Exploring your Proof of Concept environment for CaaS Platform – Part 3

By Andrew Gracey, SUSE
Andrew.gracey@suse.com

Table of Contents

This is part three in a three-part guide focused on setting up and exploring what is provided with and how to use SUSE CaaS Platform.

Part 1 covers the setup and installation of the platform:

1. Environment Prep
2. Install Base System(s)
3. Provision Cluster

Part 2 covers some Kubernetes basics and looking at what you just installed:

4. A Quick Look Under the Hood
5. Using Helm
6. Building & Deploying your First Container

Part 3 covers a few other topics to be aware of:

7. Storage
8. Networking
9. Security
10. Suggested Additional Reading

NOTE: In a few instances, the command specifies a backtick or minus. If you are copying and pasting, make sure that these don't get converted into other characters (otherwise, it could cause odd behavior).

In Part 1, we built a SUSE CaaS Platform Cluster and in Part 2, we installed some applications into it. In this final part, we can start looking at how to pull everything together and explore how Kubernetes can fit into the rest of your infrastructure.

7. Storage

One of the mistaken architectural tenets of a Cloud Native application is to have components be stateless. This is great, but the reality is that most applications have some state, so at least one of your components will need to hold that. The component that is holding your state will need to have storage attached, in order to store the data in a way that will survive pod restarts.

In Kubernetes, containers can be given persistent storage though the Container Storage Interface (CSI). This pluggable interface enables storage vendors to easily expose chunks of space as volumes.

There are many storage vendors out there. If you have an existing storage solution, I suggest finding out if they have a storage plug-in. SUSE Enterprise Storage is (naturally) a supported storage provider.

Components

Storage in Kubernetes consists of these components:

- The StorageClass object will be defined by the plugins that are installed. This allows for different parts of an application to have different storage needs (think read-only vs. read-write or high-speed cache vs. cheap, long-term cache).
- A PersistentVolume (PV) object will be defined as a chunk of disk space that can be passed around as needed.

- A PersistentVolumeClaim (PVC) object is created for a pod, to ask for what type of storage it needs.
- The Pod Specification itself will specify which PVCs get mounted and where they are located in the filesystem that is exposed to the containerized application.
- A provisioner is a component that can automate the creation of PVs for a specific storage class when a PVC is created. This means you can reduce a lot of the maintenance headache when dealing with storage.

Simple NFS Setup

For the purposes of this guide and most of my sandbox systems, I tend to use a simple NFS server.

Setting up an NFS share is outside the scope of this guide, but here are the nfs share settings for the wholly unsecure and not production ready server (which, in my experience, works with minimal hassle):

```
rw,insecure,sync,no_subtree_check,no_root_squash
```

With the NFS share set up, we can install the nfs-provisioner. This is done with helm (which should be set up via Part 2 of this guide):

```
$ helm install stable/nfs-provisioner --name nfs-provisioner --namespace nfs-provisioner –set
storageClass.defaultClass=true --set nfs.server=x.x.x.x --set nfs.path=/exported/path
```

Once the pods in the nfs-provisioner namespace are started, you should be able to create a PVC and have the PV provisioned and bound to it automatically.

Example

At this point, we can now mount storage to pods. To show this, let’s run a pod, mount a volume to it, get shell access, change a file, delete the pod (but not the Claim), and then spin up a new pod with the same volume attached to it to see that the same data persists.

NOTE: This flow is not a common workflow. It’s for demonstration purposes only.

The first thing we need is a Persistent Volume Claim. To make one, create a file containing:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-claim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Then add it to k8s, using:

```
$ kubectl apply -f /path/to/pvc.yaml
```

You should be able to see the Volume get created and bound using:

```
$ kubectl get pv,pvc
```

You should also see a folder get created on the nfs server to host the new volume.

To attach this to a Pod, we can create a new deployment to use it:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deploy
spec:
  selector:
    matchLabels:
      app: sample
  replicas: 1
  template:
    metadata:
      labels:
        app: sample
    spec:
      volumes:
        - name: persisted-storage
          persistentVolumeClaim:
            claimName: test-claim
      containers:
        - name: something-long-living
          image: tomcat
          volumeMounts:
            - mountPath: "/persist"
              name: persisted-storage
```

Start this with:

```
$ kubectl apply -f /path/to/deploy.yaml
```

You can wait for the new pod to get created with:

```
$ kubectl get pods -w
```

Once you see that it’s ready, copy the pod name and get shell access using:

```
$ kubectl exec -it sample-deploy-<random id> -- /bin/bash
```

This will give you a bash shell running in the container. To see the folder that was mounted, you can use:

```
$ ls /persist/
```

Create a file inside that folder using:

```
$ echo “Hello from the container” > /persist/sample
```

Then end your shell with:

```
$ exit
```

Now that we have data inside the container, let’s see it persist. Kill the pod with:

```
$ kubectl delete pod sample-deploy-<random id>
```

The deployment will create a new pod with a new random id.

If we repeat the process above to get a bash shell into it, we can see that the file still exists:

```
$ kubectl exec -it sample-deploy-<new random id> -- /bin/bash

$ ls /persist/

$ cat /persist/sampleHeading
```

StatefulSets

One of the problems with this approach is that a database that has been scaled to multiple instances will expect that the mapping of the host name to the data never changes. With our example above, you can see that when we deleted the pod, the name changed. This problem is solved by using a StatefulSet deployment instead. This will ensure that there aren't any unexpected changes between pod restarts.

It’s outside the scope of this guide to show how to use them, but with what you know now, it shouldn’t be hard to pick up. You can find the Kubernetes Tutorial here, https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/.

8. Networking

With all of these moving parts, the networking in a Kubernetes cluster can get a bit complicated. However, as with everything, tools have been created to simplify the management.

Kubernetes exposes networking between pods, services and the outside world though a pluggable interface known as the Container Networking Interface (CNI). Similar to the CSI, there are several vendors that offer solutions.

NOTE: Services act as internal load balancers for groups of pods.

In SUSE CaaS Platform, we include Cilium as the built-in CNI plugin. We did this for a variety of reasons, but the two most obvious are improved performance and security.

Networks

Three networks exist in a cluster: the host network, the pod network and the services network. Each of these gives access to different parts of the system.

Due to the need for sandboxing, pods are given their own Virtual Network Interface Connection (veth), which are the bridge with the outside world.

You can see what this looks like by SSH'ing into a worker node and running:

$ ip link list

The addresses exposed inside the pod are controlled by the CNI and are dependent on how the CNI developer decides to expose pools of addresses.

NOTE: Dial stack support for IPv6 is in an alpha state in Kubernetes and will hopefully be ready soon.

Network Control

By default, networking between pods is wide open and any pod can talk to any other pod or service. It can be locked down using Network Policies, which enable an operator to use pod labels, traffic types and requested ports to limit both ingress and egress.

With many CNI providers, additional options are exposed through their own custom configuration objects. Cilium (the built-in CNI provider in SUSE CaaS Platform) offers filtering based on the OSI Layer 7 (application) traffic type as well.

These controls enable you to create zone-based security definitions. They also enable you to gain more fine-grained control, which previously required an extremely powerful firewall to do hair-pinning with all of your traffic.

External Load Balancing

There are three ways to expose traffic to consumers outside of the cluster:

- Use the **kubectl command** to set up a proxy directly to the pod. This is only used for debugging or accessing internal UIs.
- Use the **NodePort service type**, which will expose the served port on the host interface directly. A drawback of this is that if the pod gets restarted onto a new host, its IP address will change.
- Use the **Load Balancer service type**. This will use the configured load balancer to assign an IP from a pool of external addresses. In Part 2 of this guide, you will have installed MetalLB to do this. There are several vendors of hardware and software load balancers that Kubernetes can use. Those will all likely be better than MetalLB in a production environment.

Network Policy Example

Next, I'll reuse the same deployment from the storage example and show how to set up a network policy that allows only a single type of pod to connect to it.

Before we start, we need to create a service to allow name resolution to make this easier.

```
$ kubectl expose deployment sample-deploy --port 8080
```

First, we can attempt to get access to the tomcat pod from another pod with:

```
$ kubectl run --generator=run-pod/v1 access -it --image busybox --rm -- /bin/bash
```

This will run a pod with shell access and delete it when you exit the shell.

From this pod's shell, we can use `wget` to test access to the deployed pods:

```
$ wget sample-deploy:8080 -O - -q -T 5
```

You should see the default tomcat html returned to you. (This is likely not what you want your security people want to see.)

The first step is to add a deny all policy. Create a file with the following yaml and apply it with `kubectl apply -f <filename>`

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

In network policies, the `{}` will act as select all. This means that the above policy doesn't allow Ingress or Egress to/from any pod.

If you run the same test from above again, you should get a timeout after five seconds:

```
$ kubectl run --generator=run-pod/v1 access -it --image busybox --rm -- /bin/bash
$ wget sample-deploy:8080 -O - -q -T 5
```

Now we need to open up the connection again, but specifically for this access node. Again, create a file with the following: *yaml and apply it with `kubectl apply -f <filename>`*

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-pod
```

```
spec:
  podSelector: {}
policyTypes:
  - Ingress
    - from:
       - podSelector:
           matchLabels:
             run: access
```

This policy allows ingress to the pods with the label of `app: sample` from all pods with the selector of `run: access.` If you did a `kubectl describe pod access` while the pod was still running, you would see that the `kubectl run …` command automatically added this label to the container based on its name. This let's our example shell pod access any pods in this namespace.

Again, test it out by running the same set of commands:

```
$ kubectl run --generator=run-pod/v1 access -it --image busybox --rm -- /bin/bash
$ wget sample-deploy:8080 -O - -q -T 5
```

To start using these rules, you need to switch the example policies above to be of type CiliumNetworkPolicy. The documentation for this can be found, at https://docs.cilium.io/en/v1.6/kubernetes/policy/.

9. Security

Kubernetes is not a silver bullet for creating secure solutions. It is possible to create a very secure system, but as we saw above with the networking, the defaults are not typically secure.

There are several places where we need to pay attention to security, but for the purposes of this guide, we will limit it to host security, user management, the container lifecycle and networking.

Host Security

Because all container processes are sharing the same kernel, we need to make sure that no one can improperly access the host. With this in mind, your host should be secured just like any other host. For example, you need to stay up to date on patches and CPU mitigations and limit access (both physically and remotely).

Also, in Part 1 of this guide, we turned off the firewall. This is obviously not secure and should not be done in a production environment. At a minimum, you need to allow port 6443 for the kubelet, 31000 and 31001 for Dex/Gangway, and 80 and 443 for the ingress controller.

User Management

Kubernetes uses a system of Users, Roles and Role Bindings to enable a least-privileged style of user management. It also has the ability to use Service Accounts to give access to external tools and internal pods.

There are two parts to think about when it comes to any user management: Authentication (being sure who a user is) and Authorization (giving access only to the users that need it). These should be kept as separate concerns, since not all users can be trusted equally.

Authentication is done through tokens (or certs) and is typically handled through a kubeconfig file. As we saw in Part 1 of this guide, the admin version of this file is generated on bootstrap of the master node. If we want other users to be able to log in, we can either manually create users and RSA certs or we can let the users use a self-service application.

There are two components that work together to provide this self-service: Dex and Gangway. Gangway enables Kubernetes to use the OpenID Connect (OIDC) authentication scheme, while Dex enables you to hook into a huge variety of SSO providers.

Due to the number of options in potential integrations, instructions for setting this up is outside the scope of this guide.

Container Lifecycle

It's important to make sure that the container you are running is the one that you are expecting to be running, that it only contains the application you expect, and that it hasn't been modified since being created. Here are a few ways to maintain control of your system. Many of these are overkill for a random test system, so be careful.

- Build your own base containers or use only ones from trusted vendors. While that one random container on DockerHub might have everything you want, it might also carry some malware.
- Sign your containers and verify at runtime. There are tools that hook into the container runtime to ensure that the signatures match and nothing has been changed between build time and runtime.
- Run your own internal container registry. This will allow for greater control and minimize external network traffic. Don't, however, run your registry inside the cluster, because it creates a "chicken and egg" problem if a host dies and loses the container to run the registry.
- Don't run as root and use privileged mode sparingly. While the container root uid is no longer the same as that of the host, it is still a bad idea. (To ease management and integration headaches, choose port 8080 for all of your containers. It will be remapped outside the container anyway.)

Network Security

The goal here is to limit any footholds that an attacker manages to get. You can do this by making sure that any pod can only talk to pods that it depends on and only in the right protocols. For example, no one should be able to SSH into the database pod and the only access should be from the API pods via the correct port.

NOTE: This takes actually designing your architecture and knowing where data should flow before just building!

If you are using namespaces to segregate your pods (and you should), limit communication between them as much as possible.

10. Suggested additional resources

The Kubernetes world is huge and growing fast. Don't worry about not knowing it all—no one does. That said, here are some good places to learn more:

- https://github.com/kelseyhightower/kubernetes-the-hard-way

  This is a very in-depth guide about how to build a cluster from scratch. I recommend browsing through it, to get to know all the pieces and how they relate to each other. Obviously, a distro (such as SUSE CaaS Platform) will have all of this done for you in a way that doesn't require you to maintain it.

- https://github.com/walidshaari/Kubernetes-Certified-Administrator

  This is a set of resources to prepare to take the Kubernetes Certified Administrator Exam. Even if you aren't planning to take the test, there's a wealth of information here.
- https://kubernetes.io/docs/concepts/architecture/

  This is a discussion on the high-level architecture of Kubernetes from the team itself.
- https://blog.risingstack.com/the-history-of-kubernetes/

  This is a good timeline of all the events that led to where we are now. I personally find it useful to know where we came from and why things were created in certain ways.
- https://microservices.io/patterns/microservices.html

  This is a collection of common microservice patterns for designing systems that work well in Kubernetes.