IP: 10.7.195.150

NAME in artifactory to get chart: fiscd-core-tc-aks-runtime

wget --user e5694382 --ask-password <https://artifactory.fis.dev/artifactory/fiscdcore-helm-snapshot-local/fiscd-core-tc-aks-runtime/fiscd-core-tc-aks-runtime-202403.0.0-devfeature-feature-FISCD-149007-tc.1.tgz>

**K8S** originated from Google and it was founded and remained an open source project. Based on its origin, it shares lots of DNA with other Google products, such as Borg and Omega. Later, it was given to Cloud Native Computing Foundation. Its code is written in Go/Golang language.

One of K8s definitions is that K8S is orchestrator of micro-services apps. From a high perspective, there are two main components that make K8S cluster: **Master** (Control Plane) and **Worker** nodes. Interesting fact about Master nodes is that there is always one Leader node, while remaining ones are followers that act as a proxy. In case currently selected leader goes down, a new one is elected. Each Master node contains further components:

1. **API Server**
2. **Cluster Store** (only persistent component where it persists cluster state and config – based on etcd distributed NoSQL DB)
3. **Controller Manager** (controls all other controllers and observes the state of a cluster with a desired state)
4. **Scheduler** (assigns work to worker nodes)

On the other hand, Worker nodes possess next components:

1. **Kubelet** (main K8S agent, registers node with cluster, watches API server for work tasks, executes Pods and reports back to Masters)
2. **Container Runtime** (contains instructions how to run containers, it is pluggable (CRI) and it usually runs Docker, ContainerD, CRI-O)
3. **Kube Proxy** (takes care of networking like make sure every Pod gets IP, load balancing)

**Pod** is shared execution environment and containers that run in it share that environment. Usually, each container is packed inside of a single Pod. Only in special uses cases multiple Containers run inside the same Pod (this is the case when you need to compliment your Services, like exposing logs for telemetry, encrypt the traffic, etc.). Once the Pod is created, it gets registered within K8s cluster DNS. Pod DNS is visible in: *etc/resolv/*conf file.

GENERAL COMMANDS

Display all available k8s clusters: secret

**kubectl config get-contexts**

Select cluster:

**kubectl config use-context [CLUSTER\_NAME]**

Set default namespace based on the cluster context you are in:

**kubectl config set-context –current –namespace=[NAMESPACE\_NAME]**

Grep the content of file and display the specific number of lines after the key word:

cat output.yaml | grep "imagePull" -A 2

PODS COMMANDS

Commands to watch Pods:

***kubectl get pods***

***kubectl get pod –watch***

***kubectl get pods -o wide***

***kubectl describe pods [POD\_NAME]***

Commands to trace log of Pods:

*kubectl logs [POD\_NAME]*

*kubectl logs [POD\_NAME] -c [CONTAINER\_NAME]*

Enter the container shell upon the pod creation:

*kubectl run -it [POD\_NAME]*

Get a pod specification and divert the spec into output file:

*kubectl get pod [POD\_NAME] -o yaml > [OUTPUT\_FILE\_NAME]*

*kubectl get pod schema-evolution-archive-5fbc594fc-plclj -o yaml > schema.yaml*

**Deployments** have two types: Deployment for **Stateless** and **Stateful** apps. They enable to have self-healing, rolling updates, scailing, versioning, rollbacks, etc.

K8s supports several types of strategies for Deployments: *Blue/Green*, *Canary, Rolling* and *Rollback* Deployments.

**Blue/Green Deployments** is strategy for checking the viability of a Deployment before it’s publicly available. Basically, there are two production environments, where each environment has its Pods, Services, etc. New application (green) is deployed alongside the old app (blue) and it is being tested, where clients don’t have an access to (only development). Once the green proves to be working as expected, traffic gets routed from blue to green (swap of services, pods and everything else relevant).

***Canary*****Deployments** is also a strategy for checking the viability of Deployment, before it goes to production. It relies on public Service that clients are using, but that Service is able to route the traffic between stable (production) and Canary deployments. However, only small portion of public traffic gets routed to Canary deployment (it differentiates from production deploy, by having smaller value for replica set). In that way the traffic gets routed less to Canary deployment. Once everything proves to be working, corresponding Service starts to send traffic only to Canary, which becomes a production.

**Rolling Updates Deployments** (default strategy) on a higher-level function as in the following. Let’s say we start off with 3 Pods that have V1 app running. When rolling out V2 app, new Pod with V2 app gets deployed alongside those 3 V1 Pods. Then, one of V1 Pods gets removed. The procedure continues until we switch all V1 Pods with V2.

**Recreate Deployment** is exactly what the naming suggests, where Pods that run old version of app, get removed by new Pods with newer version of app (some amount of down-time). This strategy is practiced when there are significant changes in app (let’s say API). So, if you would simply do Rolling Deployment strategy here, end-user could suffer from some functionality breaking down.

**Replica Set** in k8s is a controller whose main job is to ensure that a specified number of identical Pods are running at any given time. If a Pod crashes, is deleted, or becomes unresponsive, the ReplicaSet automatically creates a new one to maintain the desired state. This provides high availability, fault tolerance, and scalability for your applications.

App deployment in K8S looks like this: start off with app code, out of which you build a container image, store the image in a repo, defined it in a K8S manifest and post that to API Server. In some more details:

1. Start off with app code (make sure Dockerfile exists and that ports in app and Dockerfile match)
2. Build a container image (docker image build -t)
3. Push the crated image to repo (docker image push)
4. Define K8S manifest file (check that exposed pod port matches with app port)
5. Post the manifest to API Server (kubectl apply -f)

DEPLOYMENT COMMANDS

Show labels for deployments:

kubectl get deploy –show-labels

kubectl get deploy –show-labels -l [LABEL\_NAME]

Create custom Deployment based on the predefined template:

*kubectl create deploy [DEPLOY\_NAME] –image=[IMAGE\_NAME] –dry-run=client -o yaml > deploy.yml*

Create Deployment with converted annotation from yaml to JSON:

*kubectl create -f [DEPLOYMENT\_FILE] –save-config*

Record the command in Deployment revision history:

*kubectl apply -f [DEPLOYMENT\_FILE] –record=true* OR *kubectl annotate deployment [DEPLOYMENT\_NAME] Kubernetes.io/change-cause=”Change details” –overwrite=true*

Check Deployment configuration in form of yaml file:

kubectl get deploy [DEPLOYMENT\_NAME] -o yaml

Scale Deployment:

*kubectl scale deploy [DEPLOYMENT\_NAME] –replicas=1 -l* fis.cd.tc.online=true

*kubectl scale deploy ref-loader --replicas=1*

Load referentials:

k exec deploy/ref-loader -- groovysh -e 'LoaderShell.uploadTradeClearingAndBackOfficeFISProvidedData()'

Edit Deployment configuration live, when there is no Deploy.yaml:

*KUBE\_EDITOR=”nano” kubectl edit deploy/[DEPLOYMENT\_NAME]*

Change Deployment image:

*kubectl set image deployment/nginx nginx*

Rollback commands:

Check Deployment history:

*kubectl rollout history deploy [DEPLOY\_NAME]*

Undo Deployment to a previous or selected revision:

*kubectl rollout undo -f [DEPLOYMENT\_FILE]*

*kubectl rollout undo -f [DEPLOYMENT\_FILE] –to-revision=2*

Display only an annotation of Deployment:

*kubectl get deploy [DEPLOY\_NAME] -o json |jq ‘.metadata.annotations.”Kubernetes.io/change-cause”’*

Get info about specific Deployment:

kubectl rollout status deployment [DEPLOYMENT\_NAME]

kubectl scale deploy --replicas=1 -l fis.cd.tc.prereq=true

kubectl scale deploy --replicas=1 -l fis.cd.tc.online=true

**HELM**

Helm is a package manager, which is the best way to find, share and use software built for K8s. These K8s packages are called **Charts** and they are bundle of information used to create an instance of K8s app. Alongside with Charts, there are also **Config** and **Release** objects. Config contains config info that is merged with packaged Chart, to create releasable object. And Release is running instance of Chart.

HELM COMMANDS

List all helm templates:

*helm list*

List added repos:

*helm repo list*

Search for Helm package:

*helm search hub [PACKAGE\_NAME]*

Add a repo:

*helm repo add [CUSTOM\_REPO\_NAME] [REPO\_URL]*

*helm repo add marko* [*https://artifactory/fis.dev/artifactory/fiscdcore-helm-snapshot-local*](https://artifactory/fis.dev/artifactory/fiscdcore-helm-snapshot-local)

helm repo add markorepo [https://artifactory.fis.dev/artifactory/fiscdcore-helm-snapshot-local --username e5694382](https://artifactory.fis.dev/artifactory/fiscdcore-helm-snapshot-local%20--username%20e5694382)

Delete repo:

Helm repo remove [REPO\_NAME]

Refresh the helm list of added repos:

*helm repo update*

*helm dep up .*

Show Chart values:

*helm show values [CHART\_NAME]*

List all Helm installations:

helm list

Search for Charts:

*helm search repo [REPO\_NAME]*

helm search repo fiscd-core-tc-aks-runtime -l --devel | grep FISCD-149013

Install Helm Release:

*helm install [CUSTOM\_RELEASE\_NAME] [CHART\_NAME]*

*helm install [CUSTOM\_RELEASE\_NAME] [CHART\_NAME] –values=[FILE\_WITH\_OVERRIDES] -n [NAMESPACE] –version=15.0.9*

*helm install marko-feature fiscd-core-tc-aks-runtime-202403.0.0-devfeature-feature-FISCD-149007-tc.2.tgz -f values.yaml -n markoshelm install [CUSTOM\_RELEASE\_NAME] [CHART\_NAME] –version=13.1.5 -n dev –set replicaCount=5*

*Uninstall Helm Release:*

*helm uninstall [RELEASE\_NAME]*

Pull chart to system as a .tar, but doesn’t install it:

*helm pull [CHART\_NAME] –version=15.0.9 –untar*

*helm pull stable/fiscd-core-brm-aks-runtime –version 202403.0.0-bo-development.365*

helm pull markorepo/fiscd-core-tc-aks-runtime --version 202404.0.0-devfeature-feature-FISCD-149013.13

Upgrade Helm Release:

*helm upgrade [RELEASE\_NAME] [CHART\_NAME] –version=13.1.8*

*helm upgrade install –name [CUSTOM\_RELEASE\_NAME] [CHART\_NAME] -f aks-values.yaml -n [NAMESPACE]*

*helm upgrade marko-feature fiscd-core-tc-aks-runtime-202403.0.0-devfeature-feature-FISCD-149007-tc.2.tgz -f values.yaml*

*helm template [CHART\_NAME] -f aks-values.yaml > mydata.yaml*

*helm template fiscd-core-tc-aks-runtime-202403.0.0-devfeature-feature-FISCD-149007-tc.2.tgz -f values.yaml > output.yaml*

**SERVICES**

**Service** provides a stable DNS name and IP to attached Pods that also acts as a Load Balancer. How to know Pod is attached to what Service? Simply, via **Labels**! 3 types of services:

**ClusterIP** -> Exposes the service **inside the cluster**. Other Pods can access it, but it’s not reachable externally

**NodePort** -> External access via node IP, opens a static port on every node’s IP, simple external access for development or testing

**LoadBalancer** Service is for external access via cloud load-balancer

How it works in practice:

Diagram

Description automatically generated with medium confidence

SERVICE COMMANDS

Test connection of Service to a Pod:

*curl [SERVICE\_NAME]:[LISTENING\_PORT]*

Change Service’s selector label:

kubectl set selector svc [SERVICE\_NAME] ‘role=green’

Expose K8s resource to network:

*kubectl expose [RESOURCE\_TYPE] [DEPLOYMENT\_NAME] –port=9000 –target-port=80 –type=NodePort –name=nginx-svc*

**INGRES**

They let Load Balancers to expose multiple Services on Internet. Only used for HTTP/HTTPS (ports 80 or 443). Two things to mention: **Object Spec** (defines a rules) and **Controller** (implements the rules). K8s doesn’t ship with Controller, so it is required to install one (same as for Network Policies).

How Ingress knows how to redirect the traffic call from outside to inside, to some Cluster IP? It watches over HTTP headers in requests. They are tightly integrated with Load Balancers.

Ingress Class is a way to run more than one Ingress Controller on the same cluster.

Diagram

Description automatically generated

List Ingress Class:

*kubectl get ingressclass*

**VOLUMES**

Implementation of volumes for data persistence in K8s is dead simple. It is just required to create an actual volume (iSCSI LUN, NFS, bucket, etc.) on your onPrem or cloud external storage system and expose it to your K8s cluster as a Persistent Volume (PV) that links back to the original volume. Next, claim the created PV to your Pod via K8s API object called **Persistent Volume Claim** (PVC) and app can start using it. This is the break-down perspective:

Diagram

Description automatically generated

Worth pointing out, what plugin you will be using, depends on what type of external storage is behind and each storage vendor usually offers one (in the example below it is *gcePersistentDisk*). In the old days, the logic and code of these storage plugins or **Container-Storage Interface** (CSI), were baked right inside of K8s code tree. However, this approach demonstrated that CSI was built just to work (ad-hoc), without caring about maintaining it, which was recognized as a huge problem later, due to tight coupling with K8s, so the CSI was moved outside of K8s code and became its own independent layer.

Below are examples of ps-pv.yaml (exposes external storage), ps-pvc.yaml (binds to PV) and ps-pod.yaml (references to the PVC and creates a volume from it) in static provisioning use-case:

Timeline

Description automatically generated with low confidence

Once a Pod makes a claim in PV as a bound, no other Pod can claim it, but every container in a Pod can access it.

K8s Persistent Volume sub-system has 3 major API objects what make external storage available to Pods and containers: **Persistent Volume**, **Persistent Volume Claim** and **Storage Class**.

**Persistent Volume** is one to one mapping to actual storage resources. Meaning if we want to attach 20 different volumes from an external storage, we will need 20 different PVs. Same goes for **PVS** that will match PV specification. However, **Storage Class** object makes this work-load dynamic, meaning it dynamically creates volumes and creates different classes/tiers of those volumes. An example is below:

Graphical user interface, text, application

Description automatically generatedGraphical user interface

Description automatically generated with low confidence

Important to note is that Storage Class is immutable, meaning, once it is created, it can’t be edited. If something needs to change inside of it, the Storage Class must be destroyed and re-deployed.

There are several type of volumes like Raw Block Volumes, Volume Clones, etc.

**MULTI-CONTAINER POD**

Multi-Container patterns:

1. Init pattern
2. Sidecar pattern
3. Adapter pattern
4. Ambassador pattern

**Init pattern** enables preparation of environment, before the Main Container (app) is started. It is run only once, after which the next Init or Main Container launches. If there happens to be multiple Init Containers, they will be run one after the other and, if one of them fails to be executed, the Pod will be restarted.

Text

Description automatically generated

**Sidecar pattern** has Main and Sidecar Container run in parallel. This Sidecar or Helper Container adds additional functionality. For instance, main app Container runs a web service, where sidecar pulls up-to-date content from web server. Additionally, there is something called **Service Mesh**, which at high-level, encrypts the traffic between Pods, expose telemetry and metrics.

Text

Description automatically generated

Diagram

Description automatically generated

**Adapter pattern** is a form of Sidecar pattern, but the difference is it takes non-standardized output from the Main Container and transforms the data into format known by some external system.

Text

Description automatically generated

**Diagram

Description automatically generated**

**Ambassador pattern** also a form of Sidecar pattern, where it proxies the connection to outside world. It sits between the Main app Container and some external system.

A screenshot of a computer

Description automatically generated with medium confidenceText

Description automatically generated

Diagram

Description automatically generated

Some of available public cloud storage services options are: Elastic Block Store (AWS), GCP Persistent Disks and Azure files.

**SERVICE ACCOUNTS**

Two security entities in K8s: **Users** and **Applications**. Users are managed outside of K8s (identity management system of client choice) with **User Account**. Applications are authenticated with **Service Accounts** each Pod is associated with and every Service Accounts has a token (multiple Pods can share Service Account). Service Account may be found under */var/run/secrets/Kubernetes.io/serviceaccount* on Container. Each request, whether it comes from the inside or outside of K8s cluster, must go through API Server. API Server recognizes the given command and it tries to answer who is making the given command (authentication), by inspecting the certificate of Kubectl (signed by the cluster or certificate authority that is trusted). Then, the request gets forwarded to authorization plugin (RBAC), which questions is the entity allowed to do what is requesting to do. If it is authorized, then the command will be processed.

List all secrets:

*kubectl get secret*

Get token/pass in encoded format for specific secret:

kubectl get secret [SECRET\_NAME] -ojsonpath='{.data.\.dockerconfigjson}'

*kubectl get secret registrycred -ojsonpath='{.data.\.dockerconfigjson}'*

Get token/pass in unencoded (decrypted) format for specific secret:

*kubectl get secret [SECRET\_NAME] -ojsonpath='{.data.\.dockerconfigjson}' | base64 -d*

*kubectl get secret registrycred -ojsonpath='{.data.\.dockerconfigjson}' | base64 -d*

kubectl create secret docker-registry registrycred --docker-server="fiscd-docker-dev.docker.fis.dev" --docker-username="svcacct-leobuild" --docker-password="AKCp5fTjaYApjcqRTwuqsuJaEVWsVpBvuK4vHaF3SWBMEnrcDpqWgrhHXwFvaCXmPvw4dzBGK" -n markos

**JOBS AND CHRON JOBS**

Jobs run set number of Pods to their completion. Controller responsible for watching and kicking off chron jobs, re-scans the environment every 10 seconds, which means, if property *startingDeadlineSeconds* inside of Chron Job YAML specification is below 10 seconds, controller will probably miss some jobs to start.

Text

Description automatically generatedText

Description automatically generated

**JOBS AND CHRON JOBS COMMANDS:**

Get the list of chron jobs:

*kubectl get chronjobs*

Get the list of jobs:

*kubectl get jobs*

**NETWORK POLICIES**

Network Policies lock down Pod-to-Pod traffic within K8s cluster, because out-of-box, K8s cluster is wide open. Network Policies consist of **Ingress** and **Egress** rules. Ingress is about traffic coming in to a Pod, while Egress is traffic going out from a Pod.

Setting up a Network Policies is all about allowing something to communicate with something else. Meaning, if there are 5 Pods and Pod 1 is only allowed to talk to Pod 2, that means that every other traffic, except to Pod 2, is forbidden to the Pod 1.

Also, pay attention to Network Policy configuration (watch out for ‘-’ sign in them):

Graphical user interface, application

Description automatically generated

To reference any Network Policy to any Namespace, this label must be used in form: **kubernetes.io/metadata.name**.

**MINIKUBE COMMANDS**

Fetch Minikube IP and service’s NodePort:

*minikube service [SERVICE\_NAME] --url*

Switch to proper Namespace in K8s cluster:

*kubectl config set-context –current –namespace=[NAMESPACE\_NAME]*

Create an alias:

*alias k=”kubectl”*