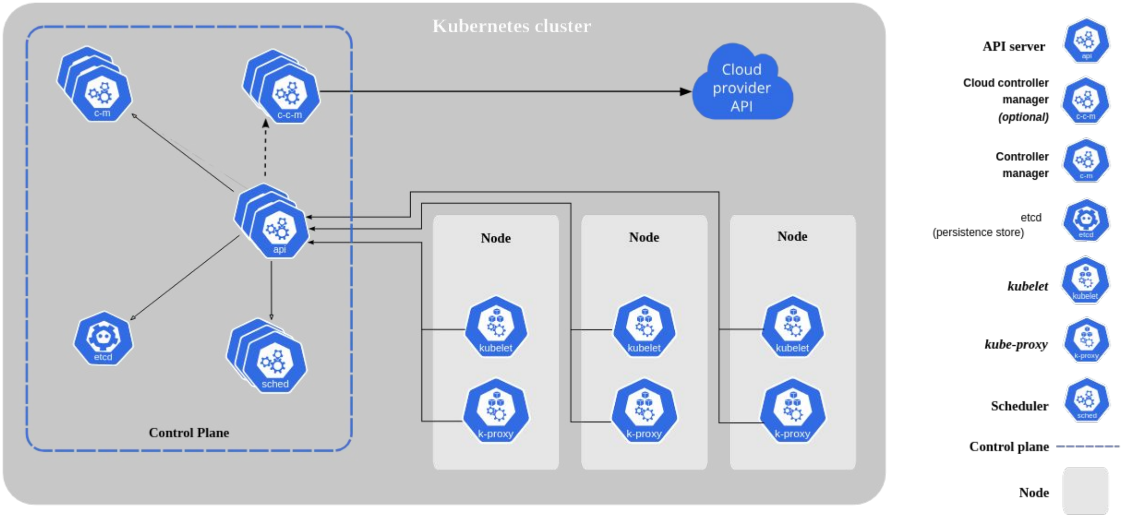
Kubernetes Operators

A deep dive into K8 native workload management

Kubernetes Architecture



The Controlplane:

- API Server
- Controller Manager
- Scheduler
- ETCD KV Store
- Cloud Controller Manager

The Dataplane (nodes):

- Kubelet (node agent)
- Kube Proxy (IPTables or IV mode)

Source: https://kubernetes.io/docs/concepts/overview/components/

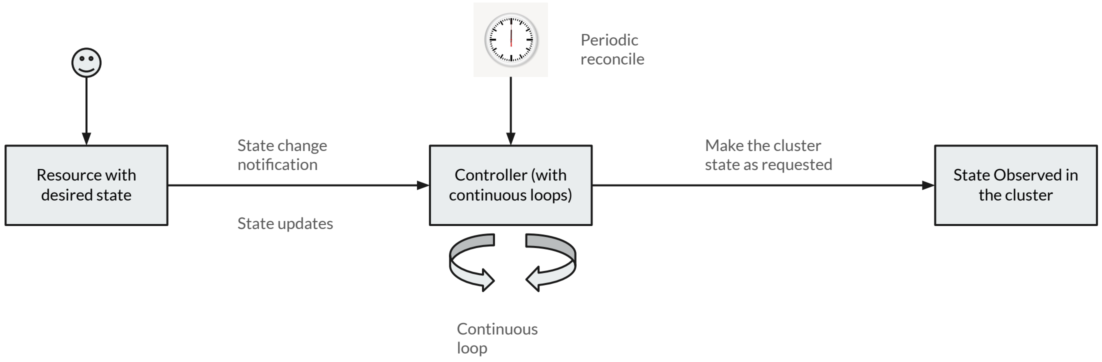
Foundation: Understanding the ‘Control Loop’

Control loops are implemented by the controller and it’s responsibility is to ‘watch’ the state of the cluster (Kubernetes API objects) and make or request a change so as to bring the ‘observed’ state closer to the ‘desired’ state defined

[Kubernetes official documentation: https://kubernetes.io/docs/reference/glossary/]

‘Control Loop’ in Kubernetes

1. Read the state of the resources (using events *streamed* over *watches*)
2. Change or request a change in state for the resource
3. Update the status of the resource to the API Server
4. Repeat



Reference reading: Programming Kubernetes from O’Reilly

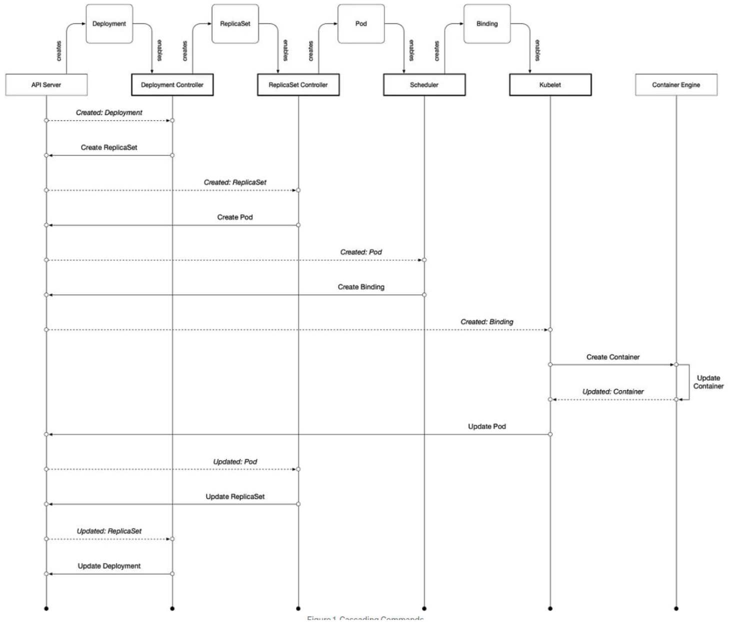
Building Blocks of a Control Loop

A Kubernetes Controller/Control Loop has 3 fundamental building blocks:

a. The Informer : Watches the desired state, implements resync and reconciliation
b. The Work Queue: Queuing the state changes, implement retries if needed
c. The Events*: The state changes (add/update/delete) itself

**Events here do not refer to the Event API, which are ephemeral resources stored in ETCD (upto an hour) and purged later, these objects merely act as a user friendly logs and are often created by other controllers [kubectl get events -n my-namespace]*

Kubernetes Event Reference



Kubernetes control plane employes events & a loosely coupled architecture over RPC calls, controllers watches the changes & executes the business logic inside it

The diagram on the left shows what happens when a pod is launched through a deployment in Kubernetes

A number of controllers communicating over events while for an end user it’s something as simple as kubectl create deployment command!

Source: [Refer to Andrew Chen’s & Dominik Tornow’s blog]

Uncovering the Go Code

```go
func (rsc *ReplicaSetController) manageReplicas(ctx context.Context, filteredPods []*v1.Pod, rs *apps.ReplicaSet) error
	diff := len(filteredPods) - int(*(rs.Spec.Replicas))
	rsKey, err := controller.KeyFunc(rs)
	if err != nil {
		utilruntime.HandleError(fmt.Errorf("couldn't get key for %v %#v: %v", rsc.Kind, rs, err))
		return nil
	}
	logger := klog.FromContext(ctx)
	if diff < 0 {
		diff *= -1
		if diff > rsc.burstReplicas {
			diff = rsc.burstReplicas
		}
		// TODO: Track UIDs of creates just like deletes. The problem currently
		// is we'd need to wait on the result of a create to record the pod's
		// UID, which would require locking *across* the create, which will turn
		// into a performance bottleneck. We should generate a UID for the pod
		// beforehand and store it via ExpectCreations.
		rsc.expectations.ExpectCreations(logger, rsKey, diff)
		logger.V(2).Info("Too few replicas", "replicaSet", klog.KObj(rs), "need", *(rs.Spec.Replicas), "creatin
		// Batch the pod creates. Batch sizes start at SlowStartInitialBatchSize
		// and double with each successful iteration in a kind of "slow start".
		// This handles attempts to start large numbers of pods that would
		// likely all fail with the same error. For example a project with a
		// low quota that attempts to create a large number of pods will be
		// prevented from spamming the API service with the pod create requests
		// after one of its pods fails.  Conveniently, this also prevents the
		// event spam that those failures would generate.
		successfulCreations, err := slowStartBatch(diff, controller.SlowStartInitialBatchSize, func() error {
			err := rsc.podControl.CreatePods(ctx, rs.Namespace, &rs.Spec.Template, rs, metav1.NewController
			if err != nil {
				if apierrors.HasStatusCause(err, v1.NamespaceTerminatingCause) {
					// if the namespace is being terminated, we don't have to do
					// anything because any creation will fail
					return nil
				}
			}
			return err
		})
```

What's going on here?

- Calculate the diff i.e; current state (pods) vs what's in the spec
- Then if diff < 0 (less than needed) ~ Create new pods
- But if diff > 0 (more than needed) ~ Delete some pods

Under the hoods it does more intelligent work like picking pods to delete and all but those are the implementation details, however message is clear enough..”**match the desired state to observed state**”

[Kubernetes source code on GitHub : https://github.com/kubernetes/kubernetes/blob/master/pkg/controller/replicaset/replica_set.go#L565]

Optimistic Concurrency

- Kubernetes uses optimistic concurrency to carry out concurrent operations without locking
- API server detects concurrent writes and rejects the latter of the two operations
- API Client's responsibility is to handle this and probably retry the operation
- The client code provided by the API machinery libs use resource versions to determine if another process in the cluster updated the resource before client.Update() was called by the controller

Detecting changes : https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes

Operators: Control Loops & Operational Intelligence

- Operators were first introduced by CoreOS in 2016
- Inspired by how DevOps engineers used their ‘domain’ knowledge to run software in production
- Operator is a “workload” specific control loop which embodies the operational knowledge needed to run workloads reliably in production
- At the very foundation of any operator, resides Kubernetes resources & controllers but it’s value is enhanced through ‘workload aware’ automation

Introducing Operators a blog from 2016 by CoreOS :
https://www.redhat.com/en/blog/introducing-operators-putting-operational-knowledge-into-software

Operators: The Building Blocks

- Kubernetes API extensions (A.K.A Custom Resources)
- Custom controllers

Reference : https://kubernetes.io/docs/reference/using-api/api-concepts/

Quick Introduction: Custom Resources

- Available since Kubernetes 1.7
- Serves a way to extend Kubernetes API and declare a custom objects
- Custom Resources also serves as abstraction to hide lower level Kubernetes details
- Used in the cloud native landscape to provide a “Kubernetes-first” experience
- Istio, Linkerd, Flux, Argo and many other CNCF hosted projects use Custom Resources

Quick Introduction: Custom Resource Definition

- CRDs form the base for creating CRs, that is Custom Resources
- It's an API natively available in Kubernetes to help you define the anatomy of your Custom Resource
- When it comes to persisting & serving your CRs defined via CRDs, API server makes no distinction
- State of a Custom Resource is also persisted in ETCD alongside the state of native Kubernetes resources like a Deployment or a Service
- As a provider of the CR, you can control versioning & also instruct API server to do conversions between the versions when a CR is requested (e.g. moving from v1beta1 to v1)

Reference : https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/

Example: CRD & Custom Resource

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: crontabs
    # singular name to be used as an alias on the CLI and for display
    singular: crontab
    # kind is normally the CamelCased singular type. Your resource manifests use this.
    kind: CronTab
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - ct
```

```yaml
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
```

What this may look like:

1. Define the CR using the CRD
2. Apply the CRD to the cluster
3. Create a CR manifest
4. Apply the CR to the cluster

Reference : https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/#create-a-customresourcedefinition

What's Next?

- Define a Custom Resource Definition (CRD) and create a Custom Resource following the schema provided by the CRD
- But Kubernetes as such does not know what to do with it
- That's where Custom Controllers come in
- Operator SDK (part of the Operator Framework) offers an easy way to bootstrap Custom Controllers & deploy to your Kubernetes cluster

Kubebuilder Framework

- Kubebuilder maintained by the API Machinery SIG provides an easier and efficient way to write custom controllers
- Kubebuilder provides Go modules/packages to help you simplify the operator development:
  - Manager
    - Client
    - Cache
  - Controller
    - Reconciler
    - Predicate
  - Webhook
    - Admission Request
    - Validator

Further Reading: https://book.kubebuilder.io/introduction

Operator SDK

- Bases on Kubebuilder, CoreOS/RedHat put together the Operator Framework
- Operator SDK is a part of the Operator Framework
- Operator SDK uses the Kubernetes ‘Controller-Runtime’ library to make operator development easier, scalable, automated and more effective
    - Scaffolding
    - Automated testing
    - Code generation & simple bootstrapping
    - Extensions
    - API abstraction
    - Supports writing operators using Ansible, Helm or Go

Operator Framework : https://sdk.operatorframework.io/

Operator Initialization & Code Generation

Key Commands:

operator-sdk init --domain acme.io --repo github.com/acme/redis-operator

operator-sdk create api --group cache --version v1alpha1 --kind Redis --resource --controller

make generate

make manifests

make install run

kubectl apply -f <custom-resource-file>.yaml

kubectl patch redis redis-cache -n conf42-init -p '{"spec":{"size": 4}}' --type=merge

Reference: https://sdk.operatorframework.io/docs/building-operators/golang/tutorial/