PowerDNS Cloud Control

Overview

Apr 24, 2024

Release 2.5.2

Contents

1 Cloud Control

Cloud Control facilitates orchestration, management & monitoring of PowerDNS products in Kubernetes deployments. PowerDNS products supported in this version are:

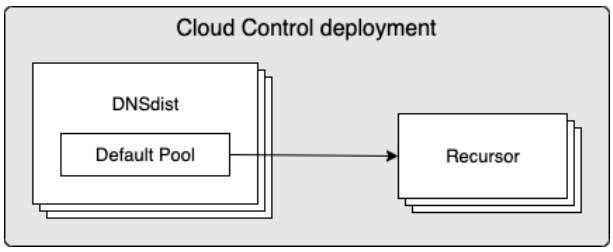
- PowerDNS DNSdist - A DNS, DoS and abuse-aware loadbalancer that brings out the best possible performance in any DNS deployment.
- PowerDNS Recursor - A high-performing, low latency DNS resolver.
- PowerDNS Authoritative Server - A versatile authoritative server for hosting domain names.
- PowerDNS ZoneControl - A graphical web-based interface for managing domains on the PowerDNS Authoritative Server.

In additions, several PowerDNS Add-ons are available to deploy alongside above products:

- PowerDNS dstore-dist - Component which can send protobuf messages to different destinations and acts as a distributor of the protobuf messages generated by PowerDNS Recursor and DNSdist
- PowerDNS ixfrdist - Component which transfers zones from an authoritative server and re-serves these zones over AXFR and IXFR

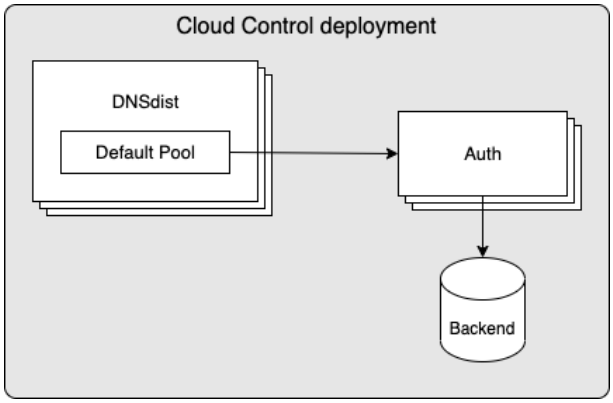
1.1 Simple deployment - Recursor

Cloud Control can be used to roll out a set of Recursor instances, with a set of DNSdists in front. In the below diagram you can see a set of DNSdist instances, with a default pool sending all traffic to a set of Recursor instances:



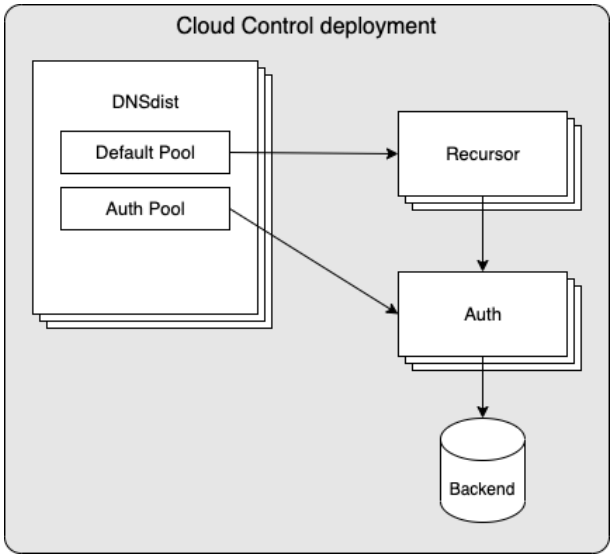
1.2 Simple deployment - Auth

Cloud Control can be used to roll out a set of Auth instances, with a set of DNSdists in front. In the below diagram you can see a set of Auth instances, with a default pool sending all traffic to a set of Auth instances:



1.3 Complex deployment

In a more complex deployment you can deploy both Recursor & Auth instances, having DNSdist using multiple pools to send traffic to the different instances based on the incoming queries/traffic. In the below example you see a setup where both Recursor & Auth are deployed, with DNSdist using rules to send some traffic to Auth, while defaulting to sending queries to Recursor. The Recursor > Auth arrow signifies the use of forward zones, which instructs the Recursor to forward queries for certain zones to Auth.



1.4 Rules & Actions

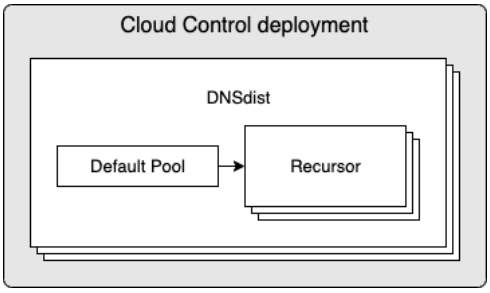
Deciding which traffic to send to each pool is handled by DNSdist's packet policies, which offers a mechanism to define rules and corresponding actions. In the context of the above diagram, such rules & actions could be:

Rule	Action
'QPS' of requests from the sender has exceeded a certain value	Answer request with 'REFUSED'
'Opcode' of request is 'Notify'	let Auth pool handle the request
'Qtype' of request is 'AXFR'	let Auth pool handle the request

Note: By default, all requests will be handled by the 'Default Pool'

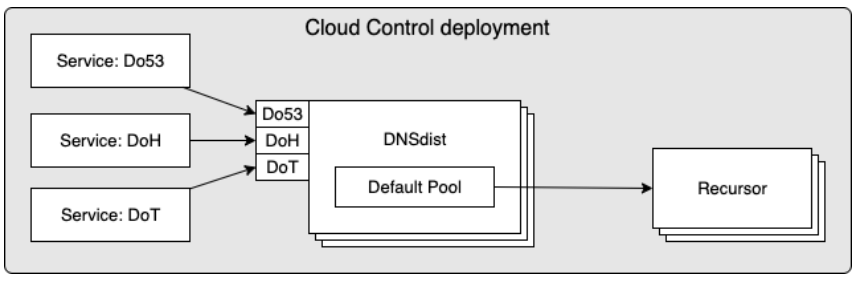
1.5 DNSdist with co-hosted Recursors

In a high load environment, the overhead on Kubernetes network components from the DNSdist to Recursor traffic can potentially become a bottleneck and/or lead to unacceptable latency. For these scenarios it is possible to have 1 or more Recursor instances running within the same Pod as DNSdist. Such a deployment would look as follows:



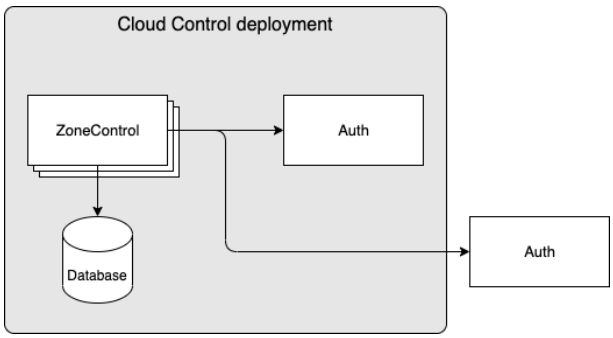
1.6 DNSdist with DoH and/or DoT listeners

Inbound traffic to DNSdist is supported not only via the standard UDP & TCP over port 53 (Do53), but also via DoH and DoT. When configured, you can have a deployment that looks as follows:



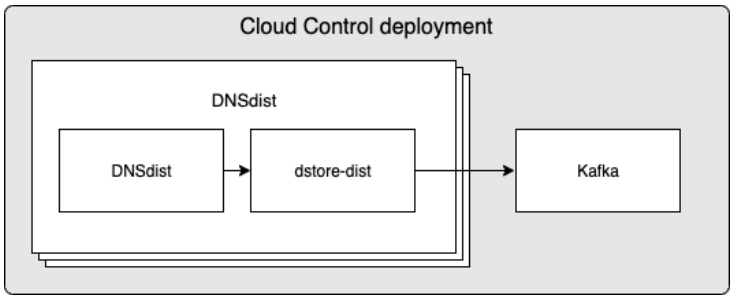
1.7 ZoneControl deployment

Cloud Control can be used to roll out a set of ZoneControl instances and configure the endpoints of Auth instances that it should be able to manage. In the below diagram you can see a set of ZoneControl instances, configured to manage 2 sets of Auth instances, one within the same Cloud Control deployment and another in a separate deployment:

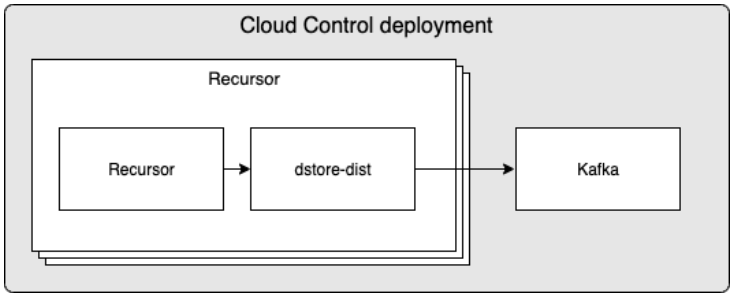


1.8 Dstore-dist

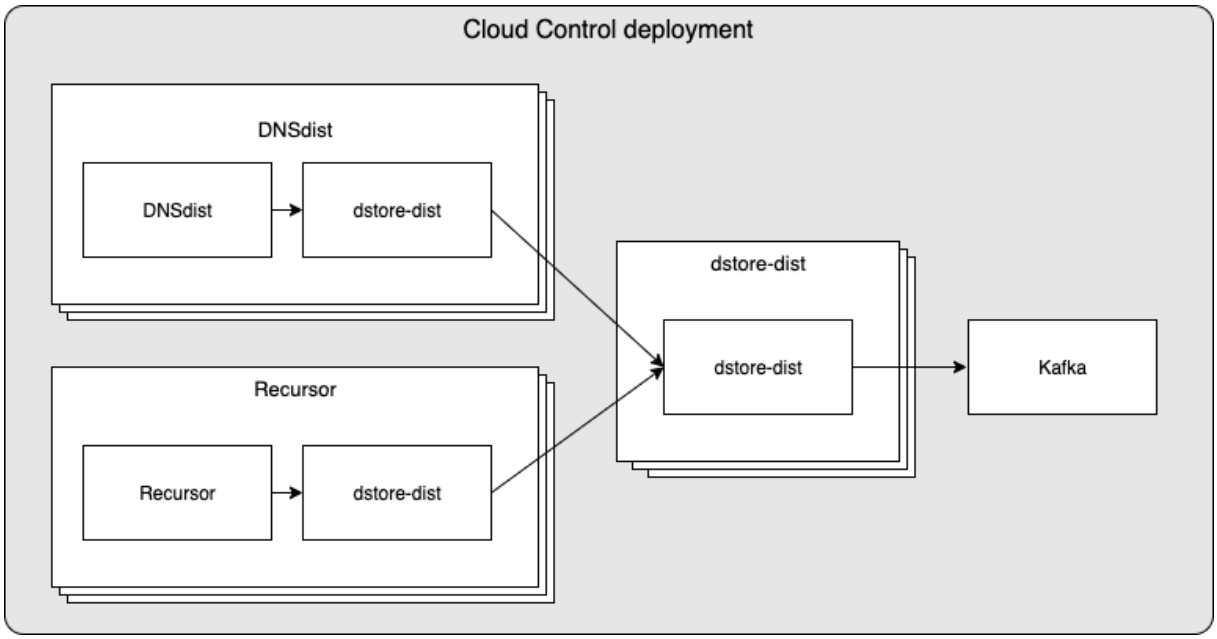
To enable the distribution and filtering of protobuf messages generated by DNSdist and/or Recursor, dstore-dist can run as a sidecar in DNSdist and Recursor. In the below diagram you can see a set of DNSdist instances with a dstore-dist sidecar configured to distribute messages to a Kafka deployment.



When deployed as a sidecar in a Recursor pod with a Kafka destination:

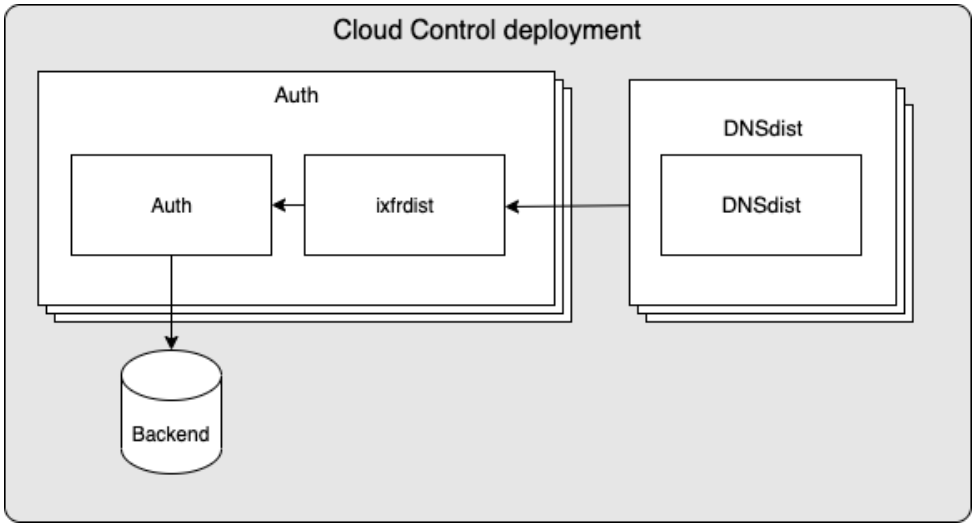


Alternatively, dstore-dist can also be deployed as a standalone set of Pods, to perform additional aggregation, filtering and distribution of protobuf messages sent by multiple sets of DNSdist and/or Recursor pods:



1.9 Ixfrdist

Ixfrdist can be enabled as a sidecar in a set of Auth pods and exposed via DNSdist. A basic deployment of Auth with ixfrdist is shown in the below diagram:



2 Cloud Control on Kubernetes

Cloud Control provides a Helm Chart which allows for the definition & configuration of the following:

- **auth** - Definition of a set of PowerDNS Authoritative Server instances and corresponding configuration
- **dnsdist** - Definition of a set of PowerDNS DNSdist instances and corresponding configuration
- **recursor** - Definition of a set of PowerDNS Recursor instances and corresponding configuration
- **resolver** - Definition of a set of external resolver endpoints
- **ruleset** - Definition of a set of rules which can be applied to DNSdist instances
- **zonecontrol** - Definition of a set of PowerDNS ZoneControl instances and corresponding configuration

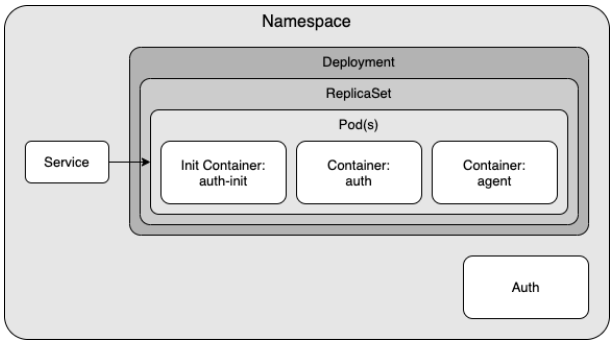
The following sections discuss each in more detail.

2.1 Auth

For each `auth` defined in the input to the Helm Chart, objects of the following types (aka `kind` in Kubernetes terminology) will be created in Kubernetes:

Kind	API Group	Description
Auth	`cloudcontrol.powerdns.com`	Object which holds configuration of the Auth instances
Deployment	`core`	Deployment of Auth pods (including ReplicaSet)
Service	`core`	Service which can be discovered by DNSdist & Recursor agents to direct traffic to the Auth pods

When an `auth` instance is configured using the Helm Chart, it will deploy the following to Kubernetes:



As the diagram shows an Auth pod will consist of 2 containers + 1 init container:

- **auth-init** - Prepares configuration for Auth.
- **auth** - Container running PowerDNS Authoritative Server.
- **agent** - Contains an agent that watches several kinds of objects in Kubernetes within the namespace. If any watched objects are created/updated/removed, the agent will sync any corresponding configuration items to the running Auth instance. The agent is described in detail in the next chapter.

2.1.1 Auth agent

The Auth agent is responsible for keeping the configuration of the running Auth process in sync with the desired configuration. If any configuration changes are needed, the agent will attempt to synchronize them without restarting the Auth process.

Items which are watched by the agent are:

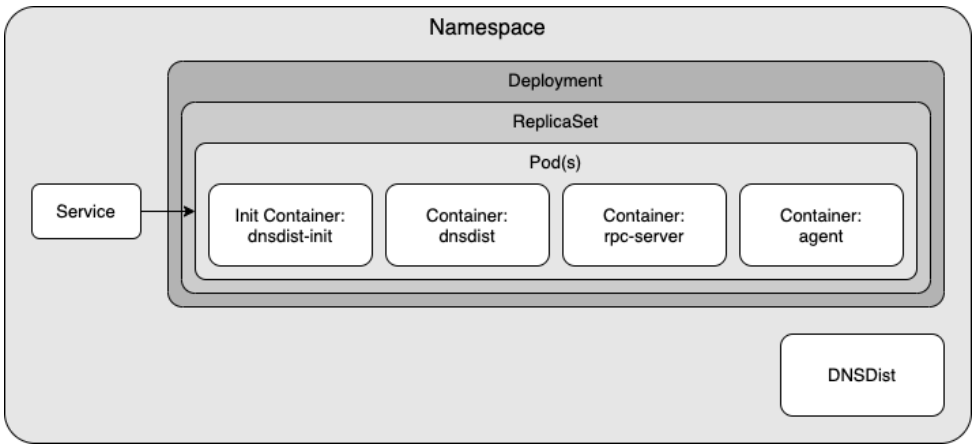
Kind	Purpose
Auth	The object which contains the configuration details for an Auth deployment. If any updates are detected the agent will attempt to update the configuration of Auth without having to restart it.
Pod	The agent watches the pod which it is a part of. Particularly the statuses of each container inside the pod are observed, to ensure the agent can synchronize an Auth instance again if it's container was recycled for any reason.
GeoIP zonefiles	The agent watches for changes in the GeoIP zonefiles that can be configured for the GeoIP backend using the *domains* attribute. If any changes are detected the agent will instruct Auth to reload the zonefiles.

2.2 DNSdist

For each `dnsdist` defined in the input to the Helm Chart, objects of the following types (`kind` in Kubernetes) will be created in Kubernetes:

Kind	API Group	Description
DNSDist	`cloudcontrol.powerdns.com`	Object which holds configuration of the DNSdist instances
Deployment	`core`	Deployment of DNSdist pods (including ReplicaSet)
Service	`core`	Service which can be used to direct traffic to the DNSdist pods

When a `dnsdist` instance is configured using the Helm Chart, it will deploy the following to Kubernetes:



As the diagram shows a DNSdist pod will consist of 3 containers + 1 init container:

- **dnsdist-init** - Prepares configuration for dnsdist.
- **dnsdist** - Container running PowerDNS DNSdist.
- **rpc-server** - Runs an API that is responsible for handling JSON messages over HTTP from the agent and forwarding them to dnsdist.
- **agent** - Contains an agent that watches several kinds of objects in Kubernetes within the namespace. If any watched objects are created/updated/removed, the agent will sync any corresponding configuration items to the running dnsdist instance. The agent is described in detail in the next chapter.

2.2.1 DNSdist agent

The DNSdist agent is responsible for keeping the configuration of the running DNSdist process in sync with the desired configuration. If any configuration changes are needed, the agent will attempt to synchronize them without restarting the DNSdist process. These configuration changes range from performance parameters defined in the DNSDist object to adjusting server pools according to changes observed in Recursor, Auth & Resolver deployments.

Items which are watched by the agent are:

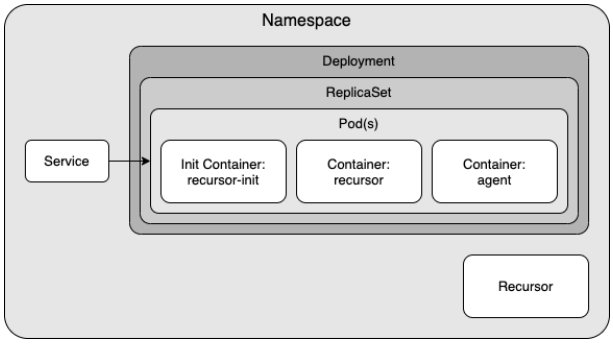
Kind	Purpose
DNSDist	The object which contains the configuration details for a DNSdist deployment. If any updates are detected the agent will attempt to update the configuration of DNSdist without having to restart it.
Pod	The agent watches the pod which it is a part of. Particularly the statuses of each container inside the pod are observed, to ensure the agent can synchronize a DNSdist instance again if it's container was recycled for any reason.
DNSDistRule	Any rule objects which match the RuleSelector on the DNSDist object are watched and synchronized to the DNSdist process if needed. Any new rules that match the RuleSelector are also applied as soon as they are observed by the agent.
Service & Endpoints	The agent watches for changes in the Endpoints of any Service objects which match the ServiceSelector of the DNSDist object. This allows the agent to discover the servers that should be part of the pool(s) in DNSdist and works for Recursor, Auth & Resolver deployments.

2.3 Recursor

For each `recursor` defined in the input to the Helm Chart, objects of the following types (aka `kind` in Kubernetes terminology) will be created in Kubernetes:

Kind	API Group	Description
Recursor	`cloudcontrol.powerdns.com`	Object which holds configuration of the Recursor instances
Deployment	`core`	Deployment of Recursor pods (including ReplicaSet)
Service	`core`	Service which can be discovered by DNSdist agents to direct traffic to the Recursor pods

When a `recursor` instance is configured using the Helm Chart, it will deploy the following to Kubernetes:



As the diagram shows a Recursor pod will consist of 2 containers + 1 init container:

- **recursor-init** - Prepares configuration for Recursor.
- **recursor** - Container running PowerDNS Recursor.

- **agent** - Contains an agent that watches several kinds of objects in Kubernetes within the namespace. If any watched objects are created/updated/removed, the agent will sync corresponding configuration items to the running Recursor instance. The agent is described in detail in the next chapter.

2.3.1 Recursor agent

The Recursor agent is responsible for keeping the configuration of the running Recursor process in sync with the desired configuration. If any configuration changes are needed, the agent will attempt to synchronize them without restarting the Recursor process.

Items which are watched by the agent are:

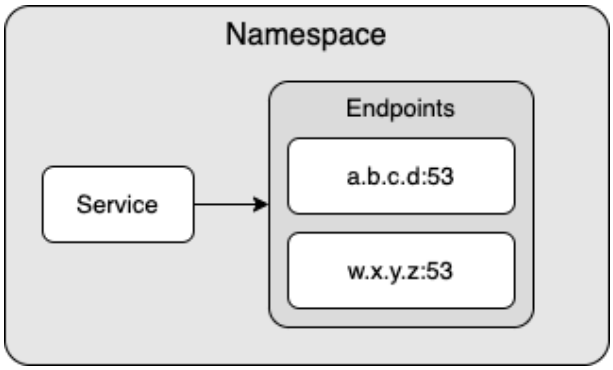
Kind	Purpose
Recursor	The object which contains the configuration details for a Recursor deployment. If any updates are detected the agent will attempt to update the configuration of Recursor without having to restart it.
Pod	The agent watches the pod which it is a part of. Particularly the statuses of each container inside the pod are observed, to ensure the agent can synchronize a Recursor instance again if it's container was recycled for any reason.
Service & Endpoints	The agent watches for changes in the Endpoints of any Service objects which match the ServiceSelector of the Recursor object. This allows the agent to discover the endpoints that should be part of the forward zones in Recursor.

2.4 Resolver

For each `resolver` defined in the input to the Helm Chart, objects of the following types (aka `kind` in Kubernetes terminology) will be created in Kubernetes:

Kind	API Group	Description
Endpoints	`core`	Object that holds each IP:port combination defined for the resolver
Service	`core`	Service which can be discovered by DNSdist & Recursor agents to direct traffic to the resolver's endpoints

When a `resolver` instance is configured using the Helm Chart, it will deploy the following to Kubernetes:



2.5 Ruleset

For each `ruleset` defined in the input to the Helm Chart, objects of the following types (aka `kind` in Kubernetes terminology) will be created in Kubernetes:

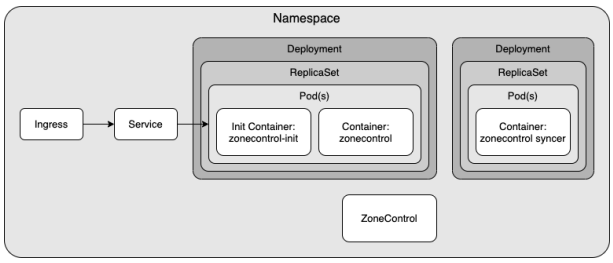
Kind	API Group	Description
DNSDistRule	`cloudcontrol.powerdns.com`	Object which holds configuration of a set of rules which can be discovered by DNSdist agents and applied to DNSdist without restarting

2.6 ZoneControl

For each `zonecontrol` defined in the input to the Helm Chart, objects of the following types (aka `kind` in Kubernetes terminology) will be created in Kubernetes:

Kind	API Group	Description
ZoneControl	`cloudcontrol.powerdns.com`	Object which holds configuration of the ZoneControl instances
Deployment	`core`	Deployment of ZoneControl pods (including ReplicaSet)
Service	`core`	Service which can be used to expose ZoneControl instances
Ingress	`networking.k8s.io`	Ingress which can be used to expose ZoneControl instances outside of the cluster via HTTP(S)

When a `zonecontrol` instance is configured using the Helm Chart, it will deploy the following to Kubernetes:



As the diagram shows a ZoneControl instance will consist of a ZoneControl deployment with 1 container + 1 init container and a ZoneControl Syncer deployment. The ZoneControl Deployment contains the GUI and can have multiple replicas, while the ZoneControl Syncer deployment has a single replica and is used to synchronise configuration changes to the ZoneControl instances.

- **zonecontrol-init** - Prepares configuration for ZoneControl.
- **zonecontrol** - Container running PowerDNS ZoneControl.
- **syncer** - Contains an operator that watches ZoneControl objects in Kubernetes within the namespace. If any watched objects are updated, the syncer will synchronise any corresponding configuration items to the running ZoneControl instances.

2.6.1 ZoneControl Syncer

The ZoneControl Syncer agent is responsible for keeping the configuration of the running ZoneControl processes in sync with the desired configuration. If any configuration changes are needed, the syncer will attempt to synchronize them without restarting the ZoneControl process.

Items which are watched by the syncer are:

Kind	Purpose
ZoneControl	The object which contains the configuration details for a ZoneControl deployment. If any updates are detected the syncer will attempt to update the configuration of ZoneControl without having to restart it.

3 Cloud Control on OpenShift

Cloud Control provides an OpenShift compatibility mode which will make sure default settings adhere to a set compatible with policies present on a default OpenShift cluster.

To enable this compatibility mode, ensure the following is present in your Helm values overrides:

```
global:
  openshift:
    enabled: true
```

When enabled, Cloud Control will be configured with the following:

3.1 podSecurityContext

All pods will have a 'podSecurityContext' enforcing 'runAsNonRoot' and nothing else (UID, GID & fsGroup will all be auto-assigned at runtime)

To adjust the podSecurityContext configured for pods in the OpenShift compatibility mode, you can use the following configuration:

```
global:
  openshift:
    enabled: true
    podSecurityContext:
      <Contents of Pod Security Context>
```

4 Helm Charts

CloudControl has several Helm Charts available to manage & deploy PowerDNS environments to Kubernetes. The main charts are as follows:

- **powerdns-crds**: Chart to install/upgrade the CloudControl CRDs
- **powerdns**: Chart to install/upgrade CloudControl deployments
- **powerdns-operators**: Chart that allows for installation of optional operators

4.1 Helm Chart: powerdns-crds

This chart is used to deploy & upgrade the CRDs used by PowerDNS CloudControl deployments. Having these CRDs deployed to the cluster is a prerequisite to being able to install an environment using the *powerdns* Helm chart.

Scope of objects: cluster-scoped, requires cluster privileges on *CRD* objects.

4.2 Helm Chart: powerdns

This chart is used to deploy & upgrade the PowerDNS CloudControl deployments.

Scope of objects: namespace-scoped, does not require any cluster privileges.

4.3 Helm Chart: powerdns-operators

This optional chart is used to deploy auxiliary Kubernetes Operators that may be used to easily deploy additional components to support CloudControl PowerDNS deployments. Due to the complexity of persistent storage in a Kubernetes environment we recommend you leverage any existing facilities you may have to provide the services offered by this chart instead of using this chart to deploy them.

Currently contains Operators for:

- **Postgres**: Allows for automated creation of Postgres databases, potentially used by Auth & ZoneControl deployments.

Scope of objects: cluster-scoped & namespace-scoped, requires cluster privileges on *CRD*, *ClusterRole* and *ClusterRoleBinding* objects.

OpenShift: The Postgres Operator is based on an opensource project, which is currently not compatible with OpenShift. OpenShift users will be unable to use this Operator and will need to provision their own Postgres databases.

5 Getting Started

5.1 Install Tools

You will need the following software on the machine from which you want to deploy CloudControl:

- Kubectl (Configured for your target Kubernetes cluster)
- Helm (3.8.0 or newer - https://helm.sh/docs/intro/install/)

5.2 Download Helm Charts

CloudControl Helm Charts are available on the Open-Xchange registry, located at: registry.open-xchange.com.

There are several methods for obtaining Helm Charts using Helm's CLI, in this chapter we are using a method that copies the chart locally to your filesystem prior to using it. Any Helm-supported method will work, but you will need to adjust the commands in this guide accordingly if you wish to utilise a different method.

First step will be to login to the OX registry (replace username & password with your OX registry credentials):

```
helm registry login registry.open-xchange.com --username=REGISTRY_USERNAME_HERE \
--password=REGISTRY_PASSWORD_HERE
```

Once helm has been logged in to the OX registry you can access the CloudControl Helm Charts. To pull the powerdns Helm Charts and unpack them to your current working directory use the following commands:

```
# Pull & unpack CRDs chart
helm pull oci://registry.open-xchange.com/cloudcontrol/powerdns-crds \
--version=2.5.2 --untar

# Pull & unpack Powerdns chart
helm pull oci://registry.open-xchange.com/cloudcontrol/powerdns \
--version=2.5.2 --untar

# Pull & unpack Operators chart (optional)
helm pull oci://registry.open-xchange.com/cloudcontrol/powerdns-operators \
--version=2.5.2 --untar
```

5.3 Install/Upgrade CloudControl CRDs

The CloudControl CRDs can be installed or upgraded using the *powerdns-crds* Helm Chart. While the chart only deploys cluster-scoped objects (CRDs), you need to provide a namespace to allow Helm to store the relevant information about this deployment. This ensures you can easily upgrade to a newer version in the future.

To install the CRDs with a Helm release name of 'pdnscrds' stored in a namespace 'pdnscrds':

```
helm install pdnscrds ./powerdns-crds --namespace pdnscrds
```

Note: you can add `--create-namespace` if the namespace does not exist yet and you have privileges to create it

---

Using kubectl you should now be able to see the corresponding Kubernetes objects created:

```
# Kubectl command to show CRD objects (filtered for 'cloudcontrol')
kubectl get crd | grep cloudcontrol

# Kubectl output
dnsdistrules.cloudcontrol.powerdns.com              <timestamp of creation>
zonecontrols.cloudcontrol.powerdns.com              <timestamp of creation>
auths.cloudcontrol.powerdns.com                     <timestamp of creation>
recursors.cloudcontrol.powerdns.com                 <timestamp of creation>
dnsdists.cloudcontrol.powerdns.com                  <timestamp of creation>
```

Result should be a list of CRDs within the *cloudcontrol.powerdns.com* group as shown above.

To upgrade the CRDs, you can use the *helm upgrade* command. For example:

```
helm upgrade pdnscrds ./powerdns-crds --namespace pdnscrds
```

Note: Since the Helm upgrade command needs to have awareness of the previous install/upgrade, it is crucial to specify the same *release* and *namespace* (both 'pdnscrds' in this example). If you try to upgrade but do not specify the existing *release* and *namespace*, the upgrade of the CRDs will fail (if it does fail, Helm will tell you and the old CRDs will remain untouched)

5.4 Install/Upgrade CloudControl

The CloudControl Helm Chart has a large amount of configurable options, which are detailed in the reference documentation. In the next few chapters the most important parts are discussed.

5.4.1 Registry Credentials

Since the CloudControl images are in a protected repository there is a requirement to supply credentials. There are several ways to configure these, for the remainder of this guide we will use the method which allows you to specify them directly in the override values:

```
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
```

Make sure the username & password match your credentials for the OX registry.

Note: Alternatively, you can make use of the methods described in the reference guide under 'Private Registries'

5.4.2 Cluster Networking

To be able to support Kubernetes clusters with IPv4, IPv6 or dual stack (IPv4 & IPv6) configurations, it is required to ensure the 'ipFamily' configuration in the helm values matches your cluster. The 'ipFamily' section contains the following parameters:

- **ipv4**: Whether or not your cluster has IPv4 enabled (Default: true)
- **ipv6**: Whether or not your cluster has IPv6 enabled (Default: false)
- **families**: Preference of IP families on your cluster, if it is a dualstack cluster

To ensure your deployment is correctly configured, you need to provide one of the 4 possible variations:

IPv4 only (default)

```
# Networking configuration
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

Note: 'families' is ignored in this configuration. It is only used in a dualstack setup.

IPv6 only

```
# Networking configuration
ipFamily:
  ipv4: false
  ipv6: true
  families:
    - "IPv4"
    - "IPv6"
```

Note: 'families' is ignored in this configuration. It is only used in a dualstack setup.

Dualstack - IPv4 primary

If you are running a dualstack cluster, you can check any Pod to see if your cluster has a preference for IPv4 or IPv6. Your pods will have a 'podIP' and 2 values for 'podIPs'. If the 'podIP' is an IPv4 address as shown in the example below, then you are running a cluster with IPv4 as primary:

```
podIP: 172.17.183.4 # IPv4
podIPs:
  - ip: 172.17.183.4 # IPv4
  - ip: fd43:128b:8658:b73b:3eb7:2e30:8815:3f6 # IPv6
```

Configuration for dualstack with IPv4 primary:

```
# Networking configuration
ipFamily:
  ipv4: true
  ipv6: true
  families:
    - "IPv4" # IPv4 is primary
    - "IPv6"
```

Dualstack - IPv6 primary

If you are running a dualstack cluster, you can check any Pod to see if your cluster has a preference for IPv4 or IPv6. Your pods will have a 'podIP' and 2 values for 'podIPs'. If the 'podIP' is an IPv6 address as shown in the example below, then you are running a cluster with IPv6 as primary:

```
podIP: fd43:128b:8658:b73b:3eb7:2e30:8815:3f6 # IPv6
podIPs:
  - ip: fd43:128b:8658:b73b:3eb7:2e30:8815:3f6 # IPv6
  - ip: 172.17.183.4 # IPv4
```

Configuration for dualstack with IPv6 primary:

```
# Networking configuration
ipFamily:
  ipv4: true
  ipv6: true
  families:
```

(continues on next page)

(continued from previous page)

```
    - "IPv6" # IPv6 is primary
    - "IPv4"
```

For the remainder of the guide we will assume the cluster is running on the 'IPv4 only' scenario. If your cluster has a different setup please make sure you substitute accordingly.

5.4.3 Deploying Recursor

To deploy a set of Recursor instances, include an entry in the YAML file under the 'recursors' parent, such as:

```
recursors:
  myrecursor:
    replicas: 3
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

The above file will create a set of Recursor instances named 'myrecursor' and the Deployment in Kubernetes will have a ReplicaSet with replicas=3. If you save this file as 'values.yaml' in your current working directory you should be able to use the Helm Chart to create the Recursor instances:

```
# The namespace
CC_NAMESPACE=my-namespace
HELM_RELEASE=ccdemo

helm install $HELM_RELEASE ./powerdns --namespace $CC_NAMESPACE --create-namespace \
--values ./values.yaml
```

Note: you can remove `--create-namespace` if you have an existing namespace to deploy into

Using kubectl you should now be able to see the corresponding Kubernetes objects created:

```
# Kubectl command to show all objects in a namespace
kubectl get all --namespace=$CC_NAMESPACE

# Kubectl output
NAME                                READY   STATUS    RESTARTS   AGE
pod/myrecursor-589559675d-d57jk     1/1     Running   0          3m12s
pod/myrecursor-589559675d-m779s     1/1     Running   0          3m12s
pod/myrecursor-589559675d-xxrvc     1/1     Running   0          3m12s

NAME                            TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)     AGE
```

(continues on next page)

(continued from previous page)

```
service/recursor-myrecursor   ClusterIP   None         <none>        5353/TCP   3m12s

NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/myrecursor    3/3     3            3           3m12s

NAME                                     DESIRED   CURRENT   READY   AGE
replicaset.apps/myrecursor-589559675d    3         3         3       3m12
```

Result should be a deployment + replicaset + service + a number of pods equal to the 'replicas' value from the values.yaml file.

5.4.4 Adding DNSdist

To add a set of DNSdist instances to our deployment, include an entry in the YAML file under the 'dnsdists' parent, such as:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
        packetcache:
          maxEntries: 200000
recursors:
  myrecursor:
    replicas: 3
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

The above will add a set of DNSdist instances named 'mydnsdist' and the Deployment in Kubernetes will have a ReplicaSet with replicas=2. The 'pools' configuration instruct DNSdist's agent to make sure all instances of 'myrecursor' are added to the default pool in DNSdist. The 'packetcache' with 'maxEntries' configuration ensures the cache for this pool will be able to hold 200000 entries.

Save the values.yaml file and upgrade the environment using the Helm Chart:

```
# The namespace
CC_NAMESPACE=my-namespace

# Helm release name
HELM_RELEASE=ccdemo
```

(continues on next page)

(continued from previous page)

```
helm upgrade $HELM_RELEASE ./powerdns --namespace $CC_NAMESPACE --values=./values.yaml
```

Using kubectl you should now be able to see the corresponding Kubernetes objects created for DNSdist:

```
# Kubectl command to show all objects in a namespace
kubectl get all --namespace=$CC_NAMESPACE

# Kubectl output
NAME                                 READY   STATUS    RESTARTS   AGE
pod/mydnsdist-775cbf55d9-qjtks       3/3     Running   1          15m
pod/mydnsdist-775cbf55d9-t8fbk       3/3     Running   1          15m
pod/myrecursor-589559675d-d57jk      1/1     Running   0          27m
pod/myrecursor-589559675d-m779s      1/1     Running   0          27m
pod/myrecursor-589559675d-xxrvc      1/1     Running   0          27m

NAME                            TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
service/recursor-myrecursor     ClusterIP   None         <none>        5353/TCP   27m

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mydnsdist      2/2     2            2           15m
deployment.apps/myrecursor     3/3     3            3           27m

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/mydnsdist-775cbf55d9      2         2         2       15m
replicaset.apps/myrecursor-589559675d     3         3         3       27m
```

5.4.5 Adding an external Resolver

To add a set of external resolvers to our deployment, include an entry in the YAML file under the 'resolvers' parent, such as:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
          - group: myresolver
        packetcache:
          maxEntries: 200000
recursors:
  myrecursor:
    replicas: 3
resolvers:
  myresolver:
    ips:
      - 9.9.9.9
      - 149.112.112.112
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
```

(continues on next page)

(continued from previous page)

```
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

The above will add a Service named 'myresolver' in Kubernetes which will have an Endpoints object containing the IP addresses (in this example the Quad9 IPs). By adding 'myresolver' to the 'default' pool in DNSdist, traffic will be loadbalanced between the Recursor & resolver endpoints (not a realistic scenario, which will be tackled in the next chapter).

Save the values.yaml file and upgrade the environment using the Helm Chart:

```
# The namespace
CC_NAMESPACE=my-namespace

# Helm release name
HELM_RELEASE=ccdemo

helm upgrade $HELM_RELEASE ./powerdns --namespace $CC_NAMESPACE --values=./values.yaml
```

Using kubectl you should now be able to see the corresponding Kubernetes objects created for resolver (the service object named 'myresolver'):

```
# Kubectl command to show all objects in a namespace
kubectl get all --namespace=$CC_NAMESPACE

# Kubectl output
NAME                                 READY   STATUS    RESTARTS   AGE
pod/mydnsdist-775cbf55d9-qwvrq       3/3     Running   0          22s
pod/mydnsdist-775cbf55d9-swz2w       3/3     Running   0          22s
pod/myrecursor-589559675d-5sqmg      1/1     Running   0          22s
pod/myrecursor-589559675d-cv6bl      1/1     Running   0          22s
pod/myrecursor-589559675d-sptfh      1/1     Running   0          22s

NAME                            TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
service/recursor-myrecursor     ClusterIP   None          <none>        5353/TCP   22s
service/resolver-myresolver     ClusterIP   None          <none>        53/TCP     22s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mydnsdist      2/2     2            2           22s
deployment.apps/myrecursor     3/3     3            3           22s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/mydnsdist-775cbf55d9      2         2         2       22s
replicaset.apps/myrecursor-589559675d     3         3         3       22s
```

5.4.6 Adding a DNSdist rule

To add more logic to DNSdist instances you can create rules under the 'rulesets' parent and assigning them to DNSdist objects, such as:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
          - group: myresolver
        packetcache:
          maxEntries: 200000
    rulegroups:
      - traffic-filters
recursors:
  myrecursor:
    replicas: 3
resolvers:
  myresolver:
    ips:
      - 9.9.9.9
      - 149.112.112.112
rulesets:
  block-traffic-ruleset:
    group: traffic-filters
    type: DNSDistRule
    priority: 100
    rules:
      - name: Block ANY
        combinator: AND
        selectors:
          - QType: ANY
        action:
          RCode:
            rcode: "REFUSED"
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

The above will add a DNSDistRule object named 'block-traffic-ruleset' in Kubernetes. This rule will select incoming queries with QType='ANY' and send a response 'REFUSED'. This rule is tagged with 'group' = 'traffic-filters', which is also added to the 'mydnsdist' rulegroups list, associating this rule to the DNSdist instances. More details on the specification of rules can be found in the reference guide.

Save the values.yaml file and upgrade the environment using the Helm Chart:

```
# The namespace
CC_NAMESPACE=my-namespace

# Helm release name
HELM_RELEASE=ccdemo

helm upgrade $HELM_RELEASE ./powerdns --namespace $CC_NAMESPACE --values=./values.yaml
```

Using kubectl you should now be able to see the corresponding Kubernetes objects if you specifically request them (since kubectl will not show any custom object types with 'get all'):

```
# Kubectl command to show all DNSDistRule objects in a namespace
kubectl get dnsdistrule --namespace=$CC_NAMESPACE

# Kubectl output
NAME                     AGE
block-traffic-ruleset    6s
```

5.4.7 Using DNSdist rules to route traffic

In a previous step we added recursors & resolvers to the default pool, but it would make more sense to have them in separate pools so they can serve different purposes. Rules allow this behaviour to be configured, such as:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
        packetcache:
          maxEntries: 200000
      external:
        serverGroups:
          - group: myresolver
        packetcache:
          maxEntries: 200000
    rulegroups:
      - traffic-filters
      - traffic-routers
recursors:
  myrecursor:
    replicas: 3
resolvers:
  myresolver:
    ips:
      - 9.9.9.9
      - 149.112.112.112
rulesets:
  route-traffic-ruleset:
    group: traffic-routers
    type: DNSDistRule
    priority: 200
    rules:
```

(continues on next page)

(continued from previous page)

```
      - name: External IPv6 resolution
        combinator: AND
        selectors:
          - QType: AAAA
        action:
          Pool:
            poolname: "external"
  block-traffic-ruleset:
    group: traffic-filters
    type: DNSDistRule
    priority: 100
    rules:
      - name: Block ANY
        combinator: AND
        selectors:
          - QType: ANY
        action:
          RCode:
            rcode: "REFUSED"
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

In the above example we moved the 'myresolver' group to a new pool named 'external'. Also, a new ruleset 'route-traffic-ruleset' was added which will match any queries with 'QType' = 'AAAA' and assign the pool named 'external' to handle those queries.

Save the values.yaml file and upgrade the environment using the Helm Chart:

```
# The namespace
CC_NAMESPACE=my-namespace

# Helm release name
HELM_RELEASE=ccdemo

helm upgrade $HELM_RELEASE ./powerdns --namespace $CC_NAMESPACE --values=./values.yaml
```

Using kubectl you should now be able to see the new Kubernetes objects if you specifically request them (since kubectl will not show any custom object types with 'get all'):

```
# Kubectl command to show all DNSDistRule objects in a namespace
kubectl get dnsdistrule --namespace=$CC_NAMESPACE

# Kubectl output
NAME                    AGE
block-traffic-ruleset   33m
route-traffic-ruleset   2s
```

5.4.8 Separating config into multiple files

As you start adding more instances & configuration options to the Helm Chart input file it becomes harder to make sense of the config. A recommended approach to improving this is to make use of Helm's ability to add multiple values files to the arguments of the helm command line. For example:

generic.yaml:

```
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

rulesets.yaml:

```
rulesets:
  block-traffic-ruleset:
    group: traffic-filters
    type: DNSDistRule
    priority: 100
    rules:
      - name: Block ANY
        combinator: AND
        selectors:
          - QType: ANY
        action:
          RCode:
            rcode: "REFUSED"
```

instances.yaml:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
        packetcache:
          maxEntries: 200000
    rulegroups:
      - traffic-filters
recursors:
  myrecursor:
    replicas: 3
```

You can then run helm as follows:

```
# The namespace
CC_NAMESPACE=my-namespace

# Helm release name
HELM_RELEASE=ccdemo

helm upgrade $HELM_RELEASE ./powerdns --namespace $CC_NAMESPACE \
--values=./generic.yaml --values=./rulesets.yaml --values=./instances.yaml
```

5.4.9 Exposing dnsdist

We now have a set of dnsdist instances running, but to complete the setup we need to make sure we have a method to direct traffic to the dnsdist instances. You can find out the different methods to expose dnsdist instances by reading the chapter 'Exposing dnsdist' in the reference guide.

5.4.10 Deploying ZoneControl

If you have one or more deployments of Auth running, you can deploy ZoneControl to manage the zones and records using a graphical user interface. This can be done by including an entry under the 'zonecontrols' parent.

Since this will require a Postgres database, we either need to have an existing database available for usage, or the extra Helm chart named *powerdns-operators* can be used to provision an Operators that creates Postgres databases for us. In the below example we will make use of the operator approach. To do so, we need to make sure the operator is installed, which can be done as follows:

```
# The release we're working with
CCTAG=2.5.2

# The namespace
CCOPS_NAMESPACE=ccops

# Helm release name
HELM_RELEASE=ccops

# Ensure repo data is up-to-date
helm repo update

# Pull the Helm Chart & unpack
helm pull cloudcontrol/powerdns-operators -d . --version=$CCTAG --untar

# Deploy the operator
helm install $HELM_RELEASE ./powerdns-operators --namespace $CCOPS_NAMESPACE
```

As a result there should be a Postgres Operator running in the 'ccops' namespace. We can then deploy ZoneControl:

generic.yaml:

```
registrySecrets:
  registry: registry.open-xchange.com
```

(continues on next page)

(continued from previous page)

```
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

zonecontrols.yaml:

```
zonecontrols:
  myzonecontrol:
  replicas: 2
  postgres:
    operator: true
  authEndpoints:
    - name: auth1
      url: https://auth1.example.com
      key: "apiKeyForAuth1"
    - name: auth2
      url: https://auth1.example.com
      key: "apiKeyForAuth2"
```

The above example assumes there are 2 deployments of Auth, named 'auth1' and 'auth2', with the Auth API endpoints accessible via the corresponding url and key. For more configuration options you can refer to the reference guide.

You can deploy these as follows:

```
# The namespace
ZC_NAMESPACE=zonecontrol

# Helm release name
HELM_RELEASE=ccdemo

helm install $HELM_RELEASE ./powerdns --namespace $ZC_NAMESPACE \
--values=./generic.yaml --values=./zonecontrols.yaml
```

Note: In the above example we deploy ZoneControl in a dedicated namespace 'zonecontrol'. Whilst not strictly necessary, it is generally advisable to deploy ZoneControl in a dedicated namespace to keep the management & delivery functions of CloudControl separated.

6 Advanced Examples

6.1 DNSdist: DoH

To deploy a set of DNSdist instances with DoH enabled, include a 'doh' configuration node in the dnsdist instance. The example below shows a basic DoH-enabled deployment of a set of DNSdist instances with Recursors:

```
dnsdists:
  mydohdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
        packetcache:
          maxEntries: 200000
    doh:
      - name: mydoh
        certificates:
          - key: |-
              -----BEGIN RSA PRIVATE KEY-----
              << CONTENTS OF PRIVATE KEY HERE>>
              -----END RSA PRIVATE KEY-----
            cert: |-
              -----BEGIN CERTIFICATE-----
              << CONTENTS OF CERTIFICATE HERE>>
              -----END CERTIFICATE-----
              -----BEGIN CERTIFICATE-----
              << CONTENTS OF ANY INTERMEDIATE CERTIFICATE(S) HERE>>
              -----END CERTIFICATE-----
recursors:
  myrecursor:
    replicas: 2
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

Note: Make sure to replace the contents of the 'key' and 'cert' with the data of a valid pair.

The above will result in a DNSdist deployment with the regular 'dnsdist-mydohdist' Service created, plus an additional Service named 'dnsdist-mydohdist-doh-mydoh'. This additional Service will have (by default) an inbound listener for traffic over port '443'.

You can refer to the 'Reference' guide for all available options to configure DoH. Options available include the configuration of STEK tickets (enabled & rotated by default) and loading certificates from pre-existing TLS Secrets to leverage a certificate manager such as certmanager.

6.2 DNSdist: DoT

To deploy a set of DNSdist instances with DoT enabled, include a 'dot' configuration node in the dnsdist instance. The example below shows a basic DoT-enabled deployment of a set of DNSdist instances with Recursors:

```
dnsdists:
  mydotdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
        packetcache:
          maxEntries: 200000
    dot:
      - name: mydot
        certificates:
          - key: |-
              -----BEGIN RSA PRIVATE KEY-----
              << CONTENTS OF PRIVATE KEY HERE>>
              -----END RSA PRIVATE KEY-----
            cert: |-
              -----BEGIN CERTIFICATE-----
              << CONTENTS OF CERTIFICATE HERE>>
              -----END CERTIFICATE-----
              -----BEGIN CERTIFICATE-----
              << CONTENTS OF ANY INTERMEDIATE CERTIFICATE(S) HERE>>
              -----END CERTIFICATE-----
recursors:
  myrecursor:
    replicas: 2
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

Note: Make sure to replace the contents of the 'key' and 'cert' with the data of a valid pair.

The above will result in a DNSdist deployment with the regular 'dnsdist-mydotdist' Service created, plus an additional Service named 'dnsdist-mydotdist-dot-mydot'. This additional Service

will have (by default) an inbound listener for traffic over port '853'.

You can refer to the 'Reference' guide for all available options to configure DoT. Options available include the configuration of STEK tickets (enabled & rotated by default) and loading certificates from pre-existing TLS Secrets to leverage a certificate manager such as certmanager.

6.3 DNSdist: Co-hosted Recursor

To deploy a set of DNSdist instances with co-hosted Recursor instances, include a 'recursor' configuration node in the dnsdist instance and specify the number of Recorsor instances. The example below shows an example of DNSdist instances with 2 co-hosted Recursors:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        packetcache:
          maxEntries: 200000
    recursor:
      replicas: 2
registrySecrets:
  registry: registry.open-xchange.com
  username: REGISTRY_USERNAME_HERE
  password: REGISTRY_PASSWORD_HERE
  email: admin@registry.open-xchange.com
ipFamily:
  ipv4: true
  ipv6: false
  families:
    - "IPv4"
    - "IPv6"
```

The above will result in a DNSdist deployment where each DNSdist Pod also contains 2 Recursor containers (+ a Recursor agent container to keep the Recursors synchronised)

Note: If you do not specify an amount of 'replicas' there will be no embedded recursor instances deployed.

6.4 DNSdist: Lua script

To deploy a set of DNSdist instances with custom Lua script included, include a *luaScript* configuration node in the dnsdist instance. The example below shows a basic deployment of a set of DNSdist instances with Recursors and a dynamic rule which will answer *refused* for 60 seconds if they are measured to be generating > 5 QPS on queries with type *ANY*:

```
dnsdists:
  mydnsdist:
    luaScript: |-
      function maintenance()
        addDynBlocks(exceedQTypeRate(DNSQType.ANY, 5, 10), "Exceeded ANY rate", 60)
      end
```

(continues on next page)

(continued from previous page)

```
      setDynBlocksAction(DNSAction.Refused)
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
        packetcache:
          maxEntries: 200000
recursors:
  myrecursor:
    replicas: 2
```

For more information on the capabilities of Lua scripting you can refer to the product documentation at: https://dnsdist.org/

6.4.1 Lua script from file

Helm also allows injecting the contents of a separate file into a configuration node in the *helm install* & *helm upgrade* commands. This has several benefits, including not having to indent it inside your main YAML file.

For example, if you have a directory with these 2 files:

overrides.yaml:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor
        packetcache:
          maxEntries: 200000
recursors:
  myrecursor:
    replicas: 2
```

script.lua:

```
function maintenance()
  addDynBlocks(exceedQTypeRate(DNSQType.ANY, 5, 10), "Exceeded ANY rate", 60)
end

setDynBlocksAction(DNSAction.Refused)
```

Then you could inject the Lua script as follows (*dnsdists.mydnsdist.luaScript* is the path to the *luaScript* node for the dnsdist instance named *mydnsdist*):

```
helm install myrelease ./powerdns --namespace mynamespace \
    --values overrides.yaml --set-file dnsdists.mydnsdist.luaScript=script.lua
```

Note: This method assumes *overrides.yaml* and *script.lua* are in the same directory

6.5 Recursor: Lua script & config

To deploy a set of Recursor instances with custom Lua script and/or Lua config included, include a *luaScript* and/or *luaConfig* ` configuration node in the Recursor instance. The example below shows a basic deployment of a set of Recursors instances with both a Lua script and config included:

```
recursors:
  myrecursor:
    replicas: 2
    luaScript: |-
      function preresolve(dq)
        if dq.qname:equal("somerecord.example.com") then
          dq.rcode = 5
          return true
        end
        return false
      end
    luaConfig: |-
      addAllowedAdditionalQType(pdns.MX, {pdns.A, pdns.AAAA})
```

For more information on the capabilities of Lua scripting and Lua configuration you can refer to the product documentation at: https://doc.powerdns.com/recursor/

6.5.1 Lua script and config from file

Helm also allows injecting the contents of a separate file into a configuration node in the *helm install* & *helm upgrade* commands. This has several benefits, including not having to indent it inside your main YAML file.

For example, if you have a directory with these 3 files:

overrides.yaml:

```
recursors:
  myrecursor:
    replicas: 2
```

script.lua:

```
function preresolve(dq)
  if dq.qname:equal("somerecord.example.com") then
    dq.rcode = 5
    return true
  end
  return false
end
```

lua.config:

```
addAllowedAdditionalQType(pdns.MX, {pdns.A, pdns.AAAA})
```

Then you could inject the Lua script as follows (*recursors.myrecursor.luaScript* is the path to the *luaScript* node for the recursor instance named *myrecursor*):

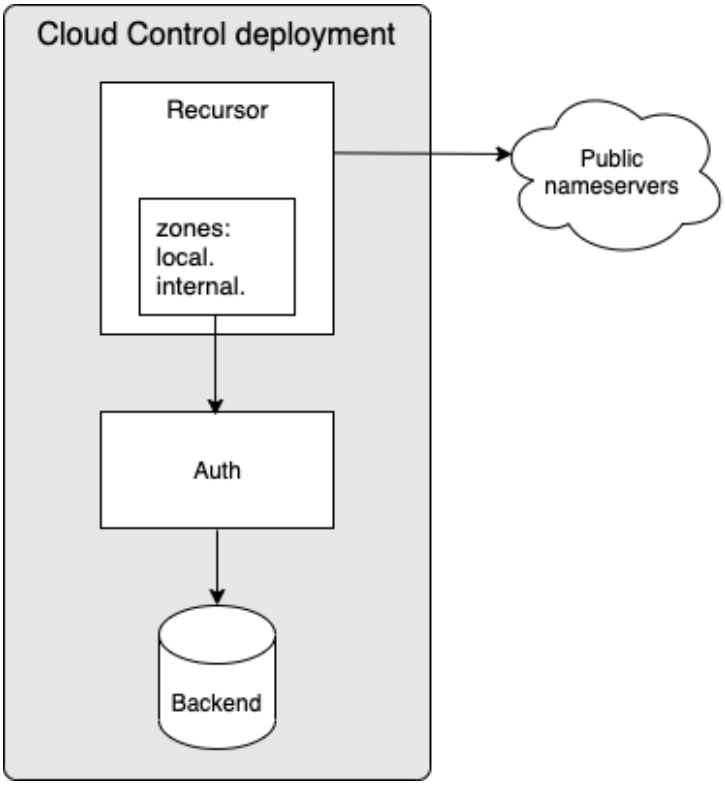
```
helm install myrelease ./powerdns --namespace mynamespace \
    --values overrides.yaml \
    --set-file recursors.myrecursor.luaScript=script.lua \
    --set-file recursors.myrecursor.luaConfig=lua.config \
```

Note: This method assumes *overrides.yaml*, *script.lua* and *lua.config* are in the same directory

6.6 Recursor: Forwarding zones

By default, a Recursor will attempt to resolve via public nameservers. If you have any zones which you would like to be resolved by a specific set of nameservers and/or delegate to other recursive resolvers you can use the *forward* parameter.

A popular use case for forwarding is the following:



In this setup we tell Recursor to forward all traffic for zones matching 'local.' and 'internal.' to a set of Auth instances, while the rest of the traffic will be handled as per default resolution via public nameservers. To configure this scenario, you can use the following forwarding configuration:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - zones:
          - "local."
          - "internal."
        serverGroups:
```

(continues on next page)

(continued from previous page)

```
        - group: myauth

auths:
  myauth:
    replicas: 2
    backends:
      - type: ls
        access_key: MY_ACCESS_KEY
        secret_key: MY_SECRET_KEY
        bucket: mybucket
        endpoint: https://my.s3.endpoint
```

6.6.1 Automatically learning forward zones from Auth

If you have a set of Auth instances you would like to forward traffic to, you can also instruct Recursor to learn the zones for which these Auth instances are authoritative. This can be done via the *learnFrom* parameter. For example:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - learnFrom: myauth

auths:
  myauth:
    replicas: 2
    backends:
      - type: ls
        access_key: MY_ACCESS_KEY
        secret_key: MY_SECRET_KEY
        bucket: mybucket
        endpoint: https://my.s3.endpoint
```

The above example will automatically forward all zones learned from the *myauth* instances towards them.

Note: You can only reference the name of a set of Auth instances managed via CloudControl in the *learnFrom* parameter, as this integration depends on the use of the Auth API.

6.6.2 Filtering learned zones from Auth

Besides forwarding all zones to an Auth, you can also filter them. A common use case for this is to have a predefined set of zones which you can use freely inside Auth to test, without exposing them to the Recursor instances. This can be done via 2 parameters: *exclude* and *include*

An example using *exclude*:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - learnFrom: myauth
```

(continues on next page)

(continued from previous page)

```
        exclude:
          - ".test."
          - "local.$"
```

The above example will learn all zones from Auth, and remove any zones which match:

- ".test." any zone which contains ".test." anywhere within the name
- "local.$" any zone which ends in "local."

And an example using *include*:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - learnFrom: myauth
        include:
          - "internal.$"
```

The above example will learn all zones from Auth, but will only forward zones which match:

- "internal.$" any zone which ends in "internal."

Also, you can use both *exclude* and *include* together:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - learnFrom: myauth
        include:
          - "internal.$"
        exclude:
          - ".test."
          - "local.$"
```

When both are used, all zones are learned, then reduced to the zones matching the *include* parameter and finally any zones matching *exclude* are removed from the forwarding.

Notes:

- You can use *$* at the end of your filter to signal this filter should only apply to zones ending with this
- You can use ^ at the beginning of your filter to signal this filter should only apply to zones starting with this
- These filters match against the FQDN of a zone, hence you need to take into account the trailing period in a zone, ie: "powerdns.com.", not "powerdns.com"

6.6.3 Forwarding zones to another Recursor

If you have multiple sets of Recursor instances you can also forward from one set to another. For example:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - zones:
          - "internal."
        serverGroups:
          - group: internalrecursor
        recurse: true
  internalrecursor:
    replicas: 2
```

In the above example you can see we are telling the set of Recursors named *myrecursor* to forward *internal.* to the set of Recursors named *internalrecursor*. Since Recursor by default assumes we are forwarding to authoritative nameservers, we set *recurse: true* to ensure the request for recursion is preserved.

6.6.4 Forwarding to external resolvers and/or authoritative nameservers

If you have resolvers and/or authoritative nameservers deployed in another location and would like to forward traffic to them, you can use the *resolvers* type of instances. For example:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - zones:
          - "."
        serverGroups:
          - group: externalresolver
        recurse: true

resolvers:
  externalresolver:
    ips:
      - 1.2.3.4
      - 5.6.7.8
```

In the above example all zones matching "." (all of them!) will be forwarded to the IPs configured in the resolver object named *externalresolver* with the request for recursion preserved.

Similarly, we can also forward to an external nameserver:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - zones:
          - "local."
        serverGroups:
```

(continues on next page)

(continued from previous page)

```
        - group: externalnameserver

resolvers:
  externalnameserver:
    ips:
      - 9.8.7.6
      - 5.4.3.2
```

6.6.5 Forwarding & DNSSEC

Forwarding to a set of Auth instances can run into problems when you are forwarding zones which would otherwise be validated using DNSSEC. If this is the case, you can opt to have negative trust anchors applied to all zones in a forward configuration, for example:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - zones:
          - "local"
          - "internal"
        serverGroups:
          - group: myauth
        nta: true
```

The *nta: true* in this example will enable the creation of negative trust anchors for each entry in *zones:*

6.6.6 Priority

Forwarding to multiple sets of Auth instances and/or sets of Resolvers may lead to a scenario where you have multiple sets with a destination for the same zone. To manage the ordering of precedence in such a situation the *priority* field is available. The rules regarding the *priority* field:

- If multiple entries have a destination for the same zone, the entry with the lowest value for *priority* will be selected
- If multiple entries have the same priority and both have a destination for the same zone, the entry which is listed first in the *forward:* list will be selected
- By default each entry has priority = 100
- Priority must be a positive integer

For example:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - zones:
          - "my.zone.com"
```

(continues on next page)

(continued from previous page)

```
        serverGroups:
          - group: resolver1
      - zones:
          - "my.zone.com"
        serverGroups:
          - group: resolver2
```

Both entries have no priority specified, so they both default to 100. Since they are equal and both have a destination for *my.zone.com*, *resolver1* will be selected due to its higher position in the *forward:* list.

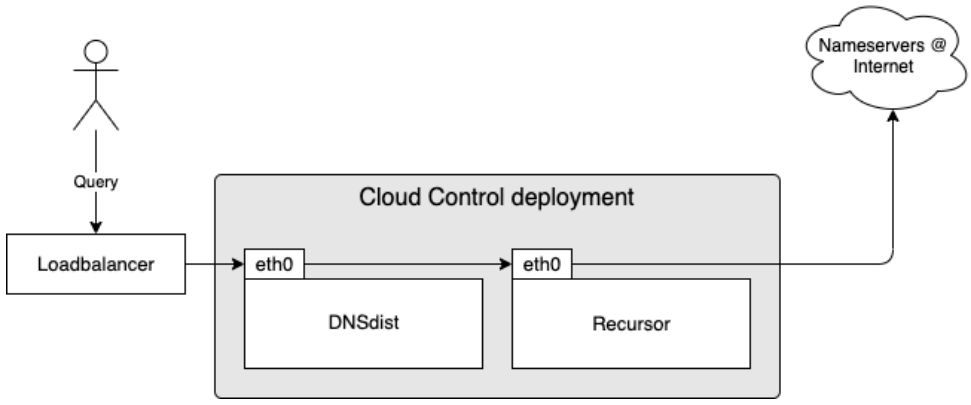
Example with priorities specified:

```
recursors:
  myrecursor:
    replicas: 2
    forward:
      - zones:
          - "my.zone.com"
        priority: 900
        serverGroups:
          - group: resolver1
      - zones:
          - "my.zone.com"
        priority: 50
        serverGroups:
          - group: resolver2
```

Now *resolver2* will be selected, because it has a lower value for *priority*.

6.7 Multi-homed pods

Container network interface (CNI) plugins such as Multus CNI allow you to attach multiple network interfaces to pods (ie: multi-homed pods). Without multi-homed pods, you are limited to the pod network (indicated by eth0 interfaces) as shown in the below diagram:

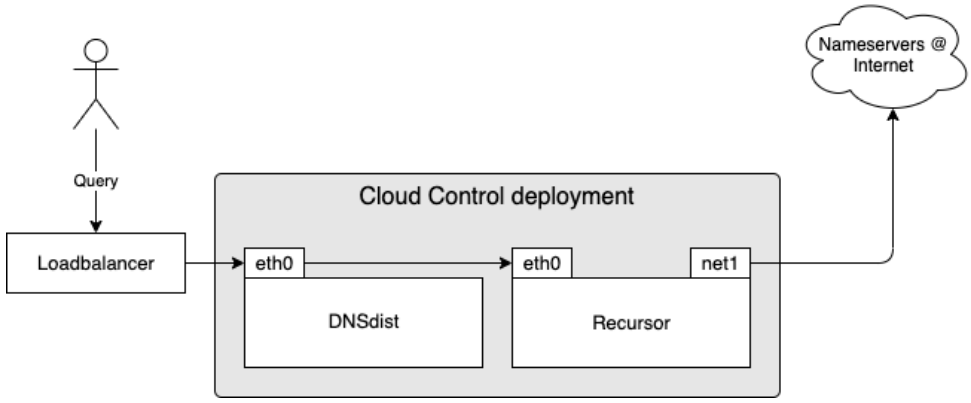


You can see this dnsdist + recursor example has the following traffic flows:

- dnsdist inbound from users: eth0

- dnsdist outbound to recursor: eth0
- recursor inbound from dnsdist: eth0
- recursor outbound to internet: eth0

Whether or not your Kubernetes cluster can accomodate for all the above traffic flows over the pod network depends on many factors and often the last flow (recursor outbound to internet) presents a problem. For this purpose using a multi-homed Recursor pod is a good alternative. An example of how this can be used:



Now the example has the following traffic flows:

- dnsdist inbound from users: eth0
- dnsdist outbound to recursor: eth0
- recursor inbound from dnsdist: eth0
- recursor outbound to internet: net1 (the additional interface)

6.7.1 Configuring multi-homed Recursor pods

Making a Recursor pod multi-homed is a simple task, since this only involves adding an annotation to the pods. Your CNI plugin should take care of the rest.

For example using the Multus CNI plugin we can attach a Network named *testnetv4* which is defined in namespace *kube-system*:

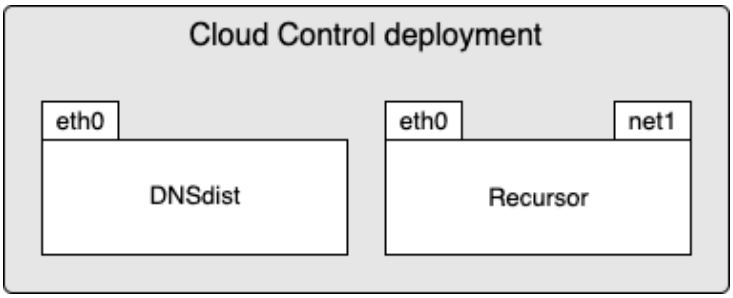
```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor

recursors:
  myrecursor:
    podAnnotations:
      k8s.v1.cni.cncf.io/networks: '[{
          "name": "testnetv4",
          "namespace": "kube-system"
```

(continues on next page)

(continued from previous page)

```
        }]'
    replicas: 2
```

This results in the following setup (your additional nic might have a different name):



By default, the Recursor will not make efficient use of the additional interface though, as Kubernetes default routing will prioritise the pod network's eth0. To force the Recursor to use the additional interface for outbound traffic you can configure the 'outboundInterfaces' parameter, for example:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor

recursors:
  myrecursor:
    podAnnotations:
      k8s.v1.cni.cncf.io/networks: '[{
          "name": "testnetv4",
          "namespace": "kube-system"
        }]'
    replicas: 2
    outboundInterfaces:
      - net1
```

The Recursor will now be able to:

- Receive traffic from the pod network over eth0
- Receive traffic from the additional network over net1
- Send traffic to the additional network over net1

Note: The Recursor will no longer be able to send traffic to destinations on the pod network with this configuration. If you need to have the Recursor able to send traffic to destinations both internal and external to your Kubernetes cluster, the suggested approach is to solve this via routing modifications rather than having multiple interfaces.

If you are running a dualstack Kubernetes cluster and wish to assign an interface for both an outbound IPv4 and IPv6 address, you can specify both of the additional interfaces in the 'outboundInterfaces' list:

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor

recursors:
  myrecursor:
    podAnnotations:
      k8s.v1.cni.cncf.io/networks: '[{
          "name": "testnetv4",
          "namespace": "kube-system"
        },
        {
          "name": "testnetv6",
          "namespace": "kube-system"
        }]'
    replicas: 2
    outboundInterfaces:
      - net1
      - net2
```

Assuming the IPv4 NIC is assigned as 'net1' and the IPv6 NIC as 'net2', Recursor will now attempt to use both for outbound traffic based on the type of address it needs to communicate to (IPv4 or IPv6).

In the above examples we configured Recursor's outbound interface to our additional network, but it might not be desirable to allow inbound traffic from the internet to reach the Recursor pod. How to handle that situation is specific to the larger architecture/infrastructure in which the Kubernetes cluster resides, but if it is desirable then it is possible to stop Recursor from listening to the additional interface. An example which shows how to configure this (and more):

```
dnsdists:
  mydnsdist:
    replicas: 2
    pools:
      default:
        serverGroups:
          - group: myrecursor

recursors:
  myrecursor:
    podAnnotations:
      k8s.v1.cni.cncf.io/networks: '[{
          "name": "testnetv4",
          "namespace": "kube-system"
        }]'
    replicas: 2
    inboundInterfaces:
      - eth0
    metricsInterfaces:
      - eth0
    outboundInterfaces:
      - net1
```

(continues on next page)

(continued from previous page)

```
    readiness:
      bindInterfaces:
        - "eth0"
```

In the above deployment we ignore all the defaults and override each inbound & outbound traffic flow to utilize a specific interface:

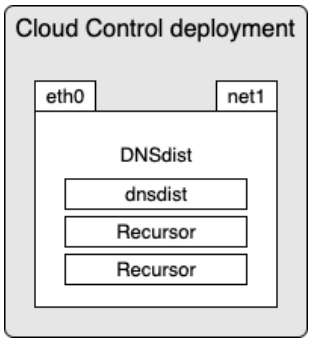
- Inbound traffic to Recursor: eth0 (pod network)
- Inbound traffic to metrics aggregator: eth0 (pod network)
- Inbound traffic to readiness probe: eth0 (pod network)
- Outbound traffic from Recursor to nameservers: net1 (additional interface)

6.7.2 Configuring multi-homed DNSdist with co-hosted Recursor pods

Similar to above, making a dnsdist + co-hosted recursor pod requires the addition of an annotation:

```
dnsdists:
  mydnsdist:
    podAnnotations:
      k8s.v1.cni.cncf.io/networks: '[{
          "name": "testnetv4",
          "namespace": "kube-system"
        }]'
    replicas: 2
    recursor:
      replicas: 2
```

This results in dnsdist pods as follows (your additional nic might have a different name):



The defaults for this scenario are slightly different, since Recursor is embedded within the Dnsdist pod. The enabled traffic flows are:

- Dnsdist: Receive traffic from the pod network over eth0
- Dnsdist: Send traffic via loopback to embedded Recursor containers
- Recursor: Receive traffic via loopback from Dnsdist
- Recursor: Send traffic to the additional network over net1

And the *utility* traffic flows:

- Inbound traffic to Dnsdist readiness: eth0 (pod network)
- Inbound traffic to Recursor readiness: eth0 (pod network) & net1 (additional interface)
- Inbound traffic to metrics aggregator: eth0 (pod network)

Suppose we want to implement a common scenario, where all inbound traffic is limited the the pod network, while recursor's outbound traffic uses the additional interface. Then we would want the following traffic flows:

- Dnsdist: Receive traffic over pod network: eth0
- Recursor: Send traffic to nameservers over additional network: net1
- Utilities: Receive traffic over pod network: eth0

The above deployment can be finetuned as follows to accomodate this scenario:

```
dnsdists:
  mydnsdist:
    podAnnotations:
      k8s.v1.cni.cncf.io/networks: '[{
          "name": "testnetv4",
          "namespace": "kube-system"
        }]'
    replicas: 2
    recursor:
      replicas: 2
      outboundInterfaces:
        - net1
      readiness:
        bindInterfaces:
          - "eth0"
```

6.8 Auth: Backends

Within Cloud Control deployments, Auth supports 4 backends:

- Postgres
- MySQL
- GeoIP
- LMDB with LightningStream

6.8.1 Postgres

If you have a Postgres (or compatible) cluster available, you can configure Auth to store its data in a Postgres database. A simple example using the postgres backend:

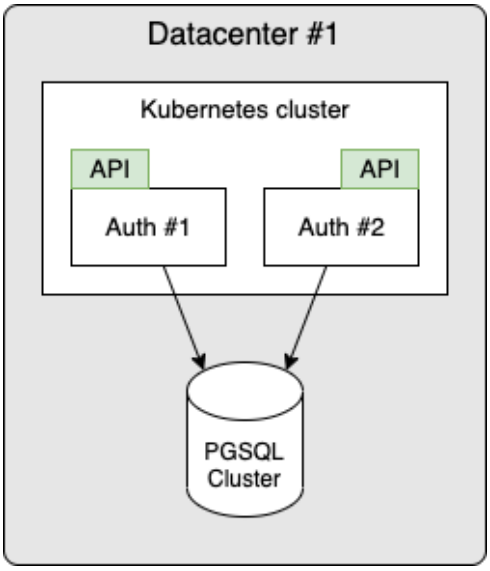
```
auths:
  myauth:
    replicas: 2
    backends:
      - type: postgres
```

(continues on next page)

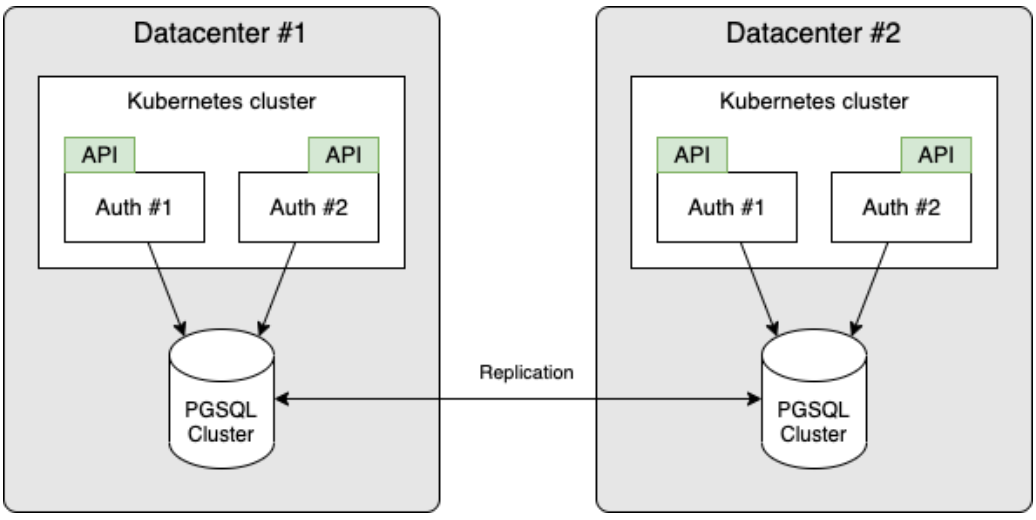
(continued from previous page)

```
        host: pg.host.local
        dbname: mydb
        user: some_user
        password: some_password
```

When deployed, you will have an environment as follows:



If you have a Postgres cluster available with replication features, you can utilise this to deploy Cloud Control in multiple datacenters with shared data:



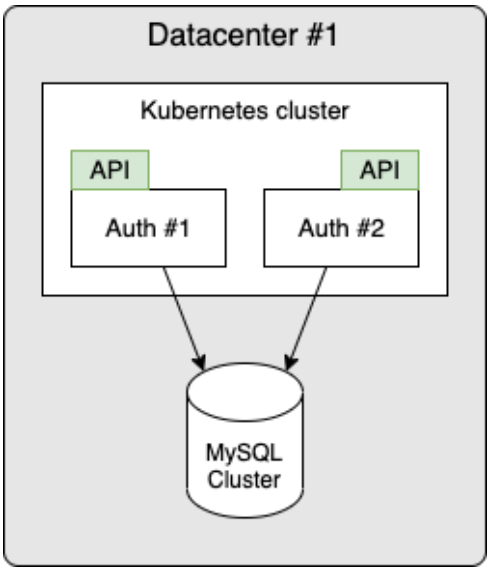
For further configuration options regarding the Postgres backend you can read the corresponding chapter in the reference guide.

6.8.2 MySQL

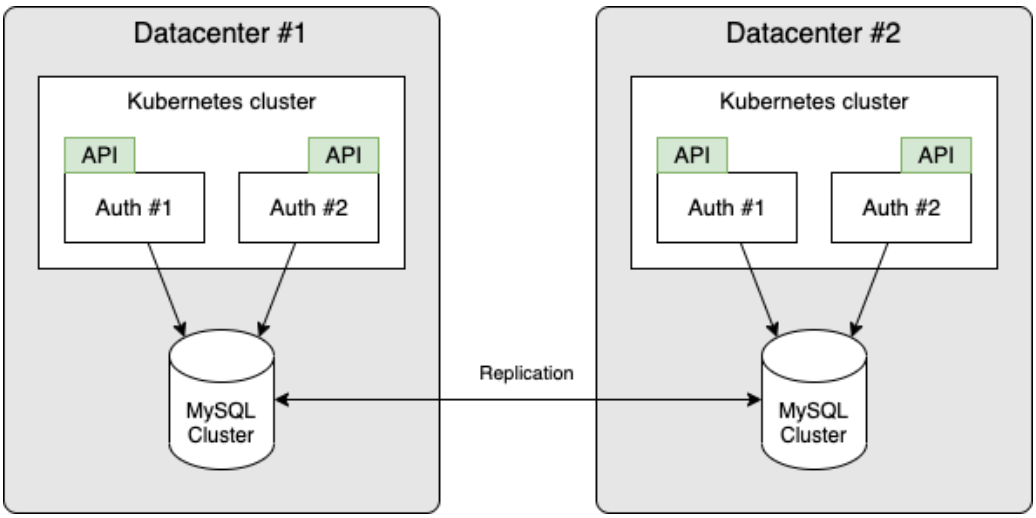
If you have a MySQL (or compatible) cluster available, you can configure Auth to store its data in a MySQL database. A simple example using the mysql backend:

```
auths:
  myauth:
    replicas: 2
    backends:
      - type: mysql
        host: mysq.host.local
        dbname: mydb
        user: some_user
        password: some_password
```

When deployed, you will have an environment as follows:



If you have a MySQL cluster available with replication features, you can utilise this to deploy Cloud Control in multiple datacenters with shared data:



For further configuration options regarding the MySQL backend you can read the correspond-

ing chapter in the reference guide.

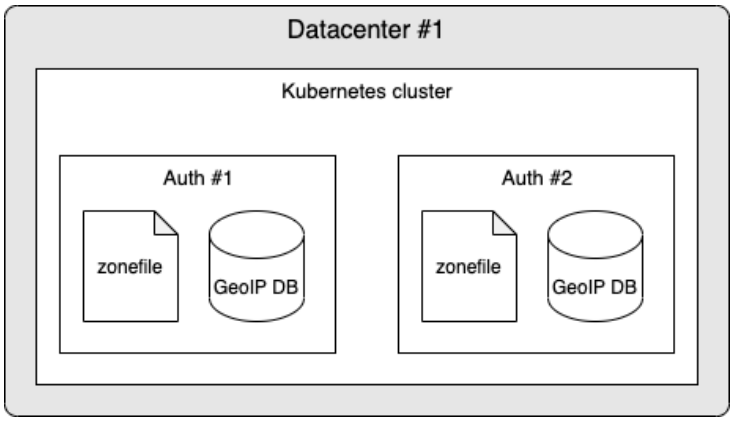
6.8.3 GeoIP

This backend allows zones to be managed via a YAML format included in the backend configuration and the inclusion of a GeoIP database. The records managed via this backend can make use of the GeoIP database to respond to DNS queries with an answer based on the request's origin. High-level configuration looks as follows:

```
auths:
  myauth:
    replicas: 2
    backends:
      - type: geoip
        databases: <Configuration of one or more GeoIP databases>
        domains: <Configuration of domains>
```

Note: Configuration of this backend is quite complex, for more detailed examples please refer to the reference guide.

When deployed, you will have an environment as follows:



Since the Auth instances have a copy of the zonefile(s) and GeoIP databases locally, you can replicate this deployment across multiple datacenters simply by deploying the same configuration to each datacenter.

6.8.4 LMDB with LightningStream

This backend allows you to utilise an S3 bucket to store & replicate data between Auth instances within a datacenter and across multiple datacenters. Locally each Auth instance writes to an LMDB database, which the LightningStream component then synchronises bi-directionally with the snapshots stored in the S3 bucket. Configuration looks as follows:

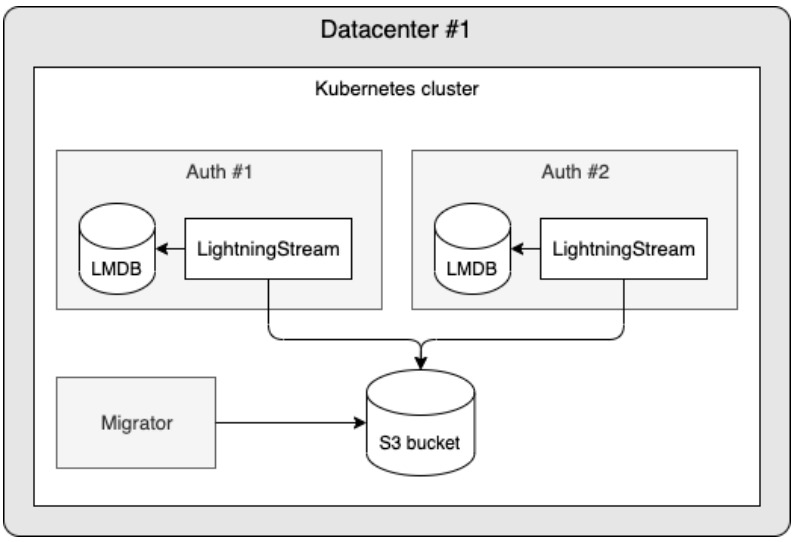
```
auths:
  myauth:
    replicas: 2
    backends:
      - type: ls
        access_key: MY_ACCESS_KEY
```

(continues on next page)

(continued from previous page)

```
        secret_key: MY_SECRET_KEY
        bucket: mybucket
        endpoint: https://my.s3.endpoint
```

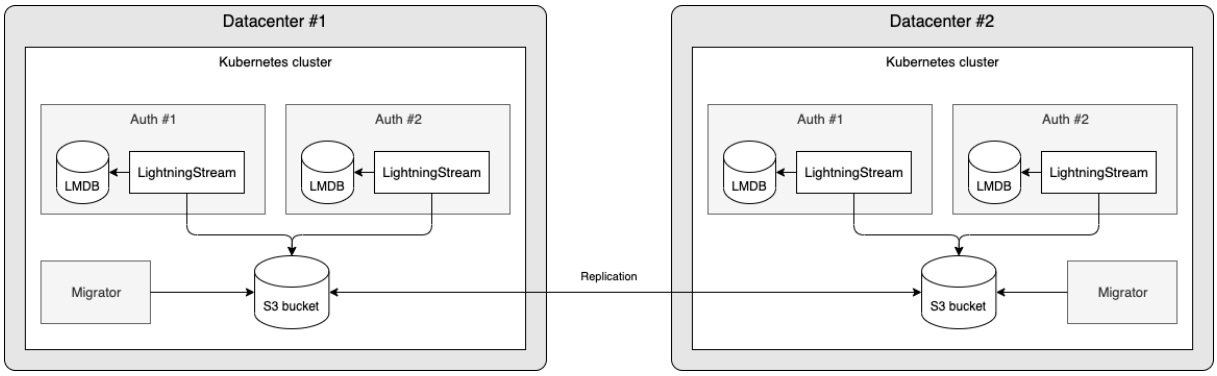
When deployed, you will have an environment as follows:



The additional Migrator pod will be deployed to monitor the different schema versions (based on Auth version) for which snapshots are available in the S3 bucket. When you initially deploy an environment with this backend the migrator will not have anything to do. However, once you upgrade to a newer version of Cloud Control, there might be changes to the schema used by Auth. The Migrator will ensure a seamless transition to the new schema without having to bring the environment offline for schema upgrading and/or manual maintenance.

If you have a distributed S3 bucket available (or multiple buckets with replication) you can keep multiple deployments across datacenters in sync.

Example diagram of such a setup with replicated S3 buckets:



Because of the Migrator pod, you can safely upgrade Cloud Control in each datacenter sequentially. When you upgrade datacenter #1, all known data will be migrated to a new schema version (if required) and any writes handled by the deployment in datacenter #2 (old schema version) will be migrated, albeit with a minor delay of 10-15 seconds. Once all deployments are upgraded to the same version, the Migrator will no longer have anyhing to do and will idle until a new upgrade is detected.

Once the migrator has detected that the previous schema version has not been updated for a certain period (Default: *28 days*) since it was last migrated successfully, the old schema will be removed. This behaviour can be modified via the following parameters on the backend:

```
auths:
  myauth:
    replicas: 2
    backends:
      - type: ls
        access_key: MY_ACCESS_KEY
        secret_key: MY_SECRET_KEY
        bucket: mybucket
        endpoint: https://my.s3.endpoint
        removeOldSchemaAge: 336h
```

The use of *removeOldSchemaAge* will instruct the migrator to cleanup an old schema once it has been migrated successfully and since then has had no changes observed for *336* hours.

To disable the cleanup mechanism and keep the old schemas indefinitely, you can disable it altogether by setting *removeOldSchemaEnabled* to *false*:

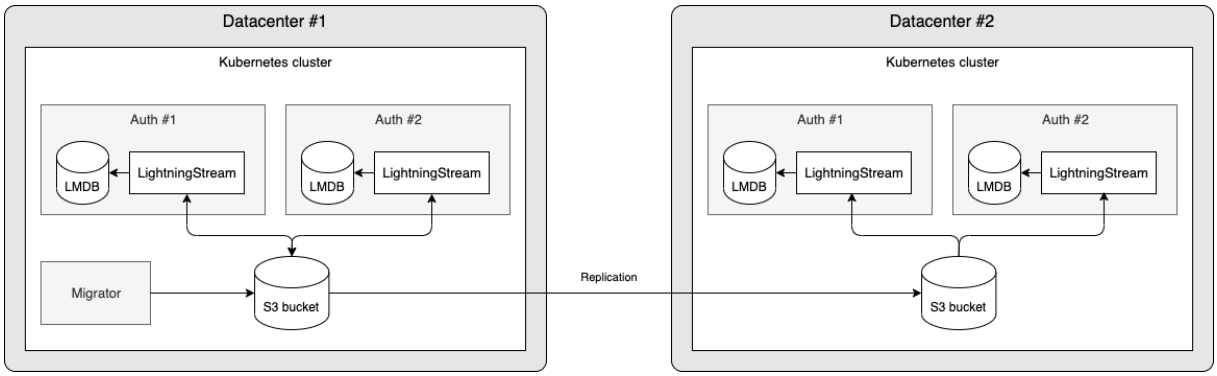
```
auths:
  myauth:
    replicas: 2
    backends:
      - type: ls
        access_key: MY_ACCESS_KEY
        secret_key: MY_SECRET_KEY
        bucket: mybucket
        endpoint: https://my.s3.endpoint
        removeOldSchemaEnabled: false
```

Synchronization modes

In addition to the default *sync* mode described above, you can also deploy LightningStream in *receive* mode. In this configuration, the following changes are active:

- LightningStream only pulls changes from the s3 bucket, it no longer writes to the bucket
- Migrator Pod is not deployed, since this deployment will not be writing changes
- Less privileges are required, since LightningStream will only need to fetch the contents of the s3 bucket

When you have a deployment with the default *sync* mode (left in diagram) and another with the *receive* mode (right in diagram), both using the same s3 bucket, you will have a deployment as follows:



Cleanup

LightningStream includes a cleanup component, which will periodically check the S3 bucket to locate old snapshots which are no longer required. If it locates snapshots which are no longer needed, they will be removed from the S3 bucket. By default this cleanup mechnism is *enabled* and it is highly advisable to keep it this way to prevent the bucket from occupying more storage than necessary. Having the cleanup enabled also allows LightningStream to sync faster, as it has fewer snapshots to consider. Intervals and restrictions used by the cleanup component are included in the reference guide chapter for configuring LightningStream.

Handling S3 bucket outages/errors

If for any reason the S3 bucket becomes unavailable or unreachable by LightningStream, the corresponding Auth instance will no longer be able to keep in sync with the other instances. Modifications made to the data by this particular Auth instance will not be lost though, as soon as the S3 bucket is available again this data will be synced.

While the S3 bucket is not available/reachable, there are several options that control how Cloud Control will deal with this situation:

- unreadyIfError: If LightningStream has detected problems with the S3 bucket for a period exceeding the error threshold, the LightningStream container (and hence the Auth pod) will be marked as *NotReady*. Consequence is that this Auth Pod will be removed from the Service and stop receiving traffic.
- unreadyIfWarning: If LightningStream has detected problems with the S3 bucket for a period exceeding the warning threshold, the LightningStream container (and hence the Auth pod) will be marked as *NotReady*. Consequence is that this Auth Pod will be removed from the Service and stop receiving traffic.
- waitForInitialSync: When an Auth Pod is spawned, mark the Pod as *NotReady* until LightningStream has successfully completed its initial sync with the S3 bucket.

The defaults for LightningStream are as follows:

- unreadyIfError: true
- unreadyIfWarning: false
- waitForInitialSync: true

This is typically a good configuration for an Auth deployment which should not be allowed to be available if data consistency across all instances is preferred over availability of all Auth instances.

Another potential combination could be as follows:

- unreadyIfError: false
- unreadyIfWarning: false
- waitForInitialSync: true

This could be used when an Auth instance primarily relies upon another backend and has LMDB with LightningStream as a secondary backend defined. Also, if you prefer availability of the Auth Pods over data consistency you could use this configuration.

If you also want the Auth Pod to register to the Service before LightningStream has completed its initial sync, you can set *waitForInitialSync* to *false*.

Note: Configuring the error and warning thresholds is possible via the *health* setting on the backend. This is documented in more detail in the reference guide in chapter 'LightningStream - Health Configuration'

S3 buckets with versioning

LightningStream does not require any versioning to be enabled on S3 buckets to function, but it might be enabled if your S3 provider needs this to support your desired replication topology.

If versioning is applied to your bucket, it is advisable to also enable lifecycle management to make sure a limited amount of versions are kept. In addition, lifecycle management on your versioned S3 buckets can ensure objects are actually deleted (instead of marked for deletion) after LightningStream issues a delete operation for them.

6.9 Auth: ixfrdist

To deploy a set of Auth instances with ixfrdist enabled, include a 'ixfrdist' configuration node in the Auth instance with 'enabled' set to 'true'. The example below shows a basic Auth deployment with ixfrdist enabled and a DNSdist deployment to expose ixfrdist:

```
auths:
  myixfrauth:
    replicas: 2
    backends:
      - type: ls
        access_key: MY_ACCESS_KEY
        secret_key: MY_SECRET_KEY
        bucket: mybucket
        endpoint: https://my.s3.endpoint
    ixfrdist:
      enabled: true
      domains:
        - domain: "zone1.ixfrdist.local"
        - domain: "zone3.ixfrdist.local"

dnsdists:
```

(continues on next page)

(continued from previous page)

```
  myixfrdnsdist:
    replicas: 2
    readiness:
      healthCheck: false
    pools:
      default:
        serverGroups:
          - group: myixfrauth
            component: ixfrdist
```

The above will result in an Auth deployment with a Lightning Stream backend. Inside each Auth Pod, there will be an 'ixfrdist' container which will attempt to transfer and re-serve the defined zones ('zone1.ixfrdist.local' and 'zone3.ixfrdist.local' over AXFR and IXFR).

To expose ixfrdist, a DNSdist deployment is included with some configuration specific to ixfrdist:

- 'healthCheck: false' under readiness will ensure that DNSdist does not try to perform the default readiness probes for 'a.root-servers.net' to determine if the DNSdist instance is ready for traffic (as ixfrdist will not be able to answer these queries succesfully)
- 'component: ixfrdist' under serverGroups will instruct DNSdist to send this traffic to the ixfrdist container instead of the Auth container

You can refer to the 'Reference' guide for all available options to configure ixfrdist.

6.10 Dstore-dist: Recursor

To deploy a set of Recursor instances with dstore-dist enabled, include a 'dstoredist' configuration node in the Recursor instance with a set of destinations and routes configured. The example below shows a basic Recursor deployment with dstore-dist enabled to distribute protobuf messages to a Kafka endpoint:

```
recursors:
  myrecursor:
    replicas: 2
    dstoredist:
      destinations:
        mydestination:
          type: kafka
          kafka:
            addresses:
              - my.kafka.local:9092
            topic: my_topic
      routes:
        myroute:
          destinations:
            - mydestination
```

The above will result in a Recursor deployment with a dstore-dist sidecar. By default, when dstore-dist is enabled in a Recursor pod all inbound queries and corresponding answers will be sent to dstore-dist and then distributed according to the configured routes and destinations. In this example, for all inbound queries and corresponding answers a protobuf message will be generated and distributed to a kafka topic named 'my_topic' via a Kafka endpoint at 'my.kafka.local:9092'.

You can refer to the 'Reference' guide for all available options to configure dstore-dist within a Recursor pod.

6.11 Dstore-dist: DNSdist

To deploy a set of DNSdist instances with dstore-dist enabled, include a 'dstoredist' configuration node in the DNSdist instance with a set of destinations and routes configured. The example below shows a basic DNSdist deployment with dstore-dist enabled to distribute protobuf messages to a Kafka endpoint:

```
dnsdists:
  mydnsdist:
    replicas: 2
    dstoredist:
      destinations:
        mydestination:
          type: kafka
          kafka:
            addresses:
              - my.kafka.local:9092
            topic: my_topic
      routes:
        myroute:
          destinations:
            - mydestination
    luaScript: |-
      addResponseAction(QNameRule("example.test.local"),␣
↪RemoteLogResponseAction(ccdstoredist))
```

The above will result in a DNSdist deployment with a dstore-dist sidecar. In addition, a 'RemoteLogger' object named 'ccdstoredist' becomes available for use in DNSdist Lua script and via the DNSDistRule objects.

In the example, the 'ccdstoredist' RemoteLogger is configured to send protobuf messages to the dstore-dist sidecar, which will then distribute them to a kafka topic named 'my_topic' via a Kafka endpoint at 'my.kafka.local:9092'.

The 'luaScript' will ensure a protobuf message is generated by calling the action 'RemoteLogResponseAction(ccdstoredist)' when the response is generated (the 'addResponseAction') for a query matching the 'QNameRule' of 'example.test.local'.

The same example using DNSDistRule instead of luaScript:

```
dnsdists:
  mydnsdist:
    replicas: 2
    dstoredist:
      destinations:
        mydestination:
          type: kafka
          kafka:
            addresses:
              - my.kafka.local:9092
            topic: my_topic
      routes:
```

(continues on next page)

(continued from previous page)

```
        myroute:
          destinations:
            - mydestination
    rulegroups:
      - dstore-dist-rules

rulesets:
  dstoredist-action-ruleset:
    group: dstore-dist-rules
    type: DNSDistRule
    priority: 100
    rules:
    - name: Example logger
      combinator: AND
      selectors:
        - QName: "example.test.local"
      action:
        RemoteLog:
          remoteLogger: ccdstoredist
```

Lastly, you can also enable dstore-dist in a DNSdist Pod with embedded Recursor instances to capture traffic from both DNSdist and recursor. To do this, configure DNSdist as usual to enable dstore-dist, then add a 'dstoredist' configuration to the nested recursor configuration. For example:

```
dnsdists:
  mydnsdist:
    replicas: 2
    dstoredist:
      destinations:
        mydestination:
          type: kafka
          kafka:
            addresses:
              - my.kafka.local:9092
            topic: my_topic
      routes:
        myroute:
          destinations:
            - mydestination
    luaScript: |-
      addResponseAction(AllRule(), RemoteLogResponseAction(ccdstoredist))
    recursor:
      replicas: 2
      dstoredist:
        inbound:
          enabled: true
        outbound:
          enabled: true
```

Note: The destinations and routes do not need to be configured on the nested recrusor, as this is derived from the main DNSdist dstoredist configuration.

In the above example, DNSdist will send protobuf messages for responses to all queries (the 'AllRule()' rule) and Recursor will send protobuf messages for all queries and corresponding answers for inbound traffic (received queries) and outbound traffic (queries sent during resolution

and/or forwarding).

You can refer to the 'Reference' guide for all available options to configure dstore-dist within a DNSdist pod.

6.12 Dstore-dist: Standalone

To deploy a set of dstore-dist instances in dedicated pods, configure the instances under the top-level 'dstoredists' node. The example below shows a standalone dstore-dist deployment enabled to distribute protobuf messages to a Kafka endpoint:

```
dstoredists:
  mydstoredist:
    replicas: 2
    destinations:
      mydestination:
        type: kafka
        kafka:
          addresses:
            - my.kafka.local:9092
          topic: my_topic
    routes:
      myroute:
        destinations:
          - mydestination
```

The above will result in a dstore-dist deployment configured to distribute all received protobuf messages to a kafka topic named 'my_topic' via a Kafka endpoint at 'my.kafka.local:9092'.

To configure Recursor and DNSdist deployments to use this standalone dstore-dist for aggregation, filtering and/or distribution, you can reference it by name:

```
dstoredists:
  mydstoredist:
    replicas: 2
    destinations:
      mydestination:
        type: kafka
        kafka:
          addresses:
            - my.kafka.local:9092
          topic: my_topic
    routes:
      myroute:
        destinations:
          - mydestination

recursors:
  myrecursor:
    replicas: 2
    dstoredist:
      inbound:
        enabled: true
      outbound:
        enabled: true
```

(continues on next page)

(continued from previous page)

```
      dstoredists:
        - group: mydstoredist

dnsdists:
  mydnsdist:
    replicas: 2
    dstoredist:
      dstoredists:
        - group: mydstoredist
    luaScript: |-
      addResponseAction(AllRule(), RemoteLogResponseAction(ccdstoredist))
```

In the above example the DNSdist and Recursor instances will have a dstore-dist sidecar which will distribute all generated protobuf messages to the dstoredists instance set named 'mydstoredist'

7 Security

7.1 Verification of OCI artifacts

CloudControl Helm Charts and container images are made available as OCI artifacts. To allow verification of authenticity of these artifacts, each of them is signed using cosign (https://docs.sigstore.dev/signing/quickstart/).

The following public key can be used to verify the CloudControl OCI artifacts:

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE8V1VF5mq63jGEe8vfUg87pQKQ/qN
201vIRUbfaJrbYgToDfCIg+q90FKSLvxssho8AyWgvGoEf1UQycf/QbyJA==
-----END PUBLIC KEY-----
```

This verification can be performed via several methods, below we will show how to do this manually using the cosign CLI. Prerequisite to be able to do this manually is to download and install *cosign*, which can be done via the above link.

First, make sure to save the above public key to a local file, we will use */tmp/cc.pub* in the below examples.

Then, make sure you have a login configuration locally for the OX registry, this can be done either via Docker (if installed) or cosign using the login command. Example using cosign:

```
cosign login registry.open-xchange.com --username=REGISTRY_USER --password=REGISTRY_PASS

# Note: You can also feed the password in via stdin
# See `cosign login --help` for more options.
```

Now you can use cosign to verify the signatures of the CloudControl OCI artifacts. To do this, you can use the following command:

```
cosign verify --key=/tmp/cc.pub registry.open-xchange.com/<repository>/<name>:<tag>
```

Expected output upon successful verification should include:

```
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - Existence of the claims in the transparency log was verified offline
  - The signatures were verified against the specified public key

<JSON encoded details of the verification>
```

8 Troubleshooting

8.1 Accessing DNSdist console

DNSdist offers a commandline console which allows for debugging of issues and retrieving statistics. In Cloud Control deployments this is enabled by default and can be accessed via kubectl's exec command. This chapter will show how to gain access to the console and a few sample commands. For full documentation on the DNSdist console you can refer to: DNSdist reference guide

---

Note: While DNSdist's console exposes methods to modify a running instance we highly encourage users NOT to do this. Any change made to a running instance using the console will not persist and will not be synchronized to other DNSdist instances.

The following command can be used to gain access to the console:

```
# Pod name (make sure to replace with an existing DNSdist pod's name)
POD=mydnsdist-775cbf55d9-qjtks

# The namespace
CC_NAMESPACE=my-namespace

# Kubectl command to access the DNSdist console
kubectl exec -it $POD --namespace=$CC_NAMESPACE -c dnsdist -- dnsdist -c \
--config=/config/dnsdist.conf
```

You should then be presented with a console session as follows:

```
* dnsdist-state loaded
* Control socket set to 127.0.0.1:5199 with provided key
>
```

To see the status of the `recursor` and/or `resolver` instances that DNSdist will send queries to use `showServers()`:

```
> showServers()
#   Name                Address             State  Qps  Ord Wt  Queries  Pools
0   Endpoints/my-namespa 10.244.1.7:5353        up  0.0    1  1      546
1   Endpoints/my-namespa 10.244.1.8:5353        up  0.0    1  1        0
2   Endpoints/my-namespa 10.244.1.9:5353        up  0.0    1  1        0
3   Endpoints/my-namespa 149.112.112.112:53     up  0.0    1  1        0  external
4   Endpoints/my-namespa 9.9.9.9:53             up  0.0    1  1        0  external
All                                                0.0              546
```

Show the pools using `showPools()`:

```
> showPools()
Name        Cache   ServerPolicy       Servers
                    leastOutstanding   10.244.1.7:5353, 10.244.1.8:5353, 10.244.1.9:5353
external            leastOutstanding   149.112.112.112:53, 9.9.9.9:53
```

List all rules with `showRules()`:

```
> showRules()
#   Name        Matches     Rule            Action
0               0           qtype==ANY      set rcode 5
1               0           qtype==AAAA     to pool external
```

8.2 Pod Events

Cloud Control pods, primarily DNSdist, emit events to indicate potential problematic behaviour and provide tracability into the synchronisation processes.

There are many ways to list events in a namespace, for a pod, etc.. In the below example we'll use kubectl's `get event` to show the events for a specific pod, but in a production setting we recommend capturing these in your logging/monitoring infrastructure.

```
# Pod name (make sure to replace with an existing DNSdist pod's name)
POD=mydnsdist-775cbf55d9-qjtks

# The namespace
CC_NAMESPACE=my-namespace

# Kubectl command to list recent events emitted by a pod in a given namespace
kubectl get event --namespace=$CC_NAMESPACE --field-selector involvedObject.name=$POD
```

Examples of events generated by DNSdist pods (reformatted to fit):

```
# Event emitted by agent when a rule is updated
Type: Normal
Reason: DNSDistRuleUpdated
Object: pod/mydnsdist-775cbf55d9-gvjwk
Message: DNSDistRule 'my-namespace/block-traffic-ruleset' has been synchronised

# Event emitted by agent when a recursor/resolver endpoint changes
Type: Normal
Reason: EndpointsUpdated
Object: pod/mydnsdist-775cbf55d9-gvjwk
Message: Endpoints 'my-namespace/recursor-myrecursor' has been synchronised

# Event emitted by Kubernetes when a readiness probe fails
Type: Warning
Reason: Unhealthy
Object: pod/mydnsdist-775cbf55d9-gvjwk
Message: Readiness probe failed: HTTP probe failed with statuscode: 500
```

9 Compatibility

9.1 Kubernetes

All Cloud Control releases are tested against the Kubernetes versions which are actively supported by the Kubernetes project when the Cloud Control release is made available. As Cloud Control development does not follow the same cadence as Kubernetes, some Cloud Control releases may have been validated against Kubernetes releases that are subsequently no longer in active support by the Kubernetes project.

The releases supported by the Kubernetes project can be found at the following location: https://kubernetes.io/releases/

9.1.1 Validated releases

This version of Cloud Control has been validated against the following Kubernetes releases:

- 1.29
- 1.28
- 1.27
- 1.26
- 1.25

9.2 OpenShift

As of release 2.5.0, Cloud Control releases are tested against the OpenShift versions which are actively supported by Red Hat and available for deployment via 'Red Hat OpenShift Service on AWS' (ROSA) at the time of each specific Cloud Control release.

9.2.1 Validated releases

This version of Cloud Control has been validated against the following OpenShift releases:

- 4.14
- 4.13
- 4.12