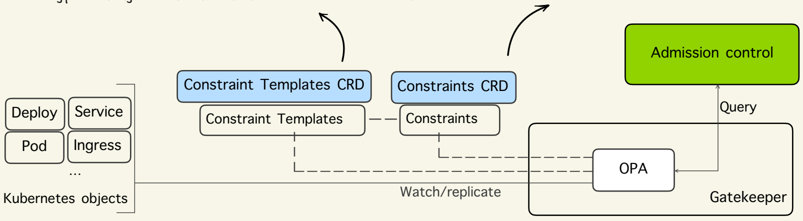
Gatekeeper

Kubernetes provides admission controller webhooks as a mechanism to decouple policy decisions from the API server. These webhooks intercept admission requests before they are persisted as objects in k8s, allowing custom logic and policies to be enforced. Gatekeeper was specifically designed to facilitate customizable admission control through configuration, rather than requiring code changes. It brings awareness to the overall state of the cluster, going beyond evaluating a single object during admission. Gatekeeper integrates with Kubernetes as a customizable admission webhook. It leverages the Open Policy Agent (OPA), which is a policy engine hosted by the Cloud Native Computing Foundation (CNCF), to execute policies in cloud-native environments.

Constraint Templates are Kubernetes Custom Resource Definitions (CRDs) that define a set of constraints or policies that can be applied to Kubernetes objects. They act as a template or blueprint for creating individual Constraints. A Constraint Template defines the structure, parameters, and validation rules for a specific type of constraint that can be applied to Kubernetes resources.Constraint Templates allow you to define reusable policies that can be applied to multiple resources across your cluster. They provide a way to centralize and standardize the enforcement of constraints

Constraints are instances of Constraint Templates. They are created based on the defined template and applied to specific Kubernetes resources. Constraints enforce policies by validating the resources against the defined rules and conditions in the Constraint Template. If a resource violates any of the defined constraints, it is considered non-compliant



When a user tries to create/update a resource in the cluster, the request first goes to the gatekeeper (as an admission webhook). Gatekeeper checks if the resource satisfies all the defined constraints and rejects the request if any policy is violated

Enforcing Resource Limits and Requests for Pods using Gatekeeper

To enforce a policy where all Pods must have resource limits and requests set using Gatekeeper, you would create a ConstraintTemplate and then a Constraint using that template. Here's how you can do it:

1. Create a ConstraintTemplate, which defines the schema and the Rego logic for the policy. The ConstraintTemplate specifies that the Pods must have resource limits and requests

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredresources
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredResources
      validation:
        openAPIV3Schema:
          properties:
            resources:
              type: array
              items: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredresources

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not container.resources.limits.memory
          msg := sprintf("container <%v> has no memory limit", [container.name])
        }

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not container.resources.limits.cpu
          msg := sprintf("container <%v> has no CPU limit", [container.name])
        }
```

2. Create a Constraint based on the ConstraintTemplate you defined. The Constraint specifies the name and the kind of resources to which the policy applies

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredResources
metadata:
  name: pod-must-have-limits
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

3. After applying the Constraint, any new Pods that do not have resource limits and requests will be rejected by the Gatekeeper admission webhook. Existing pods will not be affected by this policy

this ConstraintTemplate is defining a constraint that requires all containers in Kubernetes resources to have resource limits defined. If any container violates this constraint, Gatekeeper will prevent the resource from being created or modified.

The first violation rule checks whether a container in the input resource's specification (spec) has defined memory resource limits. If there are no memory resource limits defined, it generates a violation with a message indicating that the container lacks memory resource limits.

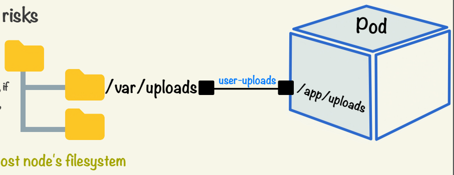
Storage

In Kubernetes, containers are typically considered to be ephemeral and immutable, meaning that they are designed to be short-lived and replaceable. This approach is well-suited for stateless applications that don't store or modify persistent data, but it can be challenging for stateful applications that require persistent storage.
To address this challenge, Kubernetes provides various ways to persist data, ranging from simple to complex solutions. Here are some of the approaches to persist data in k8s.

HostPath volumes:

HostPath volumes allow you to mount a directory from the host node's filesystem into a Pod. This approach is useful for testing and development purposes, but it is not recommended for production environments as it can create security risks

One important thing to note about HostPath volumes is that they are only accessible from the node where the pod is running. This means that if the pod is rescheduled to a different node, it will not have access to the files on the original node's filesystem. Also, if multiple pods are scheduled on the same node and they use the same HostPath volume, they will be able to read and write to the same files on the host node's filesystem




```yaml
apiVersion: v1
kind: Pod
metadata:
 name: web-app
 namespace: dev
spec:
 containers:
     - name: web-app
       image: my-web-app-image
       volumeMounts:
           - mountPath: /app/uploads
             name: user-uploads
 volumes:
     - name: user-uploads
       hostPath:
          path: var/uploads
          type: DirectoryOrCreate
```

2 We then mount the user-uploads volume to the container's /app/uploads directory using the volumeMounts field in the container specification. This allows the web application to access the user-uploaded files stored in the /var/uploads directory on the host node's filesystem

1 We are creating a HostPath volume named user-uploads that maps to the /var/uploads directory on the host node's filesystem

type field specifies that the directory should be created if it doesn't already exist

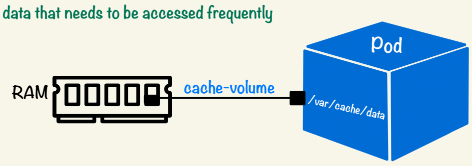
EmptyDir volumes:

EmptyDir volumes are a type of temporary storage volume that are created and attached to a Pod when the Pod is created. The data stored in an EmptyDir volume exists only for the lifetime of the Pod and is deleted when the Pod is deleted. These volumes are commonly used for storing temporary data that is needed by a Pod, such as cache files or temporary log

When you define an EmptyDir volume, you can specify a size limit for the volume. If you don't specify a limit, the application running in the pod can generate any amount of data, which can cause the disk to become full and potentially cause the node to become unavailable

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: monitoring-pod
spec:
  containers:
  - name: monitoring-container
    image: monitoring-image
    volumeMounts:
    - name: logs-volume
      mountPath: /var/log/monitoring-app
  volumes:
  - name: logs-volume
    emptyDir:
      sizeLimit: 1Gi
```

EmptyDir volume can be configured to store its data in memory instead of on disk. This provides faster access to the data in the volume, which can make it useful for caching data that needs to be accessed frequently



The medium field is used to indicate the underlying storage medium for a volume. By setting the medium to "Memory", the cache-volume volume will be created using the host node's RAM as the storage medium.

/var/cache/data directory inside the container is mounted to an EmptyDir volume named cache-volume. The cache-volume volume is configured with a sizeLimit of 1 gigabyte, which means that it can store up to 1 gigabyte of data in memory during the lifetime of the Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ML-app
spec:
  containers:
  - name: video-conv
    image: video-conv
    volumeMounts:
    - name: cache-volume
      mountPath: /var/cache/data
  volumes:
  - name: cache-volume
    emptyDir:
      medium: Memory
      sizeLimit: 1Gi
```

ConfigMaps and Secrets:

ConfigMaps and Secrets are Kubernetes objects that allow you to store configuration data and sensitive information such as credentials and keys, respectively. They can be mounted as volumes in a Pod, allowing the Pod to access the data as files Go to page 18

Persistent Volumes (PVs) and Persistent Volume Claims (PVCs):

PVs are independent storage volumes that can be provisioned from different storage providers such as cloud storage or on-premise storage systems, and PVCs are used to request storage resources from the PVs. The PVs and PVCs allow you to abstract the underlying storage infrastructure from your application, providing a layer of indirection. You can use PVs and PVCs to store data persistently, even if a Pod is deleted or restarted. PVs and PVCs can be used with different storage backends like NFS, iSCSI, Ceph, etc

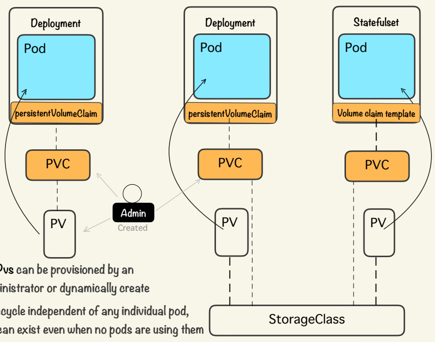
To connect PVs and PVCs to pods, you need to follow these steps:

Provision a Persistent Volume (PV): As an administrator, you'll define and create a PV object, specifying the storage capacity, access modes, and other properties. This involves interacting with your underlying storage infrastructure, whether it's local disks, network storage, or cloud storage.

Create a Persistent Volume Claim (PVC): A user or developer creates a PVC object, specifying their desired storage capacity, access modes, and any additional requirements. The PVC will be used by the pod to request storage.

Binding PVC to PV: Once the PVC is created, Kubernetes matches it with an available PV that meets the requested criteria. The binding process ensures that the PVC and PV are associated with each other.

Mounting the PV to a pod: In the pod's specification, you specify the PVC as a volume source. When the pod is scheduled and runs, Kubernetes mounts the PV associated with the PVC to a specified path within the pod's filesystem.



Pvs can be provisioned by an administrator or dynamically create

PVs have a lifecycle independent of any individual pod, meaning they can exist even when no pods are using them

PVC is a request for storage by a pod. It is a way for pods to dynamically request a specific amount and type of storage without having to know the details of the underlying storage infrastructure

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: nfs-pv1-40g-rw
spec:
  capacity:
    storage: 40Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs_server_ip
    path: /mnt/nfs_share/pv1-40g-rw
  persistentVolumeReclaimPolicy: Retain
```

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-pvc-20g
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 15Gi
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: my-image
      volumeMounts:
        - name: my-volume
          mountPath: /data
  volumes:
    - name: my-volume
      persistentVolumeClaim:
        claimName: nfs-pvc-20g
```

accessModes is a field that is used to specify how the volume can be mounted and accessed by a pod

- ReadOnlyMany (ROX): This access mode allows the volume to be mounted as read-only by multiple nodes in a cluster. This means that the volume can be mounted by multiple pods at the same time, but cannot be modified. This mode is typically used for shared read-only storage resources, such as configuration files or static data
- ReadWriteMany (RWX): This access mode allows the volume to be mounted as read-write by multiple nodes in a cluster. This means that the volume can be mounted by multiple pods at the same time, and can be modified. This mode is typically used for shared read-write storage resources, such as file shares or databases
- ReadWriteOnce (RWO): This access mode allows the volume to be mounted as read-write by a single node in a cluster. This means that the volume can be mounted by only one pod at a time, and is typically used for storage resources that can only be accessed by one node or pod at a time, such as local storage or block storage.

The persistentVolumeReclaimPolicy determines what happens to the contents of a Persistent Volume (PV) when it is released, specifying whether the contents should be retained or deleted

- Retain: The PV's contents are retained even after the PV is released. This means that the PV can be reused by creating a new PVC that requests the same storage capacity and access modes as the original PVC that used the PV
- Delete: The PV's contents are deleted when the PV is released. This means that the PV cannot be reused by creating a new PVC that requests the same storage capacity and access modes as the original PVC that used the PV
- Recycle (deprecated): The PV's contents are deleted when the PV is released, but the PV is made available for reuse. However, this value is deprecated and should not be used in newer versions of Kubernetes