Contents

[Introduction 2](#_Toc36374073)

[Architecture 2](#_Toc36374074)

[Background 2](#_Toc36374075)

[Deployment Options 3](#_Toc36374076)

[Multi-tenancy 3](#_Toc36374077)

[APICast/GW architecture 3](#_Toc36374078)

[Network partition 8](#_Toc36374079)

[Install APICast on OpenShift 9](#_Toc36374080)

[Create helm file for APICast 10](#_Toc36374081)

[**values.yaml:** 11](#_Toc36374082)

[deployment.yaml 12](#_Toc36374083)

[service.yaml: 13](#_Toc36374084)

[Managing 13](#_Toc36374085)

[Getting started 13](#_Toc36374086)

[Adding you app 14](#_Toc36374087)

[Useful Parameters 16](#_Toc36374088)

[Testing 17](#_Toc36374089)

[Scenarios performed 17](#_Toc36374090)

[Support Considerations 17](#_Toc36374091)

# Introduction

This document outlines the architecture, installation steps and configuration of 3Scale APICasts (GWs). It is intended for a general audience, so you can skip the parts that aren’t relevant to you.

To follow the instructions in the guide, you should have:

* OpenShift 4.2+ installed
* An access to 3Scale 2.7+ from the OpenShift you are planning to work one
* A user for 3Scale with the needed permissions (If in Azure – the Cloudlet team manages the 3Scale, if in Army you should talk to 3Scale team in IDFCTS)

# Architecture

## Background

To get a basic understanding of 3Scale, please take a look here:

<https://developers.redhat.com/products/3scale/overview>

If you are going to implement some of the components outlined here, a deeper understanding is recommended. You should read at least the ‘getting started’ guide at RedHat’s site.

If you are a manager, or just want a shallow understanding:

3Scale is RedHat’s API Management solution. It allows you to wrap you APIs, and provide:

* Metrics about usage
* Call limiting for you APIs
* Authentication with various methods
* Policies that filter requests

The product allows you to customize the access to your private APIs, saving you from implementing it yourself.

It is a software that consists if multiple parts, most easily divided as:

* Main 3Scale. A 14-container deployment which allows all 3Scale’s functionality. It stores the APIs, the keys, all your settings, the users, etc.
* An APICast/GW. This container is responsible for passing requests to your API. It validates the request, can apply some policies to filter requests, and forwards only the authorized requests to your API. If you use 3Scale, your application will not be accessible from anywhere except the GW.

When the GW boots, it downloads all its configuration for the main 3Scale

## Deployment Options

In the previous part you understood what a APICast/GW is and what are the capabilities of 3Scale. Now we will talk about how we should use the product in the Cloudlet.

### Multi-tenancy

In 3Scale, a tenant is a logical unit that can be responsible for its own users, APIs and permissions. A tenant has an admin, which can add users and manage them. In a single tenant, no 2 APIs with the same endpoint can exist. The best use case for tenants are different organizations - for example, in the IDFCTS 3Scale, the Navy can have its own tenant and manage it.

An important thing to note: tenants are only a logical division, all of them ‘exist’ only in the main 3Scale. In the case of a network split, the same functionality will be available as if you belonged to one big tenant.

In our case, it doesn’t make sense to use a tenant per Cloudlet. There is an overhead for managing multiple tenants and no benefit. The sensible architecture is one of:

* Using the same tenant as ‘Mamram’, with all other ‘Tikshuv’ applications running on 3Scale
* A dedicated tenant for all cloudlets.

In practice, the difference between both options is very small.

### APICast/GW architecture

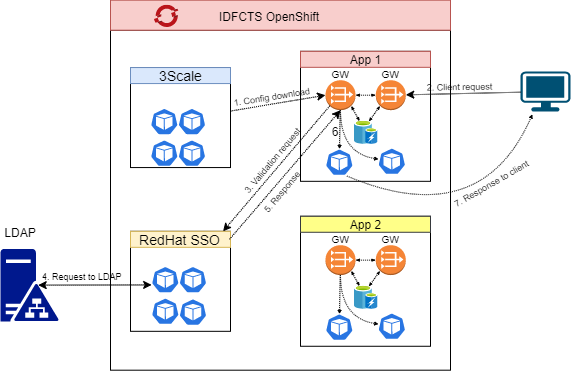
From now we will call the APICast GW for simplicity’s sake.

There are 3 options for deploying GWs (There is a fourth option of using 3Scale default GWs, which are deployed in the main 3Scale namespace, but we will not consider it for production uses):

1. Dedicated GWs per application, in the application’s namespace
2. Dedicates GWs per application, in a different namespace
3. Shared GWs

Before diving into each option, let’s overview the flow of C2B (for UI) in the main OpenShift. Take note that the GW deployment is not necessary in the App namespace and will be discussed later.

In the app’s namespace, the developers should deploy 2 GWs (for HA), after configuring all their APIs in the 3Scale web portal. In addition, a Redis container is deployed (Only necessary if using OpenID, which we plan to do). The Redis is storing authentication information, needed for the session to stay active.

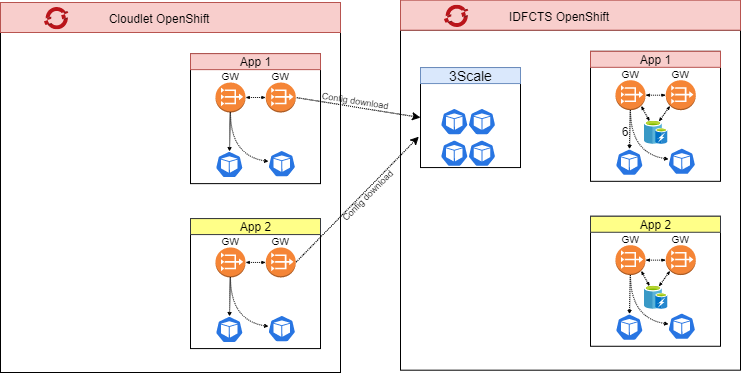


1. After initialization, the GWs download their config from 3Scale
2. A client requests the UI of the application
3. The GW talks with RedHat SSO to authenticate the client
4. RedHat SSO validates using LDAP (Mamram’s AD)
5. Redhat SSO returns a response
6. If the client is allowed to access the UI by the API policy, the request is forwarded to the application.
7. The client receives a response.

The flow of C2B for APIs (‘B2B’, where one application is the client of another) is even simpler. If an application provides the right token for the requested API, and the request is allowed by the policy, the request is forwarded (without getting into TLS termination), no RedHat SSO in the flow.

**A. Dedicated GWs per application, in the application’s namespace**

Now that we (hopefully) understand the flow, let’s return to the GW deployment options. In the following architecture, the GW-per-application approach is depicted (without SSO in the Cloudlet, thus the GWs are without an attached Redis container). In this use-case, each application deploys its own GWs in its namespace.



Pros:

* Communication between GWs and app is very simple (in networking terms, because they sit in the same namespace)
* Best security separation
* A heavy load by one application’s clients doesn’t affect other applications’ GWs

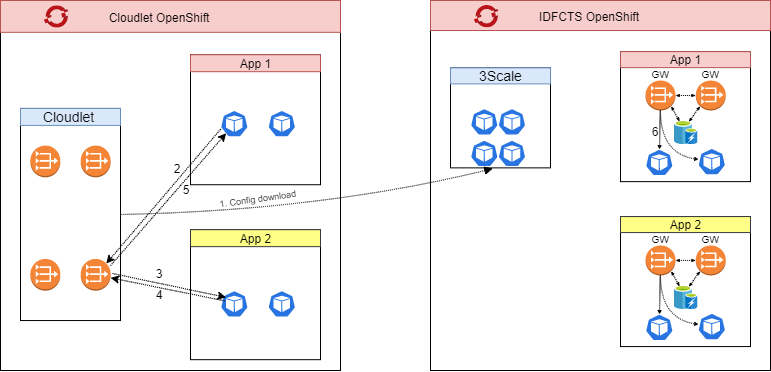
Cons:

* More work for the application developers (as will be disscussed in the next scenario)
* Harder to manage the GW lifecycle for the Cloudlet team
* A large amount of GWs (harder to manage)

1. **Dedicates GWs per application, in a different namespace**

In this scenario, each application will have it’s own pair of GWs. The GWs will be created in a dedicated namespace, managed by the Cloudlet team. Each GW will have access only to the namespace of the application it is responsible for, using network policy and labels. E.g., Ztube’s GWs will have a label ‘Ztube’, and there will be a network policy to allow communication to Ztube’s namespace from each pod with label ‘Ztube’ in Cloudlet’s namespace.

This is the architecture that the 3Scale team currently plans to use to manage the applications in the main OpenShift. That means that a similar solution has a big advantage for the developers.



Pros:

* Easier to manage (from one namespace)
* Less work for developers
* No performance dependence between GWs of different applications.

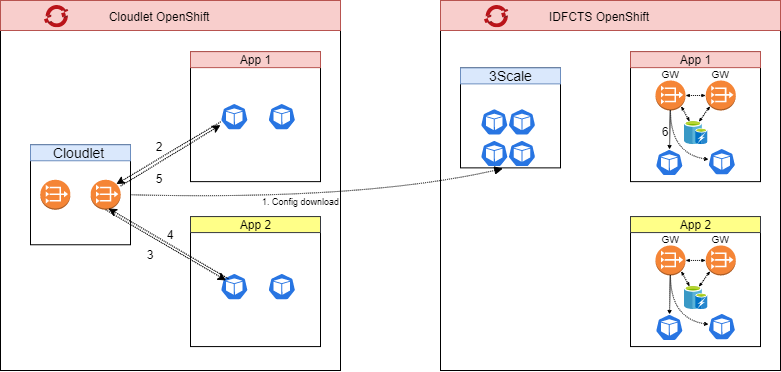
Cons:

* A large amount of GWs
* Security is less ‘tight’
* Networking is more complex

1. **Shared GWs**

In this scenario, there will be only one set of GWs (with 2 or more, depending on performance needs) deployed. The GWs will download the configuration for all APIs, and their pods will be allowed to communicate with all the namespaces of applications that expose APIs (again using labels and network policies).

This is the easiest solution to manage at scale, with the least amount of work for developers and the Cloudlet team. The biggest drawback is the security issue, allowing the GW pods to communicate with all the namespaces in the OpenShift. In my opinion, this issue is minor, the communication will be available only on the API ports. An intruder who gained control over the GW can’t do much, he will still need the correct API keys and certificates. It is still a lot more secure than the current architecture in the IDFCTS (pre-3Scale).



Pros:

* Easiest to deploy and manage, both for developers and Cloudlet team
* Least amount of resources

Cons:

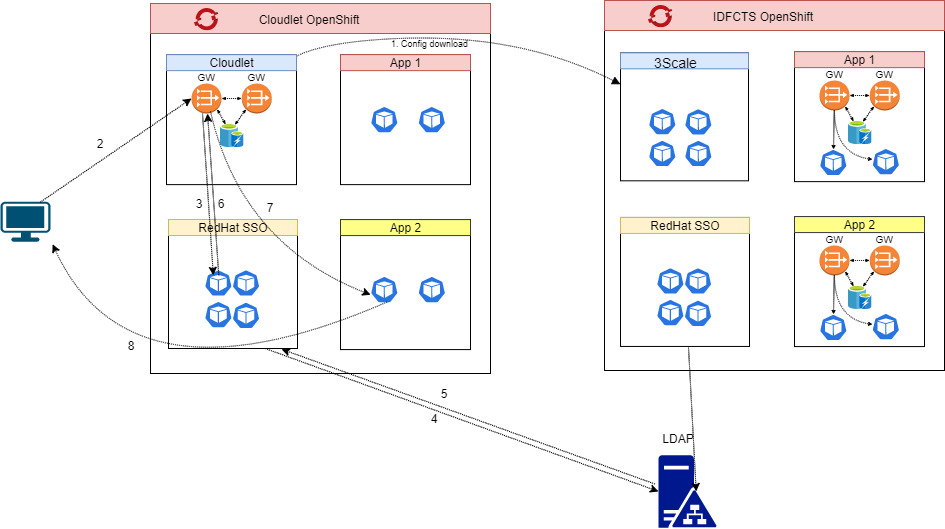
* Performance issues may arise, with one application ‘bombing’ the GWs with requests from its clients. This can be solved by deploying multiple GWs, even 5 or 6.
* Security issues, as mentioned above
* Networking is more complex

**To summarize**, I think the last scenario is best for our case. The second scenario is also plausible, by will require more work from our team.

RedHat SSO integration

Read here about RedHat SSO: <https://access.redhat.com/products/red-hat-single-sign-on>

Depicted in the drawing is the flow of a client’s request to an application running on the Cloudlet. The GW deployment is in the application namespace, but it doesn’t matter for the RedHat SSO topic and would’ve behaved similarly in the other deployment types. In all of them there is a Redis container alongside the GWs in needed.



1. The GW download its config from the main 3Scale.
2. The client initiates a request.
3. The GW authenticates the client with RedHat SSO
4. RedHat SSO checks the identity in the provided LDAP (Mamram’s Microsoft Active Directory in our case)
5. The LDAP responses
6. RedHat SSO responses to the GW, which checks if the given identity is allowed to access the API
7. If it does, the GW forwards the request to the application
8. The client receives the answer.

The biggest issue in that architecture is the location of the LDAP. If the LDAP is not located at the Cloudlet, in case of a network partition the users will not be able to authenticate. Yaakov suggested an elegant solution to our problem.

RedHat SSO can be operated in two modes:

1. As a proxy, authenticating users using an LDAP
2. As the LDAP itself, mapping the information from a given LDAP to RedHat SSO’s DB

The second use-case is suitable for us. In the case of a network partition, no just-created users will be able to authenticate, and all of the signed-in users will need to re-authenticate.

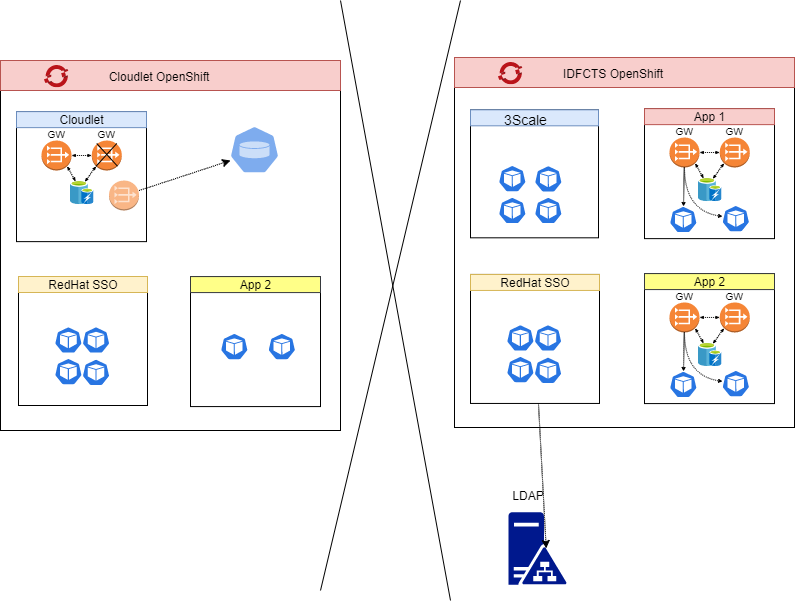
The issue of the copying strategy needs to be figured out, what’s possible in RedHat SSO and what are our needs. Copying all the AD every X hours\days is a huge amount of data, and maybe our use-case is to allow authentication only for previously authenticated users.

### Network partition

We already answered one issue with network partition, which is the authentication of users. The other big issue it the downloading of configuration for the GWs. As seen in every drawing, the first step for running a GW is to download its config from the main 3Scale. In case the main 3Scale is not available, currently, there is no way to deploy a new GW. That means that if a GW crashed, or needs an update, it stays down.

A solution for that use-case is the ability to boot a GW from an OpenShift PV, which will hold the latest configuration. We opened a RFE for RedHat, using Yaakov Preiger’s help. That’s the case:

<https://access.redhat.com/support/cases/#/case/02614837>



# Install APICast on OpenShift

Follow this guide:

<https://access.redhat.com/documentation/en-us/red_hat_3scale_api_management/2.7/html/installing_3scale/install-threescale-on-openshift-guide#using-apicast-with-threescale-on-openshift>

The section “Connecting APIcast from a different OpenShift cluster” is the relevant one, with some corrections.

1. Creating the token: After creating the token as shown in the steps, first check if it works using the following command:

*curl -v -k "https://3scale-admin.apps.anton-openshift.cloudlet-dev.com/admin/api/services.json?access\_token=TOKEN*

If you get *“{"error":"Your access token does not have the correct permissions"}”,* it means your token is incorrect.

First, check here of the token exists:

<https://3scale-admin.apps.anton-openshift.cloudlet-dev.com/p/admin/user/access_tokens> (replace the address and the user name with the relevant ones).

If it doesn’t it may be that you created the token in a different places (for example in the master api, for managing tenants).

1. After you verified the token works correctly, create a secret called **apicast-configuration-url-secret**. The apicast will use this secret to authenticate to the main 3Scale.

oc create secret generic apicast-configuration-url-secret --from-literal=password=https://TOKEN@3scale-admin.apps.anton-openshift.cloudlet-dev.com \

--type=kubernetes.io/basic-auth

1. Download the API-cast yml from here, or install directly:

*oc new-app -f https://raw.githubusercontent.com/3scale/apicast/master/openshift/apicast-template.yml*

1. In the Workloads->Deployment configs, change the following parameters of the apicast config:

APICAST\_CONFIGURATION\_CACHE: 60

APICAST\_PATH\_ROUTING: true

Read [here](#_Useful_Parameters:) for explanation

**Note:** You can change the values **before** the creation of the app, for that you’ll need to download the template and edit it.

1. Create a route that exposes the service that is automatically created (expose the one at 8080).

## Create helm file for APICast

Helm is a tool for managing your Kubernetes resources. You can learn more here: <https://helm.sh/docs/chart_template_guide/>

Follow these steps to deploy your app using helm.

1. Go to the project you want to deploy the APICast to (or create a new one).
2. Make sure the secret “apicast-configuration-url-secret” exists in your current project. Use the steps [here](#_Install_APICast_on).
3. Create a new helm:

*helm create HELM\_NAME*

1. You will see a folder HELM\_NAME created. Delete all the content inside the values.yml folder, and all the files inside the “templates” directory.
2. Now you need to create your templates, based on the latest apicast version.

I used the one from here:

<https://raw.githubusercontent.com/3scale/apicast/master/openshift/apicast-template.yml>

* First copy the deployment config section to a “deployment.yaml” file in the templates directory. Take notice that in the original yaml it was an element inside the ‘object’ array, and you in your file it should be a standalone object. So first thing fix the indentation, remove the hyphen, and delete the first 2 spaces in each line.
* Next, you need to change the reference to the parameters, which we will insert in the values.yml in a later step. In Helm, you can reference the values passed to the chart (from the values file or other methods, like the cli) using the following syntax:

*{{ .Values.PARAM\_NAME }}*

To make stuff easier and not reference the Values object in each line, we can put this line at the beginning of our file:

*{{- with .Values }}*

And this one at the end:

*{{- end }}*

Now each time we reference a parameter, it will be searched only in the Values object.

* In addition, you need to change the apiVersion to:

*apps.openshift.io/v1*

By default, helm uses Kubernetes resources, and deplyomentConfig isn’t one of them, so you need to specify it is an OpenShift object. You can see an example here: <https://github.com/sclorg/nodejs-ex/blob/master/helm/nodejs/templates/deploymentconfig.yaml#L2>

* Now do the same changes for the service.yaml, using the Service object in the apicast template.
* The last thing to do is to convert the parameters section to key-value pairs, looking like:

*PARAMETER: "X"*

I used Notepad++ for it. Change only 2 values from the default:

*CONFIGURATION\_CACHE: "60"*

*PATH\_ROUTING: "true"*

Add all your parameters to the values.yaml object.

My files looked like this:

**values.yaml:**

*CONFIGURATION\_URL\_SECRET: apicast-configuration-url-secret*

*CONFIGURATION\_FILE\_PATH:*

*IMAGE\_NAME: 'quay.io/3scale/apicast:master'*

*DEPLOYMENT\_ENVIRONMENT: production*

*APICAST\_NAME: apicast*

*APICAST\_WORKERS:*

*RESOLVER:*

*SERVICES\_LIST:*

*CONFIGURATION\_LOADER: boot*

*LOG\_LEVEL: "warn"*

*PATH\_ROUTING: "true"*

*RESPONSE\_CODES: "false"*

*CONFIGURATION\_CACHE: "60"*

*REDIS\_URL:*

*MANAGEMENT\_API: "status"*

*OPENSSL\_VERIFY: "false"*

*REPORTING\_THREADS: "0"*

### deployment.yaml

*apiVersion: apps.openshift.io/v1*

*kind: DeploymentConfig*

*metadata:*

*{{- with .Values }}*

*name: "{{.APICAST\_NAME}}"*

*spec:*

*replicas: 2*

*selector:*

*deploymentconfig: "{{.APICAST\_NAME}}"*

*strategy:*

*type: Rolling*

*template:*

*metadata:*

*labels:*

*deploymentconfig: "{{.APICAST\_NAME}}"*

*spec:*

*containers:*

*- env:*

*- name: THREESCALE\_PORTAL\_ENDPOINT*

*valueFrom:*

*secretKeyRef:*

*name: "{{.CONFIGURATION\_URL\_SECRET}}"*

*key: password*

*- name: THREESCALE\_CONFIG\_FILE*

*value: "{{.CONFIGURATION\_FILE\_PATH}}"*

*…… (All the environment variables)*

*…….*

*image: "{{.IMAGE\_NAME}}"*

*imagePullPolicy: Always*

*name: "{{.APICAST\_NAME}}"*

*livenessProbe:*

*httpGet:*

*path: /status/live*

*port: management*

*initialDelaySeconds: 10*

*timeoutSeconds: 1*

*readinessProbe:*

*httpGet:*

*path: /status/ready*

*port: management*

*initialDelaySeconds: 15*

*timeoutSeconds: 1*

*ports:*

*- name: proxy*

*containerPort: 8080*

*protocol: TCP*

*- name: management*

*containerPort: 8090*

*protocol: TCP*

*triggers:*

*- type: ConfigChange*

*{{- end }}*

### service.yaml:

*apiVersion: v1*

*kind: Service*

*metadata:*

*name: "{{.Values.APICAST\_NAME}}"*

*spec:*

*ports:*

*- name: proxy*

*port: 8080*

*protocol: TCP*

*targetPort: 8080*

*- name: management*

*port: 8090*

*protocol: TCP*

*targetPort: 8090*

*selector:*

*deploymentconfig: "{{.Values.APICAST\_NAME}}"*

* Now you are ready to create the files. First run a dry-run to see you didn’t miss anything:

*helm install CURR\_INSTALL\_NAME(CHOOSE) HELM\_NAME --debug --dry-run*

If everything is correct, you can run it without the last two parameters.

* If you didn’t miss anything, the APICast DC should be up and running, with two pods.

# Managing

## Getting started

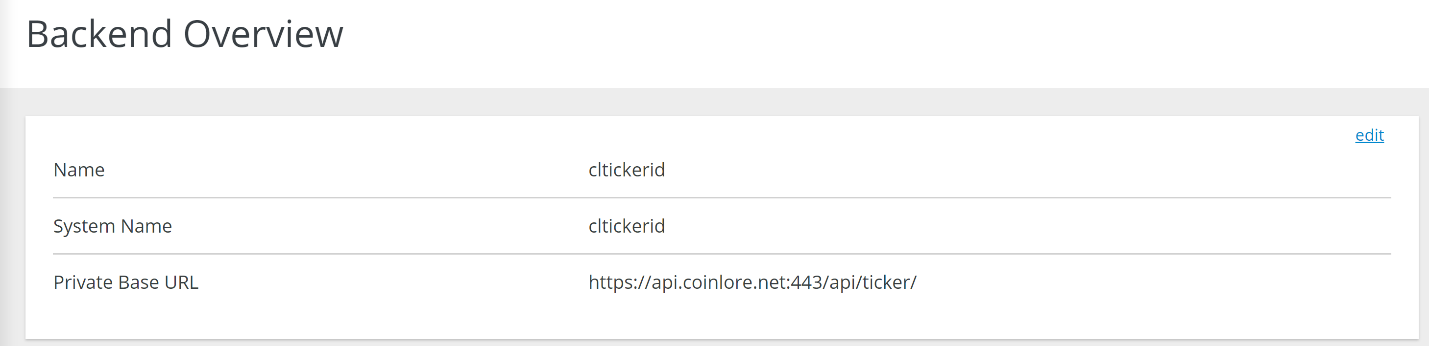
A simple setup of an api can be performed by the wizard here:

<https://3scale-admin.apps.anton-openshift.cloudlet-dev.com/p/admin/onboarding/wizard/intro>

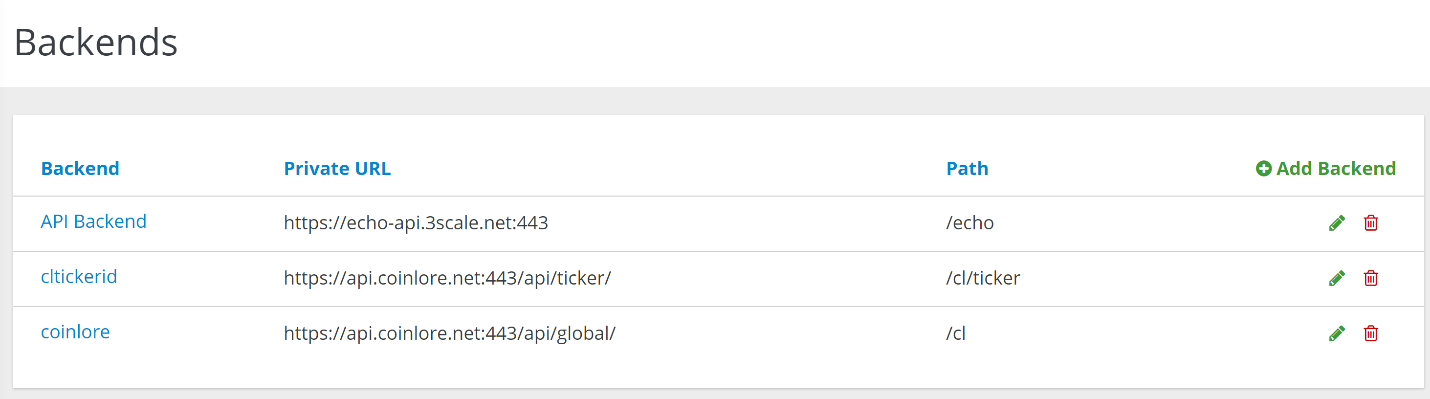
Backend: a collection of internal APIs from your application. containing at least the URL of your API. The backend can optionally have mapping rules, methods and metrics.

Products: Customer facing APIs, that expose your backends. It’s where you define the application plans, and configure APIcast (GW). Can have one or more backends.

For example, if we create a backend like this one:



And add the backend to our product, with the path: /cl/ticker as shown:



After publishing the API in the staging and prod environments (in the configuration tab, in integration menu of your product), we will be able to access the API using the following link:

<https://api-3scale-apicast-staging.apps.anton-openshift.cloudlet-dev.com/cl/ticker/?app_id=b68664e6&app_key=e99a268a04b216aed27da00de0f86071&id=91>

as you can see, I access the general path of the staging API-GW, then the path for the backend, and in the end I can add parameters. The API-GW will pass the query to the private base URL, with the additional parameters.

## Adding you app

The following steps outline how to add your own app, after you already have an APICast (GW) deployed in your OpenShift.

1. Create a project for the app, and deploy it

*oc new-project demo*

*oc new-app --name=demo https://github.com/wicksy/openshift-demo-app*

1. Get the address of the service you want to expose.

You can use:

*oc get svc*

For example: 172.30.186.220:8080 (the port is the one you expose the app on)

1. Create a network policy that allows only the 3scale GW to access your project. As a prerequisite, you need to assign the 3scale project (if not assigned before) a label that you can filter by.

*oc label namespace 3scale-app project=3scale-app*

1. Create a file with the following network policy:

*kind: NetworkPolicy*

*apiVersion: networking.k8s.io/v1*

*metadata:*

*name: allow-from-apicast*

*spec:*

*podSelector: {}*

*ingress:*

*- from:*

*- namespaceSelector:*

*matchLabels:*

***project: 3scale-app***

*podSelector:*

*matchLabels:*

***deploymentconfig: apicast***

And apply the network policy.

*oc apply -f FILENAME*

After applying, only the API GW will be able to talk to your application. If you want your pods to be able to talk with each other, you need to add an additional policy.

For example:

*kind: NetworkPolicy*

*apiVersion: networking.k8s.io/v1*

*metadata:*

*name: allow-same-namespace*

*spec:*

*podSelector:*

*ingress:*

*- from:*

*- podSelector: {}*

1. Now you are finished with OpenShift, and you can move to 3Scale UI.

In your main dashboard, in the “Backends” tab, create a backend. Name it as you want, and in the Private Endpoint enter the service address you got earlier (with http/s according to your app).

1. Go back you the dashboard, and from the product tab create a product.
2. In the integration->backends menu, add the backend you created with the desired path.
3. In the Application->applications plan menu, create an application plan to allow applications to subscribe to your product.
4. In the Audience, click the Group you want to create an application for. Click on “Applications”, and create a new one. Make sure to subscribe to the application plan in the product you created earlier.
5. Go to integration->settings, and change the deployment option to “APICast self-managed”.

Enter the url of your API GW.

1. Now you need to add a mapping rule. To understand why:

We set the parameter “APICAST\_PATH\_ROUTING” (See [here](#_Useful_Parameters:)) in our API GW to true, to allow using the same address for different products (because different apps share the same API).

To know which product (or service, sometimes called) will answer the request, 3Scale go over all the mapping rules and find one that matches. That means the if you leave the default “/” mapping, all the request will go to the first product.

To successfully enable the path routing, you need to match a mapping rule for each backend you have, or follow a convention (for example - /ztube/\* for ztube product, /startrack/\* for startrack product). Make sure there are no conflicts between mapping rules, otherwise you’ll have chaos.

In your case, we will create a mapping rule for /demo, to count hits (doesn’t really matter which metric the mapping rule is applied to).

1. Now go to integration->configuration, and promote your configuration.

<https://developers.redhat.com/blog/2019/07/29/3scale-toolbox-deploy-an-api-from-the-cli/>

# Useful Parameters

APICAST\_CONFIGURATION\_CACHE

Specifies the interval (in seconds) that the configuration will be stored for. The value should be set to 0 (not compatible with boot value of APICAST\_CONFIGURATION\_LOADER) or more than 60. For example, if APICAST\_CONFIGURATION\_CACHE is set to 120, the gateway will reload the configuration from the API manager every 2 minutes (120 seconds). A value < 0 disables reloading.

APICAST\_PATH\_ROUTING

When this parameter is set to *true*, the gateway will use path-based routing in addition to the default host-based routing. The API request will be routed to the first service that has a matching mapping rule, from the list of services for which the value of the Host header of the request matches the *Public Base URL*

# Testing

## Scenarios performed

1. Create a self-managed APICast and expose an API from a private project using Network Policy.
2. Shutdown the main 3Scale and see that the GW works.

## Support Considerations

Per RedHat’s response:

*“3scale API Management Hosted with APIcast API Gateway self-managed is being tested and supported with the latest two versions of APIcast API Gateway made available as part of the latest 3scale releases. Bug fixes and Security fixes will be provided on the latest release of APIcast, and only security fixes will be provided on the previous release.”*

**Take note** that 3Scale 2.7 is officially supported only up to OpenShift 4.2.

Sources:

<https://access.redhat.com/articles/2798521>

<https://access.redhat.com/articles/2798521#3scale-api-management-27-2>

This means that we must upgrade all the GWs periodically, depending on the 3Scale upgrades. Because those upgrades will be managed by IDFCTS’s team, we need to compose a policy which outlines the upgrading process.

**Stil in progress, working with Shiri Morshtein From IDFCTS.**