Pods

Motivation

- Pod=group of whales.
- Simplest and most important unit in K8s cluster.
- Abstraction level above containers.
- This gives also flexibility in the choice of container platform (`rkt` for instance).

So what is a pod, then?

- Collection of application containers and volumes running in the same execution environment.
- All containers in a pod land on the same machine.
- Applications in the same port share:
    - IP addresses and port space (network namespace).
    - hostname (UTS namespace).
    - System V IPC / POSIX message queues (IPC namespace).
- However, applications in different pods are completely isolated from each other, as if they were on different servers.

Pod lifecycle

- A pod's current state is in the *Status* field, which has a *Phase* subfield.
- When the state changes, the `kubelet` process in the node updates the field in the `etcd` entry.

Phases

- **Pending:** The API Server has validated the pod and created an entry for it in etcd, but its containers haven't been created or scheduled yet.
- **Running:** All of a pod's containers have been created, the pod has been scheduled on a node in the cluster, and it has running containers.

Phases (cont.)

- **Succeeded:** Every container in the pod has finished executing without returning errors or failing. The pod is considered to have completed successfully. None of the containers are going to be restarted.
- **Failed:** Every container has finished executing, but some of them have exited with a failure code.
- **Unknown:** The pod's status could not be obtained.

What goes on a pod?

- Not always easy to find out.
- Ask yourself *Will this application work correctly if the containers land in different machines?*
- If *no*, then you should create a pod for all these containers.
- If *yes*, multiple pods might be a better solution.

Why do we need pods?

- Suppose you have a container serving a REST API, that reads from a filesystem shared with a *sidecar* Git service that syncs the common filesystem.
- **Trivia:** How should we wrap our application?
- **Trivia**: Should a WordPress server and a MySQL database live on the same pod?

Answer

- A WordPress server and a MySQL db can work well if on separate machines, because communication happens through a network connection anyway.
- Putting them together is an *antipattern* for pod construction:
  - They are not symbiotic.
  - They scale differently: WordPress is mostly stateless, so can be scaled in a separate way as response to increased load. Scaling a MySQL is better done by increasing resources to its container.
- However, this is **not** the case for the Git+REST combo.

Demo

- Fire up GCP and clone the repository (if you haven't already). `https://github.com/jpmaldonado/orchestrate-with-kubernetes/`
- Creating and managing pods

Pod configuration

Custom command and arguments

- Use **command** and **args** to define shell commands in YAML that will execute when the container runs.

```yaml
#custom-command.yaml
kind: Pod
apiVersion: v1
metadata:
name: custom-command-pod
spec:
containers:
- name: command-container
image: alpine
command: ["/bin/sh"]
args: ["-c", "while true; do date; sleep 5; done"]
```

Environment variables

- Pass environment variables.

```yaml
kind: Pod
apiVersion: v1
metadata:
name: environment-variables
spec:
containers:
- name: env-var-container
image: nginx:1.7.9
# The env field lets you set environment variables
# for the container
env:
- name: BASE_URL
value: "https://api.company.com/"
- name: CONNECTION_STRING
value: "mongodb://localhost:27017"
```

Healthchecks

Is there anybody out there?

- Containers may not have an exit code that accurately tells Kubernetes whether the container was successful.
- Container probes are small processes that run periodically.
- The result of this process determines Kubernetes' view of the container's state.

Healthchecks

- **Liveness** probes are responsible for determining if a container is running or when it needs to be restarted.
- **Readiness** probes indicate that a container is ready to accept traffic.
    - Once all of its containers indicate they are ready to accept traffic, the pod containing them can accept requests.

Healthchecks (cont.)

- Passing a *readiness* check tells Kubernetes that a pod is available to receive traffic.
  - If it fails the readiness probes, Kubernetes will stop sending it traffic.
- *Liveness* checks are used to tell Kubernetes when to restart a pod.
  - If a pod fails three liveness checks kubernetes will restart it.
- The `kubelet` daemon of each node takes care of this.

Implementing probes

- One way is to use HTTP requests, which look for a successful status code in response to making a request to a defined endpoint.
- You can also use TCP sockets, which returns a failed status if the TCP connection cannot be established.
- The final, most flexible, way is to define a custom command, whose exit code determines whether the check is successful

Example (Readiness probe)

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: liveness-readiness-pod
spec:
  containers:
    - name: server
      image: python:2.7-alpine
      readinessProbe:
        initialDelaySeconds: 5
        failureThreshold: 1
        httpGet:
          path: /
          port: 8000
      env:
      - name: DELAY_START
        value: "45"
      command: ["/bin/sh"]
      args: ["-c", "echo 'Sleeping...'; sleep $(DELAY_START);
echo 'Starting server...'; python -m SimpleHTTPServer"]
```

Example (Liveness probe)

```
kind: Pod
apiVersion: v1
metadata:
        name: liveness-readiness-pod
spec:
  containers:
    - name: server
      image: python:2.7-alpine
      initialDelaySeconds: 30
      failureThreshold: 4
      httpGet:
      path: /
      port: 8000
      livenessProbe:
        initialDelaySeconds: 60
        periodSeconds: 5
        exec:
          command: ["ls", "index.html"]
      env:
      - name: DELAY_START
          value: "45"
      command: ["/bin/sh"]
      args: ["-c", "echo 'Sleeping...'; sleep $(DELAY_START);
      echo 'Starting server...'; python -m SimpleHTTPServer"]
```

Demo (http)

- Monitoring health checks.
- Why and how does this app fail?

Defining security context in pods

- A **security context** is an object that configures roles and privileges at the container level.
- If not specified for a container, it inherits the security context from the parent pod.
- Specifies the user that should run the pod, and the group for filesystem access.

Example

```yaml
kind: Pod
apiVersion: v1
metadata:
        name: security-context-pod
spec:
    securityContext:
        runAsUser: 42
        fsGroup: 231
    volumes:
    - name: simple-directory
      emptyDir: {}
    containers:
    - name: example-container
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /etc/directory/file.txt; sleep 5; done"]
    volumeMounts:
    - name: simple-directory
      mountPath: /etc/directory
```

```
kubectl exec security-context-pod -- ls -l /etc/directory
```

Multi-container pod design patterns

- Pods usually have one container, and that is enough.
- The fact that you can put many containers on a pod does not mean that you should, but there are exceptions.
  - Sidecar pattern.
  - Adapter pattern.
  - Ambassador pattern.

Sidecar pattern

- This pattern consists of your web application plus a helper container with a responsibility that is useful to your application, but is not necessarily part of the application itself.
- The most common sidecar containers are logging utilities, sync services, watchers, and monitoring agents.

Adapter pattern

- Standardize and normalize application output or monitoring data.
- Example: request timing in Ruby is [DATE] - [HOST] - [DURATION], but in Node this is [HOST] - [START_DATE] - [END_DATE].
- Good alternative to annoy a developer :)

Ambassador pattern

- An ambassador container is essentially a proxy that allows other containers to connect to a port on localhost.
- For example, to connect to a database: your main application only cares about connecting to a port in localhost, the ambassador container will handle the connections.
- Alternative to using environment variables to store connection strings for different environments.

Your turn!

- Implement a pod with a sidecar pattern with the following containers:
  - Main application that writes the current date to a log file every 5 seconds.
  - A sidecar nginx container that serves that file.
- Once the pod is running, connect to the sidecar, install `curl` and use it to access the log file.

```yaml
# Sidecar container (hint)
- name: sidecar-container
# Simple sidecar: display log files using nginx.
image: nginx:1.7.9
  ports:
  - containerPort: 80
```