ToiletPaper #85

Integrating cdk8s with Argo CD

Author: Maximilian Brenner / DevOps Engineer / Office Leipzig

After having a look at how to integrate *cdk8s* with *Flux* in my ToiletPaper #88, I was asked whether I could do the same with Argo CD. Let's go!

✗ Argo CD

But at first, what is Argo CD? Similar to Flux, it realizes Kubernetes Manifest deployments by following the GitOps paradigm. It states that the desired condition of your system should be kept in one or more Git repositories. Tools like Argo CD take care of keeping your system (Kubernetes in this case) and its state in sync. In its simplest form, think of a repository containing a single Kubernetes Manifest that defines a *Deployment* and a *Service*. Argo CD will make sure to deploy them to your cluster and continue to do so for any new change.
To control Argo CD, it ships with a CLI tool and a really nice web UI that allows you to check for the status of your deployments in real time. Both allow you to create new applications or setup new cluster connections. The latter allow you to use the multi-cluster functionality to orchestrate deployments over several clusters with a single instance of Argo CD. Setting one up, would be the next step.

✓ Preparation

If you want to follow along, you gonna need a running Argo CD instance. Feel free to use my snippet to set one up in 5 minutes. Otherwise you can go through the official Getting Started guide.
In case you are not familiar with *cdk8s*, make sure to check out this little rundown or my dedicated blog post that goes into more detail. Otherwise let's dive right into action.
At first, we need some *cdk8s* code that we can deploy later on. We are going to use the argocd-example-apps repository as a starting point. It contains the same configuration in different formats – like plain *K8s* manifests, a *Helm* Chart or *kustomize*-files – all deploying a simple guestbook application consisting of one *Deployment* and one *Service*. We are going to put our code into a new folder called `cdk8s-guestbook`.

As with every *cdk8s* application, we start by executing `cdk8s init python-app`. I'm going to use Python but feel free to go with *TypeScript* alternatively. Afterwards, we define the *Deployment* and *Service* objects in the *main.py*.

```
...
  label = {"app": "guestbook-ui"}

  k8s.Service(self, 'service',
              spec=k8s.ServiceSpec(
                type='LoadBalancer',
                ports=[k8s.ServicePort(port=80,
target_port=k8s.IntOrString.from_number(80))],
                selector=label))

  k8s.Deployment(self, 'deployment',
                 spec=k8s.DeploymentSpec(
                   replicas=1,
                   selector=k8s.LabelSelector(match_labels=label),
                   template=k8s.PodTemplateSpec(
                     metadata=k8s.ObjectMeta(labels=label),
                     spec=k8s.PodSpec(containers=[
                       k8s.Container(
                         name='guestbook-ui',
                         image='gcr.io/heptio-images/ks-guestbook-demo:0.2',
                         ports=[k8s.ContainerPort(container_port=80)])]))))
...
```

After this is done, we push the code into a repo that is preferably publicly accessible. Otherwise we need to pass Argo CD the credentials to the repo later on. Afterwards, we can continue with the integration.

➔ Integration

Currently, *cdk8s* is not supported by Argo CD out-of-the-box. To be able to use it, we need to register *cdk8s* as a custom config management plugin. This works by simply creating/updating the *argocd-cm* Kubernetes ConfigMap with something like the following content:

```
# config.yml
data:
  configManagementPlugins: |
    - name: cdk8s # the name of the plugin that we'll later use to reference it
      init: # some optional preprocessing commands
        command: ["bash"]
        args: ["-c", "pipenv install && cdk8s import -l python && cdk8s synth"] # making sure
everything is installed and generating the K8s manifest(s)
      generate: # the output of this command will be deployed onto the target cluster
        command: ["bash"]
        args: ["-c", "cat dist/*"] # printing the generated Kubernetes manifests
kind: ConfigMap
apiVersion: v1
metadata:
  annotations:
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
  name: argocd-cm
  namespace: argocd
  selfLink: /api/v1/namespaces/argocd/configmaps/argocd-cm
```

Apply this with ~~`kubectl apply -f config.yml`~~ and our plugin is ready to use.

So let's try it. I'm going to use the Argo CD CLI tool but creating the application using the web UI will work as well. After issuing this command …

```
$ argocd app create guestbook \ # creating an application called guestbook
      --repo https://github.com/brennerm/argocd-example-apps.git \ # the URL of our repo
      --path cdk8s-guestbook \ # the path to the folder containing our config
      --dest-server https://kubernetes.default.svc \ # the cluster we want to deploy to
      --dest-namespace default \ # the namespace we want to deploy to
      --sync-policy automated \ # enabling automatic sync of changes in the repo
      --config-management-plugin cdk8s # make sure to use our cdk8s plugin
```

… we will end up with the following error:

> *FATA[0006] rpc error: code = InvalidArgument desc = application spec is invalid: InvalidSpecError: Unable to generate manifests in cdk8s-guestbook: rpc error: code = Unknown desc = 'bash -c pipenv install && cdk8s import -l python && cdk8s synth' failed exit status 127: bash: pipenv: command not found*

Having some background knowledge of Argo CD's internals, make this issue somewhat predictable. Each config management plugin is being executed in a component called the *argocd-repo-server*. To make our custom plugin work, we also need to make sure that the tools we use are available in this environment. In our case, these are *pipenv* and *cdk8s*. The proposed solutions are the following:

- using volume mounts containing the necessary binaries
- providing a custom image for the argocd-repo-server

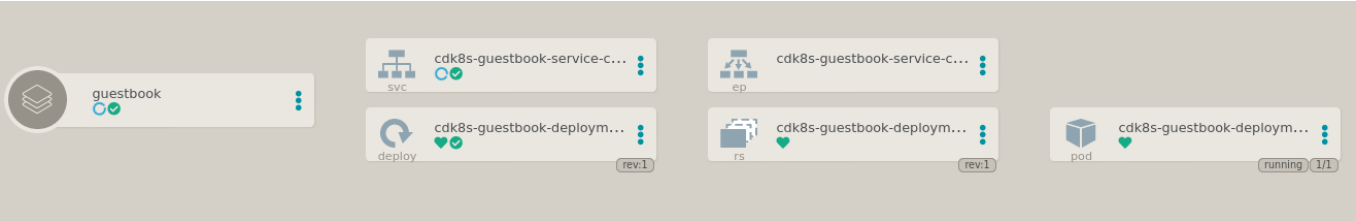
If you've read my last post, you know that Flux has the exact same problem and that I'm pretty disappointed by these options. But that's what we have to work with. I decided to go with the custom image as it appears easier to me. Below you can find my Dockerfile adding the missing binaries:

```
FROM argoproj/argocd:latest

USER root

RUN apt-get update && \
    apt-get install -y \
        curl \
        python3-pip && \
    apt-get clean && \
    pip3 install pipenv

RUN curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | apt-key add -
RUN echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee
/etc/apt/sources.list.d/yarn.list
RUN apt-get update && apt-get install -y yarn
RUN yarn global add npm cdk8s-cli

USER argocd
```

After building this Dockerfile and pushing the resulting image to a Docker registry of your choice (pro tip: if you are working with kind use the really nice load feature) we need to update the *argocd-repo-server* Kubernetes Deployment to use the new image, e.g. with `kubectl edit -n argocd deployments.apps argocd-repo-server`.
After ensuring that the *argocd-repo-server* pod has been recreated with the new image, we can create our application again. This time, everything should work and we'll end up with the *guestbook* pod being started.

```
$ kubectl get pods
NAME                                                READY   STATUS              RESTARTS   AGE
cdk8s-guestbook-deployment-967cec91-65b878495d-jcczj   0/1     ContainerCreating   0          16s
```

To make sure Argo CD is properly syncing changes, let's set the *replicaCount* in the *main.py* to 2, push the change et voila:



live update of the replica count change in Argo CD's Web UI

Conclusion

There you go – a fully functional integration of *cdk8s* with Argo CD.
The main steps in a nutshell:

1. registering the *cdk8s* configuration management plugin
2. making the necessary tools available in the *argocd-repo-server*
3. using the *cdk8s* plugin when creating the Argo CD application

I'm still not satisfied with having to customize an internal service to make everything work, but AFAIK, there's currently no way around it. If you know of a better way or see room for improvement, please let me know.