OX PowerDNS Cloud Control

Overview

Dec 15, 2021

Release 2.0.0-BETA1

Contents

1 Cloud Control **1**
1.1 Simple deployment . . . 1
1.2 Complex deployment . . . 2
1.3 Rules & Actions . . . 2

2 Cloud Control on Kubernetes **3**
2.1 DNSdist . . . 3
2.1.1 DNSdist agent . . . 4
2.2 Recursor . . . 5
2.3 Resolver . . . 6
2.4 Ruleset . . . 6

3 Getting Started **7**
3.1 Install Tools . . . 7
3.2 Download Helm Chart . . . 7
3.3 Helm Chart configuration . . . 8
3.3.1 Registry Credentials . . . 8
3.3.2 Cluster Networking . . . 8
3.3.3 Deploying Recursor . . . 10
3.3.4 Adding DNSdist . . . 11
3.3.5 Adding an external Resolver . . . 12
3.3.6 Adding a DNSdist rule . . . 14
3.3.7 Using DNSdist rules to route traffic . . . 15
3.3.8 Separating config into multiple files . . . 17
3.3.9 Exposing dnsdist . . . 18

4 Troubleshooting **19**
4.1 Accessing DNSdist console . . . 19
4.2 Pod Events . . . 20

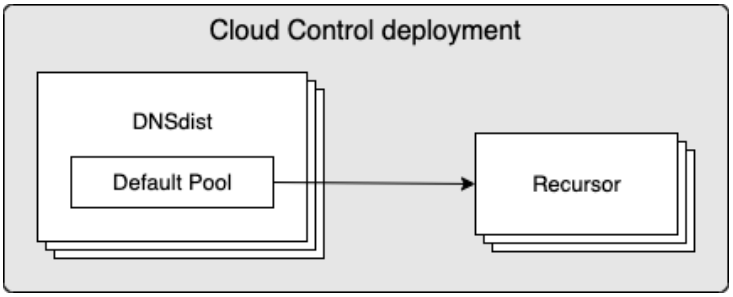
1 Cloud Control

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

- OX PowerDNS DNSdist - A DNS, DoS and abuse-aware loadbalancer that brings out the best possible performance in any DNS deployment.
- OX PowerDNS Recursor - A high-performing, low latency DNS resolver

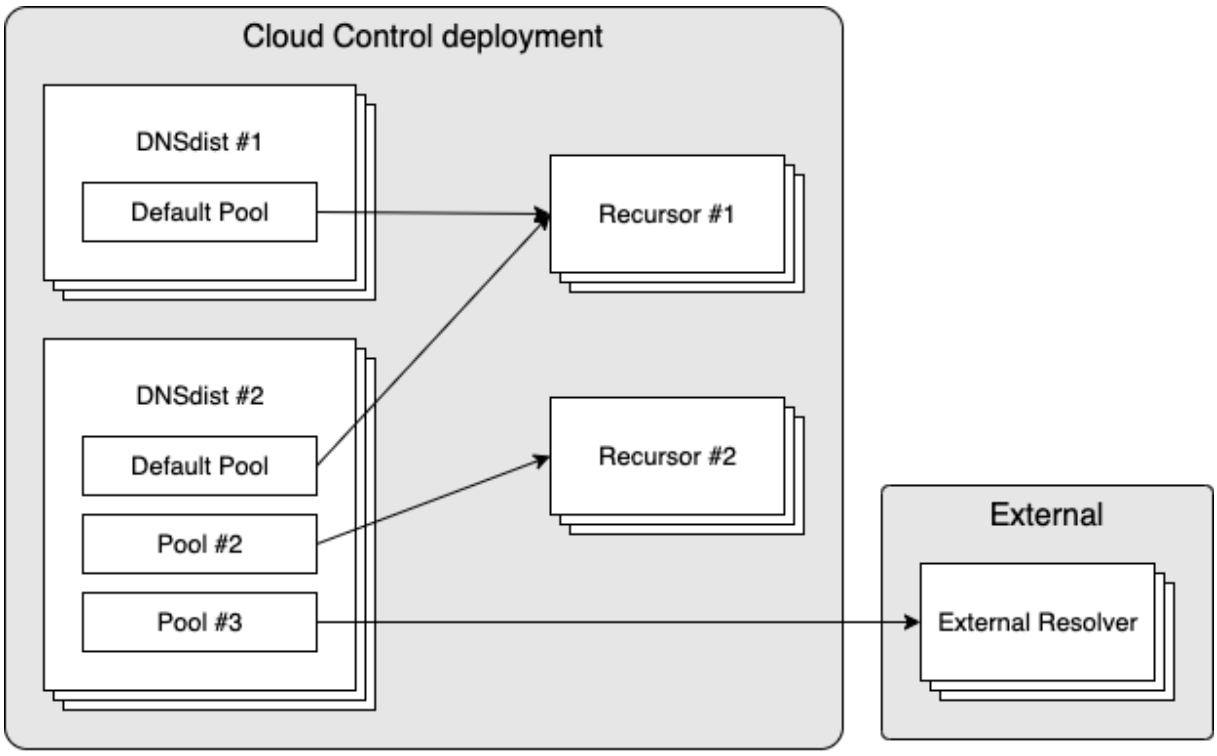
1.1 Simple deployment

In its most simple deployment scenario, 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 Complex deployment

In a more complex deployment you can deploy multiple sets of DNSdist & Recursor instances, with DNSdist using multiple pools to send traffic to the different Recursors. In addition, DNSdist can be configured to send traffic to DNS resolvers which are not part of the Cloud Control deployment.



1.3 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
'Source' of request in '130.161.0.0/16'	'let pool #2 handle the request'
'Qname' of request matches a regex	'let pool #3 handle the request'
'Qtype' of request is 'ANY'	'send response with REFUSED'

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

2 Cloud Control on Kubernetes

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

- **dnsdist** - Definition of a set of OX PowerDNS DNSdist instances and corresponding configuration
- **recursor** - Definition of a set of OX 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

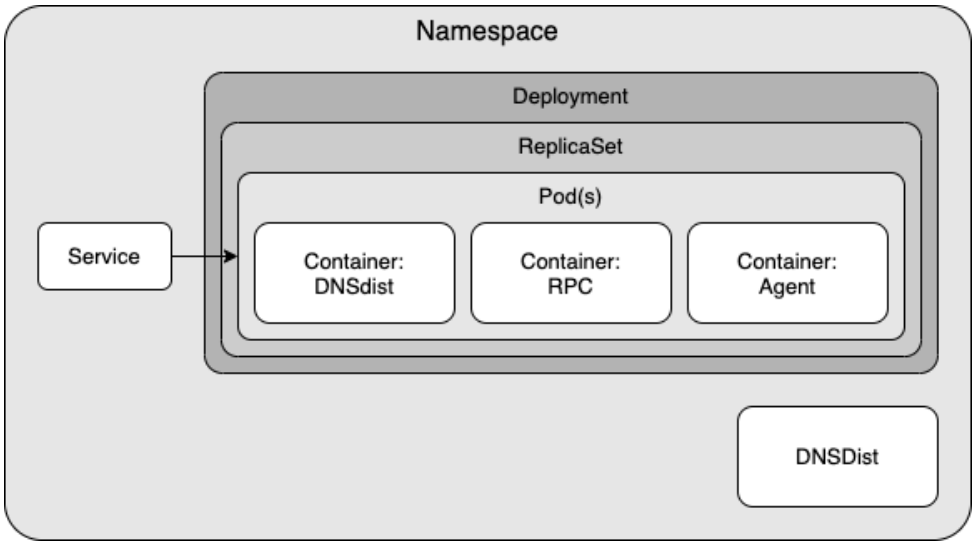
The following sections discuss each in more detail.

2.1 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:

- **DNSdist** - Runs DNSdist and is responsible for handling the actual inbound DNS queries.
- **RPC** - Runs an API that is responsible for handling JSON messages over HTTP from the agent and forwarding them to dnsdist as RPC. Future versions of Cloud Control will see the need for this container removed to allow for direct communication between the agent & 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.

2.1.1 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 deployments.

Items which are watched by the agent are:

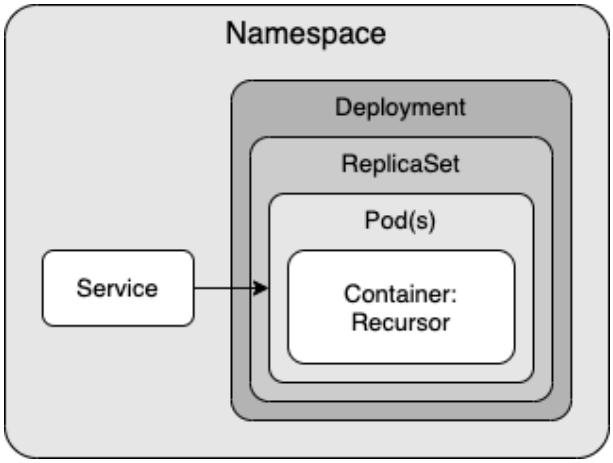
Kind	Purpose
DNSDist	The object which contains the configuration details for a DNSdist deployment. If any updates are detected the agent will attempt to update the configuration of DNSdist without having to restart it.
Pod	The agent watches the pod which it is a part of. Particularly the statuses of 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 both Recursor & resolver deployments.

2.2 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
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:

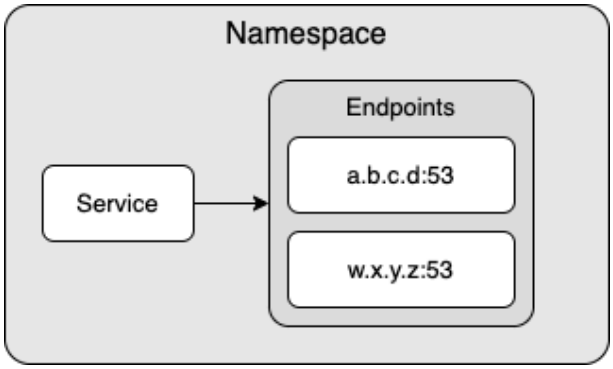


2.3 Resolver

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 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:



2.4 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

3 Getting Started

3.1 Install Tools

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

- Kubectl (Configured for your target Kubernetes cluster)
- Helm v3 (https://helm.sh/docs/intro/install/)

3.2 Download Helm Chart

Cloud Control 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 make Helm aware of the Cloud Control repository (replace username & password with your OX registry credentials):

```
helm repo add cloudcontrol https://registry.open-xchange.com/chartrepo/cloudcontrol \
--username=REGISTRY_USERNAME_HERE --password=REGISTRY_PASSWORD_HERE
```

Once the repository has been added you can pull the Cloud Control Helm Charts. To pull the powerdns Helm Chart and export it to your current working directory use the following commands:

```
# The release we're working with
CCTAG=2.0.0-BETA1

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

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

3.3 Helm Chart configuration

The Cloud Control Helm Charts have a large amount of configurable options, which are detailed in the reference documentation. In the next few chapters the most important parts are discussed.

3.3.1 Registry Credentials

Since the Cloud Control images are in a protected repository there is a requirement to configure credentials in the Helm Chart input YAML file. These need to be configured with the following block:

```
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.

3.3.2 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.

3.3.3 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.

3.3.4 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:

```yaml
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
```

3.3.5 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
```

3.3.6 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
```

3.3.7 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
```

3.3.8 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:

```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:

```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:

```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
```

3.3.9 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.

4 Troubleshooting

4.1 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
```

4.2 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
```