Why would I want to use an operator?

An operator is a design pattern for extending the API of a Kubernetes cluster to include a new resource type that you define the behaviour of. This resource can then be used in the the same manner as Kubernetes core resources.

There are two benefits to this approach. First, your new resource is available for use on the cluster in the exact same manner as core resources such as pods and services. A Kubernetes application developer who is familiar with using those core resources to provision and manage their applications is already familiar with the methods available to interact with your new resource. This allows your new resource to seamlessly integrate with the greater Kubernetes ecosystem. The other benefit is the expressive power that comes with having control over the internal reconciliation of your new resource. Rather than having your application statically deployed on a cluster, operators enable you to react both to requested changes from users and a cluster's shifting circumstances to manage your custom resource.

Tools for writing an operator

As operators are an idea, many unrelated implementations are all considered valid operators. Although you could write one from scratch using the same tools the core Kubernetes components are written with, several tools exist to make it easier to write an operator. This course makes use of:

* [Operator SDK](https://github.com/operator-framework/operator-sdk), a command-line tool for scaffolding operators that is based on Kubebuilder
* [Kubebuilder](https://github.com/kubernetes-sigs/kubebuilder), a tool for scaffolding Golang operators
* [Controller-Runtime](https://github.com/kubernetes-sigs/controller-runtime), a Golang library that is a simplified wrapper around [Client-Go](https://github.com/kubernetes/client-go), the Kubernetes API Golang client library

## Kubernetes architecure

If you're familiar with using Kubernetes, you have probably used kubectl to create resources on a cluster. With kubectl, you execute a command and the resources show up on the cluster. But what actually happens when you run kubectl create foo?

Diagram

Description automatically generated

At its heart, a Kubernetes cluster is really just a few components wrapped around an etcd key-value store. The following diagram shows the basic flow of how resources are created on a cluster:

Diagram

Description automatically generated

1. When you issue the command kubectl create foo, your local command-line interface issues a write request to the Kubernetes API server.
2. The API server performs validation to ensure that your foo object has the structure it should, but mostly just passes your request on through to the etcd store.
3. Your request creates a new entry in etcd for that resource.
4. This change is seen by a Kubernetes controller, which is a process responsible for reconciling

the desired state contained in the etcd store with the actual state running in the cluster.

1. When the controller sees your new entry, it goes and performs whatever actions are needed to create the desired state of your object foo. Once it has reconciled the desired state from the etcd with the actual state of the system, it marks the entry in the etcd as reconciled.

This loop is how all interactions on the cluster are performed -- creating resources, deleting resources, reacting to changing cluster status as components are brought up or down.

**The essence of an operator** is to reproduce this control loop with user-created components.

With an operator, you need to create equivalents for the three basic components of the loop:

* an API server to handle incoming requests
* a data store to store the desired and actual state of the system
* a controller that watches the data store and reacts to changes.

Writing all these from scratch is a lot of work, but tools such as **Operator SDK** make the task much easier.

Diagram

Description automatically generated

We replace the core roles of the control loop with our own equivalents. While operators can vary in form and function, the simple use case of an operator that adds a single new resource to a cluster using a [custom resource](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01). This new resource is managed by implementing a controller and running it inside the cluster. While this specific pattern is not the only possible way to build an operator, it covers the components and techniques that you can use to construct many other possible patterns.

## Custom resource definition

You're probably already familiar with some of the core Kubernetes resources, including pods, deployments, and services. Users declare these resource to alter the state running on the cluster. Declaring a custom resource definition (CRD) extends the Kubernetes API to include a new resource that we define the structure of. Once the resource is added to the cluster, it can be interacted with like any other resource. You can use kubectl to create and update instances of the CRD, and can store configuration of it in a YAML file. Examples of possible custom resources are Postgres and MariaDB.

By itself, a CRD only adds the resource type to the cluster. A *controller* provisions actual backend resources to reconciles the current state with the desired state.

## Controller

Users interact with Kubernetes by declaring state using Kubectl or a similar tool. These desired resources get written to the central etcd key-value store of the Kubernetes cluster. Then, one or more controllers responsible for that resource sees the new desired state and act on it to bring about the desired state somewhere in the cluster. The backend actions caused by the controller can include things like causing containers to start on a worker node to run a pod, rewriting networking rules to create a service, or writing further desired state in the etcd store itself (such as when a deployment causes pods to be created).

Controllers for core resources such as pods, services, and deployments are included in a Kubernetes cluster by default. To implement a new resource based on a CRD, you need to create a new controller that reconciles your new type. Typically, this is done by creating a program that does the reconciliaton and then runs that process within the Kubernetes cluster in a container. Golang is the most common language to use to write controllers, but Operator SDK also supports creating a controller through Ansible files or bootstrapping one from existing Helm charts

## CRD + controller

This pattern of creating a CRD and creating a controller to manage that single CRD is the most common pattern for operators, but not the only one. Other operator patterns examples include:

* Complex operators that consist of multiple related CRDs, such as a Postgres type, a Postgres user type, and a Postgres backup type.
* A simple operator could act only as a controller that modifies the behavior of existing resource types such as pods to add novel functionality to the cluster on a basic level. The controller itself might not even be running in the cluster, instead it's running on an external

system co-located where the cluster is running, or perhaps as a self-service API running on the internet. By accessing the Kubernetes API in the same manner as core Kubernetes functionality, operators offer the flexibility to work with varying architectures while enabling powerful expressiveness and a fine degree of control.