Kubernetes Design Patterns

SoC Summer Workshop
Cloud Computing with Big Data

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

Roadmap

- Behavioral Patterns
  - Stateful service -- StatefulSet
  - Self-awareness -- Downward API
- Structural Patterns
  - Init containers
  - Sidecar containers
- Lifecycle Patterns
  - Health probes
  - Managed lifecycle

Servers in the DevOps World

Pets

- nonfungible servers
  - every instance is unique
- require individual care
  - repair
  - vertical scaling
- stateful, persistent and permanent

Cattle

- identical servers
  - all instances are the same
- not need individual care
  - replaced
  - horizontal scaling
- stateless, ephemeral, and transient

StatefulSet (STS)

- Distributed stateful apps require
  - persistent storage, identity, networking, and ordinality
  - every instance is unique & has long-lived characteristics
  - e.g., big data frameworks such as Map-Reduce
- Solution: `StatefulSets` provides
  - stable, unique network identifiers: each Pod in a STS gets a unique hostname based on its ordinal index.
  - stable, persistent storage: each Pod can be associated with a PersistentVolume.
  - ordered, automated rolling updates: STS manages the deployment and scaling in an ordered & deterministic fashion

StatefulSet - how to use?

```yaml
apiVersion: v1
kind: Service
metadata:
  name: headless-service
spec:
  clusterIP: None
  selector:
    app: server
  ports:
  - port: 80
```

- Step 1: create a headless service
  - a `ClusterIP` Service without a virtual IP
- Usage case:
  - direct access to the individual pods without load balancing
- How does it work?
  - `headless-service.default.svc.cluster.local` will resolve to multiple IPs, one for each Pod.
  - `Pod-name.headless-service.default.svc.cluster.local` will resolve to the specific Pod's IP.

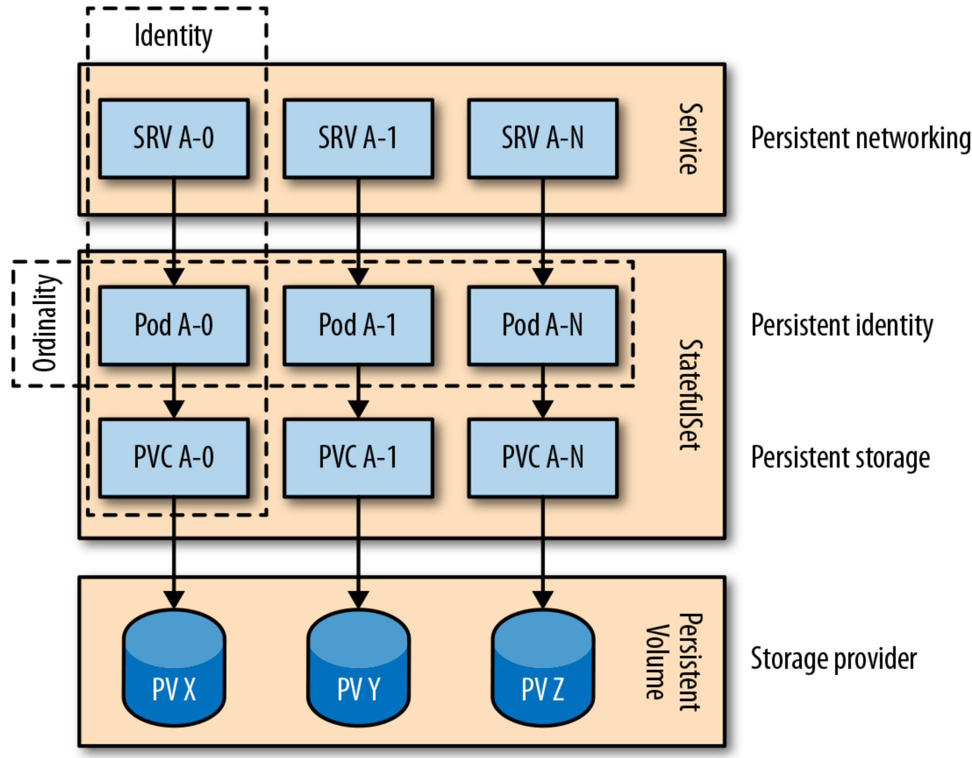
StatefulSet - how to use?

- Step 2: create a StatefulSet
  - `serviceName` matches headless service
- `volumeClaimTemplates` mechanism
  - specifies storage requirements
  - creates PVCs on the fly during
  - allows each Pod to get its own dedicated PVC during pod creation
- In contrast, Deployment & ReplicaSet
  - use a predefined PVC, suited for using `ReadOnlyMany` or `ReadWriteMany` volumes mounted on multiple replicas
  - not suited for `ReadWriteOnce` volumes

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: server-statefulset
spec:
  selector:
    matchLabels:
      app: server
  serviceName: "headless-service"
  replicas: 3
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
      - name: server-container
        image: yancanmao/server-image
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/server
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

StatefulSet - Characteristics

- STS does not manage PV
  - but manages PVCs
  - scaling up creates new Pods and associated PVCs.
  - scaling down deletes the Pods, but it does not delete any PVCs (nor PVs)
- K8s cannot free the claimed/used PV storage
  - manual deletion is needed
  - a system behavior by design

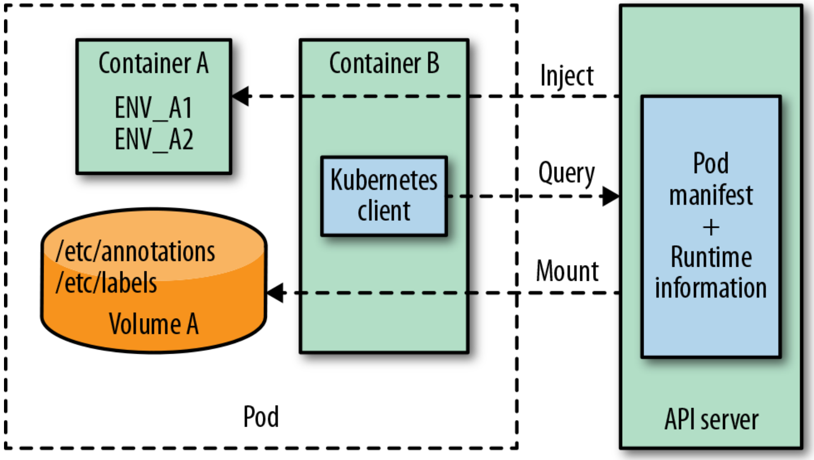


The need of self-awareness

- Scenario: apps may need to have info about themselves and their running environment
  - runtime info: Pod's name & IP, Node's hostname
  - static info: specific resource requests & limits
  - dynamic info: annotations and labels
- Use cases:
  - log information, send metrics to a central server.
  - tune thread-pool size, GC algorithm or memory allocation based on resource limits
  - discover other pods and interact with them
- Solution: Downward API
  - allows passing metadata about the Pod to the containers and the cluster through environment variables and files

Downward API - how does it work?

- Same mechanisms for passing data from ConfigMaps
  - data is not created by developers
  - specify the keys that interests us, and K8s populates the values dynamically
- Main advantage:
  - metadata is injected into Pod and made available locally
  - no need to use a client and interact with the API server
  - nonintrusive introspection and metadata injection, remain K8s-agnostic



Downward API - how to use?

- Import as environment variables
- fieldPath:fieldPath option:
  - POD_NAME, POD_NAMESPACE, POD_IP, and NODE_NAME environment variables are set using the Downward API.
- ResourceFieldRef option:
  - CPU_LIMIT and MEMORY_LIMIT are set using container resource limits.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: downwardapi-env-pod
spec:
  containers:
  - name: nginx
    image: nginx
    env:
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: CPU_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: nginx
          resource: limits.cpu
          divisor: 1m
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: nginx
          resource: limits.memory
          divisor: 1Mi
```

Downward API - how to use?

- Import as a volume
  - downwardAPI type of volume
  - all information written into files
  - all the labels and annotations retrieved as files, not for EnvVar
- Available information:

https://kubernetes.io/docs/concepts/workloads/pods/downward-api/

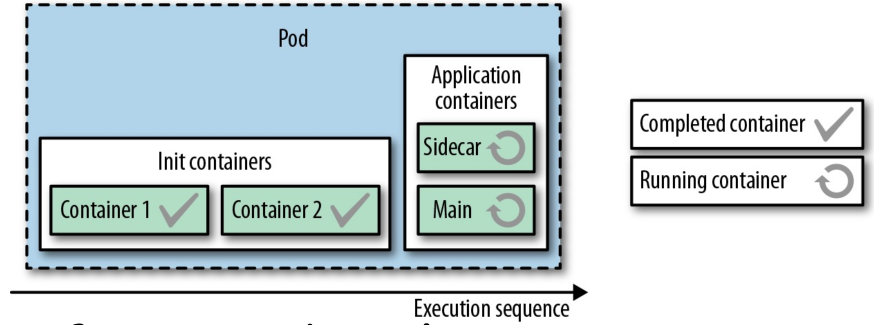
- Limitations of downward API:
  - limited info, some can only be accessed by one method

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: downwardapi-volume-pod
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: downward-api-volume
      mountPath: /etc/downward
  volumes:
  - name: downward-api-volume
    downwardAPI:
      items:
      - path: labels
        fieldRef:
          fieldPath: metadata.labels
      - path: annotations
        fieldRef:
          fieldPath: metadata.annotations
      - path: cpu_limit
        resourceFieldRef:
          containerName: nginx
          resource: limits.cpu
          divisor: 1m
      - path: memory_limit
        resourceFieldRef:
          containerName: nginx
          resource: limits.memory
          divisor: 1Mi
```

Roadmap

- Behavioral Patterns
  - Stateful service -- StatefulSet
  - Self-awareness -- Downward API
- Structural Patterns
  - Init containers
  - Sidecar containers
- Lifecycle Patterns
  - Health probes
  - Managed lifecycle

Init Containers



- Initialization is common in software development
  - e.g., constructors in OOP: executed only once at the beginning of the creation of a class instance
  - how could we appropriately initialize Pods?
- Solution: use **init containers** in the Pod definition
  - two types of containers: init and application containers
  - init containers are executed in a sequence, one by one
  - all of them have to terminate successfully before the application containers are started up
  - if an init container fails, the whole Pod is restarted, causing all init containers to run again; thus, to prevent side effects, init containers need to be idempotent

Init Containers - how to use?

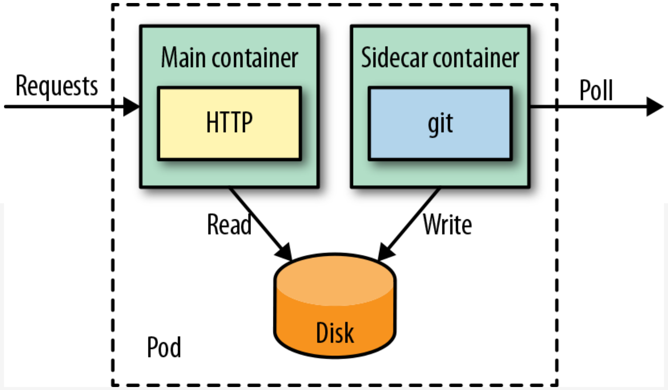
- Enable separation of concerns
  - app engineers focus on app logic
  - develop engineers focus on configuration and initialization
- Init containers are typically small, run quickly, and complete successfully
  - except when used to delay the start of a Pod while waiting for a dependency
- Init containers have separate environments, but can share volumes with app containers

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    app: www
spec:
  initContainers:
  - name: download
    image: bitnami/git
    # Clone an HTML page to be served
    command:
    - git
    - clone
    - https://github.com/mdn/beginner-html-site-scripted
    - /var/lib/data
    # Shared volume with main container
    volumeMounts:
    - mountPath: /var/lib/data
      name: source
  containers:
  # Simple static HTTP server for serving these pages
  - name: run
    image: docker.io/centos/httpd
    ports:
    - containerPort: 80
    # Shared volume with main container
    volumeMounts:
    - mountPath: /var/www/html
      name: source
  volumes:
  - emptyDir: {}
    name: source
```

Sidecar containers -- motivations

- A Pod can run multiple containers. How should we organize the application logics into containers?
  - think about a natural boundary for a unit of functionality with a distinct runtime, release cycle, API, develop team.
- A proper container behaves like a single process
  - solves one problem and does it well
  - is created with the idea of replaceability and reuse
- Single-purpose reusable containers require
  - ways of extending the functionality of a container
  - a means for collaboration among containers.

Sidecar container example



- an HTTP server and a Git synchronizer
  - the HTTP server focuses only on serving files over HTTP and does not know how and where the files are coming from.
  - the Git synchronizer's only goal is to sync data from a Git server to the local filesystem and does not care what happens to the files once synced.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  # Main container is a stock httpd serving from /var/www/html
  - name: app
    image: centos/httpd
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /var/www/html
      name: source
  # Sidecar poll every minute a given repository with git
  - name: poll
    image: bitnami/git
    env:
    - name: SOURCE_REPO
      value: https://github.com/mdn/beginner-html-site-scripted
    command: [ "sh", "-c" ]
    args:
    - |
      git clone $(SOURCE_REPO) .
      while true; do
        sleep 60
        git pull
      done
    workingDir: /var/lib/data
    volumeMounts:
    - mountPath: /var/lib/data
      name: source
  volumes:
  # The shared directory for holding the files
  - emptyDir: {}
    name: source
```

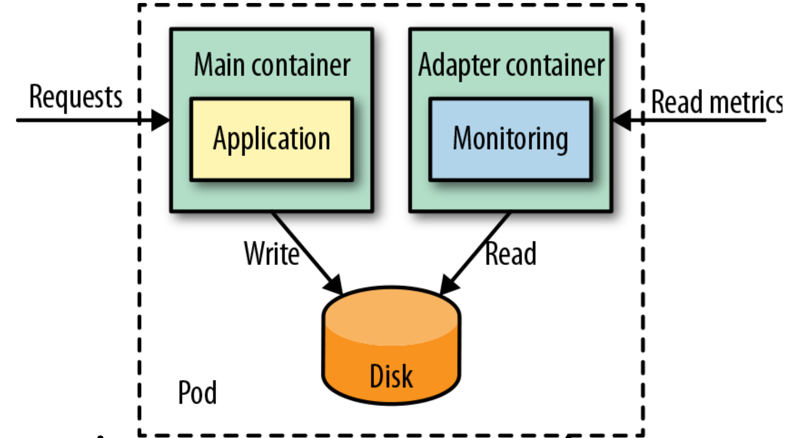
Sidecar containers – use cases

1. **Data Synchronization**: Syncing data between the main container and an external storage or database.
2. **Logging & Monitoring**: Collecting logs and metrics from the main container and sending them to external logging or monitoring systems.
3. **Service Mesh Proxies**: Injecting sidecar proxies (e.g., Envoy in Istio) to manage network traffic and enhance security, observability, and reliability.
4. **Configuration & Secrets Management**: Updating configs or secrets dynamically without restarting the main container.
5. **Security**: Providing security features like authentication, encryption, or token refresh.

Sidecars pattens: Adapter and Ambassador

- Scenario 1: you application is complicated: heterogeneous components in system from multiple teams using different technologies
  - how to utilize external service for all? it boils down to:
  - how to makes it conform to a consistent, unified interface with a standardized and normalized format that can be consumed by the outside world?
- Scenario 2: external services are complicated: heterogeneous external alternative choices, e.g., memory cache and database storage
  - how to hide complexity and providing a unified interface for accessing services outside the Pod?

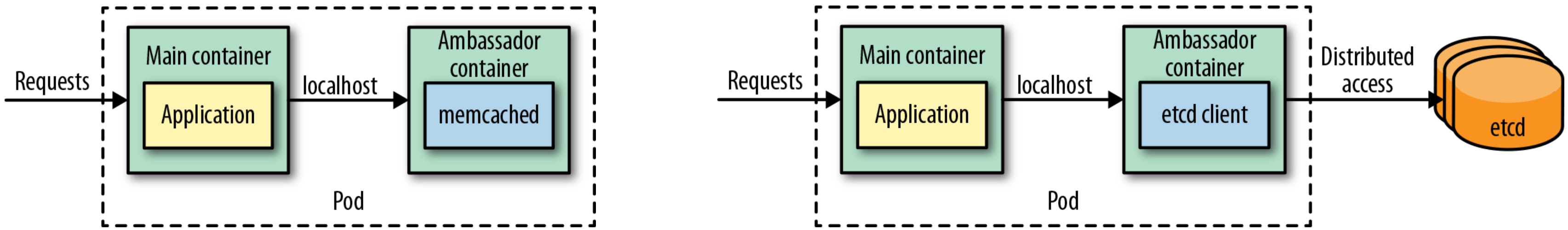
Adapter pattern



- example: when leveraging external monitoring tools
  - distributed software components written in different languages may not have the same capabilities
  - they also may not expose metrics in the same format expected by the monitoring tool

  - Solution: every service represented by a Pod, in addition to the main application container, would have another container that knows how to read the custom application-specific metrics and expose them in a generic format understandable by the monitoring tool.

Ambassador pattern



- example: when leveraging heterogeneous storage
  - an application needs to store data using different types of external storage based on different conditions
  - e.g., local memory and remote data store

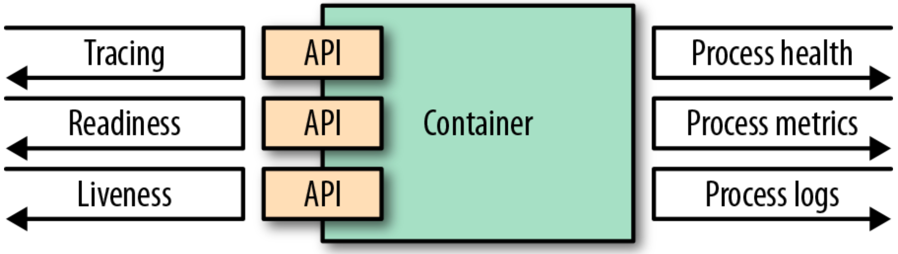
  - Solution: a sidecar container acts as a proxy and decouples the main Pod from directly accessing external dependencies.

Roadmap

- Behavioral Patterns
  - Stateful service -- StatefulSet
  - Self-awareness -- Downward API
- Structural Patterns
  - Init containers
  - Sidecar containers
- Lifecycle Patterns
  - Health probes
  - Managed lifecycle

Health probes

- How can an application communicate its health state to Kubernetes?
  - some container observability options are as follows:



- `kubelet` constantly performs a process health check on the container processes
  - container is restarted if its processes are not running
  - however, not sufficient to decide about the health of an app; e.g., a Java app may throw an OutOfMemoryError and still have the JVM process running.

Liveness probes

- Checks performed by `kubelet`
  - ask container to confirm its healthiness
  - have the checks probed from outside
- Flexible methods:
  - HTTP GET request to the container IP address and expects a successful HTTP response code between 200 and 399.
  - A TCP Socket probe assumes a successful TCP connection.
  - an Exec probe executes an arbitrary command in the container kernel namespace and expects a successful exit code (0).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: livenessprobe-pod
spec:
  containers:
    - image: yancanmao/server-image
      name: server-container
      livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

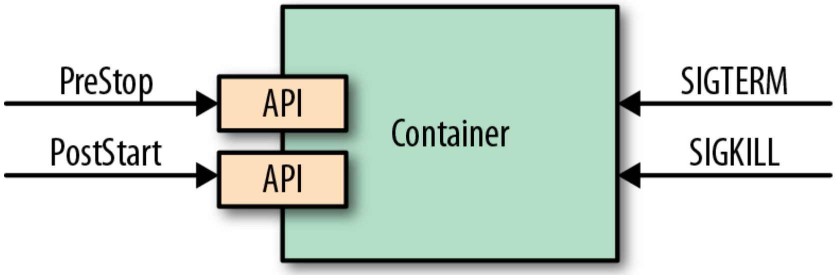
Readiness probes

```yaml
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

- Scenario: although a container is not healthy, restarting it may not help either, for example
  - when a container is still starting up and not ready to handle any requests yet
  - or a container is overloaded, and its latency is increasing, better shield it from additional load for a while.
- Different corrective action from liveness probe
  - a failed probe causes the container to be removed from the service endpoint and not receive any new traffic.
  - process health and liveness checks are intended to recover from failure by restarting; readiness check buys time for your app and expects it to recover by itself

Managed Lifecycle

- Containerized apps managed by cloud-native platforms have no control over their lifecycle
  - listen to the events emitted by the managing platform and adapt their lifecycles accordingly.
  - however, health-check APIs are read-only endpoints the platform uses to extract info from the app.
- Container can listen to and react to the following events emitted by the platform if desired:



SIGTERM and SIGKILL signals

- When K8s decides to shut down a container
  - the container receives a `SIGTERM` signal
  - the app should shut down as quickly as possible: some apps may have to complete in-flight requests, release open connections, and clean up temp files, which can take a longer time.
- After a grace period, if not shut down yet
  - the container receives a `SIGKILL` signal
  - configurable grace period at pod-level

  `.spec.terminationGracePeriodSeconds`
  - the container will be shut down forcefully
- It is up to the app to respond to signals in its logic.

```yaml
spec:
  containers:
  - image: yancanmao/server-image
    name: server-container
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postStart hook > /var/log/poststart.log"]
      preStop:
        exec:
          command: ["/bin/sh", "-c", "echo Goodbye from the preStop hook > /var/log/prestop.log"]
```

postStart & preStop hooks

- Lifecycle hooks provided by K8s,
  - invocation mechanisms: httpGet and exec
  - both `postStart` and `preStop` are blocking calls
- The `postStart` command is executed after a container is created, asynchronously with the container's process
  - the container status remains `Waiting` until the `postStart` handler completes, which in turn keeps the Pod status in the `Pending` state
  - can be used to delay the startup state of the container while giving time to the main container process to initialize, or prevent a container from starting when the Pod does not fulfill certain preconditions
- The `preStop` action must complete before the call to delete the container is sent to the container runtime,
  - a convenient alternative to a `SIGTERM` signal