Dealing with the Docker Image Pull Rate Limitation in Kubernetes

How to Configure Kubernetes Registry Authentication

Docker owns Docker Hub, the de facto standard image registry that contains millions of container images, created by different projects, people and companies to be available for free. Docker hub is the standard image registry in many solutions, hard-coded in container managers and orchestrators like Kubernetes and Docker.

In November 2020, Docker imposed the image pull rate limitation. Anonymous Docker Hub users were limited to 100 container image pull requests per six hours, and free Docker Hub users were limited to 200 container pull requests per six hours. Even if this sounds like a lot, practice is that this limit may be reached soon, in particular in Kubernetes environments where applications are failing.

Solutions for dealing with the pull rate limitation

There are different ways to deal with the image pull rate limitation:

- Avoid images from being pulled unnecessarily
- Use container images from other registries
- Run your own internal container registry
- Use a Docker Account or paid Docker subscription to get up to 5000 pulls per day.

In this article you'll read how to avoid unnecessary image pulls, and how to use Kubernetes Secrets for registry authentication.

Recognizing Image Pull Rate Limit Related Errors

Not everybody will encounter image pull rate related errors. However, if you're trying to learn Kubernetes and running many applications, you might suddenly see the following error messages while running Pods:

```
student@student-virtual-machine:~/ckad$ kubectl get all
NAME                                READY   STATUS
RESTARTS        AGE
pod/fifthnginx-855745fff-4pbww    0/1  ImagePullBackOff 0
30s
pod/sixthnginx-5d976f79-ln827     0/1  ErrImagePull     0
4s
```

In the above code listing, the ErrImagePull and ImagePullBackOff errors indicate the container image couldn't be fetched. To verify that this is because of the pull rate limitation, use **kubectl describe** to investigate the Pod events.

```
Events:
  Type     Reason           Age                   From
Message
  ----     ------           ----                  ----
-------
```

```
  Normal   Scheduled       48s                default-
scheduler  Successfully assigned kube-system/calico-kube-
controllers-85578c44bf-sdn9l to minikube
  Normal   Pulling         32s (x2 over 47s)  kubelet
Pulling image "docker.io/calico/kube-controllers:v3.26.1"
  Warning  Failed          30s (x2 over 43s)  kubelet
Failed to pull image "docker.io/calico/kube-
controllers:v3.26.1": rpc error: code = Unknown desc = Error
response from daemon: toomanyrequests: You have reached your
pull rate limit. You may increase the limit by authenticating
and upgrading: https://www.docker.com/increase-rate-limit
  Warning  Failed          30s (x2 over 43s)  kubelet
Error: ErrImagePull
  Normal   SandboxChanged  25s (x7 over 41s)  kubelet
Pod sandbox changed, it will be killed and re-created.
  Normal   BackOff         24s (x6 over 39s)  kubelet
Back-off pulling image "docker.io/calico/kube-
controllers:v3.26.1"
  Warning  Failed          24s (x6 over 39s)  kubelet
Error: ImagePullBackOff
```

In the event log output above, you see the Failed to pull iamge error, indicating that you have reached your pull rate limit.

Avoiding Unnecessary Image Pulls

In Kubernetes environments, default settings make that many unnecessary image pulls may occur. The first setting is the imagePullPolicy. This setting can be set to three values:

- IfNotPresent: the image is pulled only if it is not present locally
- Always: every time the kubelet launches a container, the image registry is queried to resolve the name to an image digest. If the kubelet already has a container image with that exact digest cached locally, this will be used, otherwise the kubelet pulls the image with the resolved digest
- Never: the image already needs to be locally present or else container startup fails

When a new Pod is created, the imagePullPolicy is set automatically according to the following rules:

- If you don't specify a tag for the container image, imagePullPolicy is automatically set to Always
- If the tag for the container image is :latest, the imagePullPolicy is automatically set to Always
- If the image has a tag that is not :latest, the imagePullPolicy is automatically set to IfNotPresent

From this follows, that you'd better specify a specific image version while working with container images. So don't use **kubectl run nginx --image=nginx** (which will automatically set the imagePullPolicy to Always), but use **kubectl run nginx --image=nginx:1.25**, which won't pull the image if the nginx 1.25 image is already present. It's easy to find the tags that can be used for images, just check the image information on the image registry.

Setting the Default imagePullPolicy

The imagePullPolicy is a property that can be set in the Pod specification. This makes it easy to define it while running applications from YAML file. Alternatively, you can use **kubectl run --image-pull-policy=' '** to set the imagePullPolicy while creating the application. Unfortunately, such an option does not exist while creating Deployments with **kubectl create deploy**. To specify the imagePullPolicy while creating Deployments, you'll have to use YAML files and set the imagePullPolicy in the Deployment spec.template.spec.

Configuring Registry Authentication

To get more than 100 Docker image pulls per 6 hours, you'll need to set up a Docker account. Free accounts can be created on hub.docker.com and allow you to go up to 200 image pulls per 6 hours, the $ 5 monthly developer account allows you to go beyond that. Accounts may also be necessary to authenticate to other registries. After creating an account, you'll need to configure Kubernetes to use it.

Secrets

While using Docker on the command line, you can use the **docker login** command to create a ~/.docker/config.json file, containing the appropriate credentials. This however doesn't work for Kubernetes, as it has its own authentication mechanism. To provide your registry credentials, you'll need a Kubernetes Secret of the docker-registry type. Next, you'll use the imagePullSecrets property on either the pod spec, or as a property of the ServiceAccount that is used to run applications.

Kubernetes Secrets are a generic solution to store sensitive values. The contents of a Secret is base64 encoded, which doesn't offer any real protection but prevents values from being exposed by accident. While creating a secret, the type must be set to either generic, tls, or docker-registry. The docker-registry secret type is used for authentication against any registry that requires authentication.

You can create this secret using command line arguments to specify docker username, email, password and other required parameters, but this increases the risk of accidental exposure (as the values provided will be part of your local command line history). A little bit better is to first login to the registry, using **docker login** or **podman login**. This will create a configuration file ~/.docker/config.json in which your credentials are stored, and next you can create the secret based on this file. The next procedure describes the steps involved:

1. Use **docker login** to provide your registry credentials. If no servername is specified, you will login to hub.docker.com, to login to another registry, provide the registryname as an argument.
2. Verify that the file ~/.docker/config.json was created
3. Use **kubectl create secret docker-registry dockercreds --from-file=.dockerconfigjson=~/.docker/config.json**
4. Verify that the secret was created, using **kubectl get secrets**

Using imagePullSecret

After creating the Secrets, you'll need to configure your applications to use it. To do so, you'll need to specify the imagePullSecret property as spec.imagePullSecret while creating Pods from YAML files, or as spec.template.spec.imagePullSecret while creating Deployments or DaemonSets from YAML files.

Code example: Deployment or DaemonSet spec using ImagePullSecrets

```
spec:
  revisionHistoryLimit: 10
  template:
    spec:
      imagePullSecrets:
      - name: dockercreds
```

While specifying the imagePullSecret for individual applications will work, it's not the most efficient way, as it will have to be repeated for every single application you'll run. A much more efficient approach is to set the imagePullSecret as the default value in the ServiceAccount resource.

Setting imagePullSecrets in ServiceAccounts

To provide applications running in Kubernetes with Permissions and some default settings, a ServiceAccount is used. In every Namespace, a ServiceAccount with the name default exists, and this ServiceAccount is used by every application that is created in that NameSpace. You'll find the current ServiceAccount configuration in the Pod spec.serviceAccount or Deployment spec.template.spec.serviceAccount field.

Without additional configuration, the default ServiceAccount provides minimal access required by applications. The ServiceAccount imagePullSecrets field can be added to configure the ServiceAccount with one or more imagePullSecrets. (Notice that imagePullSecrets is not defined in the spec section of the ServiceAccount, it's a top level configuration). After creating an imagePullSecret with the name dockercreds, you can use the **kubectl patch** command to update the ServiceAccount:

kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "dockercreds"}]}'

After updating the ServiceAccount, you can verify it:

```
student@student-machine:~/ckad$ kubectl get sa default -o yaml
apiVersion: v1
imagePullSecrets:
- name: dockercreds
kind: ServiceAccount
…
```

As the above code listing shows, the default ServiceAccount is now configured with the imagePullSecret, which will allow any currently running or new application to use the imagePullSecret it specifies.

Configuring a ServiceAccount with the imagePullSecret is the most efficient way to configure applications. You will however have to realize that this needs to be done for all Namespaces in your cluster, as ServiceAccounts as well as Secrets are Namespaced resources.

Using Image from Other Registries

Docker Hub is the default resource for getting container images. Instead of configuring authentication to access images from Docker Hub, you can also use images that are provided on registries that don't require authentication. As Docker Hub is the hardcoded image registry in Kubernetes, you will need to use a fully qualified container image name. I like using the following images, which are provided by my friend Pascal van Dam on quay.io

- Nginx: quay.io/pamvdam/nginx:pvd
- Busybox with curl: quay.io/pamvdam/busybox-curl

Obviously, you can use any other image that is provided on any other image registry.

In this article you've read how to configure Kubernetes for authenticated container registry access. This helps as a solution for the Docker Pull Rate limitation, but in general is useful as an increasing amount of container registries requires authentication.