

Kubernetes: Pods and Deployments

Course Outline

1. What is Kubernetes?

- Components
- Installation

2. Basics of Docker

- Namespaces
- Building and running Docker images

3. Pods and Deployments

- Running basic workloads in Kubernetes
- Scale, Update, Rollback

4. Advanced Pod configuration

- Args, Envs, ConfigMaps, Secrets
- Init- and sidecar containers
- Scheduling and debugging

5. Networking in Kubernetes

- What are network plugins?
- Service abstraction and ingress

6. Persistent storage

- Basics of storage: block vs. object vs. file system
- StorageClass, PVC, PV

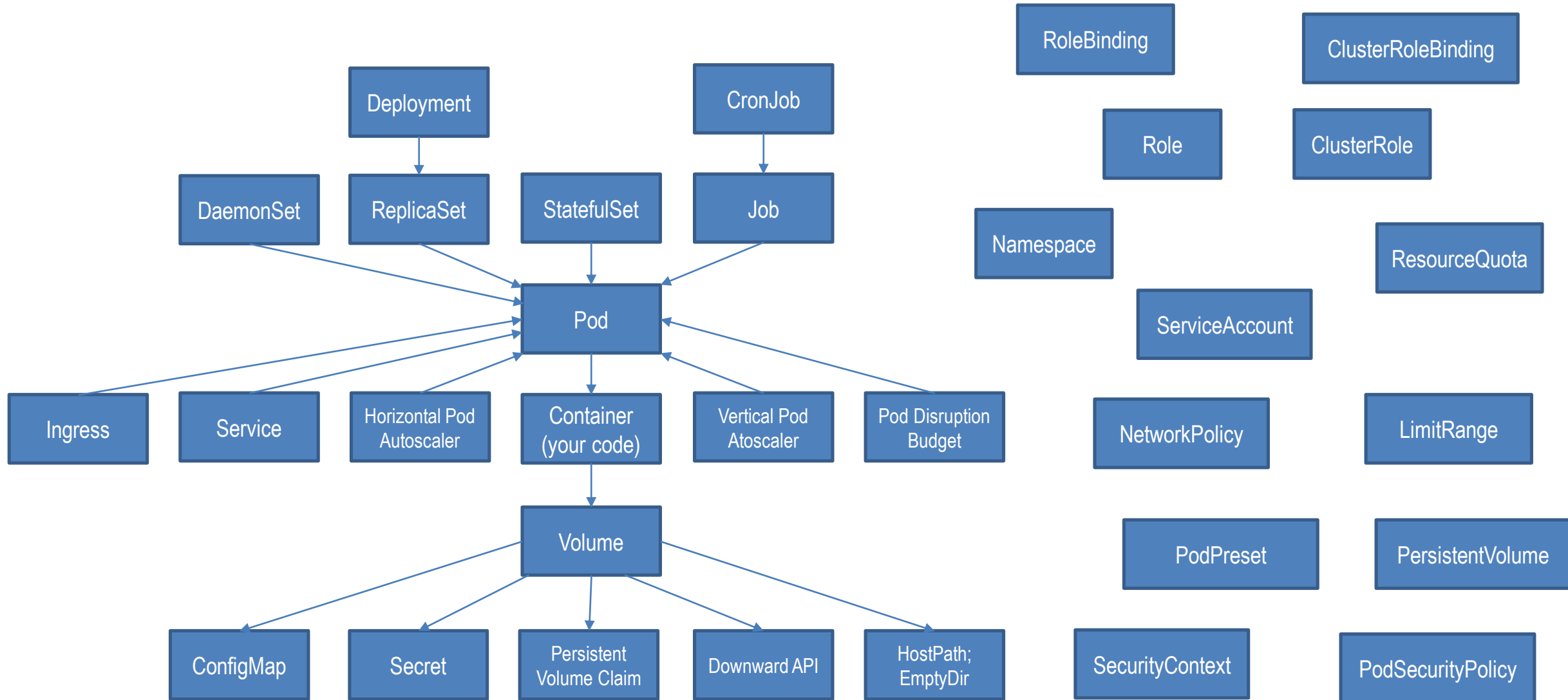
7. Security

- RBAC: Roles, ServiceAccounts, RoleBindings
- Security context and network security policy

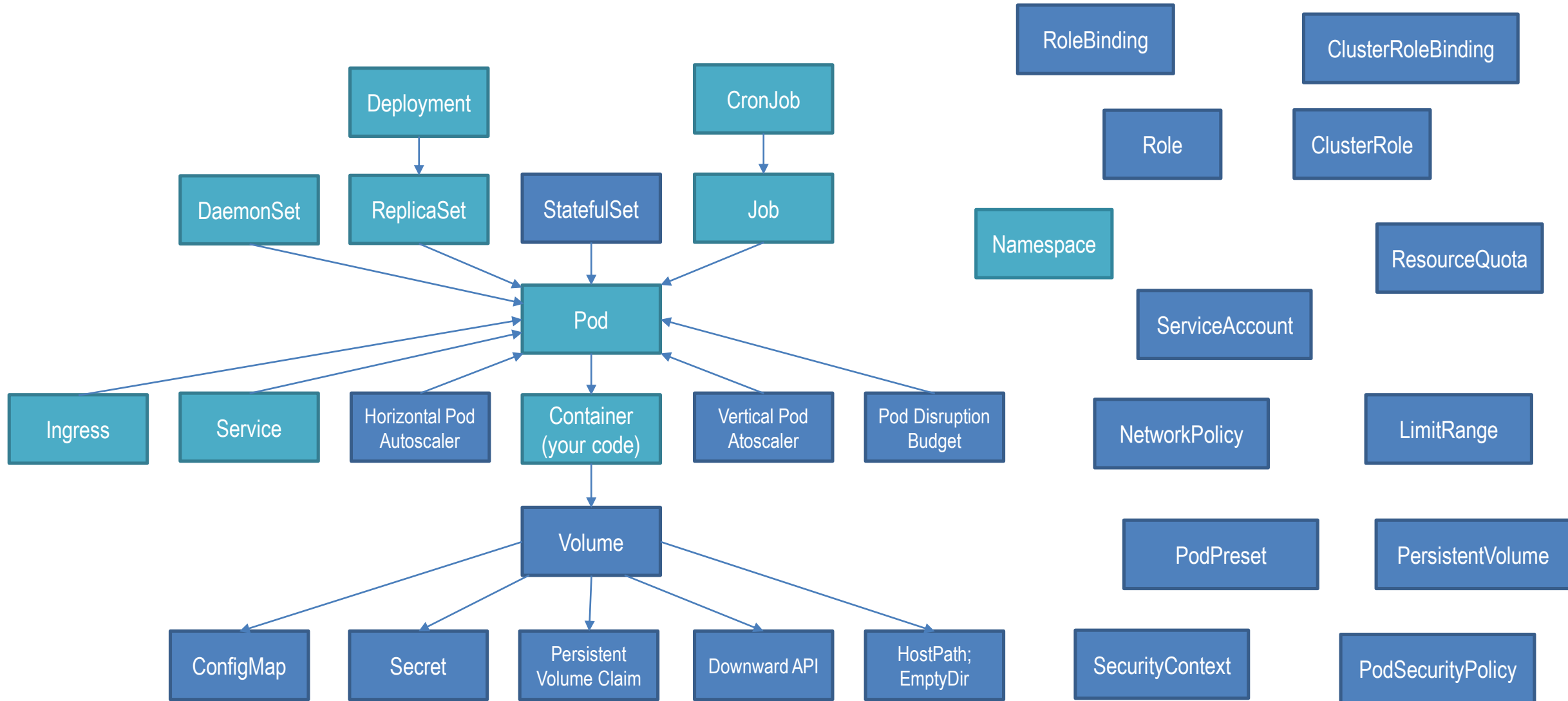
8. Advanced topics

- Helm
- Custom resources and operators

Dev/Ops Resources

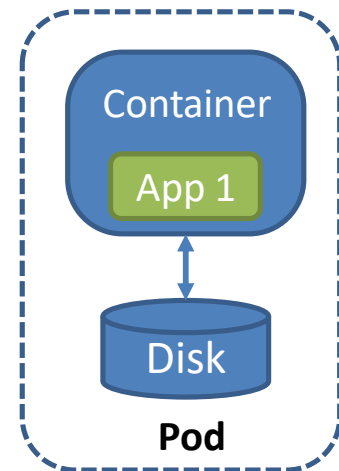


Dev/Ops Resources



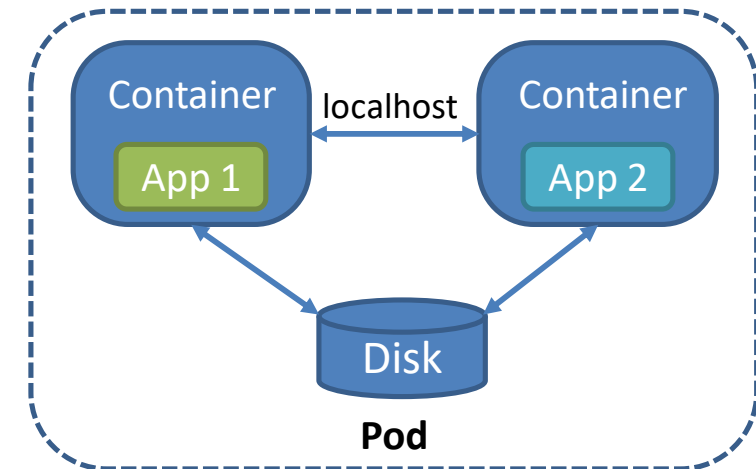
Container versus Pod

- Containers are the building blocks for Kubernetes-based cloud-native applications.
- However, Kubernetes offers another primitive: Pod
 - for managing the lifecycle of a group of containers
 - so to clarify:
 - container --> the app itself
 - Pod --> managing one or more apps
 - container cannot be started without a Pod --> Pod is the smallest unit of K8s
- Container characteristics:
 - A container image
 - is the unit of functionality that addresses a single concern.
 - is owned by one team and has a release cycle.
 - is self-contained and defines and carries its runtime dependencies.
 - is immutable, and once it is built, it does not change; it is configured.
 - has defined runtime dependencies and resource requirements.
 - has well-defined APIs to expose its functionality.
 - A container:
 - runs typically as a single Unix process.
 - A container is disposable and safe to scale up or down at any moment.



Container versus Pod?

- Pod characteristics:
 - Sandbox for 1+ containers
 - A *Pod* is an atomic unit of **scheduling**, **deployment**, and **runtime isolation** for a group of containers.
 - Scheduler tries to find a host that satisfies the requirements of all containers in the Pod (no multi Node Pods)
 - Ensures colocation of containers: localhost, shared local file system, share access to memory space
 - Pod has an IP address, name and port range that are shared by all containers in it (port clash can happen!)
 - Containers within Pod are deployed in an all-or-nothing manner
 - Why multi container Pod is a thing? When containers should be placed in the same Pod?
 - One “main” and multiple “sidecars” (log watcher, file loader, etc.)
 - Proxy, bridge, adapter
 - 3-tier app --> **Absolutely not!**
 - Multiple entities of the same container for scaling --> **Absolutely not!**
 - No auto healing or scaling
 - IP address is ephemeral
 - Pod crashes must be handled by higher level resources:
 - ReplicaSet: scaling and healing
 - Deployment: versioning and rollback
 - Service: static (non-ephemeral IP) and networking
 - Volume: non-ephemeral storage



Pod: yaml (minimal)

```
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    run: www
  namespace: development
spec:
  containers:
  - image: nginx:1.17
    name: www
```

Pod: yaml (minimal)

```
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    run: www
  namespace: development
spec:
  containers:
  - image: nginx:1.17
    name: www
```


→ imply different levels of stability and support:

- alpha:
 - The version names contain *alpha* (e.g. *v1alpha1*)
 - Maybe buggy, disabled by default
 - Support for feature may be dropped at any time without notice
 - The API may change in incompatible ways in a later release
- beta:
 - The version names contain *beta* (e.g. *v2beta3*).
 - Code is well tested, enabled by default
 - Support for the overall feature will not be dropped, though details may change
 - The schema and/or semantics of objects may change in incompatible ways, but migrating instructions are provided
 - Recommended for only non-business-critical uses
- stable:
 - The version name is *vX* where *X* is an integer.
 - Stable features will appear in released software

Pod: yaml (minimal)

```
apiVersion: v1
```

```
kind: Pod
```



Resource type, e.g. Pod, Service, ReplicaSet, Deployment, Ingress, etc.

```
metadata:
```

```
  name: www
```

```
  labels:
```

```
    run: www
```

```
  namespace: development
```

```
spec:
```

```
  containers:
```

```
  - image: nginx:1.17
```

```
    name: www
```

Pod: yaml (minimal)

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:  Information about pod that can be used for managing it
```

```
  name: www
```

```
  labels:
```

```
    run: www
```

```
  namespace: development
```

```
spec:
```

```
  containers:
```

```
  - image: nginx:1.17
```

```
    name: www
```

Pod: yaml (minimal)

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: www
```

```
  labels:
```

```
    run: www
```

```
  namespace: development
```

```
spec:  Information that is specific to the given kind of resource.
```

```
  containers:
```

```
  - image: nginx:1.17
```

```
    name: www
```

Pod: yaml (minimal)

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: www
```

```
  labels:
```

```
    run: www
```

```
  namespace: development
```

```
spec:
```

```
  containers:
```

```
  - image: nginx:1.17
```

```
    name: www
```

each object has it,
unique for that type of resource:

- only one Pod can have “name: www”
- but a Service or container can have “name: www” as well

Pod: yaml (minimal)

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: www
```

```
  labels:
```

```
    run: www
```

```
  namespace: development
```

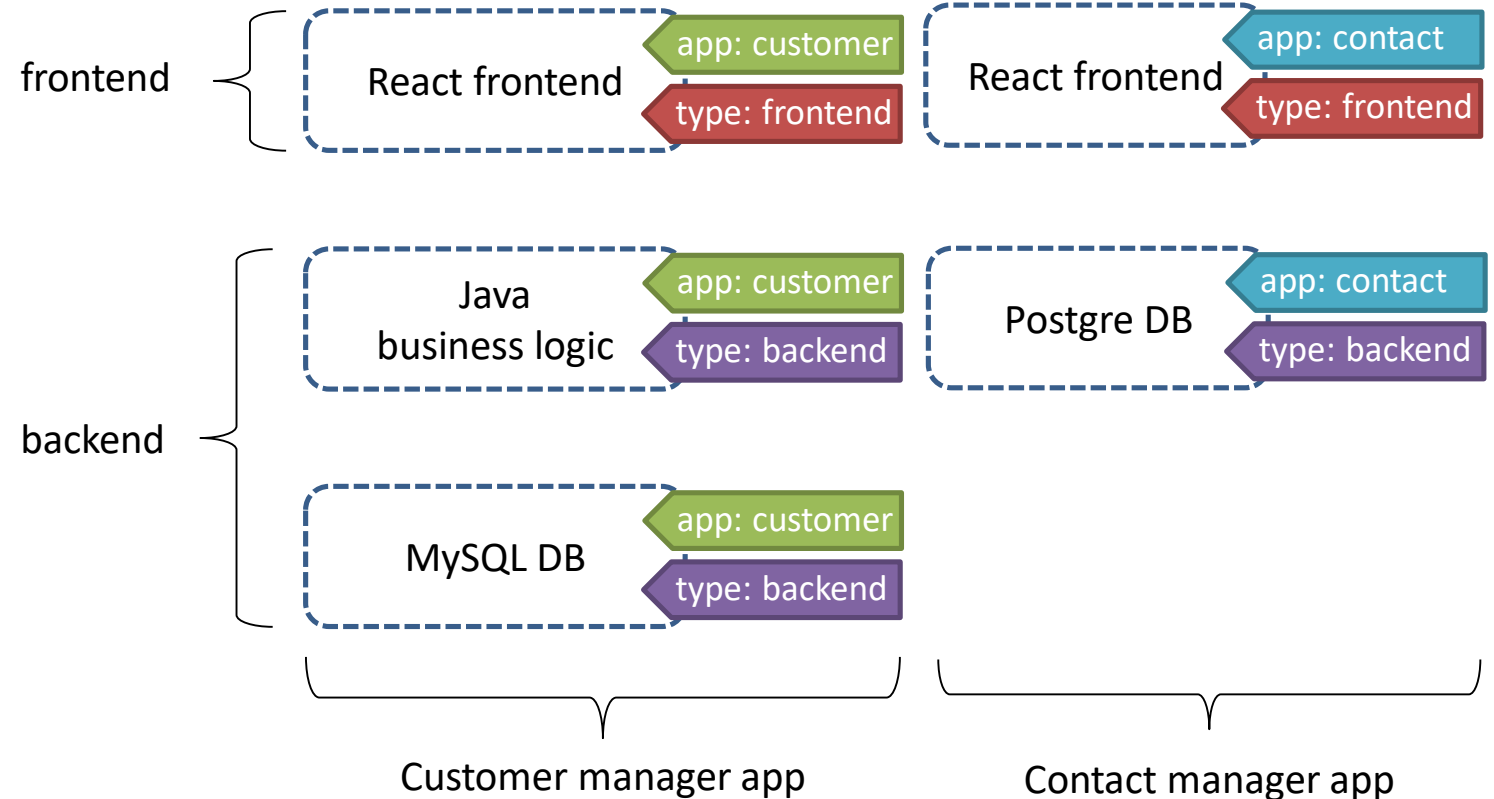
```
spec:
```

```
  containers:
```

```
  - image: nginx:1.17
```

```
    name: www
```

two primitives that can help you
define the concept of an application



Pod: yaml (minimal)

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: www
```

```
  labels:
```

```
    run: www
```

```
  namespace: development
```

```
spec: 
```

- Container images we want to run
- Several fine tuning parameters
- At least these must be specified

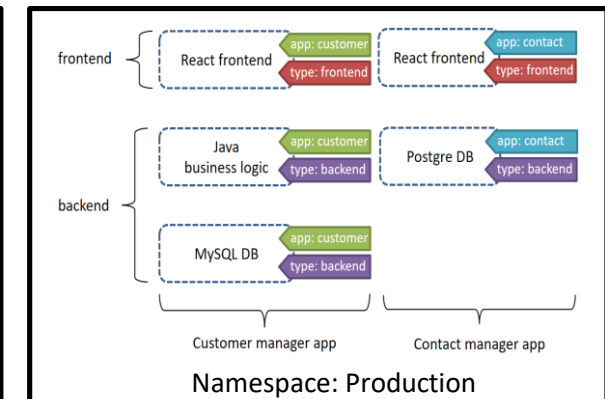
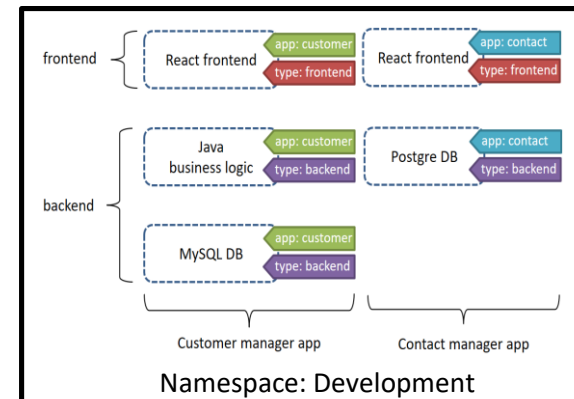
```
  containers:
```

```
  - image: nginx:1.17
```

```
    name: www
```

Namespace

- Each namespace is like a virtual cluster
- Intended for use in environments with many users with multiple teams and projects
- Provide a scope for names:
 - Names of resources need to be unique within a namespace, but not across namespaces
 - Each Kubernetes resource can only be in one namespace
 - A way to divide cluster resources between multiple users (via resource quota -> see later).
- Three pre-defined namespaces:
 - default: if not specified in yaml
 - kube-system: for internal kubernetes objects (kube-proxy, scheduler, apiserver, etc.)
 - kube-public: auto usable by all users
- Namespaces in themselves not prevent connectivity:
 - apps can reach each other
- Not all objects have namespaces:
 - Nodes, PersistentVolumes, Namespaces



- Creating a Pod imperatively
 - `kubectl run www --image=nginx:1.16`
 - no config file, intent is in the command
 - simple, no audit trail, cannot reuse it
- Imperative object config
 - `kubectl create -f my-simple-pod.yaml`
 - config file is required, intent is in the command
 - simple, robust (files can be checked into repo), can be reused
- Declaratively
 - `kubectl apply -f my-simple-pod.yaml`
 - config file is all that is required, intent is not in the command
 - most robust (everything in the file and is checked into repo)
 - K8s automatically figures out intents (reconciliation pattern)
 - can specify multiple files or directories recursively

Pod: Multiple containers

0. Pause Containers:

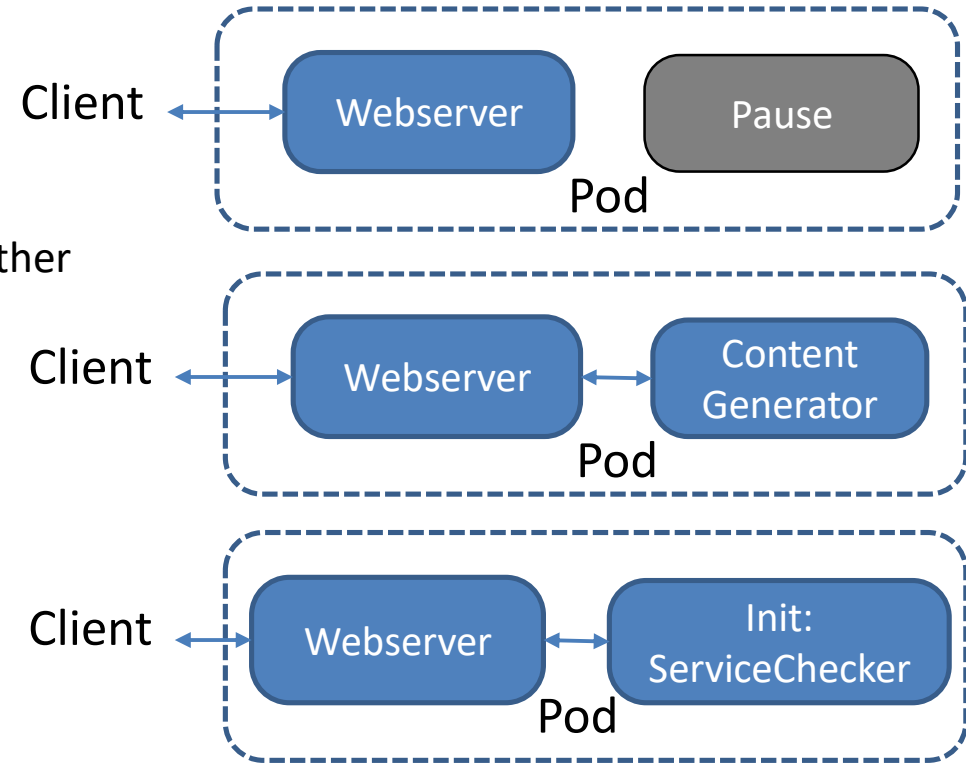
- A container that holds the network namespace for the pod
- Acquire IP address for the Pod and set up the network namespace for all other containers
- Exists in all Pods

1. ApplicationContainers:

- App or regular containers that should be placed very closely to each other

2. InitContainer:

- Run before the app containers are started
- Always run to completion
- Run serially (if there are more)
- Use-cases:
 - Usually have separate images from app containers
 - Utilities or custom code for setup that are not present in an app image
 - Can securely run utilities or custom code that would otherwise make an app container image less secure



Pod: environments that containers see in the Pod

- **Filesystem:**
 - the container image (at root)
 - attached volumes
- **Container:**
 - hostname
- **Pod:**
 - Pod name (via downwardAPI, user-defined environment variables)
- **Services:**
 - list of all services
 - more details later, but for now: kind of a load balancer with a stable IP address in front of a set of Pods

Pod: lifecycle

- Pods are only scheduled once in their lifetime (runs on the Node until stops or terminated)
- If a Node dies, Pods on that Node are scheduled for deletion after a timeout period.
 - In itself won't start again as a Pod is not self-healing (there are higher level resources for this)
- PodStatus.phase (this is printed by `kubectl get pods`):
 - Pending:
 - accepted by K8s but **one or more Containers are not ready** to run
 - waiting for scheduling or downloading images
 - Running:
 - Pod is bound to Node and **all Containers started** and **at least one is running** or in the process of starting/restarting.
 - Succeeded/Failed:
 - Success: All containers in the Pod have terminated in success, and will not be restarted
 - Failure: All containers in the Pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system
 - Unknown: Pod status could not be queried (usually Node communication error)

Higher level abstractions in K8s

- Pods usually not created directly, because:
 - No self-healing
 - No scaling
 - IP address is ephemeral
- The power of K8s comes with the higher level resources / abstractions
 - [ReplicaSet](#)
 - Deployment
 - Service
 - Job
 - CronJob
 - DaemonSet
 - etc.

ReplicaSet

- Pods:
 - Containers inside pod template
- ReplicaSet:
 - Maintaining a number of replicas:
 - Introduces self-healing & scaling
 - If a Pod crashes, starts a new one
 - Which Pods should be governed by the RS?
 - Those whose labels match with the selector
 - ReplicaSet has its own metadata field, do not confuse it with the selector!
 - After applied:
 - ReplicaSet is created
 - ReplicaSet will kick in the creation of the Pods (if it is necessary)
 - It will query Pods with the same labels as it is defined in its selector field.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    tier: frontend
spec:
  containers:
  - image: nginx:1.17
    name: nginx
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      name: frontend
      labels:
        tier: frontend
    spec:
      containers:
      - image: nginx:1.17
        name: nginx
```

- Deleting a ReplicaSet:
 - `kubectl delete` --> Every pod will be deleted that belongs to it (by default)
 - `kubectl delete -cascade=false` --> Pods remain intact
 - these pods are now without self-healing --> in case of crash there is no restart
 - a new ReplicaSet can adopt them
- Isolating Pods from a ReplicaSet:
 - Changing the labels of the Pods to not match with the selector of the RS
 - ReplicaSet will notice and start new Pods to have the required number of replicas
- Scaling:
 - Re-apply the new template
- Autoscaling a ReplicaSet:
 - HorizontalPodAutoscaler
 - Metrics Server to be correctly configured and working on the server
 - On hosted K8s services it is usually installed (On-premise: need to do manually)

Deployment

- Pods:
 - Containers inside pod template
- ReplicaSet:
 - Pod template
 - Number of replicas
 - Self-healing and scaling
- Deployment:
 - Encapsulates the spec of ReplicaSet
 - Versioning
 - Rollback
 - Advanced deployments
 - StrategyType
 - RollingUpdateStrategy

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      name: frontend
      labels:
        tier: frontend
    spec:
      containers:
        - image: nginx
          name: nginx
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
  labels:
    app: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      name: frontend
      labels:
        app: frontend
    spec:
      containers:
        - image: nginx
          name: nginx
```



- To rollout a ReplicaSet (create new pods)
- Update state of existing deployment:
 - Updating pod template: new replicas created and pods moved over in a controlled manner
 - Updating manifest (scaling or metadata):
 - Don't create new version
 - This changes cannot be rolled back easily (with rollback command)
- Rollback to earlier version:
 - Simply go to previous revision
 - One ReplicaSet for each pod template change (revision) --> with rollback only pod template will change!
- Scale up:
 - Edit the number of replicas
 - Does not change the pod template
- Pause/Resume deployments and fix bugs:
 - Imperatively: `kubectl pause/resume`
 - Change `spec.Paused` boolean
- Check status of a deployment (status field)
- Clean up old ReplicaSets that are not needed anymore

} No rollout until resume & cannot be rolled back

Deployment: deployment/rollout strategy

- `spec.strategy`: specifies the strategy used to replace old Pods by new ones
- `spec.strategy.type`: [Recreate, RollingUpdate] --> latter is the default
 - Recreate: All existing Pods are killed before new ones are created
 - RollingUpdate:
 - updates Pods in a rolling update fashion --> change the OLD and NEW ReplicaSet in parallel
 - controlled by values of `maxUnavailable` and `maxSurge`.
 - `maxUnavailable`: maximum number of Pods that can be unavailable during the update process
 - `maxSurge`: maximum number of Pods that can be created over the desired number of Pods

```
Name: nginx-deployment
Namespace: default
CreationTimestamp: Thu, 30 Nov 2017 10:56:25 +0000
Labels: app=nginx
Annotations: deployment.kubernetes.io/revision=2
Selector: app=nginx
Replicas: 3 desired | 3 updated | 3 total | 3 available | 0
unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
```

Deployment: deployment/rollout strategy

Deployments strategies:

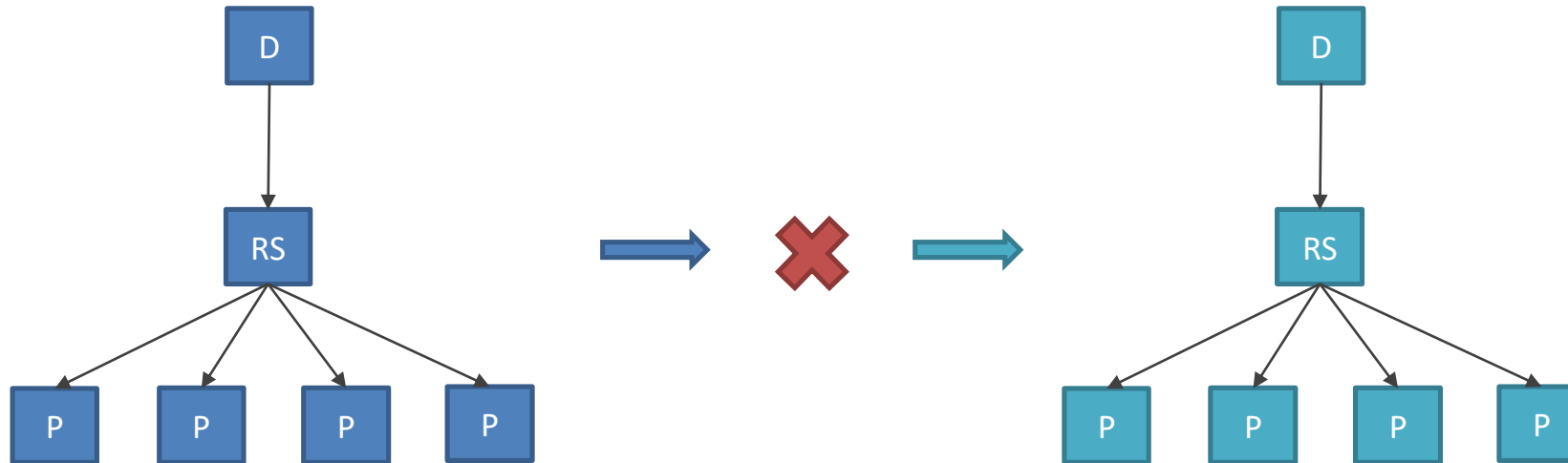
- recreate:

- terminate the old version and release the new one

```
strategy:  
  type: Recreate
```

+ Application state entirely renewed

- Downtime that depends on shutdown and boot duration of the application

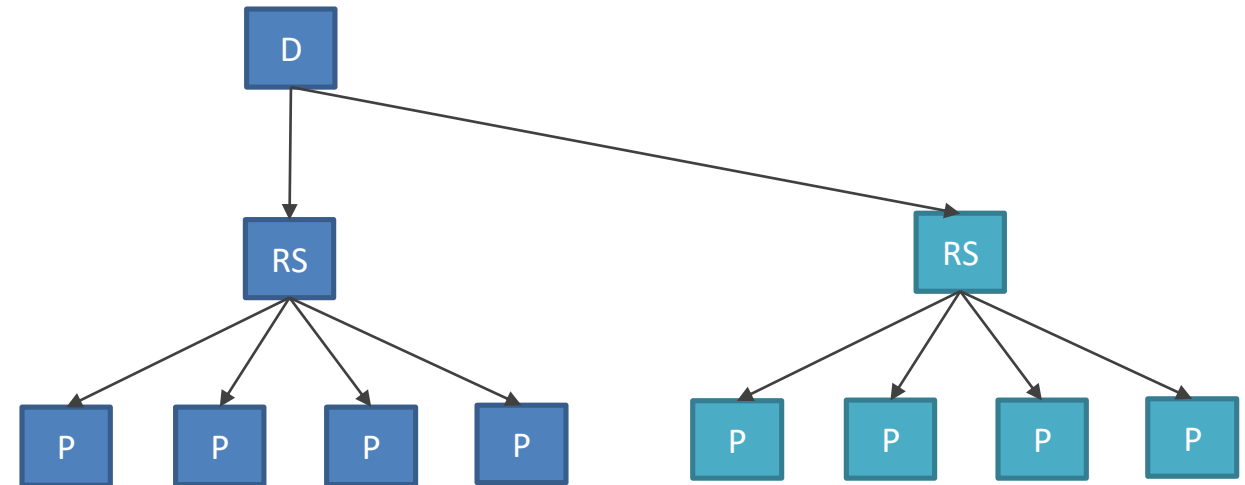


Deployment: deployment/rollout strategy

Deployments strategies:

- ramped:
 - release a new version on a rolling update fashion, one after the other
- + Version is slowly released across instances
- + Convenient for stateful applications that can handle rebalancing of the data
- rollout/rollback can take time
- supporting multiple APIs is hard
- no control over traffic

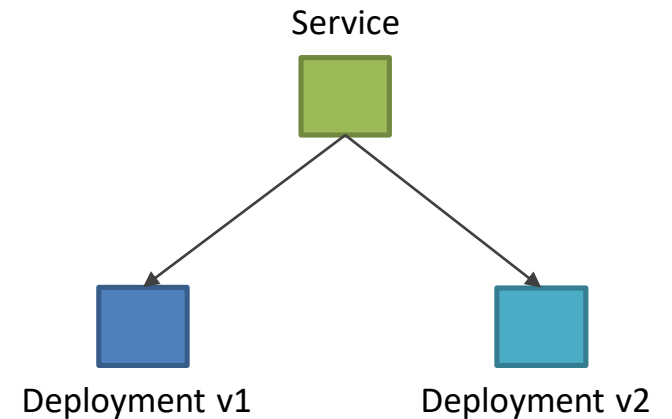
```
strategy:  
  type: RollingUpdate  
  rollingUpdate:  
    maxSurge: 2  
    maxUnavailable: 0
```



Deployment: deployment/rollout strategy

Deployments strategies:

- blue/green:
 - release a new version **alongside** the old version then switch traffic
 - done by creating a completely new deployment and use Service's selector to switch between them
- + instant rollout/rollback
- + avoid versioning issue, change the entire cluster state in one go
- requires double the resources
- proper test of the entire platform should be done before releasing to production
- handling stateful applications can be hard



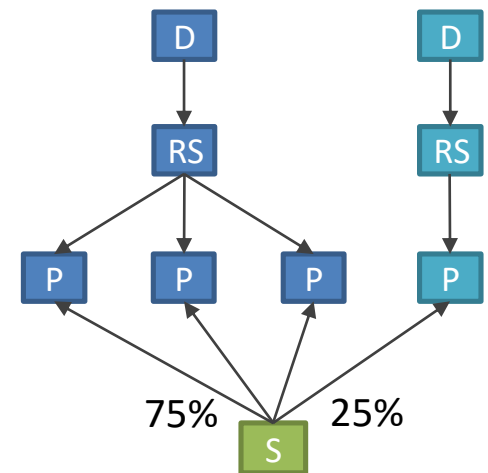
Deployment: deployment/rollout strategy

Deployments strategies:

- canary:
 - Release a new version to a subset of users, then proceed to a full rollout
 - Let client do the testing
 - Two Deployments with common pod labels and with different versions
 - If everything is fine gradually scale down the old version and scale up the new
- + version released for a subset of users
- + convenient for error rate and performance monitoring
- + fast rollback
- slow rollout
- fine tuned traffic distribution can be expensive (99% A/ 1%B = 99 pod A, 1 pod B)

```
spec:  
  replicas: 3  
  template:  
    spec:  
      containers:  
        - image: myapp:1.0
```

```
spec:  
  replicas: 1  
  template:  
    spec:  
      containers:  
        - image: myapp:2.0
```



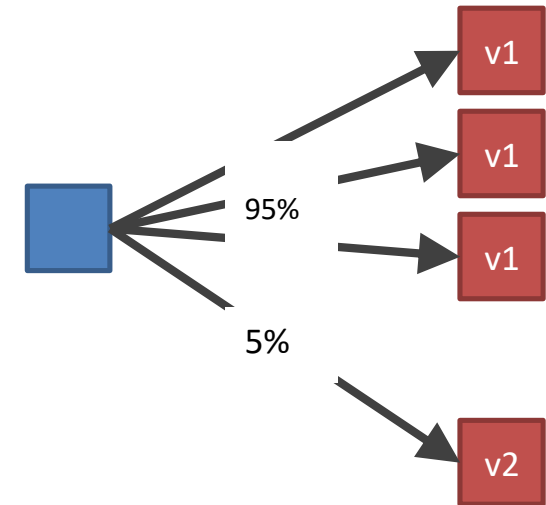
Deployment: deployment/rollout strategy

Deployments strategies:

- canary:
 - Release a new version to a subset of users, then proceed to a full rollout
 - Let client do the testing
 - Two Deployments with common pod labels and with different versions
 - If everything is fine gradually scale down the old version and scale up the new
- + version released for a subset of users
- + convenient for error rate and performance monitoring
- + fast rollback
- slow rollout
- fine tuned traffic distribution can be expensive (99% A/ 1%B = 99 pod A, 1 pod B)
- More advanced use-cases with specialized tools e.g. Istio

```
spec:  
  replicas: 3  
  template:  
    spec:  
      containers:  
      - image: myapp:1.0
```

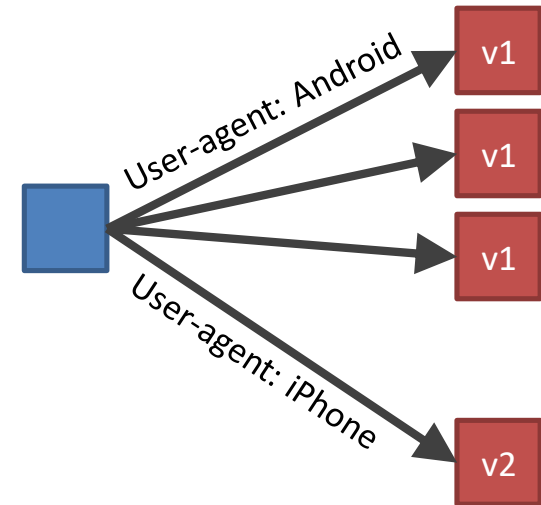
```
spec:  
  replicas: 1  
  template:  
    spec:  
      containers:  
      - image: myapp:2.0
```



Deployment: deployment/rollout strategy

Deployments strategies:

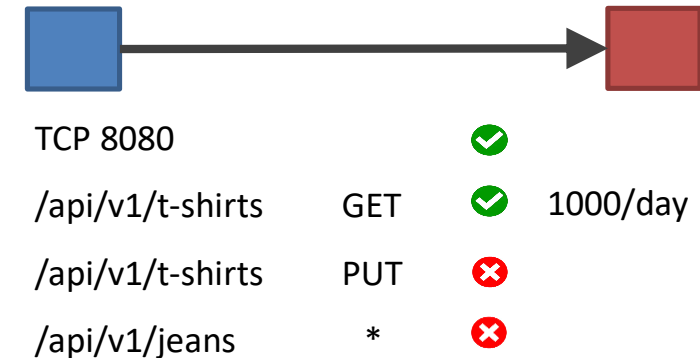
- a/b testing:
 - Release a new version to a subset of users in a precise way (HTTP headers, cookie, weight, etc.).
 - Doesn't come out of the box with Kubernetes
 - Implies extra work to setup a more advanced infrastructure (Istio, Linkerd, Traefik, custom nginx/haproxy)
 - If everything is fine gradually scale down the old version and scale up the new
- + several versions run in parallel
- + full control over the traffic distribution
- requires intelligent load balancer
- hard to troubleshoot errors for a given session, distributed tracing becomes mandatory
- not straightforward, you need to setup additional tools



Deployment: deployment/rollout strategy

Deployments strategies:

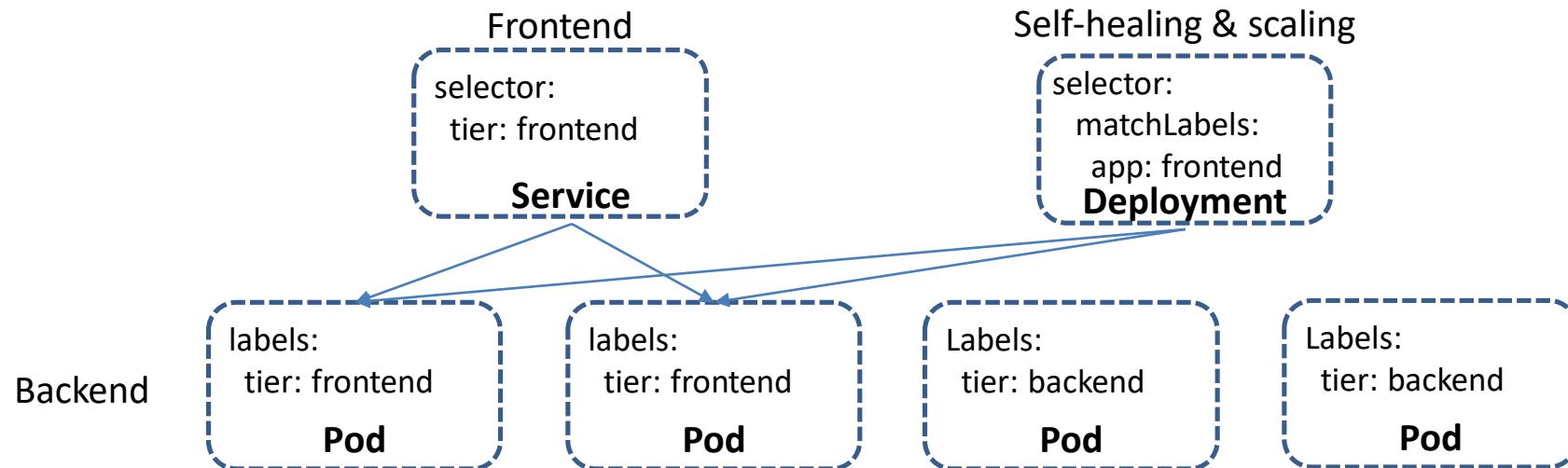
- a/b testing:
 - Release a new version to a subset of users in a precise way (HTTP headers, cookie, weight, etc.).
 - Doesn't come out of the box with Kubernetes
 - Implies extra work to setup a more advanced infrastructure (Istio, Linkerd, Traefik, custom nginx/haproxy)
 - If everything is fine gradually scale down the old version and scale up the new
- + several versions run in parallel
- + full control over the traffic distribution
- requires intelligent load balancer
- hard to troubleshoot errors for a given session, distributed tracing becomes mandatory
- not straightforward, you need to setup additional tools



Higher level abstractions in K8s

- Pods usually not created directly, because:
 - No self-healing
 - No scaling
 - IP address is ephemeral
- The power of K8s comes with the higher level resources / abstractions
 - ReplicaSet
 - Deployment
 - Service
 - Job
 - CronJob
 - DaemonSet
 - etc.

- Pod IP address keeps changing (every time RS or Deployment takes pods up/down)
- Need to find a way to hide this from client so it is enough to know one IP
- Service: logical set of backend Pods + stable front-end
 - Frontend: IP address + Port + DNS name --> independent from lifetime of backend pods
 - Backend: Logical set of pods whose label matches with the selector of the service
- Service ~ Load Balancer: route requests to the backend pods
- Endpoint: a separate object that stores all Pods that are selected by the selector



Service: without selector

- Services usually abstract access to Pods, but can also abstract other kinds of backends:
 - external database cluster in production, but in your test environment you use your own databases
 - point your Service to a Service in a different Namespace or on another cluster
 - during migration only a proportion of backend run in Kubernetes
- In any of these scenarios you can:
 - define a Service *without* a Pod selector (no Endpoint will be created)
 - create your own Endpoint object
- External Name is another option (one of the 4(5) Service types):
 - Redirection happens at the DNS level rather than via proxying or forwarding

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
---
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9376
```

Service: multiport and defined IP address



Multiport:

- When a Service needs to expose more than one port
- In this case must give all of your ports names so that these are unambiguous

Define you own IP address:

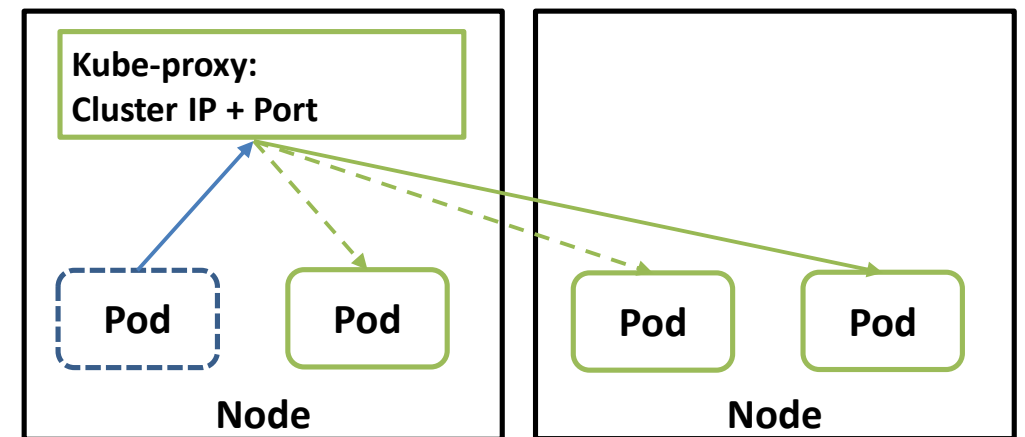
- ClusterIP in `spec.clusterIP` field
 - When:
 - Have an existing DNS entry that you wish to reuse
 - Legacy system that are configured for a specific IP address and difficult to re-configure
 - Must be a valid IPv4 or IPv6 address from within the `service-cluster-ip-range` CIDR range that is configured for the API server.
- ExternalIP in `spec.externalIPs` is also can be set:
 - If there are any that routes to one or more cluster nodes with this IP
 - It will be routed to the service that is defined with (then to its backend Pods)
 - Not managed by Kubernetes, the responsibility of the cluster administrator

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    tier: frontend
  ports:
    - name: http
      protocol: TCP
      port: 8080
      targetPort: 80
    - name: https
      protocol: TCP
      port: 443
      targetPort: 8081
```

Service: types (ClusterIP)

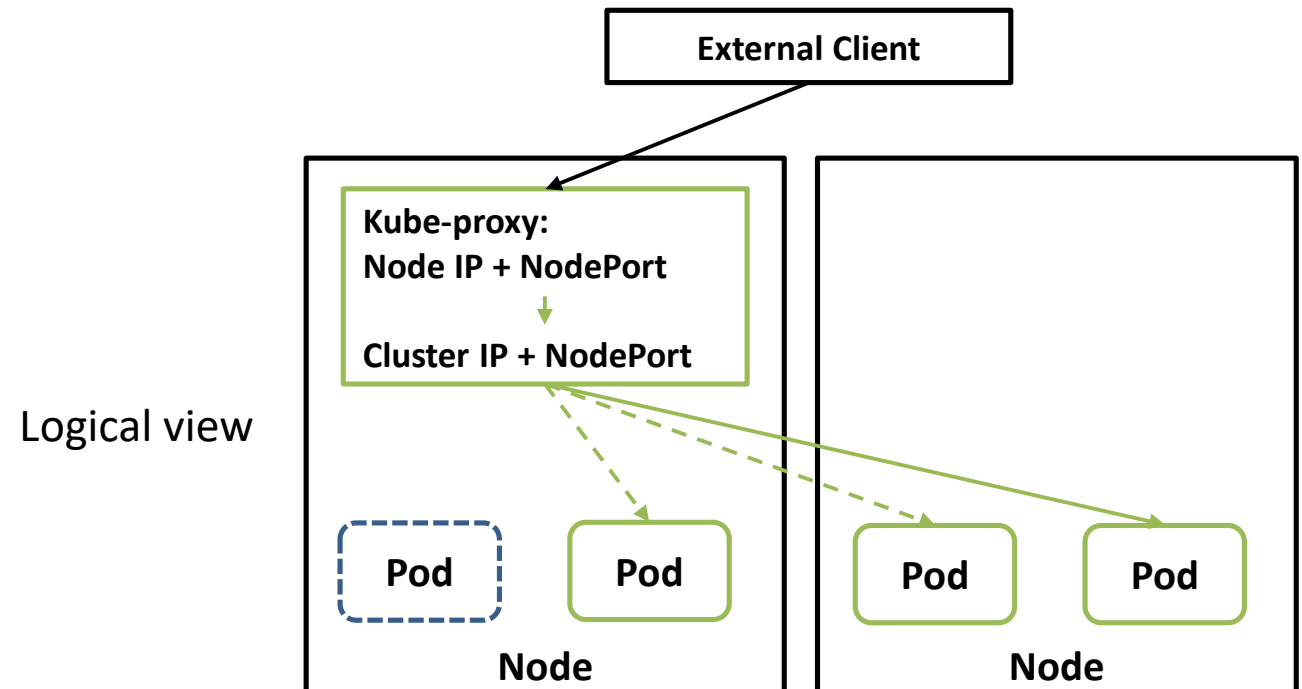
- ClusterIP (default):
 - Exposes the Service on an internal IP in the cluster.
 - ClusterIP is independent from the backend Pod IP addresses.
 - This type makes the Service only reachable from within the cluster.
 - Route: ClusterIP -> Pod.

Logical view



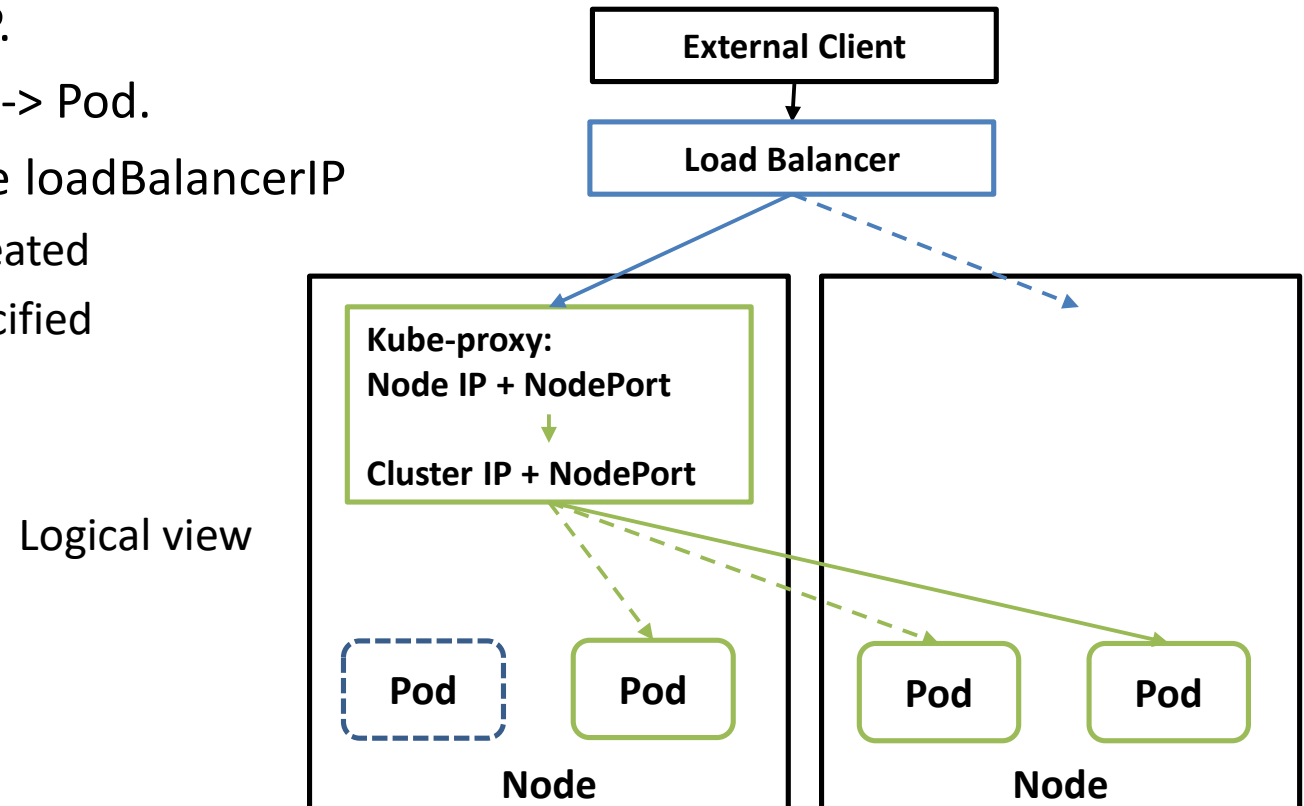
Service: types (NodePort)

- NodePort:
 - Exposes the Service on the same port of each selected Node in the cluster using NAT.
 - Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>.
 - Automatically creates ClusterIP.
 - Request is relayed from <NodeIP>:<NodePort> to <ClusterIP>:<NodePort>.
 - Route: NodePort -> ClusterIP -> Pod.



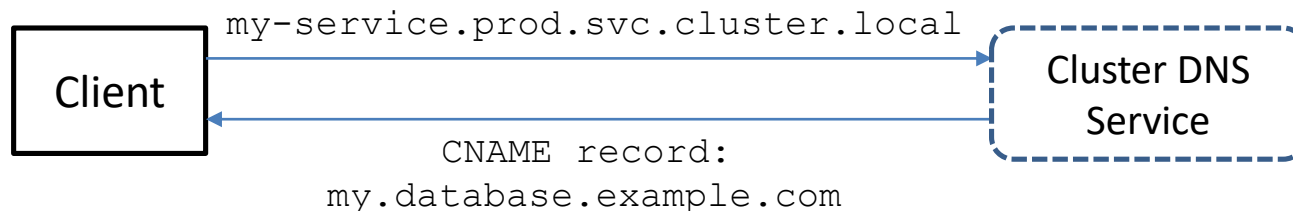
Service: types (LoadBalancer)

- LoadBalancer (Ingress is an alternative, see it later):
 - Exposes the Service externally using a cloud provider's load balancer.
 - Require tight cooperation with the cloud provider's infrastructure (via the cloud-controller-manager)
 - Assigns a fixed, external IP to the Service.
 - Automatically creates NodePort and ClusterIP.
 - Routing: External LB -> NodePort -> ClusterIP -> Pod.
 - Some cloud providers allow you to specify the loadBalancerIP
 - If yes and not specified, an ephemeral IP is created
 - If not then this field is ignored even if it is specified



Service: types (ExternalName)

- ExternalName:
 - Exposes the Service using an arbitrary name (specified in `spec.ExternalName`) by returning a CNAME record with the name.
 - No proxy is used.
 - Redirection happens at the DNS level rather than via proxying or forwarding as with other Services
 - E.g.
 1. Lookup to `my-service.prod.svc.cluster.local`
 2. Cluster DNS Service returns a CNAME record with the value



```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.db.example.com
```


Service: types + 1 (Headless)

- Headless Service:
 - When don't need load-balancing and a single Service IP (e.g. DB replicas with one master + 2 read replicas)
 - In `spec.ClusterIP: None`
 - Cluster IP is not allocated
 - Kube-proxy does not handle these Services
 - There is no load balancing or proxying done by the platform for them
 - With selectors:
 - endpoints controller creates Endpoints records
 - modifies the DNS configuration to return records (addresses) that point directly to the Pods backing the Service
 - Without selectors:
 - endpoints controller does not create Endpoints records
 - Redirection happens at the DNS level rather than via proxying or forwarding as with other Services
 - E.g.
 1. Lookup to `my-service.prod.svc.cluster.local`
 2. Cluster DNS Service returns a CNAME record with the value

Service: discovery (environment variables and DNS)

Environment variables:

- kubelet adds a set of environment variables for each active Service
- E.g. "redis-master" which exposes TCP port 6379 and allocated cluster IP address 10.0.0.11

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11  
REDIS_MASTER_SERVICE_PORT=6379  
REDIS_MASTER_PORT=tcp://10.0.0.11:6379  
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379  
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp  
REDIS_MASTER_PORT_6379_TCP_PORT=6379  
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

- If the pod would rely on this method, the service needs to be created first!

Service: discovery (environment variables and DNS)

DNS:

1. For Services (A/AAAA and SRV records):

- A DNS Service (e.g. CoreDNS) watches the API for new Services and creates records.
- Pods should be able to automatically resolve Services by their DNS names.
- For Service names (A records):
 - Service name: `my-service`, Namespace: `my-namespace`
 - Service is available
 - for pods in the same namespace: `my-service`
 - for pods in other namespaces: `my-service.my-namespace`
 - » More precisely: `my-service.my-namespace.svc.cluster.local`
 - Returns the ClusterIP
- For named ports (DNS SRV records):
 - `port-name.port-protocol.my-service.my-namespace.svc.cluster.local`
 - Port number is returned

DNS:

2. For Pods (A/AAAA records):

- `pod-ip-address.my-namespace.pod.cluster-domain.example`
- E.g.:
 - Pod namespace: `my-ns`, IP: `172.17.0.3`, cluster domain name: `cluster.local`
 - `172-17-0-3.my-ns.pod.cluster.local`
 - If the Pod is created with Deployment or DaemonSet (name: `my-d`) and exposed by a Service:
 - `172-17-0-3.my-d.my-ns.svc.cluster.local`
 - If `spec.hostname: foo` and `spec.subdomain: bar`:
 - `foo.bar.my-ns.svc.cluster.local`
 - Hostname is `metadata.name` by default but `spec.hostname` overwrites it.

Service: discovery (environment variables and DNS)

DNS:

2. For Pods (A/AAAA records):

- DNS policies can be set on a per-pod basis (so how the Pod can access the rest of the world):
- In `spec.dnsPolicy` field:
 - Default:
 - inherits the name resolution configuration from the node that the pods run on.
 - ClusterFirst:
 - DNS queries that doesn't match the configured cluster domain suffix, such as "www.kubernetes.io", is forwarded to the upstream nameserver inherited from the node.
 - ClusterFirstWithHostNet:
 - for Pods running with `hostNetwork`, you should explicitly set this DNS policy.
 - None:
 - ignore DNS settings from the Kubernetes environment. All DNS settings are supposed to be provided using the `dnsConfig` field in the Pod Spec.

Ingress

- A LoadBalancer type Service alternative
- Separate API Object (`kind: Ingress`)
- Manages external access to the Services in a cluster, typically HTTP.
 - A LoadBalancer Service does it only at the TCP level
- May provide load balancing, SSL termination and name-based virtual hosting.
- Must have an ingress controller to satisfy an Ingress:
 - There are a number of which you can choose (e.g. ingress-nginx)
 - All fit the reference specification, but in reality there are some differences
- Ingress rules:
 - Optional Host (if not specified, rule applied to all inbound HTTP)
 - List of paths (e.g. /testpath) and their associated backends
 - Backend: Service name and port
- Several Ingress Types:
 - Single Service Ingress:
 - specifying a *default backend* with no rules
 - Simple fanout:
 - single IP address to more than one Service
 - Name based virtual hosting:
 - routing HTTP traffic to multiple host names at the same IP address

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          serviceName: test
          servicePort: 80
```

Higher level abstractions in K8s

- Pods usually not created directly, because:
 - No self-healing
 - No scaling
 - IP address is ephemeral
- The power of K8s comes with the higher level resources / abstractions
 - ReplicaSet
 - Deployment
 - Service
 - DaemonSet
 - Job
 - CronJob
 - etc.

- A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod.
- As nodes are added to the cluster, Pods are added to them.
- As nodes are removed from the cluster, those Pods are garbage collected.
- Deleting a DaemonSet will clean up the Pods it created.
- Some typical use-cases:
 - cluster storage daemon on every node
 - logs collection daemon on every node
 - node monitoring daemon on every node
- YAML structure:
 - apiVersion, kind, metadata, spec.template -> Pod schema
 - spec.selector: works exactly as for a ReplicationController
- A Pod's `RestartPolicy`: `Always`
 - or be unspecified, which defaults to `Always`

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    app: fluentd
    name: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
        - image: fluentd:v2.5.2
          name: fluentd
```


- Init scripts:
 - Run daemon processes by directly starting them on a node
 - Disadvantages:
 - No monitoring and log management out of the box
 - Separate tool for managing daemons and application (instead of using only kubectl and yaml templates)
 - A DaemonSet is automatically runs in a container which is more isolated
- Bare Pods:
 - Possible to specify a particular node to run on.
 - Disadvantage:
 - Loose self-healing
 - Cannot react on dynamic Node addition
- Static Pods:
 - Create Pods by writing a file to a certain directory watched by Kubelet.
 - Cannot managed by kubectl or other K8s API clients (so could be used for cluster bootstrapping cases)
 - May become deprecated in the future
- Deployments
 - Similar concept (auto healing and pod distribution)
 - Cannot guarantee a copy of Pod always run on all or certain hosts (if there is a dynamic Node addition)

Higher level abstractions in K8s

- Pods usually not created directly, because:
 - No self-healing
 - No scaling
 - IP address is ephemeral
- The power of K8s comes with the higher level resources / abstractions
 - ReplicaSet
 - Deployment
 - Service
 - DaemonSet
 - Job
 - CronJob
 - etc.

Job

- Creates Pods and ensures that a specified number successfully terminate.
- Tracks the successful completions.
- When the specified number is reached the Job is complete.
- Deleting a Job will clean up the Pods it created.
- YAML structure:
 - apiVersion, kind, metadata, spec.template -> Pod schema
 - spec.spec.restartPolicy: [Never, OnFailure, Always]

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl"]
        args:
        - -Mbignum=bpi
        - -wle
        - print bpi(2000)
      restartPolicy: Never
```

1. Non-parallel Jobs:

- normally, only one Pod is started, unless the Pod fails.
- the Job is complete as soon as its Pod terminates successfully

2. Parallel Jobs with a *fixed completion count*:

- `spec.completions` and `spec.parallelism` defines
- multiple pods runs at the same time
- completed after completion count reaches the defined value

3. Parallel Jobs with a *work queue*:

- `spec.completions` is unset
- Pods must coordinate amongst themselves or an external service
- each Pod is capable of determining whether or not all its peers are done

Other important parameters:

- `spec.activeDeadlineSeconds`: shuts down job after exceeded
- `spec.ttlSecondsAfterFinished`: job is deleted after exceeded
- `Spec.backoffLimit`: maximum number of retries (restarts) on failure

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 2
  completions: 6
  activeDeadlineSeconds: 100
  ttlSecondsAfterFinished: 10
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl"]
        args:
        - -Mbignum=bpi
        - -wle
        - print bpi(2000)
      restartPolicy: Never
  backoffLimit: 4
```

Higher level abstractions in K8s

- Pods usually not created directly, because:
 - No self-healing
 - No scaling
 - IP address is ephemeral
- The power of K8s comes with the higher level resources / abstractions
 - ReplicaSet
 - Deployment
 - Service
 - DaemonSet
 - Job
 - CronJob
 - etc.

- A CronJob creates Jobs on a repeating schedule.
- It runs a job periodically on a given schedule, written in Cron format.
- Use-cases: creating periodic and recurring tasks, like
 - running backups
 - sending emails
 - schedule tasks (a Job) for when your cluster is likely to be idle
- YAML structure:
 - apiVersion, kind, metadata, spec.jobTemplate, spec.schedule
- Limitations:
 - creates a job object **about** once per execution time of its schedule
 - About: two jobs or none in some circumstances
 - Therefore, jobs should be **idempotent**.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello!
          restartPolicy: OnFailure
```