Kubernetes Object and Service

SoC Summer Workshop
Cloud Computing with Big Data

Richard T. B. Ma
School of Computing
National University of Singapore

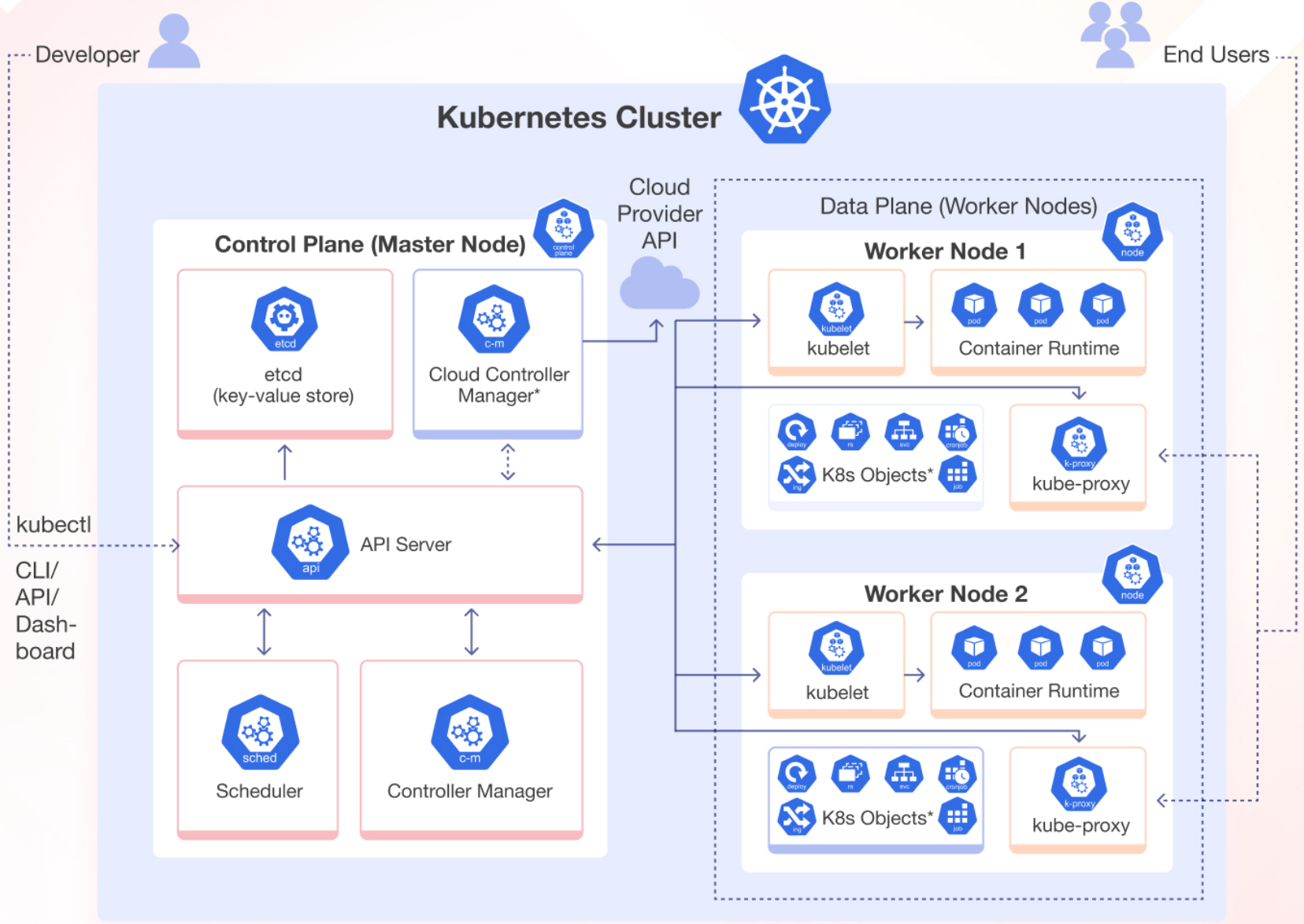
Formation of Project Groups

- Each project group consists of 4 students
  - with two groups that have 5 students
  - to be fair, each third-year student will lead a group with second-year students
- Potential topics of projects
  - traditional applications
  - new applications over the cloud
- Types of projects
  - cloud system with orchestration
  - cloud-native big data application

Roadmap

- Quick Review
- Kubernetes Object and Manifest Files
- Kubernetes Service
- We'll have a tutorial tomorrow on
  - Amazon and Google Container Engine (GKE)

Kubernetes Architecture



Kubernetes Node Components

1. **Kubelet**: an agent responsible for managing the node, communicating with the API server, and ensuring that containers are running as specified.
2. **Container Runtime**: a container runtime, such as Docker, containerd, or CRI-O, is responsible for pulling container images and running containers.
3. **Kube-proxy**: a network proxy responsible for implementing K8s service abstraction by maintaining network rules and forwarding traffic to the appropriate pods.
4. **Pod CIDR**: a range of IP addresses assigned, can be allocated to pods running on that node.

Kubernetes Node Operations

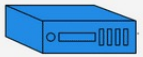
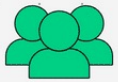
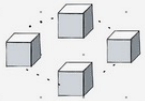
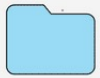
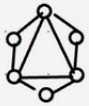
1. **Pod Scheduling**: Nodes are responsible for hosting and running pods. When a pod is scheduled to run, the K8s scheduler selects a node based on resource requirements, affinity/anti-affinity rules, node selectors, and other constraints.
2. **Status and Health**: Nodes report their status and health to the API server, e.g., resource utilization, node conditions (e.g., Ready, NotReady), and any issues or errors encountered on the node.
3. **Scaling and Capacity**: K8s clusters can dynamically add or remove nodes to accommodate changing workload demands, i.e., horizontal scaling.

Roadmap

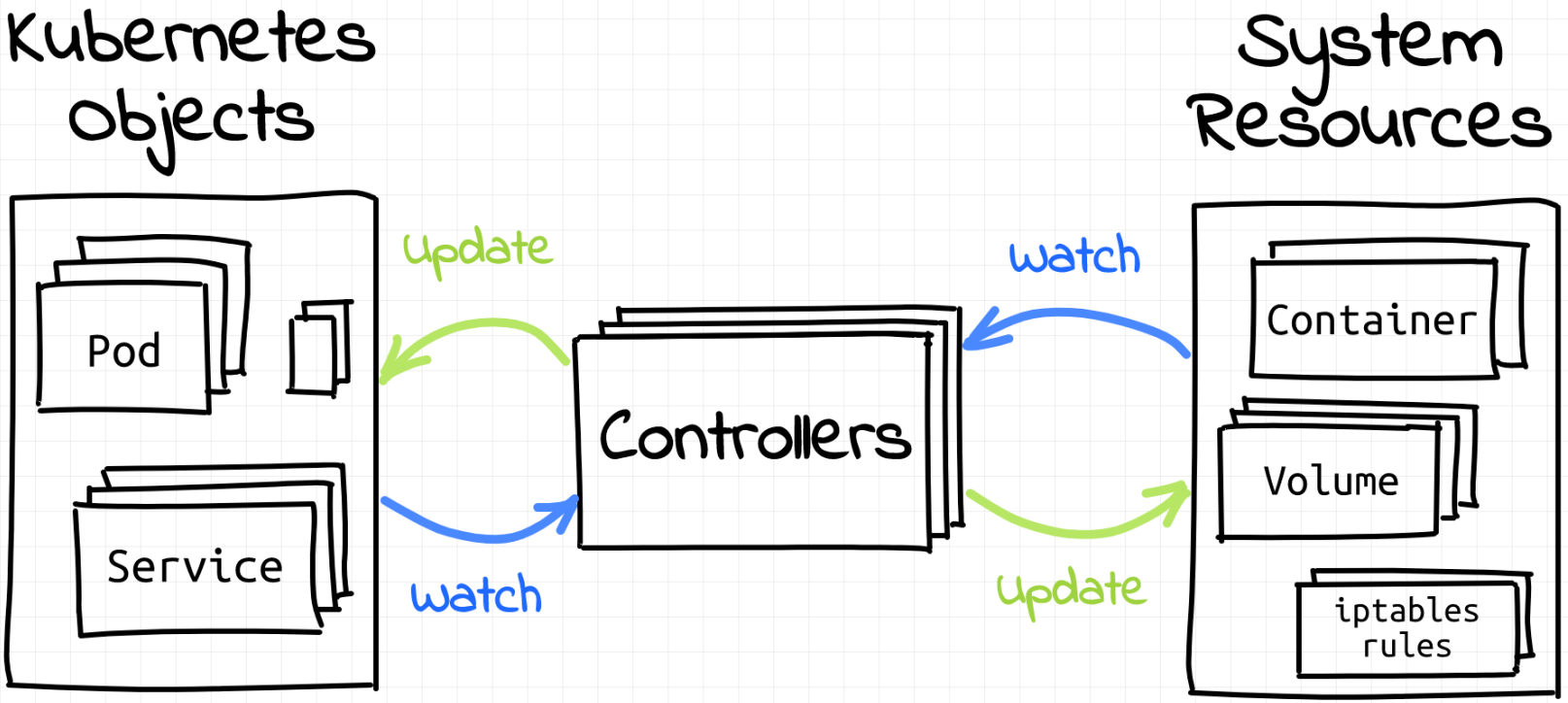
- Quick Review
- Kubernetes Object and Manifest Files
- Kubernetes Service

KUBERNETES NATIVE OBJECTS - PART 01

devopscube

Deployment Unit  Pod	Networking  Service	Storage  Volume	Isolation  Namespace
Replication  Replicaset	POD Management  Deployment	State Management  StatefulSet	Node Operation  DaemonSet
Task Execution  Job	Scheduled Tasks  CronJob	Sensitive Data  Secret	Configuration  ConfigMap
External Access  Ingress	Network Rules  NetworkPolicy	Persistent Storage  Persistent Volume	Storage Request  PV Claim
Scalability  HPA/VPA	Network endpoints  EndpointSlices	Authentication  ServiceAccount	Authorization Role/ClusterRole

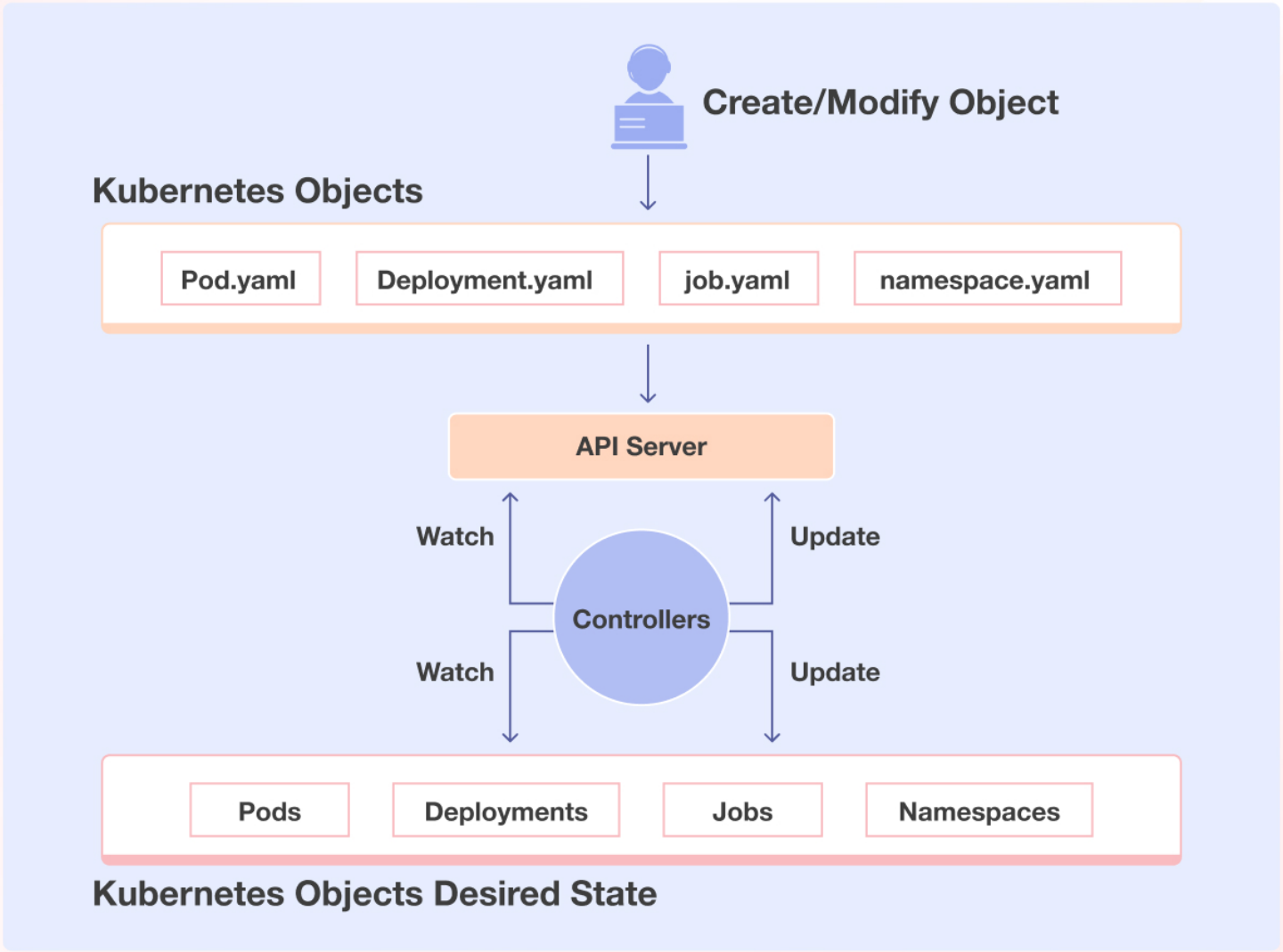
Objects and Controllers



Kubernetes Object

- Persistent entities in K8s, representing a "record of intent" that tells what you want
  - described as a "specification" or "spec"
  - K8s manages to reach the *desired state*.
- The *current* state, i.e., "status", of cluster
  - what apps are running on which nodes
  - the resources available to those apps
  - the policies around how the apps behave, e.g., restart policies, upgrades, and fault-tolerance
- Use K8s API via `kubectl` or client libs.

kube-controller-manager



Structure of YAML Manifest Files

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server-pod-v1
spec:
  containers:
  - name: server-container
    image: yancanmao/server-image
```

- apiVersion
- kind
  - type of object
- metadata
  - name, label, & etc.
- spec
  - kind-specific defs
- status
  - runtime states

```
$ kubectl get pod server-pod-v1 -o yaml
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server-pod-v1
spec:
  containers:
  - image: yancanmao/server-image
    name: server-container
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2024-05-19T03:07:31Z"
    status: "True"
    type: PodReadyToStartContainers
  - lastProbeTime: null
    lastTransitionTime: "2024-05-19T03:07:28Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2024-05-19T03:07:31Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2024-05-19T03:07:31Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2024-05-19T03:07:28Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: docker://57789f43840a65ae7e3290dc023230c381f2fc6146516b13ae366d96acbb700a
    image: yancanmao/server-image:latest
    imageID: docker-pullable://yancanmao/server-
image@sha256:1d79b0f25e226f852ccdae1b65af4a069e77699af3142b53cc084316196b7d3c
    lastState: {}
    name: server-container
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2024-05-19T03:07:30Z"
  hostIP: 192.168.49.2
  hostIPs:
  - ip: 192.168.49.2
  phase: Running
  podIP: 10.244.0.10
  podIPs:
  - ip: 10.244.0.10
  qosClass: BestEffort
  startTime: "2024-05-19T03:07:28Z"
```

Object Status

- apiVersion
- kind
  - object & non-object
- metadata
  - name, label, & etc.
- spec
  - kind-specific defs
- status
  - runtime states

Object Metadata

- Every object kind MUST have:
  - **name**: a string that uniquely identifies this object within the current namespace, used in the path when retrieving an object.
  - **namespace**: a DNS-like label that objects are subdivided into.
  - **uid**: a unique value in *time* & space used to distinguish between objects of the same name that have been deleted & recreated.
- Every object kind SHOULD have:
  - **labels**: a map of string keys and values, i.e., key-value pairs, that can be used to organize and categorize objects.
  - **annotations**: key-value pairs, can be used by external tooling to store and retrieve arbitrary metadata about this object.
  - **resourceVersion**: the internal version of this object, used to guarantee consistent updating.

Namespace

- A mechanism for isolating groups of resources within a cluster.
  - Quotas can be defined for each namespace to limit the resources consumed.
  - Unique names for resources within a namespace, but not across namespaces
- Scoping is applicable for namespaced objects, e.g., Service, not for cluster-wide objects, e.g. Node.
  - Resources within the namespaces can refer to each other with their service names.
  - Resources across namespace can be reached via:

```
<service_name>.<namespace_name>.svc.cluster.local
```

Namespace – how to use?

- Create a namespace:

```
$ kubectl create namespace <namespace_name>
```

- The active namespace is recorded in the "context"
  - a configuration file for `kubectl`
  - located at `$HOME/.kube/config`
  - also include info about the cluster and users
- Change the namespace in the current context:

```
$ kubectl config set-context --current --
namespace=<namespace_name>
```

- Change context:

```
$ kubectl config use-context <context_name>
```

Object Spec ⇔ Object Status

- Spec: the desired state of an object
  - configs from users
  - default values by system
  - initialize/updated by other components, e.g., scheduler, auto-scaler
  - persisted in storage with the API object
  - if deleted, the object will be purged from the system
- Status: current state of the object in system
  - usually persisted with the object by automated processes, generated on the fly
  - should be the most recent observations
  - may contain info such as the results of operations executed in response to the spec

Roadmap

- Quick Review
- Kubernetes Object and Manifest Files
- Kubernetes Service

How to interact with the containers?

- Remote login to execute and interact with programs inside containers

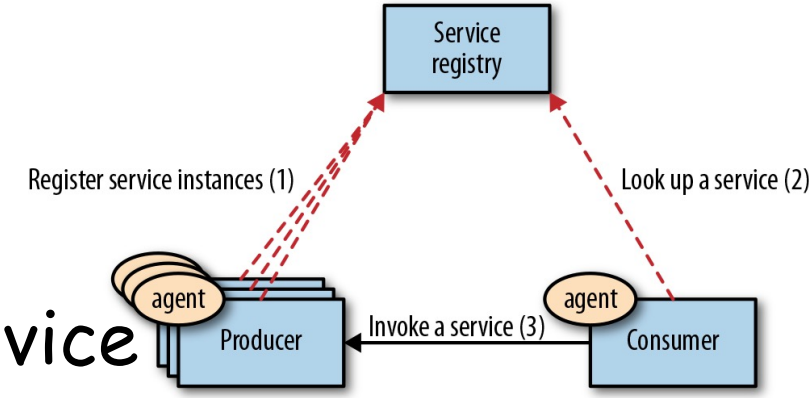
```
$ kubectl exec mypod –stdin –tty -- /bin/bash
```

- Access from another pod within the cluster
  - need to know the internal IP of the destination pod
- What if I want to access from my local machine?
  - port forwarding: access a specific port of a container running inside a Kubernetes Pod from your local machine

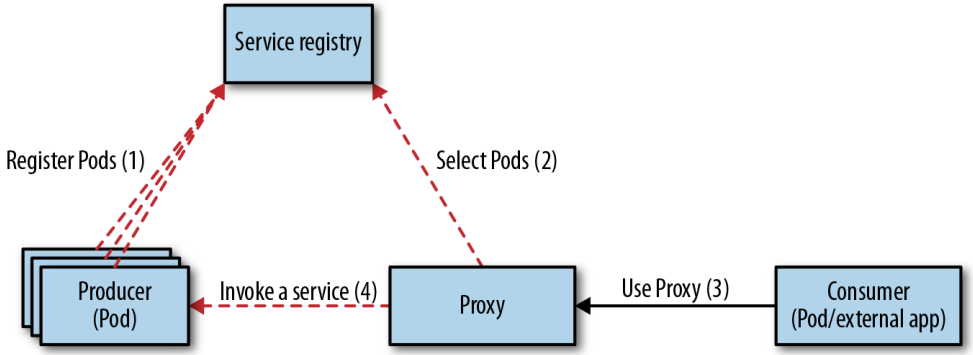
```
$ kubectl port-forward --address 0.0.0.0 mypod 8888:8080
```

  - for temporary access during development or debugging

Service Discovery



- Traditional client-side service discovery
  - an agent capable of looking at a registry for service instances and choosing one to call
- K8s: server-side service discovery



  - consumer calls a fixed virtual Service endpoint that can dynamically discover service instances

Kubernetes Service

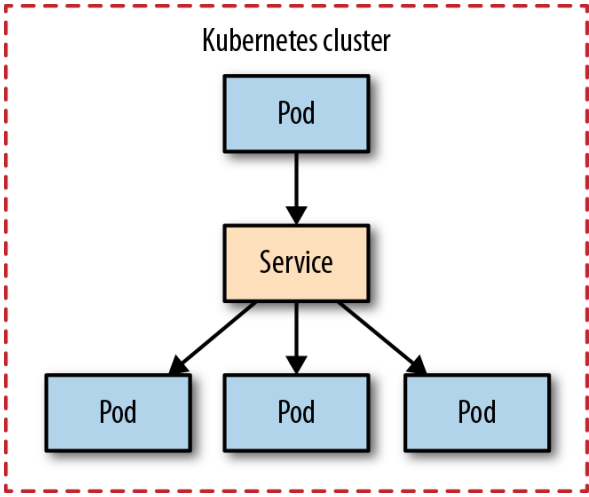
- **Problem**: K8s doesn't treat pods as unique, long-running instances; if a pod encounters an issue and dies, K8s replaces it for the app to minimize downtime.
- However, as pods are replaced, their internal names and IPs might change ➔ a pod's lifetime is not reliable.
- **Solution**: An abstraction over Pods, i.e., Service, serves as the interface the app consumers interact with.
  - A service exposes a single machine name or IP address mapped to pods whose underlying names and numbers are unreliable.
  - A service ensures that, to the outside network, everything appears to be unchanged.

Service YAML Structure

- Common spec fields
  - ports
  - type
- spec:ports:port
  - the port number on which the Service will listen for incoming traffic
  - used to communicate with the Service
- spec:ports:protocol
  - specifies the protocol used for the port
  - if not specified, TCP is used by default

```yaml
apiVersion: v1
kind: Service
metadata:
  name: http-service
spec:
  ports:
    - name: http
      port: 80
      protocol: TCP
  type: ...
```

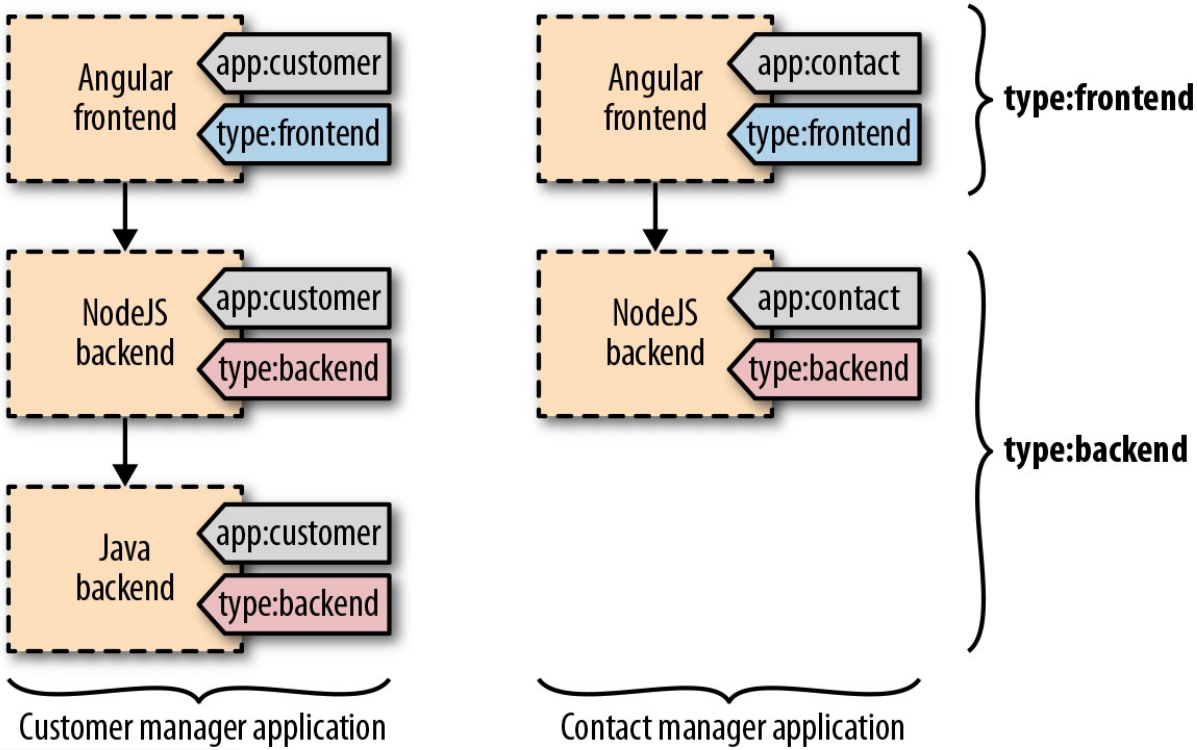
Service type: ClusterIP



- Scenario: Internal Service Discovery
  - client: pods inside K8s cluster
  - server: pods inside K8s cluster
- Type-specific field: `targetPort`
  - the port on the backend Pods to which the incoming traffic will be forwarded
  - the port that your application inside the Pod is listening on
- Type-specific field: `selector`
  - conditions target Pods need to satisfy
  - used to match Pod labels

```yaml
apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: server
```

Labels




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server-pod-v2
  labels:
    app: server
spec:
  containers:
  - name: server-container
    image: yancanmao/server-image
```

- used as an application identity for Pods

ClusterIP - how to use?

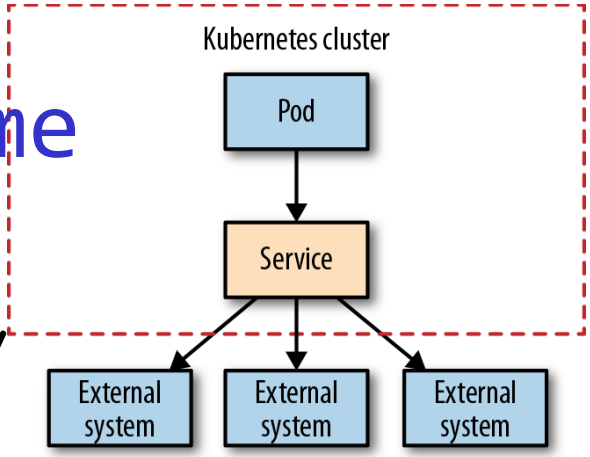
- once a Service is created, it gets a clusterIP assigned that is accessible only from within the Kubernetes cluster, and that IP remains unchanged as long as the Service definition exists.

- when a client knows the name of the Service it wants to access, it can reach the Service by a fully qualified domain name (FQDN) such as

```
SERVICE_Name.default.svc.cluster.local
```

```
$ kubectl port-forward service/$SERVICE_Name 8888:8080
```

Service type: ExternalName



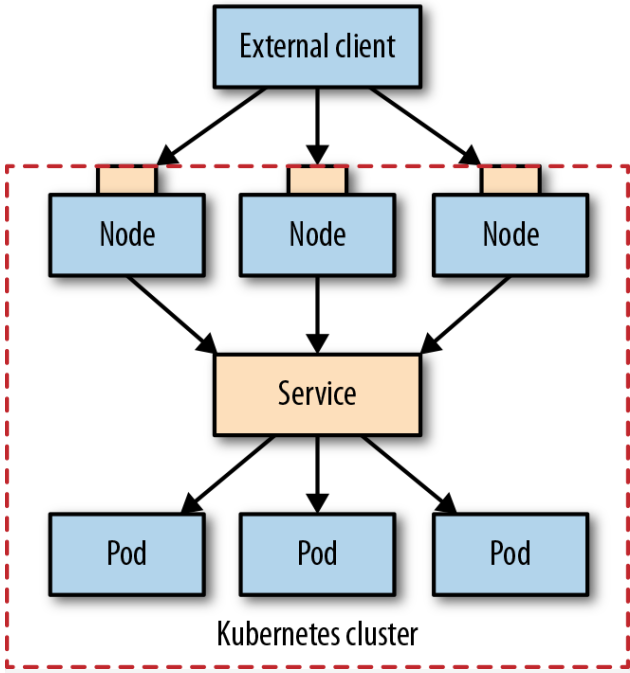
- Scenario: External Service Discovery
  - client: pods inside K8s cluster
  - server: pods outside K8s cluster
- redirect connections to external IP addresses and ports by omitting the selector definition of a Service and manually creating endpoint resources
- Type-specific field: `externalName`
  - CNAME record that points to the external name you specified
  - try minikube using `host.minikube.internal`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: externalname-service
spec:
  type: ExternalName
  externalName: my-server.com
    - port: 80
```

Service type: NodePort

- Scenario: Service Discovery from outside K8s cluster
  - client: outside K8s cluster
  - server: pods inside K8s cluster

- Type-specific field: nodePort
  - port opened on all nodes, through which pods inside the cluster are accessed
  - has a range from 30000 to 32767




```yaml
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30036
  selector:
    app: server
```