables defined in the template include Name, Namespace, Domain, Labels, and Annotations.

To configure domain template for the created Knative Services, edit your cnr-values.yml file to contain the following:

```
---
domain_template: "{{.Name}}-{{.Namespace}}.{{.Domain}}"
```

This will modify the Knative `domain-template` ConfigMap to use `domain_template` as the default domain template.

Changing this value might be necessary when the extra levels in the domain name generated are problematic for wildcard certificates that only support a single level of domain name added to the certificate's domain. In those cases you might consider using a value of `{{.Name}}-{{.Namespace}}.{{.Domain}}`, or removing the Namespace entirely from the template.

When choosing a new value, be thoughtful of the potential for conflicts, such as when users the use of characters like `-` in their service or namespace names.

`{{.Annotations}}` or `{{.Labels}}` can be used for any customization in the go template if needed.

It is strongly recommended to keep namespace part of the template to avoid domain name clashes: eg. `{{.Name}}-{{.Namespace}}.{{ index .Annotations "sub"}}.{{.Domain}}` and you have an annotation `{"sub":"foo"}`, then the generated template would be `{Name}-{Namespace}.foo.{Domain}`.

Use your existing TLS Certificate for Cloud Native Runtimes

This topic tells you how to use your existing TLS Certificate for Cloud Native Runtimes, commonly known as CNR.

Prerequisites

In order to configure TLS for Cloud Native Runtimes, you must first configure a Service Domain. For more information, see Configuring External DNS with CNR.

To configure your TLS certificate for the created Knative Services, follow the steps:

- Create a Kubernetes Secret to hold your TLS Certificate

```
kubectl create -n <Enter developer namespace here> secret tls <Enter TLS_NAME name her
e> \
  --key key.pem \
  --cert cert.pem
```

- Create a delegation. To do so, create a "tlscertdelegation.yaml" file with following contents

```
  apiVersion: projectcontour.io/v1
  kind: TLSCertificateDelegation
  metadata:
  name: default-delegation
  namespace: <Enter developer namespace here>
  spec:
     delegations:
       - secretName: <Enter the above TLS NAME here>
```

```
         targetNamespaces:
        - "<Enter developer namespace here>"
```

Apply the above yaml file by running

```
  k apply -f tlscertdelegation.yaml
```

- Include the following config in your `tap-values.yml` file and redeploy:

```
---
default_tls_secret: "<Enter developer namespace here>/<Enter the above TLS_NAME here>"
```

To redeploy run -

```
tanzu package installed update tap -p tap.tanzu.vmware.com --values-file "tap-values.y
ml"  -n tap-install
```

This will modify the Knative `default_tls_secret` ConfigMap to use `default_tls_secret` as the default tls certificate

Installing Cloud Native Runtimes with your Existing Contour Installation

This topic describes how you can configure Cloud Native Runtimes, commonly known as CNR, with your existing Contour instance. Cloud Native Runtimes uses Contour to manage internal and external access to the services in a cluster.

About Using Contour with Cloud Native Runtimes

The instructions on this page assume that you have an existing Contour installation on your cluster.

Follow the instructions on this page if:

- You have installed Contour as part of TAP and wish to configure Cloud Native Runtimes to use it.
- You see an error about `an existing Contour installation` when you install the Cloud Native Runtimes package, then follow the procedures on this page to install Cloud Native Runtimes.

Cloud Native Runtimes needs two instances of Contour: one instance for exposing services outside the cluster, and another instance for services that are private in your network. If installed as part of a Tanzu Application Platform profile, by default Cloud Native Runtimes will use the Contour installed in the namespace `tanzu-system-ingress` for both internal and external traffic. If no Contour namespaces are provided, Cloud Native Runtimes deploys an instance of Contour to the `contour-external` namespace and a second instance to the `contour-internal` namespace.

If you already use a Contour instance to route requests from clients outside the cluster, you can use that instance in place of Cloud Native Runtimes' default `contour-external` instance. Similarly, if you already use a Contour instance to route requests from clients inside the cluster, you can use that instance in place of Cloud Native Runtimes' default `contour-internal` instance.

You may use the same single instance of Contour for both internal and external traffic. However, this will cause internal and external traffic will be handled the same way. For example, if the Contour instance is configured to be accessible from clients outside the cluster, then any internal traffic will also be accessible from outside the cluster.

Note: Cloud Native Runtimes configuration options that affect the Contour installed, like the `nodeport` option, will not apply on your existing Contour instances.

If you do not want to reuse Contour instances for services, then Cloud Native Runtimes will create them.

In all of the above cases, you must allow Cloud Native Runtimes to reuse the existing Contour `CustomResourceDefinition`s.

Prerequisites

The following prerequisites are required to configure Cloud Native Runtimes with an existing Contour installation:

- Contour v1.19 (recommended) or v1.18. To identify your cluster's Contour version, see Identify Your Contour Version below.
- Contour `CustomResourceDefinition`s versions:

Resource Name	Version
`extensionservices.projectcontour.io`	v1alpha1
`httpproxies.projectcontour.io`	v1
`tlscertificatedelegations.projectcontour.io`	v1
`contourconfigurations.projectcontour.io` (v1.19 only)	v1alpha1
`contourdeployments.projectcontour.io` (v1.19 only)	v1alpha1

Note: As of CNR 1.3, not all configurations are possible if the existing Contour version is v1.18.

Tanzu Application Platform 1.2 includes Cloud Native Runtimes 1.3, but provides Contour 1.18. Therefore, using TAP provided Contour is not supported in all configurations.

See table below for details.

Identify Your Contour Version

To identify your cluster's Contour version, run:

```
export CONTOUR_NAMESPACE=CONTOUR-NAMESPACE
export CONTOUR_DEPLOYMENT=$(kubectl get deployment --namespace $CONTOUR_NAMESPACE --ou
tput name)
kubectl get $CONTOUR_DEPLOYMENT --namespace $CONTOUR_NAMESPACE --output jsonpath="{.sp
ec.template.spec.containers[].image}"
kubectl get crds extensionservices.projectcontour.io --output jsonpath="{.status.store
dVersions}"
kubectl get crds httpproxies.projectcontour.io --output jsonpath="{.status.storedVersi
ons}"
kubectl get crds tlscertificatedelegations.projectcontour.io --output jsonpath="{.stat
us.storedVersions}"
```

Where `CONTOUR-NAMESPACE` is the namespace where Contour is installed on your Kubernetes cluster.

Install Cloud Native Runtimes on a Cluster with Your Existing Contour Instances

To install Cloud Native Runtimes on a cluster with an existing Contour instance, you can add values to your `cnr-values.yml` file so that the existing Contour `CustomResourceDefinition`s from your cluster are reused. Refer to the table below for information on which config options to set.

Use your Contour for external services?	Use your Contour for internal services?	Values for `ingress` config block:	Valid with existing Contour 1.18	Valid with existing Contour 1.19
No	No	`ingress.reuse_crds=true`	No	Yes
Yes	No	`ingress.reuse_crds=true` `ingress.external.namespace=EXTERNAL-CONTOUR-NS`	No	Yes
No	Yes	`ingress.reuse_crds=true` `ingress.internal.namespace=INTERNAL-CONTOUR-NS`	No	Yes
Yes	Yes	`ingress.reuse_crds=true` `ingress.external.namespace=EXTERNAL-CONTOUR-NS` `ingress.internal.namespace=INTERNAL-CONTOUR-NS`	Yes	Yes

Where `EXTERNAL-CONTOUR-NS` and `INTERNAL-CONTOUR-NS` are the namespaces where Contour is installed on your Kubernetes cluster.

Note that in the cases where `ingress.reuse_crds` is `true`, and a value isn't provided for `ingress.external`, `ingress.internal`, or both, the install will create new instances of Contour to the namespaces `contour-external` and/or `contour-internal` using your cluster's `CustomResourceDefinition`s.

Note: If your Contour instance is removed or configured incorrectly, apps running on Cloud Native Runtimes will lose connectivity.

An example of a `cnr-values.yml` file where you wish to reuse an existing Contour instance for external traffic would look like this:

```
---
ingress:
  reuse_crds: true
  external:
    namespace: my-existing-contour-namespace
```

Securing Your Web Workloads in Cloud Native Runtimes

This topic give you an overview of securing HTTP connections using TLS certificates in Cloud Native Runtimes, commonly known as CNR, for VMware Tanzu Application Platform and helps you configure TLS (Transport Layer Security).

Overview

Cloud Native Runtimes supports both `HTTP01` and `DNS01` cert-manager challenge types. For more information about `cert-manager` challenge types, see ACME in the cert-manager documentation.

VMware recommends using Let's Encrypt as your certificate authority. However, you can integrate Cloud Native Runtimes with any ACME compatible certificate authority.

Prerequisites

You can enable HTTPS with Automatic TLS certificate provisioning for Cloud Native Runtimes.

You need the following prerequisites to use secure HTTPS connections with automatic TLS certificate provisioning:

- A cluster configured to use a custom domain. See Setting Up a Custom Domain in the Knative documentation.
- A DNS provider configured with your domain name.
- cert-manager version 1.0.0 or later. See Installing cert-manager for TLS certificates in the Knative documentation.
- `HTTP01` **challenges**: An internet-reachable cluster.
- `DNS01` **challenges**: API access to set DNS records.

Enable Auto TLS Using an HTTP01 Challenge

You can use the `HTTP01` challenge type to validate a domain with Cloud Native Runtimes. The `HTTP01` challenge requires that your load balancer be reachable from the internet via HTTP.

Note: With the `HTTP01` challenge type, you provision a certificate for each service.

To enable automatic TLS certificate provisioning using a `HTTP01` challenge, do the following:

1. Create a cert-manager `Issuer` or `ClusterIssuer` for the `HTTP01` challenge. See Issuer in the cert-manager documentation. The following example creates a `ClusterIssuer` using the Let's Encrypt Certificate Authority. See Let's Encrypt. To use a `ClusterIssuer` for the `HTTP01` challenge, run:

```
kubectl apply -f - <<EOF
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-http01-issuer
spec:
  acme:
    privateKeySecretRef:
      name: letsencrypt
    server: https://acme-v02.api.letsencrypt.org/directory
    solvers:
    - http01:
        ingress:
          class: contour
EOF
```

2. To validate that your `ClusterIssuer` was created successfully, run:

   ```
   kubectl get clusterissuer letsencrypt-http01-issuer --output yaml
   ```

3. Edit your `config-certmanager` ConfigMap in the `knative-serving` namespace to reference the `ClusterIssuer` you created. Run:

   ```
   kubectl edit configmap config-certmanager --namespace knative-serving
   ```

4. To define which ClusterIssuer will be used by Knative to issue certificates, add the following `issuerRef` block under the `data` section of the `config-certmanager` ConfigMap:

   ```
   ...
   data:
   ...
    issuerRef: |
      kind: ClusterIssuer
      name: letsencrypt-http01-issuer
   ```

5. To validate that your ConfigMap was updated successfully, run:

   ```
   kubectl get configmap config-certmanager --namespace knative-serving --output j
   sonpath="{.data.issuerRef}"
   ```

6. Edit the `config-network` ConfigMap in the `knative-serving` namespace to enable automatic TLS certificate provisioning and specify how HTTP requests are handled. Run:

   ```
   kubectl edit configmap config-network --namespace knative-serving
   ```

   Note: For `HTTP01` challenges, the `httpProtocol` field must be set to `Enabled` for the cluster to accept `HTTP01` challenge requests.

   ```
   apiVersion: v1
   kind: ConfigMap
   metadata:
    name: config-network
    namespace: knative-serving
   data:
    ...
    autoTLS: Enabled
    ...
    httpProtocol: Enabled
    ...
   ```

7. To validate that your ConfigMap was updated successfully, run:

   ```
   kubectl get configmap config-network --namespace knative-serving --output jsonp
   ath="{.data.autoTLS}"
   kubectl get configmap config-network --namespace knative-serving --output jsonp
   ath="{.data.httpProtocol}"
   ```

8. Verify that your automatic TLS certificate instance is configured correctly by deploying a sample app. See Verify Auto TLS in the Knative documentation.

Enable Auto TLS Using a DNS01 Challenge

The `DNS01` challenge validates that you control your domain's DNS by accessing and updating your domain's TXT record. You need to provide a `cert-manager` with your DNS API credentials. For a list of DNS01 providers supported for the ACME `Issuer`, see the cert-manager documentation.

Note

: You can provision certificates **per service** only.

To enable automatic TLS certificate provisioning using a `DNS01` challenge, do the following:

1. Set up credentials for `cert-manager` to access your DNS records. For information about setting up credentials for your ACME `Issuer` supported DNS provider, see Supported DNS01 providers in the cert-manager documentation. In the next step, you create an Issuer on `cert-manager` with the configuration you set up.

2. Create a cert-manager Issuer or ClusterIssuer for DNS01 challenge on the `cert-manager` Issuer you set up in the previous step. The following example uses Let's Encrypt and Google Cloud DNS. For information about other DNS providers supported by cert-manager, see the cert-manager documentation. The `Issuer` assumes that your Kubernetes secret holds credentials for the service account created. Run the following command to apply the ClusterIssuer:

   ```
   kubectl apply --filename - <<EOF
   apiVersion: cert-manager.io/v1
   kind: ClusterIssuer
   metadata:
     name: letsencrypt-dns-issuer
   spec:
     acme:
       server: https://acme-v02.api.letsencrypt.org/directory
       # This will register an issuer with LetsEncrypt.
       email: MY-EMAIL
       privateKeySecretRef:
         # Set privateKeySecretRef to any unused secret name.
         name: letsencrypt-dns-issuer
       solvers:
       - dns01:
           cloudDNS:
             project: $PROJECT_ID
             # Set this to the secret that we publish our service account key
             # in the previous step.
             serviceAccountSecretRef:
               name: cloud-dns-key
               key: key.json
   EOF
   ```

   Where `MY-EMAIL` is your email address.

3. To verify that your ClusterIssuer is created successfully, run:

   ```
   kubectl get clusterissuer letsencrypt-dns-issuer --output yaml
   ```

4. Edit your `config-certmanager` ConfigMap in the `knative-serving` namespace to reference the ClusterIssuer created in the previous step. Run:

   ```
   kubectl edit configmap config-certmanager --namespace knative-serving
   ```

5. Add an `issuerRef` block under the `data` section of your ConfigMap. This defines the ClusterIssuer Knative uses to issue certificates. Run:

   ```
   ...
   data:
   ...
    issuerRef: |
      kind: ClusterIssuer
      name: letsencrypt-dns-issuer
   ```

6. To validate that your file was updated successfully, run:

   ```
   kubectl get configmap config-certmanager --namespace knative-serving --output j
   sonpath="{.data.issuerRef}"
   ```

7. To enable automatic TLS certificate provisioning and specify how HTTP requests are handled, edit your `config-network` ConfigMap in the `knative-serving` namespace:

   ```
   kubectl edit configmap config-network --namespace knative-serving
   ```

   Note: When using the DNS01 challenge type, the `httpProtocol` field must be set to `Enabled`.

   ```
   apiVersion: v1
   kind: ConfigMap
   metadata:
    name: config-network
    namespace: knative-serving
   data:
    ...
    autoTLS: Enabled
    ...
    httpProtocol: Enabled
    ...
   ```

8. Validate that your file was updated successfully. Run:

   ```
   kubectl get configmap config-network --namespace knative-serving --output jsonp
   ath="{.data.autoTLS}"
   kubectl get configmap config-network --namespace knative-serving --output jsonp
   ath="{.data.httpProtocol}"
   ```

9. Verify that your automatic TLS certificate instance is functioning correctly by deploying a sample app. See Verify Auto TLS in the Knative documentation.

Configuring Eventing with RabbitMQ for Cloud Native Runtimes

This topic tells you how to use RabbitMQ as an event source to react to messages sent to a

RabbitMQ exchange or as an event broker to distribute events within your app for Cloud Native Runtimes, commonly known as CNR.

Overview

The integration allows you to create:

- **A RabbitMQ broker**: A Knative Eventing broker backed by RabbitMQ. This broker uses RabbitMQ exchanges to store CloudEvents that are then routed from one component to another.
- **A RabbitMQ source:** An event source that translates external messages on a RabbitMQ exchange to CloudEvents, which can then be used with Knative Serving or Knative Eventing over HTTP.

About the RabbitMQ Operators

Before you can use or test RabbitMQ eventing on Cloud Native Runtimes, you need to install the following products on your Kubernetes cluster:

- RabbitMQ Cluster Kubernetes Operator v1.8.2. See Install RabbitMQ Cluster Kubernetes Operator v1.8.2 below.
- RabbitMQ Messaging Topology Kubernetes Operator v1.2.1. See Install RabbitMQ Messaging Topology Kubernetes Operator v1.2.1 below.
- cert-manager v1.5.3 and later. See Installation in the cert-manager documentation.

Install RabbitMQ Cluster Kubernetes Operator v1.8.2

The RabbitMQ Cluster Kubernetes Operator (cluster Operator) automates the lifecycle, creation, upgrade, and shutdown, of RabbitMQ clusters on Kubernetes:

To install the cluster Operator:

1. Create the `rabbitmq-system` namespace on your Kubernetes cluster where Cloud Native Runtimes is installed:

   ```
   kubectl create namespace rabbitmq-system
   ```

2. Define the following role binding:

   ```
   kubectl apply -f - << EOF
   kind: RoleBinding
   apiVersion: rbac.authorization.k8s.io/v1
   metadata:
     name: rabbitmq-cluster-operator-psp
     namespace: rabbitmq-system
   roleRef:
     apiGroup: rbac.authorization.k8s.io
     kind: ClusterRole
     name: cnr-restricted
   subjects:
   - kind: ServiceAccount
     name: rabbitmq-cluster-operator
   ```

```
  namespace: rabbitmq-system
EOF
```

3. Install the RabbitMQ Cluster Kubernetes Operator v1.8.2 on your Kubernetes cluster where Cloud Native Runtimes is installed:

   ```
   kubectl apply -f https://github.com/rabbitmq/cluster-operator/releases/download
   /v1.8.2/cluster-operator.yml
   ```

   For general information about the RabbitMQ Cluster Kubernetes Operator, see rabbitmq/cluster-operator in GitHub.

Install RabbitMQ Messaging Topology Kubernetes Operator v1.2.1

The RabbitMQ Messaging Topology Kubernetes Operator (topology Operator) manages the topologies, or exchange types, of RabbitMQ clusters provisioned by the cluster Operator.

There are two YAMLs for the RabbitMQ Messaging Topology Kubernetes Operator:

- `messaging-topology-operator-with-certmanager.yaml`: Requires that you have cert-manager v1.5.3 installed
- `messaging-topology-operator.yaml`: Use if you want to generate and include your own certificates

To install the topology Operator:

1. Read the README.md for the topology Operator in GitHub and decide which YAML to install.
2. If you are installing the `messaging-topology-operator-with-certmanager.yaml`, then:
   1. Create the cert-manager namespace on your Kubernetes cluster where Cloud Native Runtimes is installed:

      ```
      kubectl create namespace cert-manager
      ```

   2. Define the following role binding:

      ```
      kubectl apply -f - << EOF
      kind: RoleBinding
      apiVersion: rbac.authorization.k8s.io/v1
      metadata:
        name: cert-manager-psp
        namespace: cert-manager
      roleRef:
        apiGroup: rbac.authorization.k8s.io
        kind: ClusterRole
        name: cnr-restricted
      subjects:
      - kind: ServiceAccount
        name: cert-manager
        namespace: cert-manager
      - kind: ServiceAccount
        name: cert-manager-cainjector
      ```

```
    namespace: cert-manager
  - kind: ServiceAccount
    name: cert-manager-webhook
    namespace: cert-manager
  EOF
```

3. Define the following role binding:

```
kubectl apply -f - << EOF
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rabbitmq-topology-psp
  namespace: rabbitmq-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cnr-restricted
subjects:
- kind: ServiceAccount
  name: messaging-topology-operator
  namespace: rabbitmq-system
EOF
```

4. Install the RabbitMQ Messaging Topology Operator v1.2.1 on your Kubernetes cluster where Cloud Native Runtimes is installed.

   For general information about the topology Operator v1.2.1, see the RabbitMQ Messaging Topology Operator v1.2.1 Release Notes in GitHub.

Next Steps

After completing these installations, you can:

- Verify your Knative Eventing installation using an example RabbitMQ broker. For instructions, see Verify Knative Eventing.
- Create a broker, producer, and a consumer to use RabbitMQ and Knative Eventing with your own app.

Configuring Observability for Cloud Native Runtimes

This topic tells you how to configure observability for Cloud Native Runtimes, commonly known as CNR.

Overview

You can set up integrations with third-party observability tools to use logging, metrics, and tracing with Cloud Native Runtimes for Tanzu. These observability integrations allow you to monitor and collect detailed metrics from your clusters on Cloud Native Runtimes. You can collect logs and metrics for all workloads running on a cluster. This includes Cloud Native Runtimes components or any apps running on Cloud Native Runtimes. The integrations in this topic are recommended by VMware, however you can use any Kubernetes compatible logging, metrics, and tracing platforms to monitor your cluster workload.

Logging

You can collect and forward logs for all workloads on a cluster, including Cloud Native Runtimes components or any apps running on Cloud Native Runtimes. You can use any logging platform that is compatible with Kubernetes to collect and forward logs for Cloud Native Runtimes workloads. VMware recommends using Fluent Bit to collect logs and then forward logs to vRealize. The following sections describe configuring logging for Cloud Native Runtimes with Fluent Bit and vRealize as an example.

Configure Logging with Fluent Bit

You can use Fluent Bit to collect logs for all workloads on a cluster, including Cloud Native Runtimes components or any apps running on Cloud Native Runtimes. For more information about using Fluent Bit logs, see Fluent Bit Kubernetes Logging in the Fluent Bit documentation.

Fluent Bit lets you collect logs from Kubernetes containers, add Kubernetes metadata to these logs, and forward logs to third-party log storage services. For more information about collecting logs, see Logging in the Knative documentation.

If you are using Tanzu Mission Control (TMC), vSphere 7.0 with Tanzu, or Tanzu Kubernetes Cluster to manage your cloud native environment, you must set up a role binding that grants required permissions to Fluent Bit containers in order to configure logging with any integration. Then, you can follow the instructions in the Fluent Bit documentation to complete the logging configuration. For more information about configuring Fluent Bit logging, see Installation in the Fluent Bit documentation.

To configure logging with Fluent Bit for your Cloud Native Runtimes environment:

1. VMware recommends that you add any integrations to the `ConfigMap` in both your Knative Serving and Knative Eventing namespaces. Follow the logging configuration steps in the Fluent Bit documentation to create the `Namespace`, `ServiceAccount`, `Role`, `RoleBinding`, and `ConfigMap`. To view these steps, see Installation in the Fluent Bit documentation.
2. If you are using TMC, vSphere with Tanzu, or Tanzu Kubernetes Cluster to manage your cloud native environment, create a role binding in the Kubernetes namespace where your integration will be deployed to grant permission for privileged Fluent Bit containers. For information about creating a role binding on a Tanzu platform, see Add a Role Binding in the TMC documentation. For information about viewing your Kubernetes namespaces, see Viewing Namespaces in the Kubernetes documentation. Create the following role binding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: fluentbit-psp-rolebinding
  namespace: FLUENTBIT-NAMESPACE
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name:  PRIVILEGED-CLUSTERROLE
subjects:
- kind: ServiceAccount
  name: FLUENTBIT-SERVICEACCOUNT
  namespace: FLUENTBIT-NAMESPACE
```

Where:

- `FLUENTBIT-NAMESPACE` is your Fluent Bit namespace.
- `PRIVILEGED-CLUSTERROLE` is the name of your privileged cluster role.
- `FLUENTBIT-SERVICEACCOUNT` is your Fluent Bit service account.

3. To verify that you have configured logging successfully, run the following to access logs through your web browser:

```
kubectl port-forward --namespace logging service/log-collector 8080:80
```

For more information about accessing Fluent Bit logs, see Logging in the Knative documentation.

Forward Logs to vRealize

After you configure log collection, you can forward logs to log management services. vRealize Log Insight is one service you can use with Cloud Native Runtimes. vRealize Log Insight is a scalable log management solution that provides log management, dashboards, analytics, and third-party extensibility for infrastructure and apps. For more information about vRealize Log Insight, see the VMware vRealize Log Insight Documentation.

To forward logs from your Cloud Native Runtimes environment to vRealize, you can use a new or existing instance of Tanzu Kubernetes Cluster. For information about how to configure log forwarding to vRealize from Tanzu Kubernetes Cluster, see the Configure Log forwarding from VMware Tanzu Kubernetes Cluster to vRealize Log Insight Cloud blog.

Metrics

Cloud Native Runtimes integrates with Prometheus and Tanzu Observability by Wavefront to collect metrics on components or apps. For more information about integrating with Prometheus, see Overview in the Prometheus documentation and Kubernetes Integration in the Wavefront documentation.

You can configure Prometheus endpoints on Cloud Native Runtimes components in order to be able to collect metrics on your components or apps. For information on annotations required to collect metrics on apps, see Per-Pod Prometheus Annotations in the WeaveWorks documentation.

You can use annotation based discovery with Prometheus to define which Kubernetes objects in your Cloud Native Runtimes environment to add metadata and collect metrics in a more automated way. For more information about using annotation based discovery, see Annotation based discovery in GitHub.

You can then use the Wavefront Collector for Kubernetes collector to dynamically discover and scrape pods with the `prometheus.io/scrape` annotation prefix. For information about the Kubernetes collector, see Wavefront Collector for Kubernetes in GitHub.

Note

: All Cloud Native Runtimes related metrics are emitted with the prefix `tanzu.vmware.com/cloud-native-runtimes.*`.

Tracing

Tracing is a method for understanding the performance of specific code paths in apps as they handle requests. You can configure tracing to collect performance metrics for your apps or Cloud Native Runtimes components. You can trace which aspects of Cloud Native Runtimes and workloads running on Cloud Native Runtimes are performing poorly.

Configuring Tracing

You can configure tracing for your apps on Cloud Native Runtimes. To do this, you configure tracing for both Knative Serving and Eventing by editing the ConfigMap for your Knative namespace.

To configure tracing, do the following:

1. Configure the `config-tracing` ConfigMap in your Knative component namespace. VMware recommends that you add any integrations to the ConfigMap in both your Serving and Eventing namespaces. For information on how to enable request traces in each component, see the following Knative documentation:
   - Serving. See Accessing request traces.
   - Eventing. See Accessing CloudEvent traces.

Forwarding Trace Data to a Data Visualization Tool

You can use the OpenTelemetry integration with Tanzu Observability by Wavefront to forward trace data to Tanzu Observability by Wavefront. For information about forwarding trace data, see Sending Metrics Data to Wavefront in the Wavefront documentation.

To configure to send trace data to Cloud Native Runtimes tracing with Tanzu Observability by Wavefront and the OpenTelemetry integration, do the following:

1. Use the following documentation to configure the OpenTelemetry Integration to send trace data to with Cloud Native Runtimes. For more information about sending trace data with OpenTelemetry, see OpenTelemetry Integration in the Wavefront documentation.
2. Deploy the Wavefront Proxy. For more information about wavefront proxies, see Deploy a Wavefront Proxy in Kubernetes in the Wavefront documentation.
   - Use the following .yaml file to install the Wavefront proxy in your Kubernetes cluster: wavefront.yaml.
   - Provide the URL of your Wavefront instance and a Wavefront token.
   - Uncomment the lines indicated in the yaml file to enable consumption of Zipkin traces.

Sending Trace Data to an Observability Platform

You can send trace data to an observability and analytics platform to view and monitor your trace data in dashboards.

One way to do this is to integrate Tanzu Observability by Wavefront with your Cloud Native Runtimes environment. To view your trace data in Wavefront, you configure Cloud Native Runtimes to send traces to the Wavefront proxy and then configure the Wavefront proxy to consume Zipkin spans.

For more information about using Zipkin for tracing, see the Zipkin documentation.

You can send trace data from Cloud Native Runtimes to Wavefront by using Zipkin as the backend and defining the Zipkin endpoint as the Wavefront proxy URL listening over port `9411`. You configure Cloud Native Runtimes to send traces directly to the Wavefront proxy by editing the `zipkin-endpoint` property in the ConfigMap to point to the Wavefront proxy URL. You can configure the Wavefront proxy to consume Zipkin spans by listening to port 9411.

To send trace data to Tanzu Observability by Wavefront:

1. Edit the ConfigMap to enable the Zipkin tracing integration. VMware recommends that you add any integrations to the ConfigMap in both your Serving and Eventing namespaces. Edit the Knative config-tracing ConfigMap to set `backend` to `zipkin` and pass the Wavefront proxy URL in the zipkin-endpoint field:

   ```
   Kubectl edit configmap config-tracing —namespace knative-serving apiVersion: v1
   kind: ConfigMap
   metadata:
   name: config-tracing
   ...
   data:
   backend: "zipkin"
   zipkin-endpoint: "http://wavefront-proxy.default.svc.cluster.local:9411/api/v2/
   spans"  ...
   ```

Use Wavefront Dashboards

Cloud Native Runtimes provides two Wavefront dashboards in JSON format. You can use these dashboard to monitor your apps and investigate performance issues. For information about configuring dashboards, see Create and Customize Dashboards in the Wavefront documentation.

The following Wavefront dashboards are compatible with Cloud Native Runtimes: - Application Operator Service View. See `app-operator-service-view.json` in the Cloud Native Runtimes installation .tar file. - Application Operator Revision View. See `app-operator-revision-view.json` in the Cloud Native Runtimes installation .tar file.

To import a dashboard JSON file, use one of the following methods: - Wavefront REST API - Wavefront CLIs.

You must provide the URL of your Wavefront instance and a Wavefront token. For more information about Wavefront tokens, see Generating an API Token in the Wavefront documentation.

Import Wavefront Dashboards

You can import the Wavefront dashboards using either the Wavefront API or the Ruby Wavefront CLI. For more information about Wavefront dashboard, see Import Dashboards with the Wavefront API or Import with a Ruby Wavefront CLI below.

Import Dashboards with the Wavefront API

To import a Wavefront dashboard with the Wavefront API, run:

```
curl -H "Content-Type: application/json" -H 'Authorization: Bearer <wavefront-token>'
\
    https://<wavefront-instance>.wavefront.com/api/v2/dashboard -d @observability/wave
front/app-operator-service-view.json

curl -H "Content-Type: application/json" -H 'Authorization: Bearer <wavefront-token>'
\
    https://<wavefront-instance>.wavefront.com/api/v2/dashboard -d @dashboards/wavefro
nt/app-operator-revision-view.json
```

After you run the import code, the Wavefront API creates two dashboards with the following names and URLs:

- **Title**: `Cloud Native Runtimes App Operator - Service View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Service-Level`

- **Title**: `Cloud Native Runtimes App Operator - Revision View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Revision-Level`

Import with the Ruby Wavefront CLI

To import a Wavefront dashboard with the Ruby Wavefront CLI, run:

```
export WAVEFRONT_TOKEN=<wavefront-token>
export WAVEFRONT_ENDPOINT=<wavefront-instance>.wavefront.com

wf config envvars
wf dashboard import observability/wavefront/app-operator-service-view.json
wf dashboard import dashboards/wavefront/app-operator-revision-view.json
```

After you run the import code, the Ruby Wavefront CLI creates two dashboards with a name and URL.

The Service View of the Cloud Native Runtimes App Operator dashboard will have the following title and URL:

- **Title**: `Cloud Native Runtimes App Operator - Service View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Service-Level`

The Revision View of the Cloud Native Runtimes App Operator dashboard will have the following title and URL:

- **Title**: `Cloud Native Runtimes App Operator - Revision View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Revision-Level`

Configuring Cloud Native Runtimes with Avi Vantage

This topic tells you how to configure Cloud Native Runtimes, commonly known as CNR, with Avi Vantage.

Overview

You can configure Cloud Native Runtimes to integrate with Avi Vantage. Avi Vantage is a multi-cloud platform that delivers features such as load balancing, security, and container ingress services. The Avi Controller provides a control plane. Avi Service Engines provides a data plane. The Avi Service Engines forward incoming traffic to your Kubernetes cluster's Envoy pods, which are created and managed by Contour.

For information about Avi Vantage, see Avi Documentation.

Integrate Avi Vantage with Cloud Native Runtimes

This procedure assumes that you have already installed Cloud Native Runtimes.

If you have not already installed Cloud Native Runtimes, see Installing Cloud Native Runtimes. If you already have a Contour installation on your cluster, see Installing Cloud Native Runtimes with an Existing Contour Installation.

To configure Cloud Native Runtimes with Avi Vantage, do the following:

1. Deploy the Avi Controller on any Avi supported infrastructure providers. For a list of Avi supported providers, see Avi Installation Guides. For more information about deploying an Avi Controller, see Install Avi Kubernetes Operator in the Avi Vantage documentation.

2. Deploy the Avi Kubernetes Operator to your Kubernetes cluster where Cloud Native Runtimes is hosted. See Install AKO for Kubernetes in the Avi Vantage documentation.

3. Connect to a test app and verify that it is reachable. Run:

   ```
   "curl -H KNATIVE-SERVICE-DOMAIN" ENVOY-IP
   ```

   Where:

   - `KNATIVE-SERVICE-DOMAIN` is the name of your domain.
   - `ENVOY-IP` is the IP address of your Envoy instance.

   For more information about deploy a sample application and connect to the application, see Test Knative Serving.

4. (Optional) Create a DNS record that will configure your KService URL to point to the Avi Service Engines, and resolve to the external IP of the Envoy. You can create a DNS record on any platform that supports DNS services. Refer to the documentation for your DNS service platform for more information.

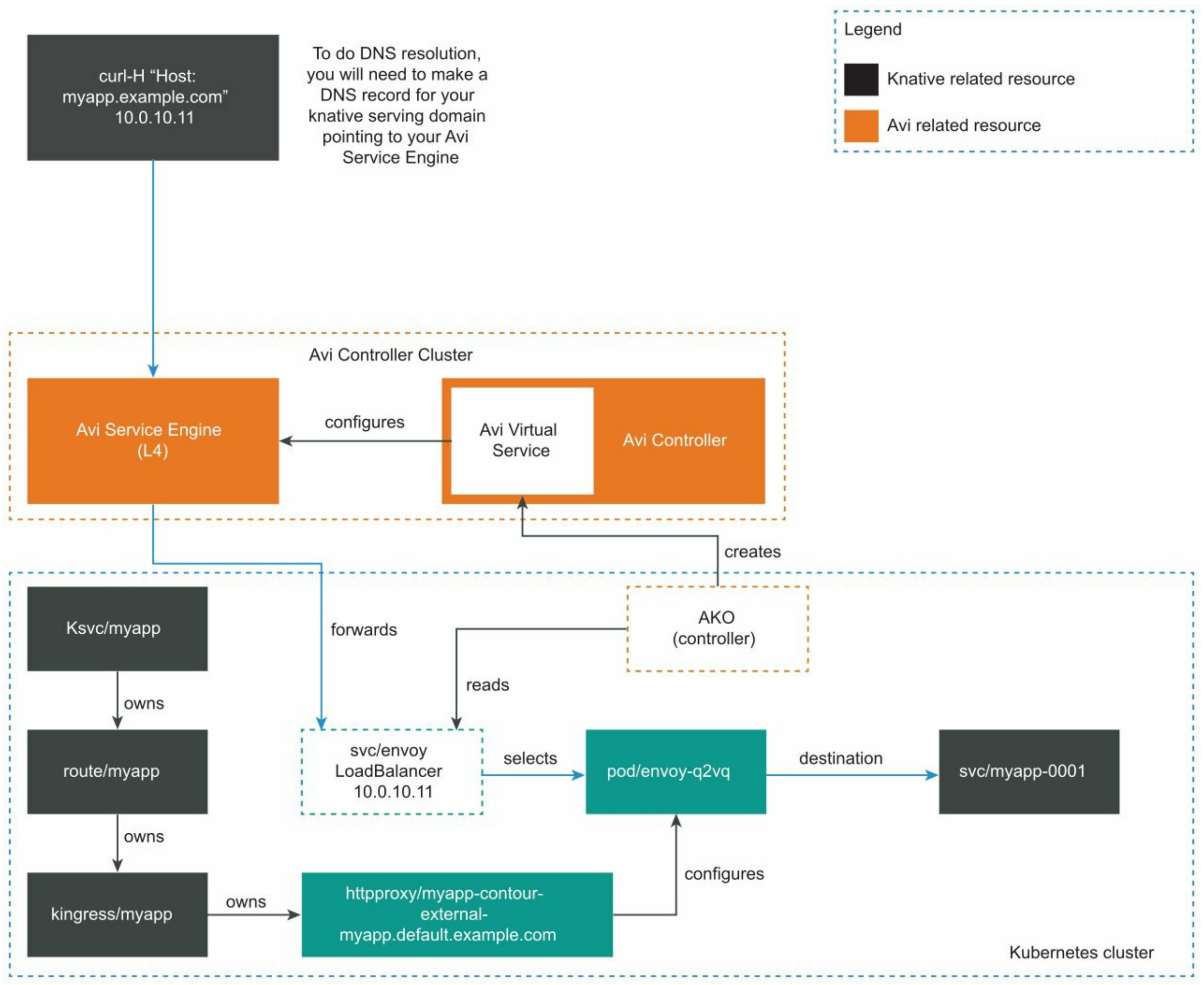
   To get the KService URL, run:

   ```
   kn route describe APP-NAME | grep "URL"
   ```

   To get Envoy's external IP, follow step 3 in Test Knative Serving in *Verifying your Serving Installation*.

About Routing with Avi Vantage and Cloud Native Runtimes

The following diagram shows how Avi Vantage integrates with Cloud Native Runtimes:



When Contour creates a Kubernetes LoadBalancer service for Envoy, the Avi Kubernetes Operator (AKO) sees the new LoadBalancer service. Then Avi Controller creates a Virtual Service. For information about LoadBalancer services, see Type LoadBalancer in the Kubernetes documentation.

For each Envoy service, Avi Controller creates a corresponding Virtual Service. See Virtual Services in the Avi Vantage documentation.

After Avi Controller creates a Virtual Service, the Controller configures the Avi Service Engines to forward traffic to the Envoy pods. The Envoy pods route traffic based on incoming requests, including traffic splitting and path based routing.

The Avi Controller provides Envoy with an external IP address so that apps are reachable by the app developer.

Note: Avi does not interact directly with any Cloud Native Runtimes resources. Avi Vantage forwards all incoming traffic to Envoy.

Configuring Cloud Native Runtimes with Tanzu Service Mesh

This topic tells you how to configure Cloud Native Runtimes, commonly known as CNR, with Tanzu Service Mesh.

Overview

You cannot install Cloud Native Runtimes on a cluster that has Tanzu Service Mesh attached.

This workaround describes how Tanzu Service Mesh can be configured to ignore the Cloud Native Runtimes. This allows Contour to provide ingress routing for the Knative workloads, while Tanzu Service Mesh continues to satisfy other connectivity concerns.

Note: Cloud Native Runtimes workloads are unable to use Tanzu Service Mesh features like Global Namespace, Mutual Transport Layer Security authentication (mTLS), retries, and timeouts.

For information about Tanzu Service Mesh, see Tanzu Service Mesh Documentation.

Run Cloud Native Runtimes on a Cluster Attached to Tanzu Service Mesh

This procedure assumes you have a cluster attached to Tanzu Service Mesh, and that you have not yet installed Cloud Native Runtimes.

Note: If you installed Cloud Native Runtimes on a cluster that has Tanzu Service Mesh attached before doing the procedure below, pods fail to start. To fix this problem, follow the procedure below and then delete all pods in the excluded namespaces.

Configure Tanzu Service Mesh to ignore namespaces related to Cloud Native Runtimes:

1. Navigate to the **Cluster Overview** tab in the Tanzu Service Mesh UI.
2. On the cluster where you want to install Cloud Native Runtimes, click **...**, then select **Edit Cluster...**.
3. Create an Is Exactly rule for each of the following namespaces:
    - CONTOUR-NS
    - knative-serving
    - knative-eventing
    - knative-sources
    - knative-discovery
    - triggermesh
    - vmware-sources
    - cloud-native-runtimes
    - rabbitmq-system
    - kapp-controller
    - The namespace or namespaces where you plan to run Knative workloads.

   Where CONTOUR-NS is the namespace(s) where Contour is installed on your cluster. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-ingress`.

Next Steps

After configuring Tanzu Service Mesh, install Cloud Native Runtimes and verify your installation:

1. Install Cloud Native Runtimes. See Installing Cloud Native Runtimes.
2. Verify your installation. See Verifying Your Installation.

Note: You must create all Knative workloads in the namespace or namespaces where you plan to run these Knative workloads. If you do not, your pods fail to start.

Troubleshooting Cloud Native Runtimes

This topic tells you how to troubleshoot Cloud Native Runtimes, commonly known as CNR, installation or configuration.

Cannot connect to app on AWS

Symptom

On AWS, you see the following error when connecting to your app:

```
curl: (6) Could not resolve host: a**********************7.us-west-2.elb.amazonaws.co
m
```

Solution

Try connecting to your app again after 5 minutes. The AWS LoadBalancer name resolution takes several minutes to propagate.

minikube Pods Fail to Start

Symptom

On minikube, you see the following error when installing Cloud Native Runtimes:

```
3:03:59PM: error: reconcile job/contour-certgen-v1.10.0 (batch/v1) namespace: contour-
internal
Pod watching error: Creating Pod watcher: Get "https://192.168.64.17:8443/api/v1/pods?
labelSelector=kapp.k14s.io%2Fapp%3D1618232545704878000&watch=true": dial tcp 192.168.6
4.17:8443: connect: connection refused
kapp: Error: waiting on reconcile job/contour-certgen-v1.10.0 (batch/v1) namespace: CO
NTOUR-NS:
  Errored:
   Listing schema.GroupVersionResource{Group:"", Version:"v1", Resource:"pods"}, names
paced: true:
    Get "https://192.168.64.17:8443/api/v1/pods?labelSelector=kapp.k14s.io%2Fassociati
on%3Dv1.572a543d96e0723f858367fcf8c6af4e": unexpected EOF
```

Where CONTOUR-NS is the namespace where Contour is installed on your cluster. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-ingress`.

Solution

Increase your available system RAM to at least 4 GB.

Pulling an image with imgpkg overwrites the cloud-native-runtimes directory

Symptom

When relocating an image to a private registry and later pulling that image with `imgpkg pull --lock LOCK-OUTPUT -o ./cloud-native-runtimes`, the contents of the cloud-native-runtimes are overwritten.

Solution

Upgrade the imgpkg version to v0.13.0 or later.

Installation fails to reconcile app/cloud-native-runtimes

Symptom

When installing Cloud Native Runtimes, you see one of the following errors:

```
11:41:16AM: ongoing: reconcile app/cloud-native-runtimes (kappctrl.k14s.io/v1alpha1) n
amespace: cloud-native-runtime
11:41:16AM:  ^ Waiting for generation 1 to be observed
kapp: Error: Timed out waiting after 15m0s
```

Or,

```
3:15:34PM:  ^ Reconciling
3:16:09PM: fail: reconcile app/cloud-native-runtimes (kappctrl.k14s.io/v1alpha1) names
pace: cloud-native-runtimes
3:16:09PM:  ^ Reconcile failed:  (message: Deploying: Error (see .status.usefulErrorMe
ssage for details))

kapp: Error: waiting on reconcile app/cloud-native-runtimes (kappctrl.k14s.io/v1alpha1
) namespace: cloud-native-runtimes:
  Finished unsuccessfully (Reconcile failed:  (message: Deploying: Error (see .status.
usefulErrorMessage for details)))
```

Explanation

The `cloud-native-runtimes` deployment app installs the subcomponents of Cloud Native Runtimes. Error messages about reconciling indicate that one or more subcomponents have failed to install.

Solution

Use the following procedure to examine logs:

1. Get the logs from the `cloud-native-runtimes` app by running:

```
kubectl get app/cloud-native-runtimes -n cloud-native-runtimes -o jsonpath="{.s
tatus.deploy.stdout}"
```

Note: If the command does not return log messages, then kapp-controller is not installed or is not running correctly.

2. Review the output for subcomponent deployments that have failed or are still ongoing. See the examples below for suggestions on resolving common problems.

Example 1: The Cloud Provider does not support the creation of Service type LoadBalancer

Follow these steps to identify and resolve the problem of the cloud provider not supporting services of type `LoadBalancer`:

1. Search the log output for `Load balancer`, for example by running:

```
kubectl -n cloud-native-runtimes get app cloud-native-runtimes -ojsonpath="{.st
atus.deploy.stdout}" | grep "Load balancer" -C 1
```

2. If the output looks similar to the following, ensure that your cloud provider supports services of type `LoadBalancer`. For more information, see Prerequisites.

```
6:30:22PM: ongoing: reconcile service/envoy (v1) namespace: CONTOUR-NS
6:30:22PM:  ^ Load balancer ingress is empty
6:30:29PM: ---- waiting on 1 changes [322/323 done] ----
```

Where CONTOUR-NS is the namespace where Contour is installed on your cluster. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-ingress`.

Example 2: The webhook deployment failed

Follow these steps to identify and resolve the problem of the `webhook` deployment failing in the `vmware-sources` namespace:

1. Review the logs for output similar to the following:

```
10:51:58PM: ok: reconcile customresourcedefinition/httpproxies.projectcontour.i
o (apiextensions.k8s.io/v1) cluster
10:51:58PM: fail: reconcile deployment/webhook (apps/v1) namespace: vmware-sour
ces
10:51:58PM:  ^ Deployment is not progressing: ProgressDeadlineExceeded (message
: ReplicaSet "webhook-6f5d979b7d" has timed out progressing.)
```

2. Run `kubectl get pods` to find the name of the pod:

```
kubectl get pods --show-labels -n NAMESPACE
```

Where `NAMESPACE` is the namespace associated with the reconcile error, for example, `vmware-sources`.

For example,

```
$ kubectl get pods --show-labels -n vmware-sources
NAME                         READY   STATUS    RESTARTS   AGE   LABELS
webhook-6f5d979b7d-cxr9k   0/1     Pending   0          44h   app=webhook,kapp.
k14s.io/app=1626302357703846007,kapp.k14s.io/association=v1.9621e0a793b4e925077
dd557acedbcfe,pod-template-hash=6f5d979b7d,role=webhook,sources.tanzu.vmware.co
m/release=v0.23.0
```

3. Run `kubectl logs` and `kubectl describe`:

   ```
   kubectl logs PODNAME -n NAMESPACE
   kubectl describe pod PODNAME -n NAMESPACE
   ```

   Where:

   - `PODNAME` is found in the output of step 3, for example `webhook-6f5d979b7d-cxr9k`.
   - `NAMESPACE` is the namespace associated with the reconcile error, for example, `vmware-sources`.

   For example:

   ```
   $ kubectl logs webhook-6f5d979b7d-cxr9k -n vmware-sources

   $ kubectl describe pod webhook-6f5d979b7d-cxr9k  -n vmware-sources
   Events:
   Type     Reason            Age                 From               Message
   ----     ------            ----                ----               -------
   Warning  FailedScheduling  80s (x14 over 14m)  default-scheduler  0/1 nodes are
    available: 1 Insufficient cpu.
   ```

4. Review the output from the `kubectl logs` and `kubectl describe` commands and take further action.

   For this example of the webhook deployment, the output indicates that the scheduler does not have enough CPU to run the pod. In this case, the solution is to add nodes or CPU cores to the cluster. If you are using Tanzu Mission Control (TMC), increase the number of workers in the node pool to three or more through the TMC UI. See Edit a Node Pool, in the TMC documentation.

Cloud Native Runtimes Installation Fails with Existing Contour Installation

Symptom

You see the following error message when you run the install script:

```
Could not proceed with installation. Refer to Cloud Native Runtimes documentation for
details on how to utilize an existing Contour installation. Another app owns the custo
m resource definitions listed below.
```

Solution

Follow the procedure in Install Cloud Native Runtimes on a Cluster with Your Existing Contour

Instances to resolve the issue.

Verifying Your Installation

This topic tells you how to verify your Cloud Native Runtimes, commonly known as CNR, installation. You can verify that your Cloud Native Runtimes installation was successful by testing Knative Serving, Knative Eventing, and TriggerMesh Sources for Amazon Web Services (SAWS).

Prerequisites

1. Create a namespace and environment variable where you want to create Knative services. Run:

   Note: This step covers configuring a namespace to run Knative services. If you rely on a SupplyChain to deploy Knative services into your cluster, skip this step because namespace configuration is covered in Set up developer namespaces to use installed packages. Otherwise, you must complete the following steps for each namespace where you create Knative services.

   ```
   export WORKLOAD_NAMESPACE='cnr-demo'
   kubectl create namespace ${WORKLOAD_NAMESPACE}
   ```

2. Configure a namespace to use Cloud Native Runtimes. If during the TAP installation you relocated images to another registry, you must grant service accounts that run Knative services using Cloud Native Runtimes access to the image pull secrets. This includes the `default` service account in a namespace, which is created automatically but not associated with any image pull secrets. Without these credentials, attempts to start a service fail with a timeout and the pods report that they are unable to pull the `queue-proxy` image.

   1. Create an image pull secret in the namespace Knative services will run and fill it from the `tap-registry` secret mentioned in Add the Tanzu Application Platform package repository. Run the following commands to create an empty secret and annotate it as a target of the secretgen controller:

      ```
      kubectl create secret generic pull-secret --from-literal=.dockerconfigjso
      n={} --type=kubernetes.io/dockerconfigjson -n ${WORKLOAD_NAMESPACE}

      kubectl annotate secret pull-secret secretgen.carvel.dev/image-pull-secre
      t="" -n ${WORKLOAD_NAMESPACE}
      ```

   2. After you create a `pull-secret` secret in the same namespace as the service account, run the following command to add the secret to the service account:

      ```
      kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "
      pull-secret"}]}' -n ${WORKLOAD_NAMESPACE}
      ```

3. Verify that a service account is correctly configured by running:

   ```
   kubectl describe serviceaccount default -n ${WORKLOAD_NAMESPACE}
   ```

   For example:

   ```
   kubectl describe sa default -n cnr-demo
   Name:                default
   Namespace:           cnr-demo
   Labels:              <none>
   Annotations:         <none>
   Image pull secrets:  pull-secret
   Mountable secrets:   default-token-xh6p4
   Tokens:              default-token-xh6p4
   Events:              <none>
   ```

   The service account has access to the `pull-secret` image pull secret.

Verify that `STATUS` is `Reconcile succeeded`.

Verify Installation of Knative Serving, Knative Eventing, and TriggerMesh SAWS

To verify the installation of Knative Serving, Knative Eventing, and Triggermesh SAWS:

1. Create a namespace and environment variable for the test. Run:

   ```
   export WORKLOAD_NAMESPACE='cnr-demo'
   kubectl create namespace ${WORKLOAD_NAMESPACE}
   ```

2. Verify installation of the components that you intend to use:

To test...	Create...	For instructions, see...
Knative Serving	a test service	Verifying Knative Serving
Knative Eventing	a broker, a producer, and a consumer	Verifying Knative Eventing
TriggerMesh SAWS	an AWS source and trigger it	Verifying TriggerMesh SAWS

3. Delete the namespace that you created for the demo. Run:

   ```
   kubectl delete namespaces ${WORKLOAD_NAMESPACE}
   unset WORKLOAD_NAMESPACE
   ```

Verifying Knative Serving for Cloud Native Runtimes

This topic tells you how to verify that Knative Serving was successfully installed for Cloud Native Runtimes, commonly known as CNR.

About Verifying Knative Serving

To verify that Knative Serving was successfully installed, create an example Knative service and test it.

The procedure below shows you how to create an example Knative service using the Cloud Native Runtimes sample app, `hello-yeti`. This sample is custom built for Cloud Native Runtimes and is stored in the VMware Harbor registry.

Note: If you do not have access to the Harbor registry, you can use the Hello World - Go sample app in the Knative documentation.

Prerequisites

Before you verify Knative Serving, you must have a namespace where you want to deploy Knative services. This namespace will be referred as `${WORKLOAD_NAMESPACE}` in this tutorial. See step 1 of Verifying Your Installation for more information.

Test Knative Serving

To create an example Knative service and use it to test Knative Serving:

1. If you are verifying on Tanzu Mission Control or vSphere 7.0 with Tanzu, then create a role binding in the `${WORKLOAD_NAMESPACE}` namespace. Run:

```
kubectl apply -n "${WORKLOAD_NAMESPACE}" -f - << EOF
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: ${WORKLOAD_NAMESPACE}-psp
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cnr-restricted
subjects:
- kind: Group
  name: system:serviceaccounts:${WORKLOAD_NAMESPACE}
EOF
```

2. Deploy the sample app using the `kn` CLI. Run:

```
kn service create hello-yeti -n ${WORKLOAD_NAMESPACE} \
  --image projects.registry.vmware.com/tanzu_serverless/hello-yeti@sha256:17d64
0edc48776cfc604a14fbabf1b4f88443acc580052eef3a753751ee31652 --env TARGET='hello
-yeti'
```

If you are verifying on Tanzu Mission Control or vSphere 7.0 with Tanzu, then add `--user 1001` to the command above to run it as a non-root user.

3. Run one of the following commands to retrieve the external address for your ingress, depending on your IaaS:

IaaS:	
	◦ Tanzu Kubernetes Grid on AWS
	◦ Tanzu Mission Control on AWS
	◦ Amazon Elastic Kubernetes Service

Run:

```
export EXTERNAL_ADDRESS=$(kubectl get service envoy -n tanzu-system-ingre
ss \
-output ‘jsonpath={.status.loadBalancer.ingress[0].hostname}’)
```

IaaS:

- vSphere 7.0 on Tanzu
- Tanzu Kubernetes Grid on vSphere/Azure/GCP
- Tanzu Kubernetes Grid Integrated Edition
- Tanzu Mission Control on vSphere
- Azure Kubernetes Service
- Google Kubernetes Engine

Run:

```
export EXTERNAL_ADDRESS=$(kubectl get service envoy -n tanzu-system-ingre
ss \
-output ‘jsonpath={.status.loadBalancer.ingress[0].ip}’)
```

IaaS: Local Kubernetes Cluster:

- Docker desktop
- Minikube

Run:

```
export EXTERNAL_ADDRESS=‘localhost:8080’
```

And, on another terminal, set up port forwarding. Run:

```
kubectl -n tanzu-system-ingress port-forward svc/envoy 8080:80
```

4. Connect to the app. Run:

```
curl -H "Host: hello-yeti.${WORKLOAD_NAMESPACE}.example.com" ${EXTERNAL_ADDRESS
}
```

If external DNS is correctly configured, you can also visit the URL in a web browser.

On success, you see a reply from our mascot, Carl the Yeti.

Delete the Example Knative Service

After verifying your serving installation, delete the example Knative service and unset the environment variable:

1. Run:

```
kn service delete hello-yeti -n ${WORKLOAD_NAMESPACE}
unset EXTERNAL_ADDRESS
```

2. If you created port forwarding in step 4 above, then terminate that process.

Verify Knative Eventing

This topic tells you how to verify that Knative Eventing was successfully installed with Cloud Native Runtimes, commonly known as CNR.

Note: The Knative eventing functionality is in beta. VMware does not recommend using Knative eventing functionality in a production environment.

About Verifying Knative Eventing

You can verify Knative Eventing by setting up a broker, creating a producer, and creating a consumer. If your installation was successful, you can create a test eventing workflow and see that the events appear in the logs.

You can use either an in-memory broker or a RabbitMQ broker to verify Knative Eventing:

- **RabbitMQ broker**: Using a RabbitMQ broker to verify Knative Eventing is a scalable and reliable way to verify your installation. Verifying with RabbitMQ uses methods similar to production environments.
- **In-memory broker**: Using an in-memory broker is a fast and lightweight way to verify that the basic elements of Knative Eventing are installed. An in-memory broker is not meant for production environments or for use with apps that you intend to take to production.

Prerequisites

Before you verify Knative Eventing, you must:

- Have a namespace where you want to deploy Knative services. This namespace will be referred as `${WORKLOAD_NAMESPACE}` in this tutorial. See step 1 of Verifying Your Installation for more information.
- Create the following role binding in the `${WORKLOAD_NAMESPACE}` namespace. Run:

  ```
  kubectl apply -f - << EOF
  apiVersion: rbac.authorization.k8s.io/v1
  kind: RoleBinding
  metadata:
    name: ${WORKLOAD_NAMESPACE}-psp
    namespace: ${WORKLOAD_NAMESPACE}
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: cnr-restricted
  subjects:
  - kind: Group
    name: system:serviceaccounts:${WORKLOAD_NAMESPACE}
  EOF
  ```

Prepare the RabbitMQ Environment

If you are using a RabbitMQ broker to verify Knative Eventing, follow the procedure in this section. If you are verifying with the in-memory broker, skip to Verify Knative Eventing.

To prepare the RabbitMQ environment before verifying Knative Eventing:

1. Set up the RabbitMQ integration as described in Integrating RabbitMQ with Cloud Native Runtimes.

2. On the Kubernetes cluster where Cloud Native Runtimes is installed, deploy a RabbitMQ cluster using the RabbitMQ Cluster Operator by running:

   ```
   kubectl apply -f - << EOF
   apiVersion: rabbitmq.com/v1beta1
   kind: RabbitmqCluster
   metadata:
     name: my-rabbit
     namespace: ${WORKLOAD_NAMESPACE}
   spec:
     replicas: 1
     override:
       statefulSet:
         spec:
           template:
             spec:
               securityContext: {}
               containers: []
               initContainers:
               - name: setup-container
                 securityContext:
                   runAsUser: 999
                   runAsGroup: 999
   EOF
   ```

   Note: The `override` section can be omitted if your cluster allows containers to run as `root`.

Verify Knative Eventing

To verify installation of Knative Eventing create and test a broker, procedure, and consumer in the `${WORKLOAD_NAMESPACE}` namespace:

1. Create a broker.

   For the RabbitMQ broker. Run:

   ```
   kubectl apply -f - << EOF
   apiVersion: eventing.knative.dev/v1
   kind: Broker
   metadata:
     name: default
     namespace: ${WORKLOAD_NAMESPACE}
     annotations:
       eventing.knative.dev/broker.class: RabbitMQBroker
   spec:
     config:
       apiVersion: rabbitmq.com/v1beta1
       kind: RabbitmqCluster
       name: my-rabbit
       namespace: ${WORKLOAD_NAMESPACE}
   EOF
   ```

   For the in-memory broker. Run:

```
kubectl create -f - <<EOF
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: default
  namespace: ${WORKLOAD_NAMESPACE}
EOF
```

2. Create a consumer for the events. Run:

```
cat <<EOF | kubectl create -f -
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-releases/knative.dev/eventing-contrib/cmd/event
_display
EOF
```

3. Create a trigger. Run:

```
kubectl apply -f - << EOF
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: event-display
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  broker: default
  subscriber:
    ref:
     apiVersion: serving.knative.dev/v1
     kind: Service
     name: event-display
     namespace: ${WORKLOAD_NAMESPACE}
EOF
```

4. Create a producer. Run:

```
cat <<EOF | kubectl create -f -
apiVersion: sources.knative.dev/v1
kind: PingSource
metadata:
  name: test-ping-source
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  schedule: "*/1 * * * *"
  data: '{"message": "Hello Eventing!"}'
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
      namespace: ${WORKLOAD_NAMESPACE}
EOF
```

5. Verify that the event appears in your consumer logs. Run:

```
kubectl logs -l serving.knative.dev/service=event-display -c user-container -n
${WORKLOAD_NAMESPACE} --since=10m --tail=50
```

Delete the Eventing Resources

After verifying your serving installation, clean up by deleting the resources used for the test:

1. Delete the eventing resources:

```
kubectl delete pingsource/test-ping-source -n ${WORKLOAD_NAMESPACE}
kubectl delete trigger/event-display -n ${WORKLOAD_NAMESPACE}
kubectl delete kservice/event-display -n ${WORKLOAD_NAMESPACE}
kubectl delete broker/default -n ${WORKLOAD_NAMESPACE}
```

2. If you created a RabbitMQ cluster:

```
kubectl delete rabbitmqcluster/my-rabbit -n ${WORKLOAD_NAMESPACE}
```

3. Delete the role binding:

```
kubectl delete rolebinding/${WORKLOAD_NAMESPACE}-psp -n ${WORKLOAD_NAMESPACE}
```

Verifying TriggerMesh SAWS for Cloud Native Runtimes

This topic tells you how to verify that TriggerMesh Sources for Amazon Web Services (SAWS) was installed successfully for Cloud Native Runtimes, commonly known as CNR.

Overview

TriggerMesh SAWS allows you to consume events from your AWS services and send them to workloads running in your cluster.

Cloud Native Runtimes includes an installation of the Triggermesh SAWS controller and CRDs. You can find the controller in the `triggermesh` namespace.

For general information about TriggerMesh SAWS, see aws-event-sources in GitHub.

The procedure below shows you how to test TriggerMesh SAWS using the example of an event source for Amazon CodeCommit. If you want to test using a different AWS service, see samples in GitHub. The basic steps are the same, regardless of the AWS service you choose: create a broker, trigger, and consumer and then test.

Prerequisites

Before you verify TriggerMesh SAWS with AWS CodeCommit, you must have:

- An AWS service account
- An AWS CodeCommit repository with push and pull access
- Have a namespace where you want to deploy Knative services. This namespace will be referred as `${WORKLOAD_NAMESPACE}` in this tutorial. See step 1 of Verifying Your Installation for more information.

Verify TriggerMesh SAWS

To verify TriggerMesh SAWS with AWS CodeCommit:

1. Create a broker:

```
kubectl apply -f - << EOF
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: broker
  namespace: ${WORKLOAD_NAMESPACE}
EOF
```

2. Create a trigger:

```
kubectl apply -f - << EOF
---
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: trigger
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  broker: broker
  subscriber:
    ref:
     apiVersion: serving.knative.dev/v1
     kind: Service
     name: consumer
     namespace: ${WORKLOAD_NAMESPACE}
EOF
```

3. Create a consumer:

```
kubectl apply -f - << EOF
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: consumer
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-releases/knative.dev/eventing-contrib/cmd/event
_display
EOF
```

4. Add an AWS service account secret:

   ```
   kubectl -n ${WORKLOAD_NAMESPACE} create secret generic awscreds \
   --from-literal=aws_access_key_id=${AWS_ACCESS_KEY_ID} \
   --from-literal=aws_secret_access_key=${AWS_SECRET_ACCESS_KEY}
   ```

   Where:

   - `AWS_ACCESS_KEY_ID` is the AWS access key ID for your AWS service account.
   - `AWS_SECRET_ACCESS_KEY` is your AWS access key for your AWS service account.

5. Create the AWSCodeCommitSource:

   ```
   kubectl apply -f - << EOF
   apiVersion: sources.triggermesh.io/v1alpha1
   kind: AWSCodeCommitSource
   metadata:
     name: source
     namespace: ${WORKLOAD_NAMESPACE}
   spec:
     arn: ARN
     branch: BRANCH

     eventTypes:
       - push
       - pull_request

     credentials:
       accessKeyID:
         valueFromSecret:
           name: awscreds
           key: aws_access_key_id
       secretAccessKey:
         valueFromSecret:
           name: awscreds
           key: aws_secret_access_key

     sink:
       ref:
         apiVersion: eventing.knative.dev/v1
         kind: Broker
         name: broker
         namespace: ${WORKLOAD_NAMESPACE}
   EOF
   ```

   Where:

   - `ARN` is Amazon Resource Name (ARN) of your CodeCommit repository. For example, `arn:aws:codecommit:eu-central-1:123456789012:triggermeshtest`.
   - `BRANCH` is the branch of your CodeCommit repository that you want the trigger to watch. For example, `main`.

6. Patch the `awscodecommitsource-adapter` service account to pull images from the private registry using the `tap-registry` secret, created during the TAP installation. Note that the `awscodecommitsource-adapter` service account was created on the previous step during the

creation of `AWSCodeCommitSource`.

```
kubectl patch serviceaccount -n ${WORKLOAD_NAMESPACE} awscodecommitsource-adapt
er -p '{"imagePullSecrets": [{"name": "tap-registry"}]}'
```

Note: It may be necessary to delete the current `awscodecommitsource-source` **Pod** so a new pod is created with the new `imagePullSecrets`.

7. Create an event by pushing a commit to your CodeCommit repository.
8. Watch the consumer logs to see that the event appears after a minute:

```
kubectl logs -l serving.knative.dev/service=consumer -c user-container -n ${WOR
KLOAD_NAMESPACE} --since=10m --tail=50
```

Upgrading Cloud Native Runtimes

This topic tells you how to upgrade Cloud Native Runtimes for Tanzu to the latest version.

New versions of Cloud Native Runtimes are available from the Tanzu Application Platform package repository, and can be upgraded to as part of upgrading Tanzu Application Platform as a whole.

Prerequisites

The following prerequisites are required to upgrade Cloud Native Runtimes:

- An updated Tanzu Application Platform package repository with the version of Cloud Native Runtimes you wish to upgrade to. For more information, see the documentation on adding a new package repository.

Upgrade Cloud Native Runtimes

To upgrade the Cloud Native Runtimes PackageInstall specifically, run:

```
tanzu package installed update cloud-native-runtimes -p cnrs.tanzu.vmware.com -v CNR-V
ERSION --values-file cnr-values.yaml -n tap-install
```

Where `CNR-VERSION` is the latest version of Cloud Native Runtimes available as part of the new Tanzu Application Platform package repository.

Uninstalling Cloud Native Runtimes

This topic tells you how to uninstall Cloud Native Runtimes.

Overview

Cloud Native Runtimes is part of the Tanzu Application Platform package repository. For information on uninstalling the entire Tanzu Application Platform package repository, see the Tanzu Application Platform uninstall documentation.

Uninstall

To uninstall Cloud Native Runtimes specifically:

1. Delete the installed package:

    ```
    tanzu package installed delete cloud-native-runtimes --namespace tap-install
    ```