04. Working With Pods

Theory

Controllers infuse Pods with self-healing, scaling, rollouts and rollbacks. Every controller has a PodTemplate defining the Pods.

Pod Theory

- Atomic unit of shceduling in Kuerbetes: Apps are deploying inside Pods

1. Write the app/code
2. Package it as a container image
3. Wrap the container image in a Pod
4. Run it on Kubernetes

- Pods augment containers
  - Labels
  - Restart policies
  - affinity
  - security policies
  - termination control
  - resource limites

```
# List all possible Pod attributes
kubectl explain pods  --recursive
kubectl explain pod.spec.restartPlicy
```

- **Labels** Let group Pods and associate them with other objects in powerful ways.
- **Annotations** let add experimental features and integrations with 3rd partyh tools
- **Probes** let test the health and status of Pods and apps
- **Affinity** give control over where in a cluster Pods are allowd to run
- **Termination** lets you gracefully terminate Pods and apps
- **Security policies** let enforce security features
- **Resource Requests** let specify a minimum and maximum values for CPU, mem, disk IO, etc. usain bolt

EVERY CONTAINER IN A POD IS GUARANTEED TO BE SCHEDULED TO THE SAME WORKER NODE

PODS PROVIDED A SHARED EXECUTION ENVIRONMENT FOR ONE OR MORE CONTAINERS:

- Shared filesystem, network stack, memory, volumes.

Deploy Pods

1. Pod manifest: Static Pods
2. Controller

static Pods have no super-powers as self-healing, scaling or rolling updates. They are monitored and managed by the worker node kubelet process. There's no control-plane process watching and capable of starting a new one on a different node.

Pods deployed via controllers have benefits of being monitored and managed by hihgly available controlle rrunning. If kubelet restart fails, the controller can start a replacement pod.

- Pods are mortal: There's no fixing them and bringing back from the dead. Pods are cattle (pets vs cattle paradigm). When die they get replaced by another. Same config, different IP address and UID. This is why apps store **state** and **data** outside the Pod.
- You almost always deploy and manage Pods via controllers
- The single-container model is the simplest, but multi-container pods are importal in production environments and vital for service meshes.

Deploying Pods

1. define it in YAML *manifest file*
2. Post the YAML to the API Server
3. The API server authenticates and authorizes
4. The config is validated
5. The scheduler deploys the Pod to a healthy worker node with available resources
6. The local kubelet monitors it

If the Pod is deployed via a controller, the config will be added to the cluster store as desired state and controller will monitor it.

anatomy of a Pod

- Pod: **execution environment shared by one or more containers**. It can be useful to think Pods as shared environments and containers as application processes.
- In Docker or containerd a Pod is a special container with containers running inside.
- The collection of resources that containers it runs inherit and share are:
  - net namespace:
    - IP Address
    - Port Range
    - Routing Table
    - etc
  - pid namespace
  - mnt namespace: filesystems and volumes
  - UTS namespace: Hostname
  - IPC namespace> unix domain sockets and shared memory

Shared Networking

Each Pod creates its own network namespace. Own P, single range of TCP and UDP ports, single rounting table. If it's a single container Pod the container has full access to the network namespace. If it's a multi container Pod, all containers share the IP, port range and rounting table.

Pod Network

The unique IP address that Pods gets are fully routable on an interneal network called pod network.

- flat: every pod can talk directly to every other Pod without ocmplex routing
- Flat security perspective: Kubernetes Network Policies to lock down access

Pods Features

- Pod deployment is an atomic operation
- Defined in a declarative YAML object yo post to the API server
- Once all containers on a Pod are pulled and running, the Pods enters the running phase
- If it's a short-lived Pod, when all containers erminate succesfully the Pod itself terminates and enters the succeeded state. -> Never or OnFailure restart policy
- If it's a long-lived Pod, the local kubelet may attempt to restart them. -> Always restart policy
- Immutability: uou cant modify them after they are deployed

Immutability Behaviors

- When updates are needed -> replace al old Pods with new ones
- When failures occur, replace failed Pods with new ones

Scaling

Multi containers pod are only for coscheduling and colocating containers, you never scale an app adding more of the same app containers to a Pod.

Multi-Container Pods

At very high-level, every container should have a single clearly defined responsibility. This is called separation of concerns. Colocating multiple containers in the same Pod allows containers to be designed with a single respondiblity but cooperate closely with others. For example, a container that puts conttent updates in a volume shared with web container. The Multiple Container Pod patterns are:

- Sidercar pattern
- Adapter pattern
- Abassador pattern
- Init pattern

Sidecar

- Most popular
- Main app container
- Sidecar container: augment or perform a secondary task for the main app container.

Adapter

Adapter pattern is a specific variation of the sidecar pattern where the helper container takes non-standarized output from the main container and rejis it into a format required by an external system. A example is NGINX logs being sent to Prometheus, who does not understand NGINX logs. A common a pproach is put an adapter container into NGINX Pod that converts NGINX logs into a format accepted by prometheus.

Ambassadorn

Variation of the sidecar. Helper container brokers connectivity to an external system. Example the main app container can dump its output to a port the ambassador container is listening. Looks like a interface.

Init

Init is NOT a form of sidecar. Runs a init container that's guaranteed to start and complete before the main app container. It's guaranteed to only run once. For example, the main app need permissions settings, an external API to be up and accepting connections.

Hands On with Pods

```
kubectl get pods
```

Pod manifest file

```
kind: Pod
apiVersion: v1
metadata:
  name: hello-pod
  labels:
    zone: prod
    version: v1
spec:
  containers:
  - name: hello-ctr
    image: nigelpoulton/k8sbook:1.0
    ports:
    - containerPort: 8080
```

- kind: type of object
- apiVersion: schema version
  - `<api-group>/<version>`
  - core group: omits the api-group part
  - Pods in the core API omits the group name so just v1 its ok
- metadata:
  - names: identify the object in the cluster
  - labels: loose couplings with other objects
  - annotations: 3rd party tools and services
  - Namespace: no specify Namespace -> default Namespace. Not good practiceuse default in RW apps
- spec: define the containers the Pod will run

Manifest files

Forces to describe applications in Kubernetes. Can be used by operations team to understand how the application works and what it requires from the environment.

Deploying Pods from a manifest file

```
kubectl apply -f pod.yml
pod/hello-pod created
kbectl get pods
```

kubectl get

- `-o wide` single line
- `-o yaml` full copy
  - desired state .spec
  - observed state .status
  - the extra information come from default values

kubectl describe

nicely formatted multi-line overview of an object

```
kubectl describe pods hello-pod
```

kubectl logs

```
kubectl logs

multipod container:
kubectl logs multipod --container syncer
```

kubectl exec: running commands in Pods

```
kubectl exec hello-pod -- ps aux
```

get shell access in a running Pod

```
kubectl exec -it hello-pod -- sh
curl localhost:8080
```

hostnames

Every container inherits its hostname from the name of the Pod

kubectl edit

```
kubectl edit pod hello-pod
```

multi container pod

```
apiVersion: v1
kind: Pod
metadata:
  name: initpod
  labels:
    app: initializer
spec:
  initContainers:
  - name: init-ctr
    image: busybox
      command: ['sh', '-c', 'until nslookup k8sbook; do echo waiting for
k8sbook service;\
      sleep 1; done; echo Service found!']
    containers:
    - name: web-ctr
      image: nigelpoulton/web-app:1.0
      ports:
        - containerPort: 8080
```

deploy

```
kubectl apply -f initpod.yml
pod/initpod created

kubectl get pods --watch
```

the init:0/1 status tells that zero out of one init containers has completed succesfully. The pod will ramain in this phase until k8sbook service is created.

```
kubectl apply -f initsvc.yml
kubectl get pods --watch
```

sidecar container

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: git-synx
  labels:
    app: sidecar
spec:
  containers:
  - name: ctr-web
    image: nginx
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/
  - name: ctr-sync
    image: k8s.gcr.io/git-sync:v3.1.6
    volumeMounts:
    - name: html
      mountPath: /tmp/git
    env:
    - name: GIT_SYNC_REPO
      value: https://github.com/nigelpoulton/ps-sidecar.git
    - name: GIT_SYNC_BRANCH
      value: master
    - name: GIT_SYNC_DEPTH
      value: "1"
    - name: GIT_SYNC_DEST
      value: "html"
      epmtyDir: {}
```

the second container it watches a GitHub repo and syncs changes into the same shared volume. If the contents of the GitHub repo change, sidecar will notice and copy the new content into the shared volume where the web container will notice and update the web page.

Clean-up

```
kubectl delete pod git-sync
pod "git-sync" deleted
```