Operator Whitepaper - Working Document

Short Link: https://bit.ly/39FZWkt

This document has moved -> https://github.com/cncf/sig-app-delivery/tree/master/operator-wg/whitepaper

How to contribute:

We are welcoming contributors to this document!

You can suggest content to each section directly from this doc. For any questions or any section you believe is missing, please reach out to us on:

- CNCF's #sig-app-delivery-operator-wg slack group or
- attend this working group's meetings (see "SIG App-Delivery Operator Working Group" under the SIG-App-Delivery calendar).

Executive Summary

TODO: Write executive Summary

Maintaining applications or infrastructure requires many repetitive actions and checks. As computers are great in doing repetitive tasks and verifying the state of an object, the preferred way to maintain an application of infrastructure should be through code. An operator provides a way to encapsulate the tasks and checks required to manage other objects.

In Kubernetes, an operator provides intelligent, dynamic management capabilities by extending the functionality of the API.

An operator allows for automation of common processes as well as reactive applications that can continually adapt to their environment. This in turn allows for more rapid development with fewer errors, lower mean-time-to-recovery, and increased engineering autonomy.

Introduction

TODO: Write introduction

The goal of this document (add this to PR)

The goal of this document is to provide a definition of operators for cloud-native applications in the context of Kubernetes and other container orchestrators.

Target Audience / Minimum Level of Experience (add this to PR)

This document is intended for application developers, Kubernetes cluster operators and service providers (internal or external) - who are looking to learn about operators and the problems they can solve. It can also help teams already looking at operators to learn when and where to use

them to best effect. It presumes basic Kubernetes knowledge such as familiarity with Pods and Deployments.

Foundation

Kubernetes primitives were not built to manage state by default. The success of Kubernetes and other orchestrators was due to their focus on containers' main capabilities and, as companies began their journey to cloud native, working with more specific use cases (microservices, stateless applications) made more sense.

As Kubernetes and other container orchestrators grew their reputation and their extensibility, requirements became more ambitious and the desire to use orchestrators' full lifecycle capabilities was also transferred to highly distributed data stores.

By relying on Kubernetes primitives alone, it's difficult to achieve a generic and automated way to manage complex stateful application requirements such as replication, failover automation, backup/restore, upgrades, etc which can occur based on events that are too specific.

We can solve this problem with the Operator Pattern. By leveraging Kubernetes built in capabilities such as self-healing and reconciliation (or any orchestrator for that matter) and extending those along with application specific complexities, it is possible to automate the lifecycle and operations of any application and turn it to a highly capable offering.

When we hear about Operators we think of Kubernetes, but the idea of an application whose management is fully automated can be exported to other platforms, therefore with this paper we aim to bring this concept to a higher level than Kubernetes itself.

Operator Design Pattern (add to PR)This section describes the pattern with abstract concepts, the next section "Kubernetes Operator Definition" will describe the implementations of the pattern in terms of Kubernetes objects and concepts

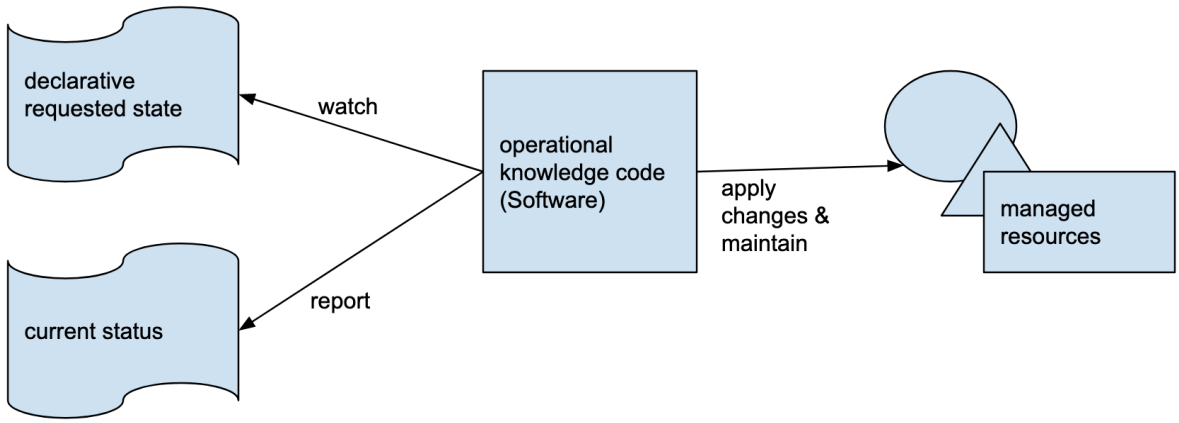
The operator design pattern defines how to manage application and infrastructure resources using declarative domain-specific knowledge. The goal of the pattern is to reduce the amount of manual imperative work which is required to keep an application in a healthy and well-maintained state.

By using the operator pattern, the knowledge on how to adjust and maintain a resource is captured in code and usually in a single service (also called a controller).

When using an operator design pattern the user should only be required to describe the desired state of the application and resources. A software pre-written should make the necessary

changes in the world so it will be in the desired state. The software will also monitor the real state continuously and take actions to keep it healthy and n the same state (preventing drifts)

A general diagram of an operator will have software that can read the desired spec and can create and manage the resources that were described.



The Operator pattern consists of three components:

- The application or infrastructure that we want to manage in a declarative way.
- Our domain specific knowledge that describes the state of the application in a declarative way.
- A controller that run continuously:
    - Reads and is aware of the state.
    - Runs actions against the application in an automated way.
    - Report the state of the application in a declarative way.

This design pattern will be applied on Kubernetes and its operators in the next sections.

Operator Characteristics (to be changed)

TODO: Make more kubernetes-agnostic or change position

The core purpose of any operator is to extend Kubernetes with new domain knowledge. Just like Kubernetes natively understands things like containers and layer 4 load balancers via the Pod and Service objects, operators add new capabilities for more complex systems and applications. As an example, prometheus-operator adds a new object type, Prometheus, which extends Kubernetes with high-level support for deploying and running a Prometheus server.

The capabilities provided by an operator can be sorted into three overarching categories: dynamic configuration, operational automation, and domain knowledge.

Dynamic Configuration

Since the dawn of time, there have been two main ways to configure software, configuration files and environment variables. In the cloud native world there are some newer options like querying a well-known API at startup, but most existing software in the world uses one or both of these options. Kubernetes naturally provides many tools to interact with these (ConfigMaps, Secrets, etc) but because they are generic, they don't understand any specifics of configuring a given application. An operator can define new custom object types (custom resources) to better express the configuration of a particular application in a Kubernetes context.

This allows for better validation and data structuring, which in turn reduces the likelihood of small configuration errors and improves the ability of teams to self-service without having as deep or complete a knowledge of either Kubernetes or the target application as would be traditionally required. This can include things like progressive defaults, where a few high level settings are used to populate a best-practices-driven configuration file, or adaptive configuration such as adjusting resource usage to match available hardware or expected load based on cluster size.

Operational Automation

Along with custom resources, most operators include at least one custom controller. These controllers are daemons that run inside Kubernetes like any other, but connect to the Kubernetes API and provide automation of common or repetitive tasks. This is the same way that Kubernetes itself is implemented, you may have seen kube-controller-manager or cloud-controller-manager mentioned in your journey so far. But as we saw with configuration, operators can extend and enhance Kubernetes with higher-level automation such as deploying clustered software, providing automated backups and restores, or dynamic scaling based on load.

By putting these common operational tasks into code, we can ensure it will be repeatable, testable, and upgradable in a standardized fashion. Keeping humans out of the loop on frequent tasks also ensures that steps won't be missed or elided, and that different pieces of the task can't drift out of sync with each other. As before, this allows for improved team autonomy by reducing the hours spent on boring-but-important upkeep tasks like application backups.

Domain Knowledge

Similar to operational automation, we can write an operator to encode specialized domain knowledge about particular software or processes. A common example of this is application upgrades. While a simple stateless application might need nothing more than a Deployment's

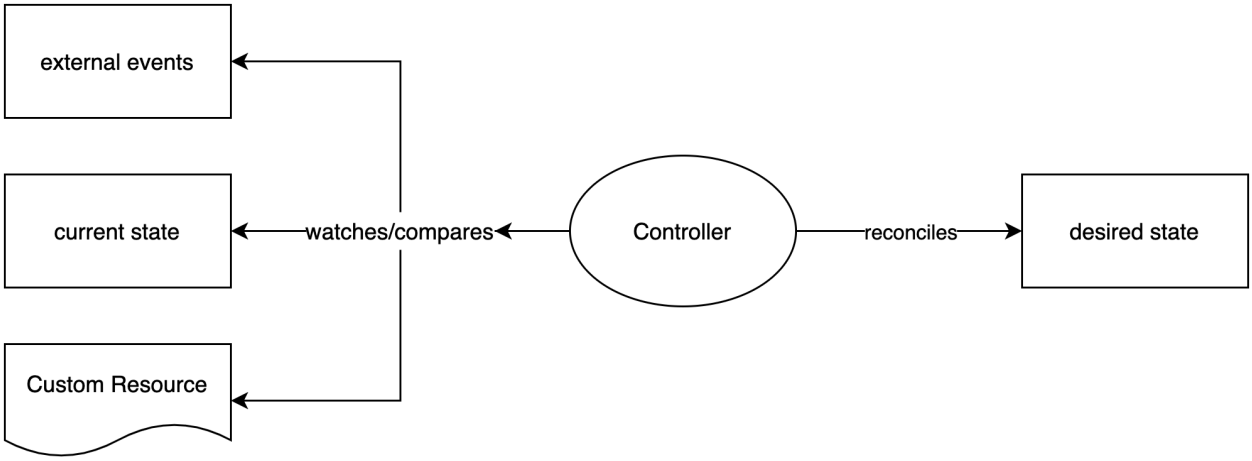
rolling upgrade, databases and other stateful applications often require very specific steps in sequence to safely perform upgrades. This can be handled autonomously by the operator as it knows your current and requested versions and can run specialized upgrade code when needed. More generally, this can apply to anything a pre-cloud-native environment would use manual checklists for, effectively using the operator as an executable runbook.

Another common place to take advantage of automated domain knowledge is error remediation. Kubernetes built in remediation behaviors mostly start and end with “restart container until it works” which is a powerful solution but often not the best or fastest solution. An operator can monitor its application and react to errors with specific behavior to resolve the error or escalate the issue if it can’t be automatically resolved. This can reduce MTTR (mean time to recovery) and also reduce operator fatigue from recurring issues.

Kubernetes Operator Components (in PR)

“An operator is a Kubernetes controller that understands 2 domains: Kubernetes and something else. By combining knowledge of both domains, it can automate tasks that usually require a human operator that understands both domains”

(Jimmy Zelinskie, https://github.com/kubeflow/tf-operator/issues/300#issuecomment-357527937).



Kubernetes Operator

Operators enable to extend the Kubernetes API with operational knowledge. This is achieved by combining Kubernetes controllers and watched objects that describe the desired state. The controller can watch one or more objects and the objects can be either Kubernetes primitives as Deployments and Services or things which reside outside of the cluster as Virtual Machines or Databases. The controller will constantly compare the desired state with the current state using the reconciliation loop which ensures that the watched objects get transitioned to the desired state in a defined way.

. The operational (or domain-specific) knowledge is usually encapsulated in one or more Kubernetes custom resources which are defined by custom resource definitions(see section below).

Kubernetes Controllers (in PR)

A Kubernetes Controller takes care of routine tasks to ensure the desired state expressed by a particular resource type matches the real-world state (current state (https://engineering.bitnami.com/articles/a-deep-dive-into-kubernetes-controllers.html, https://fntlnz.wtf/post/what-i-learnt-about-kubernetes-controller/). For instance, the ReplicaSet controller takes care that the desired amount of pod replicas is running and a new pod spins up, when one pod is deleted or fails.

Technically, there is no difference between a typical controller and an operator. Often, the difference referred to is the operational knowledge which is included in the operator. As a result, a controller which spins up a pod when a custom resource is created and the pod gets destroyed afterwards can be described as a simple controller. If the controller has additional operational knowledge like how to upgrade or remediate from errors, it is an operator.

Custom Resources and Custom Resource Definitions (in PR)

Custom resources are used to store and retrieve structured data in Kubernetes (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/). In case of an operator, a custom resource contains the required state of the application- or domain specific knowledge, but does not contain the implementation logic . Such information could be the version information of application components, but also enabled features of an application or information where backups of the application could be part of this. A custom resource definition (CRD) defines how such an object looks like, for example which fields exist and how the CRD is named. Such an CRD can be scaffolded using tools (as the operator sdk) or be written by hand.

The following example illustrates, how such an custom resource instance definition could look like:

```
apiVersion: example-app.appdelivery.cncf.io/v1alpha1
kind: ExampleApp
metadata:
  name: appdelivery-example-app
spec:
  appVersion: 0.0.1
  features:
    exampleFeature1: true
    exampleFeature2: false
  backup:
```

```
    enabled: true
    storageType: "s3"
    host: "my-backup.example.com"
    bucketName: "example-backup"
status:
  currentVersion: 0.0.1
  url: https://myloadbalancer/exampleapp/
  authSecretName: appdelivery-example-app-auth
  backup:
    lastBackupTime: 12:00
```

This example creates a custom resource with the name "appdelivery-example-app" of the kind "ExampleApp".

The "spec" section is where the user can declare the desired state. This example declares that appVersion 0.0.1 should be deployed with one feature enabled and another disabled. Furthermore, backups of this application should be made, and a s3 bucket should be used.

The "status" section is where the operator can communicate the current state of the application and other useful information back to the user. In this example, the status shows the current deployed version. If it is different from the "appVersion" in the spec, then the user can expect that the operator is working to deploy the version requested in the spec. Other common information in the status section includes how to connect to an application and the health of the application.

Control Loop (in PR)

The control (reconciliation) loop in a Kubernetes controller ensures that the state that the user declares using a CRD and other primitives matches the state of the application, but also that the transition between the states works as intended. One common use-case could be the migration of database schemes when upgrading an application. The control loop can be triggered on specific events, as a change on the crd, but also time-based, like for backing up data at a defined time.

Operator Capabilities

In the following sections, capabilities an operator could have are described.

TODO: Write Section Introduction

Install an application / take ownership of an application (add to PR)

The basic capability of an operator is to provision the required resources in order to fulfill the desired state.

An operator should be able to provision and set up all the required resources, so no manual work would be required during the installation. An operator must check and verify that resources that were provisions are working as expected, and ready to be used.

An operator should also be able to recognize resources that were provintied before the installation process, and only take ownership of them for later use. In this case, the ownership process should be seamless and not cause downtime. The ownership process purpose is to enable easy migration of resources to the operator.

Upgrade an application

TODO: Write Application upgrade section

- Ensure that the application runs in the defined state also after upgrading
    - Operator should upgrade also dependencies as the database and its schema
    - Operator should promise a rollback option and therefore be able to back up data
    - In a perfect world, the operator should do upgrades without downtime (no hard requirement). If schema upgrades of databases are necessary, short outages are acceptable

Backup

TODO: Refine Backup section, add figure
An operator's capability to back up the application state and it's data should give the user the certainty that he is able to restore from a failure if data is lost or compromised.

Therefore, the process of backing up data can be described as follows:

1. Backup gets triggered (either manually or automatically)
2. The application is set to a state where it allows consistent backups
3. Backup data and save it to an external storage
4. Write the Backup state and its location to the custom resource

At first, the application (data store) is set to consistent state (like a consistent snapshot). Afterwards, the data gets backed up using appropriate tools and are saved to an external storage, which may be an nfs share on-site, but could also be an s3 bucket in the cloud. Either if the Backup failed or succeeded, this state will be written to the state of the custom resource.

Recovery from Backup (Restore)

TODO: Write Recovery section

- Recovery path should be definable in the spec
- Data is restored from the recovery path
- Recover the same application-version as the backup

Auto-Remediation

TODO: Write Auto-Remediation section

- Operator is able to get the applications state
    - E.g. Metrics indicating errors
- The operator knows how to deal with errors / system errors
- The operator takes actions

Monitoring/Metrics - Observability (add to PR)

While the managed application should provide the telemetry data for itself, the operator could provide metrics about its own behavior and only provides a high level overview about the applications state (as it would be possible for auto-remediation). Furthermore, typical telemetry data provided by the operator could be the count of remediation actions, duration of backups, but also information about the last errors or operational tasks which were handled.

“Scaling the operator itself”

Scaling (to be changed)

Scaling is part of the day-2 operations that an operator can manage in order to keep the application / resources functional. The scaling capability doesn’t require the scaling to be automated, but only that the operator will know how to change the resources in terms of horizontal and vertical scaling.

An operator should be able to increase or decrease any resource that it owns, such as CPU, memory, disk size and number of instances.

Ideally the scaling action will be without a downtime. Scaling action ends when all the resources are in consistent state and ready to be used, so an operator should verify the state of all the resources and report it.

Auto-Scaling (to be changed)

An operator should be able to perform the scaling capability based on metrics that it collects constantly and according to thresholds. An operator should be able to automatically increase and decrease every resource that it's own

An operator should respect basic scaling configuration of min and max.

Disconnect

TODO: Write Disconnect section

- Unlink objects from the operator
- After disconnecting all objects created by the operator are still present

Uninstalling

TODO: Write Section about uninstallation section

- All objects managed by the operator are removed

Auto-Configuration Tuning

TODO: Write Auto-Configuration Tuning section

- The Operator is responsible for managing the configuration of the deployed application
- The customer should be able to "tell" the operator which configurations could be auto-tuned
- Auto-Tuning features of the operator should be documented (the user should know which configuration will be auto-tuned)

Security and Risks

General objective: let the users understand what they have to look at while installing an operator from a secuirty point of view. And let the operator developers to have a clear checklist of what they have to do for their operator to be considered secure.

- API Permissions
- Operator Scope (Cluster vs. Namespaced vs. Outside Cluster)
- Vendoring and keeping client-go updated
- Security constraints for the resources created/edited by the operator
- Audit log analysis
- Follow the recommendation on Sig Security white paper for securing workloads

Three areas

- Build trust - documentations
  - Threats
    - which threats an operator brings?
  - How do users that wants to use an operator can trust it? What API endpoints and permissions?
  - CVEs disclosure
  - How transparent the process is?
  - Audit log analysis, what should the user expect to see once this is deployed
- Security constraints - implementation
  - How can I as a customer or end user of an operator can make sure that it's a secure one?
  - API Permissions
  - What are the security features of the operator?
  - Is this operator considering security as a first class citizen?
  - Operator Scope (Cluster vs. Namespaced vs. Outside Cluster)
- Metrics - end user observation:
  - Audit log

Operator Frameworks for Kubernetes

TODO: Describe Frameworks used for creating operators (Operator Framework, KUDO, kopf, Metacontroller, …)
TODO: Describe Ansible/Helm Operators and when it would make sense to use them

Operator Frameworks for Non-Kubernetes Platforms

TODO: juju

Operator Lifecycle Management

TODO: Describe the Lifecycle of an Operator itself and how it could be maintained

Use-Cases for an Operator

TODO SIG: Think about possible use cases for an operator, possibly the whole lifecycle of an application?

An operator is used to install an application, or to provision another object which is achieved by defining a set of objects which are managed by the operator and how they work with each other. After the installation, the target application should be running without human interaction. In further consequence a controller is used for reconfiguration of a system.

To achieve this, an operator watches the current state and the definitions made in the custom resource or external events, compares them and starts reconciling the application to get to the desired state when it's needed. Changes in the custom resource could be enabling a feature or changing a version, external events could be the availability of an application update reported by an api. The current state of the application could also differ, when objects managed by the operator get deleted and so they also get recreated to get to the desired state.

When updating an application, the operator contains the logic which is needed to get to the new application version and how to transition. As described in the last chapter, this could be mechanisms to backup data before updating and updating the database schema. Therefore, the logic included in the operator knows which prerequisites are necessary to build a consistent backup, how to backup the data and how to get back to the normal state.

Finally, the operator is able to remove the application and the resulting objects.

Best Practices

TODO SIG: Define Topics

- Placement of Operators (Workloads vs. Control Processes)
- Metrics: Prometheus/OpenTelemetry
- Umbrella-Operators (Operator of Operators)
- Best Practices for Operator Implementation
    - See: https://cloud.google.com/blog/products/containers-kubernetes/best-practices-for-building-kubernetes-operators-and-stateful-apps
    - Loose coupling
    - As less features as possible (stability)
- Best Practices for securing operators
- Publishing an Operator / How to find an operator?
    - [TODO, operatorhub, artifacthub, OLM]

Technical Implementation

- How to write an operator
- Describe Demo Application (PodTato-Head)
- Implementation of different capabilities
- Delivery Pipeline for Operators

Summary / Conclusion

TODO: Write Summary

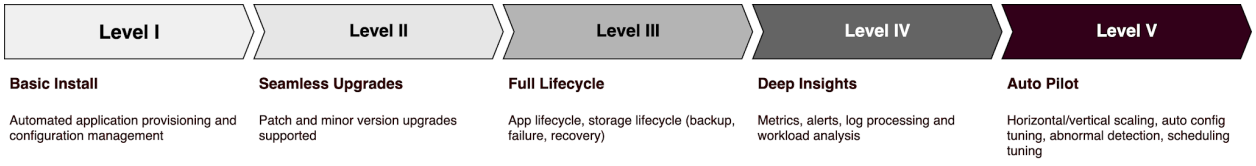
Related Work / What's next? / Where to get help?

TODO: Write about related books, articles and how this document enhances them or differs from them.

Initially, Operators were introduced by a blog post on the CoreOS Blog (Philips, 2016) . This article provides a rough overview what operators are, why the concept has been developed and how they are built. The insights of this article are mainly used for the definition of operators in this document. As the blog post only provided a concise overview, additional terms as capabilities, security and additional concepts are described more in-depth in this document.

The Book "Kubernetes Operators" (Dobies & Wood, 2020) provides a comprehensive overview about operators, which problems they solve and the different methods to develop them. Definitions made in this book flowed into this document. The same applies to the Book "Kubernetes Patterns" (Ibryam, 2019), which provides more technical and conceptual insights to operators. Definitions made in these books were summarized in this document (to provide a common declaration of operators).

Many documents describe capability levels of operators, which should describe the maturity of an operator.



Operator capability levels (Operator Framework, n.d.)

As there could be the case that an operator which supports all features of the highest level is not capable of some lower-level features, this document covers rather capabilities instead of capability levels. The capabilities used for defining capability levels were taken into consideration for this document.

Bibliography

Dobies, J., & Wood, J. (2020). *Kubernetes Operators*. O'Reilly.

Ibryam, B. (2019). *Kubernetes Patterns*. O'Reilly.

Operator Framework. (n.d.). *Operator Capabilities*. Operator Framework. Retrieved 11 2020, 24, from https://operatorframework.io/operator-capabilities/

Philips, B. (2016, 03 16). *Introducing Operators: Putting Operational Knowledge into Software*. CoreOS Blog. Retrieved 11 24, 2020, from https://coreos.com/blog/introducing-operators.html