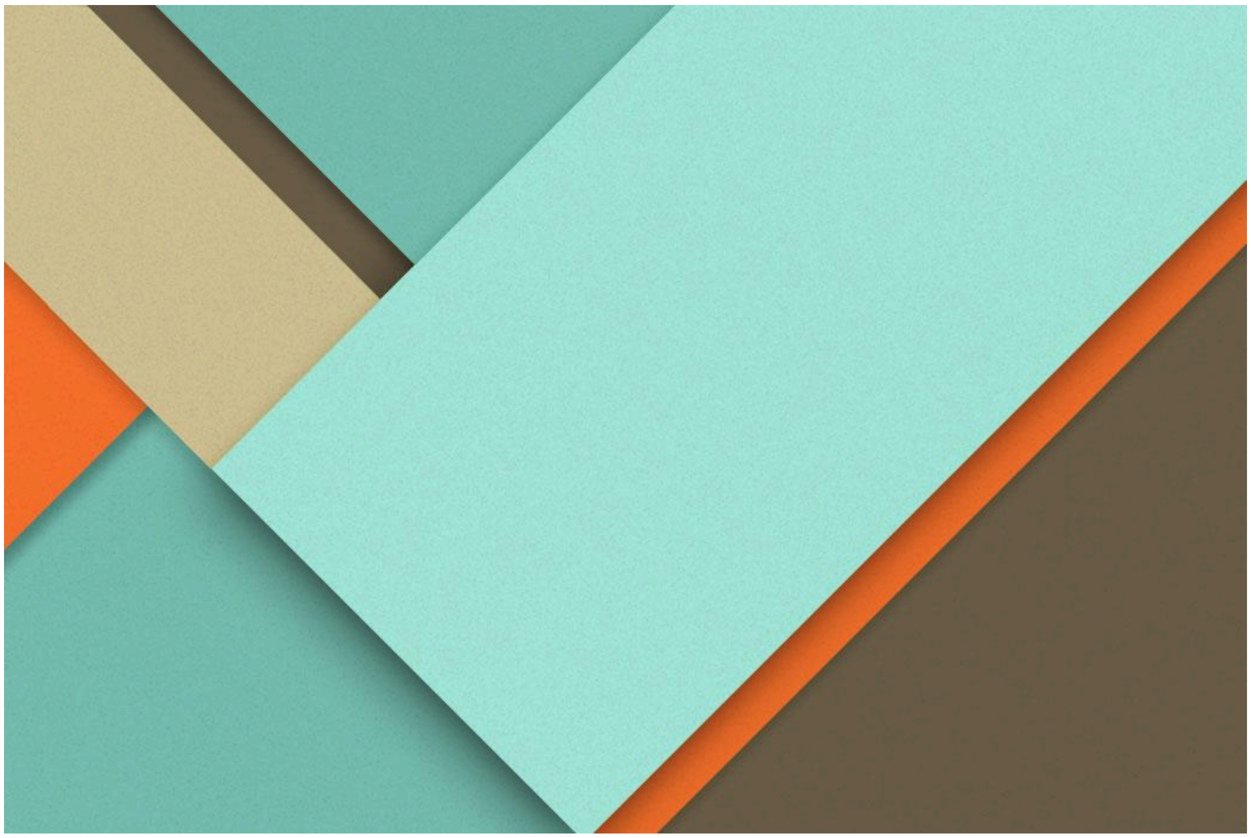


Create K8 Operators for Besu

Hyperledger Mentorship Program Project Proposal

Project Wiki Page : Link
Project Description : Link
Docker Link : Link

Email Id : sumaid.ali@students.iiit.ac.in / sumaidsyed@gmail.com
Rocket Chat / Telegram : @sumaids
Mentors : Mark Wagner, Joshua Fernande

Project Overview

What is Hyperledger Besu?

Hyperledger Besu, formerly known as Pantheon, **is a Java-based open-source Ethereum client created under the Apache 2.0 license.** It can be run both on the Ethereum public network or on private permissioned networks and test networks such as Rinkeby, Gorli, and Ropsten. Besu helps develop enterprise applications that require secure, high-transaction processing in a private network. The enterprise features supported by Besu are permissioning and privacy.

Hyperledger Besu is also an Ethereum client. An ethereum client refers to a node that verifies a blockchain and its smart contracts and everything else related to a blockchain. In other words, it is a software that implements the Ethereum protocol. **So, what does an Ethereum client contain?**

- Storage for persisting data related to transaction execution.
- APIs for application developers for interacting with the blockchain.
- An execution environment used for processing transactions in the Ethereum blockchain.
- Peer-to-peer networking to communicate with other Ethereum nodes to synchronize state.

What is Kubernetes?

Deploying applications using plain Docker containers has many issues such as :

- No management layer on top of it
  - If a container dies, nothing will restart it
  - If a virtual machine crashes!?
- No monitoring, alerts, etc.
- Not scalable, no easy way of spreading containers among VM’s

The solution to all these is the Kubernetes, which undoubtedly is the future of container orchestration. Kubernetes (also known as k8s or “kube”) is an open source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available. Kubernetes makes it very easy for deploying microservices.

Kubernetes is a good fit for blockchain networks often running on multiple clouds and on-premise. Blockchain networks require different types of nodes - validators, bootnodes, and normal network nodes, so these arrangements can be easily configured for a particular cluster.

Currently it's possible to deploy Besu on Kubernetes using Helm Charts or in kubectl. The purpose of the project is to wrap everything under Kubernetes operator, so one can provide the operator the desired state of Besu, and let it reach that state.

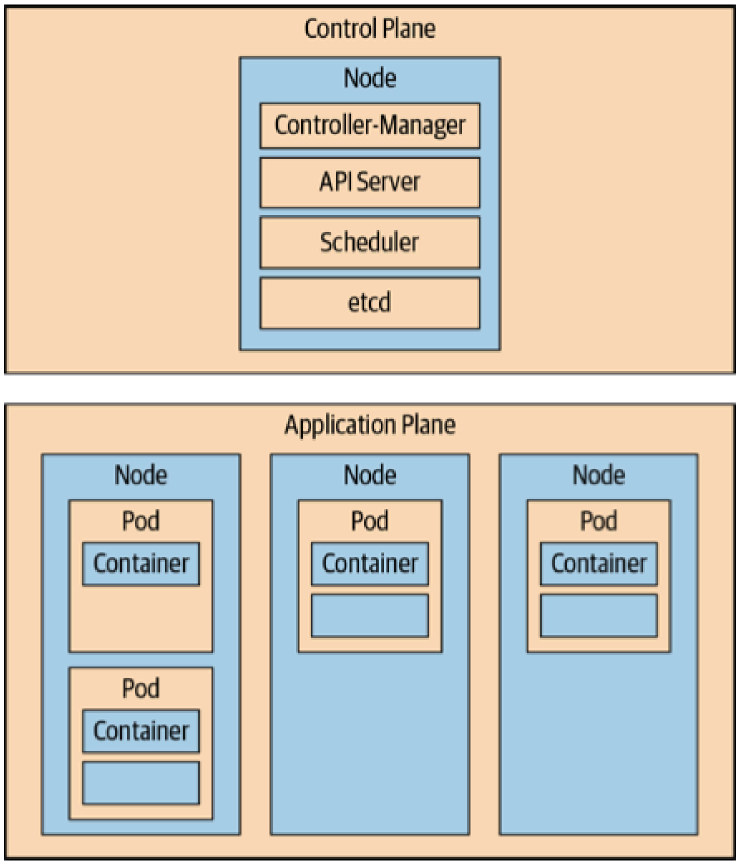
What is a Kubernetes Operator?

Whenever we deploy our application on Kubernetes we leverage multiple Kubernetes objects like deployment, service, role, ingress, config map, etc. As our application gets complex and our requirements become non-generic, managing our application only with the help of native Kubernetes objects becomes difficult and we often need to introduce manual intervention or some other form of automation to make up for it.

Operators solve this problem by making our application first class Kubernetes objects that is we no longer deploy our application as a set of native Kubernetes objects but a custom object/resource of its kind, having a more domain-specific schema and then we bake the “operational intelligence” or the “domain-specific knowledge” into the controller responsible for maintaining the desired state of this object. For example, etcd operator has made etcd-cluster a first class object and for deploying the cluster we create an object of Etcd Cluster kind. With operators, we are able to extend Kubernetes functionalities for custom use cases and manage our applications in a Kubernetes specific way allowing us to leverage Kubernetes APIs and Kubectl tooling.

Operators combine CRDs and custom controllers and intend to eliminate the requirement for manual intervention (human operator) while performing tasks like an upgrade, handling failure recovery, scaling in case of complex (often stateful) applications and make them more resilient and self-sufficient.

At a high level, Kubernetes clusters can be divided into two planes, control plane or data plane. Following diagram demonstrates the concept of two planes :



The *controllers* of the control plane implement control loops that repeatedly compare the desired state of the cluster to its actual state. When the two diverge, a controller takes action to make them match. Operators extend this behavior. An Operator continues to monitor its application as it runs, and can back up data, recover from failures, and upgrade the application over time, automatically.

Operator adds an endpoint to the Kubernetes API, called a *custom resource* (CR). A *custom resource definition* (CRD) defines a CR; it's analogous to a schema for the CR data. CRDs allow you to create domain-specific resources that correspond to your application. Using the standard Kubernetes APIs, end users interact with these resources to deploy and configure applications. An Operator is a custom Kubernetes controller watching a CR type and taking application-specific actions to make reality match the spec in that resource.

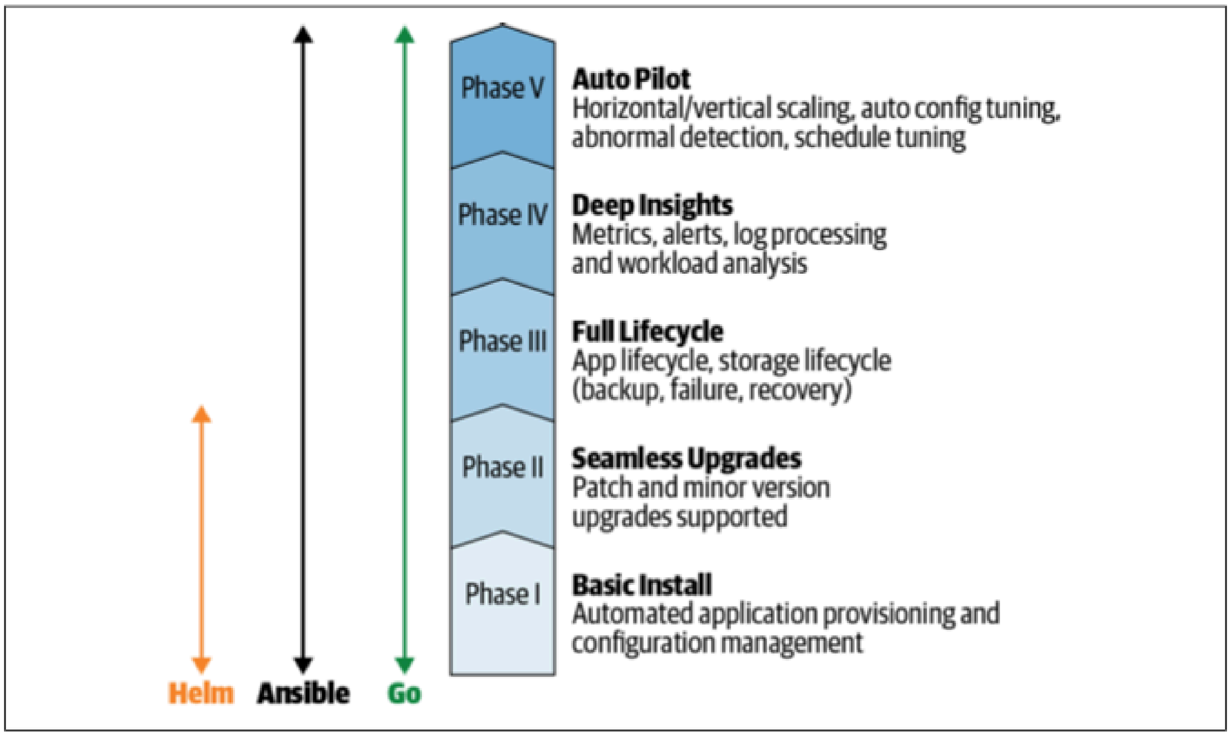
Making an Operator means creating a CRD and providing a program that runs in a loop watching CRs of that kind. What the Operator does in response to changes in the CR is specific to the application the Operator manages. The actions an Operator performs can include almost anything: scaling a complex app, application version upgrades, or even managing kernel modules for nodes in a computational cluster with specialized hardware.

Operator Framework

Operators make it easy to manage complex stateful applications on top of Kubernetes. However writing an operator today can be difficult because of challenges such as using low level APIs, writing boilerplate, and a lack of modularity which leads to duplication.

The Red Hat Operator Framework makes it simpler to create and distribute Operators. It makes building Operators easier with a software development kit (SDK) that automates much of the repetitive implementation work. The Framework also provides mechanisms for deploying and managing Operators. Operator Lifecycle Manager (OLM) is an Operator that installs, manages, and upgrades other Operators. Operator Metering is a metrics system that accounts for Operators' use of cluster resources.

The Operator Maturity Model, depicted in the following figure sketches a way to think about different levels of Operator functionality.


Phase V
Auto Pilot
Horizontal/vertical scaling, auto config tuning, abnormal detection, schedule tuning
Phase IV
Deep Insights
Metrics, alerts, log processing and workload analysis
Phase III
Full Lifecycle
App lifecycle, storage lifecycle (backup, failure, recovery)
Phase II
Seamless Upgrades
Patch and minor version upgrades supported
Phase I
Basic Install
Automated application provisioning and configuration management
Helm
Ansible
Go

The Operator SDK also provides *Adapter Operators*. Through the command-line tool, the SDK generates the code necessary to run technologies such as Helm and Ansible in an

Operator. This makes it possible to rapidly migrate the infrastructure to an Operator model without needing to write the necessary supporting Operator code. The following workflow is for a new Helm operator:

1. Create a new operator project using the SDK Command Line Interface(CLI)
2. Create a new (or add your existing) Helm chart for use by the operator's reconciling logic
3. Use the SDK CLI to build and generate the operator deployment manifests
4. Optionally add additional CRD's using the SDK CLI and repeat steps 2 and 3

While the Helm and Ansible Operators can be created quickly and easily, their functionality is ultimately limited by those underlying technologies. Advanced use cases, such as those that involve dynamically reacting to specific changes in the application require a more flexible solution i.e Go SDK.

The following workflow is for a new Go operator:

1. Define new resource APIs by adding Custom Resource Definitions(CRD)
2. Define Controllers to watch and reconcile resources
3. Write the reconciling logic for your Controller using the SDK and controller-runtime APIs
4. Use the SDK CLI to build and generate the operator deployment manifests

So definitely we should use an operator based on Golang to implement any complex features in our operator.

Timeline (Full time)

A tentative schedule of my plan is as follows.

Week	Deliverables
June 1 - June 7	• Read Documentation for HL Besu. • Read Documentation for Operator framework. • Get accustomed to the Go programming language. • Environment Setup. • Discuss the high-level design of the project with mentors.

	• Document the process
June 7 - June 21	• Prepared, automated installation of an application using operator • Document the process • Get feedback from the mentor
June 21	• Submission for 1st Quarter Evaluation
June 21 - July 7	• Add support for seamless patch and minor version upgrades. • Document the process • Get feedback from the mentor
July 14	• Submission for 2nd Quarter Evaluation
July 7 - July 14	• Add support for app/storage lifecycle, backup, failure, recovery • Document the process • Get feedback from the mentor
July 14 - July 21	• Add monitoring capabilities to the operator. • Document the process • Get feedback from the mentor
July 31	• Submission for 3rd Quarter Evaluation
July 21 - August 1	• Set up instructions for the user to deploy the operator in various scenarios • Document the process • Get feedback from the mentor
August 1 - August 21	• Code Reformatting • Error Handling • Clean up the documentation • Thorough Testing

August 21	Final Evaluations

CRDs

1. Custom Resources :

- Besu
    - Bootnodes Count
    - Validators Count
    - Member Nodes Count
- BesuNode

  Types :
    - Bootnode
    - Validator
    - Node
- Grafana
- Prometheus

Process

2. Users can provide public key and private keys of validators and bootnodes. Secrets will be created corresponding to provided keys. All the other resources required for Besu will be created by operator (eg extraData)

3. For member nodes, keys are not required

4. For bootnodes & validators, if user doesn't provide keys, then :

    - Besu will create keys
    - Besu will create secrets for each key

5. Bootnodes will be created based on secrets (for bootnode keys) and environment variables for service hosts.

7. Once bootnodes are ready, then create validators based on secrets for bootnode and validator keys and service hosts from environment variables.

8. Also, create n nodes using secretes for bootnode keys and service hosts from environment variables.

9. User can do upgrade of image and number of replicas