Pod Status and Conditions

Pod Status

Pod status tells where the pod is in its lifecycle. It can be viewed using `k get pods` command.

- **Pending** - pod is waiting to be scheduled on a node
- **ContainerCreating** - pulling the images and starting the containers for the pod
- **Running** - the containers inside the pod are running

Pod Conditions

Binary values signifying the state of a pod. It can be viewed by running `k describe pod <pod- name>` command and looking at the conditions section.

- **PodScheduled** - pod has been scheduled on a node
- **Initialized** - pod has been initialized
- **ContainersReady** - containers inside the pod are ready to run
- **Ready** - pod is ready to run (when all the containers inside the pod are ready to run)

Readiness Probes

Readiness probes allow k8s to probe the application running inside the container to check if it's ready yet or not. Only after the application is ready, k8s sets the `Ready` condition of the container to `True`.

If multiple replicas of the same pods are serving traffic for an application and a new replica pod is added, if the readiness probes are set correctly, the service will wait for the application inside the new replica container to start before sending traffic to the pod.

By default, k8s sets the `Ready` condition on the container to `True` as soon as the container starts. This means that the pod will become ready to accept requests from the service as soon as the pod's `Ready` condition becomes `True`. If the application running inside the container takes longer to start, this would cause the service to start sending requests even before the application has started, because the state of the pod (or container) is ready.

Readiness check is done at the container level in one of the following ways:

- HTTP based
- TCP based
- Shell script based

HTTP based Readiness Check

This is commonly used for containers hosting web applications. The application exposes an HTTP health check endpoint. Only if the endpoint returns a 200 status code, the container will be considered ready.

```
apiVersion: v1 kind: Pod metadata: labels: name: frontend spec: containers: -
name: webapp image: webapp ports: - containerPort: 8080 readinessProbe: httpG
et: path: /api/ready port: 8080
```

TCP based Readiness Check

This is commonly used for containers hosting databases. The container's TCP port on which the DB is exposed is checked for readiness.

```
apiVersion: v1 kind: Pod metadata: labels: name: database spec: containers: -
name: database image: database ports: - containerPort: 3306 readinessProbe: t
cpSocket: port: 3306
```

Shell Script based Readiness Check

Run a shell script inside the container to check the readiness of the application. The return code of the shell script is used to determine the readiness of the container.

```
apiVersion: v1 kind: Pod metadata: labels: name: app spec: containers: - nam
e: app image: app readinessProbe: exec: command: - cat - /app/ready
```

Configuration

```
readinessProbe: httpGet: path: /
api/ready port: 8080 initialDela
ySeconds: 10 periodSeconds: 5 fa
ilureThreshold: 5
```

`initialDelaySeconds` - start checking for readiness after some delay (when we know the application takes some time to start)

`periodSeconds` - readiness check interval

`failureThreshold` - how many times to check for readiness before declaring the status of container to failed and restart the container (default 3)

Liveness Probes

While Readiness Probes check the application running inside the container for readiness, **Liveness Probes check the application running inside the container periodically to check if the application is healthy** (live). If the application becomes unhealthy, the pod gets restarted.

Without liveness probes, the application could be stuck in an infinite loop or frozen while the status of the pod is running, making us believe that the application is working fine. In this case, the pod will not be restarted.

💡 Liveness probes are configured just like Readiness probes, we just use `livenessProbe` instead of `readinessProbe` in the pod definition.

HTTP based Liveness Check

This is commonly used for containers hosting web applications. The application exposes an HTTP health check endpoint. Only if the endpoint returns a 200 status code, the container will be considered live.

```
apiVersion: v1 kind: Pod metadata: labels: name: frontend spec: containers: -
name: webapp image: webapp ports: - containerPort: 8080 livenessProbe: httpGe
t: path: /api/ready port: 8080
```

TCP based Liveness Check

This is commonly used for containers hosting databases. The container's TCP port on which the DB is exposed is checked for liveness.

```
apiVersion: v1 kind: Pod metadata: labels: name: database spec: containers: -
name: database image: database ports: - containerPort: 3306 livenessProbe: tc
pSocket: port: 3306
```

Shell Script based Liveness Check

Run a shell script inside the container to check the liveness of the application. The return code of the shell script is used to determine the liveness of the container.

```
apiVersion: v1 kind: Pod metadata: labels: name: app spec: containers: - nam
e: app image: app livenessProbe: exec: command: - cat - /app/ready
```

Configuration

```
livenessProbe: httpGet: path: /a
pi/ready port: 8080 initialDelay
Seconds: 10 periodSeconds: 5 fai
lureThreshold: 5
```

`initialDelaySeconds` - start checking for liveness after some delay

`periodSeconds` - liveness check interval

`failureThreshold` - how many times to check for liveness before declaring the status of container to failed and restart the container (default 3)

Monitoring

Monitoring involves collecting information regarding the the cluster and its performance metrics such as memory utilization, disk utilization, network utilization etc. Monitoring data is retrieved from the Kubelet service running on each node.

K8s does not have a native monitoring solution. There are many 3rd party open-source monitoring solutions like Metrics Server, Elastic Stack, Prometheus, etc. There are also some proprietary monitoring solutions like DataDog and DynaTrace.

Metrics Server

It is an open-source **in-memory monitoring solution** built as a slim-down version of Heapster (monitoring tool used earlier). To setup metric server, clone the below repo and run `k apply -f .` inside it.

```
git clone https://github.com/kubernetes-incubator/metrics-server.git
```

We can then run `k top node` to see the nodes consuming most resources and `k top pods` to see the same for pods.

> 💡 A better way to monitor the cluster is to use a dedicated monitoring solution.

Prometheus and Grafana

- Tutorial on installing Prometheus and Grafana using Helm on a K3s cluster
- Prometheus on K8s demo (must watch for customizing prometheus installation)

Installation

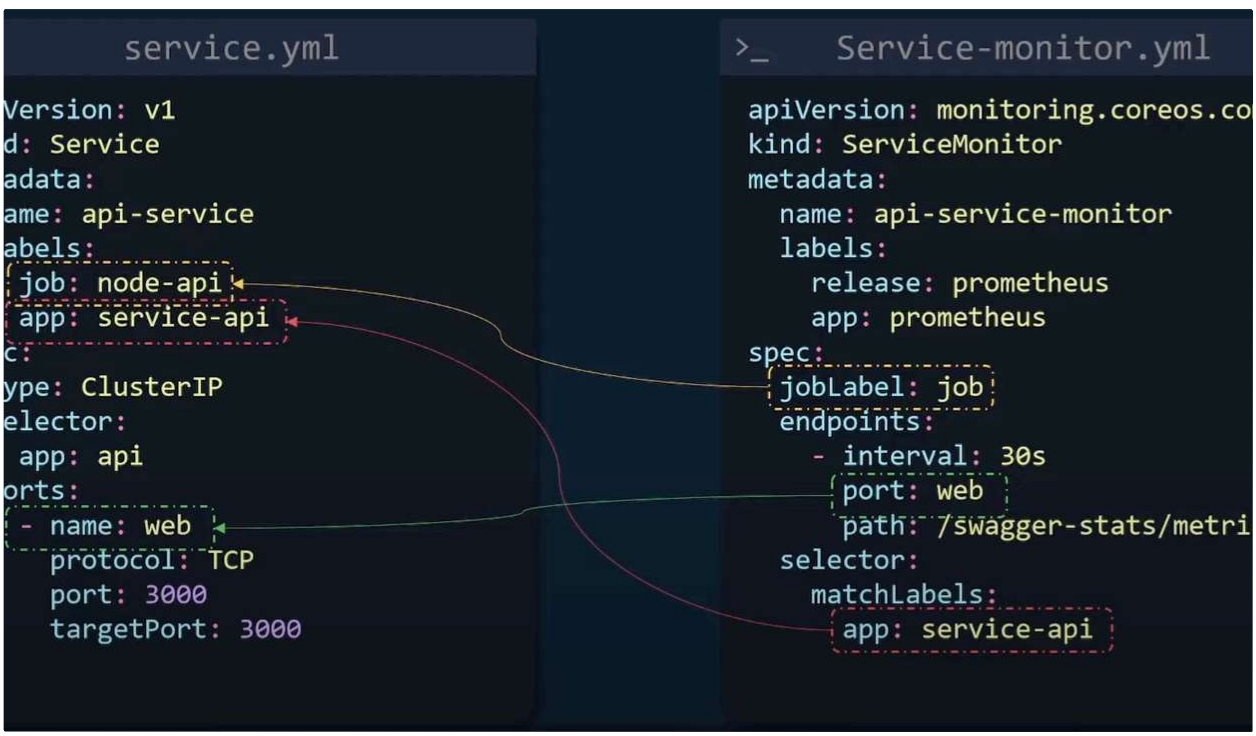
Use kube-prometheus-stack helm chart to deploy all the required components on the cluster, including grafana and alert manager. The Prometheus operator creates several CRDs to provide abstraction and allow us to configure prometheus by creating K8s manifests.

The prometheus UI is available on port 9090 on the Prometheus server (check for corresponding service for pod `prometheus-prometheus`). The helm chart installs `node-exporter` daemonset on each node which exports system level metrics for each node to the prometheus server.

`ServiceMonitor`

`ServiceMonitor` CRD (created by the Prometheus Operator) can be used to add a scrape target to Prometheus. The `kube-prometheus-stack` helm chart automatically creates some service monitors to scrape the cluster control plane components. We can also add our own service monitors to scrape metrics from applications running inside the pods.

In the example below, a service monitor is created to scrape the `api-service` every 30 seconds for metrics on port `web` (3000) at path `/swagger-stats/metrics`. The name of the scraping job will be `node-api` in this case.



`PrometheusRule`

To add new rules to Prometheus, we can create a `PrometheusRule` object (CRD created by the Prometheus Operator). The `kube-prometheus-stack` helm chart automatically creates some prometheus rules in the cluster.

Prometheus-rule.yml

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    release: prometheus
  name: api-rules
spec:
  groups:
    - name: api
      rules:
        - alert: down
          expr: up == 0
          for: 0m
          labels:
            severity: critical
          annotations:
            summary: Prometheus
target missing {{$labels.instance}}
```

`AlertManagerRule`

Alert manager rule can be added by created a `AlertManagerRule` CRD (created by the Prometheus Operator). This requires the prometheus helm chart to be installed with the following present in the `values.yaml` to match the labels under the `metadata` section.

```yaml
alertmanagerConfigSelector: matchLabels: resource: prometheus
```

Prometheus-rule.yml

```yaml
apiVersion: monitoring.coreos.com/v1alpha1
kind: AlertmanagerConfig
metadata:
  name: alert-config
  labels:
    resource: prometheus
spec:
  route:
    groupBy: ["severity"]
    groupWait: 30s
    groupInterval: 5m
    repeatInterval: 12h
    receiver: "webhook"
  receivers:
    - name: "webhook"
      webhookConfigs:
        - url: "http://example.com/"
```

Logging

Logging in containerized applications or Kubernetes involves running an agent (LogStash, FluentD, etc.) on the host (k8s nodes) to push the logs to a central database (ElasticSearch, **Loki**, etc.).

EFK Stack

- **No access** is used for log collection
- ElasticSearch is used as the DB to store the logs sent by FluentD
- Kibana is the web interface to view the logs stored in ElasticSearch

Grafana Loki

Must watch tutorial: Mastering Grafana Loki: Part 1

Grafana Loki is a log aggregation tool which uses **Promtail** as the log collecting agent by default (can be configured to use FluentBit instead). **Promtail runs as a DaemonSet** and pushes logs to Loki, which is the database that stores and indexes the logs. Once the logs are present in Loki, it can be queried by Grafana and displayed on the UI.

Unlike other logging systems, a Loki index is built from labels, leaving the original log message unindexed. This means, Loki is much more resource efficient compared to other logging tools.

Loki is built out of many component microservices, and is designed to run as a horizontally-scalable distributed system. It has three modes of operation:

- **Monolithic:** runs all of Loki's microservice components inside a single process as a single binary or Docker image, can only use filesystem for storage
- **Scalable**: separates the reads and writes to the backend datastore to improve performance, requires a managed object store such as AWS S3 or a self-hosted store such as Minio
- **Microservices**: separates each component of Loki as a separate process for maximum scalability and efficiency (default in Helm chart installations)

Helm Installation of Loki (monolithic mode)

Refer Install the monolithic Helm chart | Grafana Loki documentation.

```
helm repo add grafana https://grafana.github.io/helm-charts helm repo update
```

```
loki: commonConfig: replication_factor: 1 storage: type: 'filesystem' auth_enabled: false singleBinary: replicas: 1
```

values.yaml

```
helm install loki grafana/loki -n logging --values values.yaml
```

Helm Installation of Promtail

If you added the Helm repo in the above section, you can just install Promtail as a chart. We don’t need to update the `values.yaml` file. Refer Promtail | Grafana Loki documentation.

```
helm install loki grafana/promtail -n logging
```

Tah

Thi

log

`id`