RAPORT 15

DEPARTAMENT: OPTOSPINTRONICA

Gavrila Vlad-Theodor

Kubernetes. CRD. Operators.

**1.Definition:**

* Kubernetes is an open-source container orchestration system for automating software deployment, scaling, and management. Originally designed by Google, the project is now maintained by the Cloud Native Computing Foundation.
* Kubernetes works with various container runtimes such as containerd and CRI-O. Its suitability for running and managing large cloud-native workloads has led to its widespread adoption in the data center. There are several distributions of this platform – from independent software vendors (ISVs), as well as cloud-hosted offerings from all major public cloud providers.

**2.Architecture:**

A diagram of a software server

Description automatically generated

* Kubernetes defines a set of building blocks (“primitives”) that collectively provide mechanisms that deploy, maintain, and scale applications based on CPU, memory, or custom metrics. Kubernetes is coupled and extensible to meet the needs of different workloads. Internal components as well as extensions and containers running on Kubernetes rely on the Kubernetes API. The platform exercises control over compute and storage resources by defining resources as objects, which can then be managed as such.
* Kubernetes follows the primary/replica architecture. Kubernetes components can be divided into those that manage an individual node and those that are part of the control plane.

**2.1.Control plane:**

The Kubernetes master node handles the Kubernetes control plane of the cluster, managing its workload and directing communication in the system. The Kubernetes control plane consists of various components, each its own process, that can run both on a single master node and on multiple masters that support high-availability clusters. Different components of Kubernetes control plane are as follows:

2.1.1.etcd:

etcd is a lightweight, distributed, key-value persistent data store (originally developed for Container Linux). It reliably stores cluster configuration data, representing the overall state of the cluster at any given time. etcd favors consistency over availability in the case of a network partition (see CAP theorem). Consistency is crucial to the correct programming and operation of services.

2.1.2.API Server:

* The API Server serves the Kubernetes API using JSON over HTTP, which provides both the internal and external interface to Kubernetes. The API server processes, validates REST requests, and updates the state of API objects in etcd, allowing clients to configure workloads and containers on worker nodes. API Server uses etcd's trace API to monitor the cluster, release critical configuration changes, or restore any cluster state divergences to the desired state as declared in etcd.
* As an example, a human operator may specify that it needs to run three instances of a particular "pod" (see below), and etcd stores this fact. If the deployment controller finds that only two instances are running (conflicting with the etcd statement), it schedules the creation of an additional instance of that pod.

2.1.3.Scheduler:

* The scheduler is an extensible component that selects the node on which to run an unscheduled pod (the basic unit of workloads to be scheduled) based on resource availability and other constraints. The scheduler tracks the resource allocation on each node to ensure that the workload is not scheduled over the available resources. To this end, the scheduler must be aware of resource requirements, resource availability, and other constraints or policy directives provided by the user, such as quality of service, affinity/anti-affinity requirements, and data locality. The scheduler's role is to match the "supply" of resources with the "demand" of the workload.
* Kubernetes allows multiple schedulers to run in a single cluster. As such, scheduler plug-ins can be developed and installed as in-process extensions to the native vanilla scheduler, running it as a separate scheduler, if they conform to the Kubernetes scheduling framework. This allows cluster administrators to extend or modify the behavior of the default Kubernetes scheduler according to their needs.

2.1.4.Controllers:

* A controller is a reconciliation loop that drives the current state of the cluster to the desired state by communicating with the API server to create, update, and delete the resources it manages (for example, pods or service endpoints).
* An example controller is a ReplicaSet controller, which handles replication and scaling by running a specific number of copies of a pod in the cluster. The controller also takes care of creating replacement bridges if the base node fails. Other controllers that are part of the Kubernetes core system include a DaemonSet controller to run exactly one pod on each machine (or a subset of machines) and a Job controller to run pods that run to completion (for example, as part of a batch job). Tag selectors are often part of the controller definition that specify the set of pods that a controller manages.
* The controller manager is a single process that manages multiple core Kubernetes controllers (including the examples described above) and is distributed as part of the standard Kubernetes installation.
* Custom controllers can also be installed in the cluster, further allowing Kubernetes behavior and API to be extended when together with custom resources (see Custom Resources, Controllers, and Operators below).

**2.2.Nodes:**

A node, also known as a worker or minion, is a machine where containers (workloads) are deployed. Each node in the cluster must run a container runtime, as well as the components mentioned below, to communicate with the primary network configuration of these containers.

2.2.1.kubelet:

* kubelet is responsible for the running state of each node, ensuring that all containers on the node are healthy. It handles the starting, stopping, and maintaining application containers organized into pods according to control plane instructions.
* kubelet monitors the state of a pod, and if it is not in the desired state, the pod reinstalls itself on the same node. The state of the node is transmitted every few seconds via heartbeat messages to the API server. Once the control plane detects a node failure, a higher-level controller is expected to notice this state change and launch pods to another healthy node.

2.2.2.Container runtime:

* A container runtime is responsible for the lifecycle of containers, including launching, reconciling, and killing containers.
* kubelet interacts with container runtimes through the Container Runtime Interface (CRI), which decouples the maintenance of core Kubernetes from the actual implementation of CRI.
* Initially, the kubelet interfaced exclusively with the Docker runtime through a "dockershim". However, from November 2020 to April 2022, Kubernetes deprecated the shim in favor of directly interfacing with the container via containerd or replacing Docker with a runtime that is compatible with the Container Runtime Interface (CRI). With the release of v1.24 in May 2022, "dockershim" was removed entirely.
* Examples of popular container runtimes that are compatible with kubelet include containerd (originally supported via Docker), rkt, and CRI-O.

2.2.3.kube-proxy:

kube-proxy is an implementation of a network proxy and load balancer and supports service abstraction along with other network operations. It is responsible for routing traffic to the appropriate container based on the IP and port number of the incoming request.

**2.3.Namespaces:**

In Kubernetes, namespaces are used to separate the resources they manage into distinct, non-intersecting collections. They are intended for use in environments with many users spread over multiple teams or projects, or even separate environments such as development, test, and production.

**2.4.Pods:**

* The basic scheduling unit in Kubernetes is a pod, which consists of one or more containers that are guaranteed to be on the same node. Each pod in Kubernetes is assigned a unique IP address in the cluster, allowing applications to use ports without the risk of conflict. Inside the bridge, all containers can refer to each other.
* A container is inside a bridge. The container is the lowest level of a microservice, which holds the running application, its libraries, and its dependencies.
* A volume, such as a local disk directory or a network disk, can be defined in a pod and made accessible to the containers that reside within it. Pods can be managed manually through the Kubernetes API, or their management can be delegated to a controller. Such volumes are also the basis for the Kubernetes features of ConfigMaps (to provide access to configuration via the container-visible file system) and Secrets (to provide access to credentials needed to securely access remote resources via providing those credentials in the file system, visible only to authorized containers).

**2.5.Workloads:**

Kubernetes supports several workload abstractions that are at a higher level than simple pods. This allows users to declaratively define and manage these high-level abstractions, rather than having to manage the individual pods themselves. Some of these abstractions, supported by a standard Kubernetes installation, are described below.

2.5.1.ReplicaSets, ReplicationControllers and Deployments:

* A ReplicaSet`s reason is to hold a strong set of duplicate pods jogging at any given time. As such, it's miles regularly used to assure the provision of a unique range of equal Pods.The ReplicaSet also can be stated to be a grouping mechanism that shall we Kubernetes hold the range of times which have been declared for a given pod. The definition of a ReplicaSet makes use of a selector, whose assessment will bring about figuring out all pods which can be related to it.
* A ReplicationController, much like a ReplicaSet, serves the equal reason and behaves in addition to a ReplicaSet, that is to make sure that there'll continually be a unique range of pod replicas as desired. The ReplicationController workload changed into the predecessor of a ReplicaSet, however changed into ultimately deprecated in desire of ReplicaSet to utilize set-primarily based totally label selectors.
* Deployments are a higher-stage control mechanism for ReplicaSets. While the ReplicaSet controller manages the size of the ReplicaSet, the Deployment controller manages what takes place to the ReplicaSet – whether or not an replace must be rolled out, or rolled back, etc. When Deployments are scaled up or down, this consequences withinside the statement of the ReplicaSet changing, and this variation withinside the declared kingdom is controlled via way of means of the ReplicaSet controller. The roles and main functionalities of deployments are:

1. Desired State Management: Deployments allow defining the desired state for applications. This includes the number of application replicas to run and the container specification (image, environment variables, volumes, etc.). Kubernetes works continuously to ensure that the current state of the cluster corresponds to the desired state defined in the deployment.
2. Scaling: Deployments allow automatic or manual scaling of the number of replicas of an application. This means you can increase or decrease the number of application instances as needed.
3. Updates and Rollback: Kubernetes facilitates application updates by changing the container specification in the deployment. This may include changing the version of the container image. Kubernetes takes care of incrementally updating containers, minimizing downtime. In case of a failed update, Kubernetes can rollback to a previous version.
4. Self-healing: Kubernetes monitors and ensures application health. If a container fails, it is automatically replaced, ensuring application availability and reliability.
5. Declarative: Configuration for deployments is declarative, meaning you specify "what" to happen (the desired state), not "how" to happen. Kubernetes takes care of the "how", thereby simplifying application management.
6. Deployment Strategies: Kubernetes supports different deployment strategies, such as "RollingUpdate" (gradual update) and "Recreate" (recreating all instances at once). These policies allow you to control how updates are performed.
7. Resource Management: Deployments can also include specifications for resource allocation (such as CPU and memory) and restart policies, ensuring efficient use of resources in the cluster.
8. Isolation and Security: By using namespaces and other security mechanisms, deployments can be isolated to limit access and increase application security.

2.5.2.StatefulSets:

* StatefulSets are controllers that put in force the features of forte and ordering among times of a pod, and may be used to run stateful programs. While scaling stateless programs is handiest to be counted of including greater walking pods, doing so for stateful workloads is harder, due to the fact the state desires to be preserved if a pod is restarted. If the utility is scaled up or down, the country can also additionally want to be redistributed.
* Databases are an instance of stateful workloads. When run in high-availability mode, many databases include the belief of a number one example and secondary times. In this case, the belief of ordering of times is important. Other programs like Apache Kafka distribute the records among their brokers; hence, one dealer isn't always like another. In this case, the belief of example forte is important.

2.5.3DaemonSets:

It is the duty of DaemonSets to guarantee that a pod is formed on each and every cluster node. Most workloads typically scale in response to a desired replica count, based on the application's objectives for availability and performance. In other cases, on the other hand, it might be required to launch a pod on each and every cluster node, increasing the total number of pods as new nodes are added and removing them as they are eliminated. This is especially useful for use cases like log collecting, ingress controllers, and storage services when the workload depends in part on the host computer or node itself.

**2.6.Services:**

An interconnected group of pods, like one tier of a multi-tier application, is referred to as a Kubernetes service. Label selectors describe the set of pods that make up a service. Kubernetes offers two methods for service discovery: Kubernetes DNS or environment variables. To load balance traffic among the pods matching the selector in a round-robin fashion, service discovery assigns a stable IP address and DNS name to the service (even when failures force the pods to shift from machine to machine), nonetheless, a service may also be made available to clients outside of a cluster (for example, to access front-end pods).

A diagram of a service

Description automatically generated

**2.7.Volumes:**

By default, filesystems within the Kubernetes container offer transient storage. This implies that all data on such containers will be lost upon a pod restart, making this type of storage extremely limited for all but the most basic applications. Perpetual storage is offered by a Kubernetes volume, which lasts for the duration of the pod. Within the pod, containers can share this storage as a shared disk space. Volumes cannot mount onto or link to other volumes; instead, they are mounted at designated mount points within the container that are determined by the pod configuration. Different containers may mount the same volume at different locations within the file system tree.

**2.8.ConfigMaps and Secrets:**

* Managing and storing configuration data, some of which may contain sensitive information, is a common application difficulty. Fine-grained characteristics or coarse-grained information such as full configuration files, like JSON or XML documents, can both be considered configuration data. To address this need, Kubernetes offers two closely related methods called ConfigMaps and Secrets, which both let configuration changes be made without necessitating an application rebuild.
* Every instance of the application to which these objects have been associated will have access to the ConfigMaps and Secrets data through the Deployment. Only when a pod on that node needs it will a node provide a Secret or a ConfigMap, which will only be kept in memory on the node. All bound Secrets and ConfigMaps are destroyed from the in-memory copy as soon as the pod that depends on them is removed.
* The pod can retrieve data from a ConfigMap or Secret in one of the following ways:

1. as environment variables that, upon container launch, kubelet will consume from the ConfigMap;
2. mounted inside a volume that is reachable through the filesystem of the container, allowing for automatic reloading without requiring the container to be restarted;

* The primary distinction between a ConfigMap and a Secret is that the latter is intended to hold private and sensitive information, while the former is not encrypted by default and necessitates further configuration to provide complete security when used within the cluster. Passwords, SSH keys, image registry credentials, certificates, and other sensitive or confidential data are frequently stored in secrets.

**2.9.Labels and selectors:**

* Any API object in the system, including pods and nodes, can have keys called labels attached to it by Kubernetes clients, which can be either internal components or users. Similarly, queries against labels that return matching objects are known as label selectors.[31] The label selectors that the service router/load balancer will use to choose which pod instances to direct traffic to can be defined when a service is defined. Therefore, it is possible to control which pods receive traffic and which do not by simply altering the labels of the pods or the label selectors on the service. This allows for the support of different deployment patterns, such as A/B testing and blue-green deployments. The flexibility for services to use implementation resources in a dynamic manner allows for a loose coupling within the infrastructure.
* For example, if an application's pods have tags for a system level (with values ​​like frontend, backend, for example) and a release\_track (with values ​​like canary, production, for example), then an operation on all backend and canary nodes can use a tag selector such as:
* tier=backend AND release\_track=canary
* In the given example:
* An application's pods are assumed to have two types of tags: one for the system level (tier) and one for the release track (release\_track).
* Tier tags can have values ​​like frontend, backend, etc., which indicate the role or function of that pod within the application.
* The release\_track tags can have values ​​like canary, production, etc., which indicate the release cycle stage the pod is in (eg test, production).
* When one wants to perform an operation on all pods that are both in the backend and in the launch canary path, one can use a tag selector that combines these two criteria. The tag selector would be expressed as the code sequence above.
* This selector indicates that only resources that have the tier tag with the backend value and at the same time the release\_track tag with the canary value will be selected. This is a powerful way to organize and interact with resources in a Kubernetes cluster, enabling complex and flexible management of different application components. Tag selectors can be used in various contexts such as network policies, replica sets, Kubernetes services, and other resources to define how these resources interact and are managed.
* Field selectors allow one to choose Kubernetes resources in the same way that labels do. In contrast to labels, the selection process is not dependent on user-defined classification, but rather on the attribute values intrinsic to the item being chosen. Field selectors metadata.name and metadata.namespace will be present on all Kubernetes objects. The object/resource type determines the other selectors that are available for use.

**2.10.Add-ons:**

Add-ons are extra functionalities for the Kubernetes cluster that are implemented as internal applications. Replication Controllers, Deployments, and other systems may be in charge of the pods. There are many of extras. Among the more significant ones are:

2.10.1.DNS:

Besides the other DNS server(s) in the system, Cluster DNS is a DNS server that provides DNS records for Kubernetes services. This DNS server is automatically included in DNS searches conducted by Kubernetes-started containers.

1. Service Discovery: Kubernetes DNS is used for service discovery within a cluster. Each Kubernetes service is automatically assigned a DNS name. When an application in a bridge wants to communicate with another service, it can use this DNS name, which simplifies the process of discovery and communication between services.

2. Integrated DNS: Kubernetes comes with an integrated DNS solution (typically CoreDNS), which is automatically configured to serve DNS requests within the cluster. CoreDNS is a flexible and extensible DNS server that can be configured to serve Kubernetes-specific DNS requests.

3. DNS Names for Services: DNS names of Kubernetes services are formed according to a standard pattern. For example, a service named my-service in the namespace my-namespace will have a DNS name of the form my-service.my-namespace.svc.cluster.local, where svc.cluster.local is the standard suffix for Kubernetes services.

4. DNS for Pods: Each pod also has a DNS name based on its IP address and namespace. This allows direct communication between pods if needed.

5. External Service Name Resolution: CoreDNS in Kubernetes can be configured to redirect requests for specific domains to specific DNS servers. This is useful for resolving service names external to the cluster.

6. Custom Configuration: Although Kubernetes provides a pre-configured DNS, cluster administrators can customize DNS settings to suit the specific needs of their applications, including adding custom policies and rules.

7. Role in High Availability: DNS in Kubernetes contributes to the high availability of services by providing a consistent and reliable mechanism for locating services and bridges regardless of network infrastructure changes.

8. Security: CoreDNS in Kubernetes can be configured with security policies to control access to services and protect communications within the cluster.

2.10.2Web UI:

This is a web-based, all-purpose Kubernetes cluster user interface. It gives administrators the ability to control and troubleshoot both the cluster and the apps that are running within it.

2.10.3.Resource monitoring:

Providing reliable application runtime and the ability to scale up or down in response to workloads means being able to monitor workload performance continuously and efficiently. Container resource monitoring provides this capability by logging metrics about containers in a central database and providing a user interface for browsing that data. The cAdvisor is a component of a slave node that provides limited metric monitoring capability. There are also complete metrics pipelines such as Prometheus that can meet most monitoring needs.

2.10.4.Cost monitoring:

Kubernetes cost monitoring applications allow cost breakdown by pods, nodes, namespaces, and tags. Three key metrics to track are daily cloud spend, cost per CPU provided and requested, historical cost allocation.

2.10.5.Cluster-level logging:

Logs should have separate storage and lifecycle, independent of nodes, pods or containers. Otherwise, node or bridge failures may cause event data to be lost. The ability to do this is called cluster-wide logging, and such mechanisms are responsible for saving container logs to a central log store with a search/browsing interface. Kubernetes does not provide native storage for log data, but many existing logging solutions can integrate with your Kubernetes cluster.

**2.11Storage:**

**PV/PVC storage spaces:**

Storage spaces in Kubernetes, known as Persistent Volumes (PV) and Persistent Volume Claims (PVC), are essential concepts for managing data storage in a Kubernetes cluster. Here is a detailed description of these two components:

1. Persistent Volumes (PV): A PV is a resource in the Kubernetes cluster that represents a unit of storage. PVs are independent of the lifecycle of individual pods and provide a way to reserve storage resources in the cluster. PVs can come from various storage sources, such as local disks, NFS, iSCSI, cloud storage (eg AWS EBS, Azure Disk, Google Persistent Disk), and others.
2. Persistent Volume Claims (PVC): A PVC is a request for storage by a user. PVCs allow users to request specific storage resources, such as capacity and access mode (eg, ReadWriteOnce, ReadOnlyMany, ReadWriteMany). PVCs are associated with PVs through the binding process, where Kubernetes looks for an available PV that matches the PVC's requirements and associates it with it.
3. Binding Process: When a PVC is created, Kubernetes searches for an available PV that matches the requirements specified in the PVC (capacity, access mode, storage classes, etc.). If a compatible PV is found, it is "bound" to the PVC. Once bound, the PV is exclusive to the PVC and cannot be used by other PVCs.
4. Storage Classes: Kubernetes allows the definition of "storage classes" that specify different storage types and policies. Storage classes can be used to automate the provisioning of new PVs based on the requirements of the PVCs. Each storage class can have its own parameters and policies, such as storage backend type, snapshot policies, etc.
5. Dynamic and Static Provisioning: Kubernetes supports both static provisioning (where cluster administrators create PVs manually) and dynamic provisioning (where new PVs are automatically created when needed based on a storage class).
6. Life Cycle: PVs have a life cycle independent of the pods that use them. A PV can be "reclaimed" (recycled), "retained" (retained) or "deleted" (deleted) after the associated PVC is deleted, depending on the configured reuse policy.
7. Multiple Access: PVs can support different access modes, such as single-writer (ReadWriteOnce) or multi-reader/multi-writer (ReadWriteMany or ReadOnlyMany) access, depending on the storage type and capabilities this one.

**2.12.Storage:**

* Software can now be made portable with the help of containers. Every package required to execute a service is included in the container. Containers are incredibly portable and simple to utilize in development because to the included file system. With little to no configuration modifications, a container can be migrated from development to test or production environments.
* In the past, Kubernetes was limited to stateless services. But since many applications need persistence for their databases, Kubernetes was developed with persistent storage in mind. One of the main issues facing cloud engineers, DevOps, and Kubernetes administrators is implementing persistent storage for containers. While containers could disappear, a growing amount of data they hold is not, so it is important to make sure the data survives in the event of a hardware malfunction or container termination. Businesses frequently discover that they require persistent storage when deploying containers using Kubernetes or containerized applications. For the purpose of supporting containers, they must offer quick and dependable storage for databases, root images, and other data.
* Apart from the landscape, further material regarding Kubernetes persistent storage has been published by the Cloud Native Computing Foundation (CNCF), which also includes a blog that aids in defining the container attached storage pattern. You can think of this strategy as utilizing Kubernetes as part of the storage system or service.
* As Kubernetes gained popularity, a new type of data storage called Container Attached Storage appeared. While providing block, file, object, and interfaces to workloads running on Kubernetes, the Container Attached Storage strategy or pattern primarily depends on Kubernetes for specific capabilities.
* Using Kubernetes extensions, including custom resource definitions, and utilizing Kubernetes itself for tasks that would typically require separate development and deployment for data management or storage are common characteristics of container attached storage. Retry logic is one example of functionality that can be provided by either Kubernetes or custom resource definitions. Retry logic is normally provided by Kubernetes, while the development and upkeep of an inventory of accessible storage media and volumes is usually provided by a custom resource definition.

2.12.1.Container Storage Interface (CSI):

In Kubernetes version 1.9, the initial Alpha version of the Container Storage Interface (CSI) was introduced. Previously, volume storage plugins were included in the Kubernetes distribution. By creating a standardized CSI, the code needed to interface with external storage systems was separated from the core Kubernetes code base. Just one year later, in version 1.2.0, the CSI feature was made Generally Available (GA) in Kubernetes.

**3.API:**

* A key component of the Kubernetes control plane is the API Server, which exposes an HTTP API that can be invoked by other parts of the cluster, as well as by end users and external components. This API is a REST API and is declarative in nature and is the same API exposed to the control plane. The API server is backed by etcd to store all records persistently.
* The REST API in Kubernetes is a fundamental way that users and systems interact with the Kubernetes cluster. This provides an HTTP interface that allows manipulation of Kubernetes resources. Through the REST API, users can create, modify, delete, and obtain resources such as pods, services, replicasets, and more.
* Key functionalities of the REST API in Kubernetes include:

1. Resources and Operations: The REST API is structured around Kubernetes resources such as Node, Pod, Deployment, Service, etc. Each resource has a set of associated operations such as GET (to get information), POST (to create a new resource), PUT (to update an existing resource), PATCH (to partially update a resource), and DELETE (to delete a resource).
2. Standard URIs: Every resource in Kubernetes has a unique URI associated with it. For example, to get all the pods in each namespace, you could use a URI like /api/v1/namespaces/{namespace}/pods.
3. Resource representation: Resources in Kubernetes are represented as JSON or YAML objects. When a user makes a request to the API, the response will typically be in JSON format, containing the details of the requested resource.
4. Authentication and Authorization: The REST API includes security mechanisms to control who can access and modify Kubernetes resources. Authentication is often done through certificates, tokens, or other mechanisms, and authorization is based on defined roles and policies.
5. Watch API: Kubernetes provides a "watch" function for resources that allows users to be notified of changes to specific resources. This is useful for tracking changes in real time.
6. Client Libraries: To facilitate interaction with the REST API, Kubernetes provides client libraries in several programming languages, such as Go, Python, Java, etc., that abstract HTTP requests and object manipulation.
7. The REST API is a core element of Kubernetes, enabling developers and operators to automate operations, integrate other systems and tools, and build complex applications on Kubernetes infrastructure. For example, kubectl, the command-line tool for interacting with the Kubernetes cluster, uses the REST API to communicate with the cluster.

**3.1.API Objects:**

* In Kubernetes, all objects serve as an "intent record" of the state of the cluster and can define the desired state that the writer of the object wants the cluster to be in. As such, most Kubernetes objects have the same set. of nested fields as follows:

spec: Describes the desired state of the resource, which can be controlled by end users or other higher-level controllers;

state: Describes the current state of the resource, which is actively updated by the resource controller.

* All objects in Kubernetes obey the same API conventions. Some of these include:

It must have the following metadata under the nested object field metadata:

namespace: is a way to organize and separate resources and objects in a cluster into isolated logical spaces. It is similar to a virtual environment that allows pooling and isolation of resources within the same Kubernetes cluster. The concept of namespace is important to manage and organize resources in a way that provides logical isolation between different applications or work environments within the cluster.

Each object in Kubernetes belongs to a specific namespace, or if no namespace is specified, it is placed in the default namespace called default.

The main uses of namespaces in Kubernetes include:

1. Logical Isolation: Namespaces allow resources to be grouped and isolated based on application, environment (such as development, test, production), or other desired organizing criteria. This helps to avoid interference between different components and applications running in the same cluster.

2. Access control: Namespaces can be used to manage user or service access and permissions to cluster resources.

3. Resource throttling: It is possible to apply resource throttling (CPU, memory, etc.) at the namespace level to ensure that resources are distributed and used appropriately between different environments or applications.

4. Tagging and organizing: Namespaces can be tagged and used to organize and filter resources according to certain criteria, making it easier to manage and monitor them.

5. Lifecycle Management: Namespaces facilitate the management and removal of resources associated with a particular application or work environment, allowing a namespace to be removed along with all of its resources.

∇ name: a string that uniquely identifies the object in the defined namespace;

∇ uid: a unique string that is able to distinguish between objects of the same name in space and time (even between deletions and recreations of the same name).

× Can be handled by another controller, which is defined in the metadata.ownerReferences field:

∇ At most one other object will be the managing controller of the controlled object, which is defined by the controller field.

× Can be garbage collected if the owner is deleted:

∇ When an object is deleted, all dependent objects can also be deleted in cascade.

**3.2.Custom resources, controllers and operators:**

* Custom Resources, or things not included in the normal Kubernetes installation, allow the Kubernetes API to be expanded. Custom Resource Definitions (CRDs), a type of resource that may be dynamically registered and unregistered without stopping or restarting an operating cluster, are used to declare these custom resources.
* Like the default controllers in the standard Kubernetes controller manager that comes pre-installed, custom controllers are another extension method that communicates with the Kubernetes API. To provide a declarative API, these controllers may communicate with custom resources. Through these resources, users can specify the desired state of the world, and the custom controller's job is to monitor changes and reconcile them.
* The term "Kubernetes Operator" is frequently used to describe the combination of custom controllers and resources. The main use cases for operators are to automate the process of capturing the intent of a human operator overseeing a service or group of services, aided by a declarative API to facilitate the automation. The human operators responsible for certain applications and services has extensive understanding of how the system should function, how to implement it, and how to respond in the event of an issue.
* Operators can handle updates of the application code along with related changes, such as database schemas or additional configuration settings, and take and restore backups of the state of the program.

**3.3.API security:**

The following are the strategies that Kubernetes defines for managing access to its API:

3.3.1.Transport security:

* Utilizing CA certificates, the Kubernetes API server listens on a TCP port that handles HTTPS traffic to impose transport layer security (TLS).
* The API server used to allow listening on both HTTP and HTTPS ports in earlier iterations of Kubernetes, with HTTP having no transport security at all. In Kubernetes v1.20, this was deprecated and finally lost support.

3.3.2Authentication:

* The Kubernetes API server requires authentication for all requests, and it supports a number of authentication techniques, some of which are mentioned below:

1. Client certificates for X.509
2. Bearer tokens
3. Tokens for service accounts used for programmatic API access

* Other Kubernetes tools like kubectl and the official Kubernetes client libraries natively support kubeconfig files, which are often used by users to identify and define cluster URL details along with the appropriate credentials.

3.3.3.Authorization:

The following modes of permission are supported by the Kubernetes API:

1. Node authorization mode: Provides a predetermined set of API queries that must be fulfilled for kubelets to operate correctly.
2. The attribute-based access control (ABAC) mode allows users to be granted access rights by combining attributes in established access control policies.
3. Role-based access control (RBAC) mode: Allocates access permissions to users according to their assigned roles, with each role specifying a set of permissible behaviors.
4. Webhook mode: Asks a REST API service whether a user is allowed to carry out a specific action.

**3.4.API Clients:**

* Kubernetes offers several API clients that are either created by Open-Source communicators or distributed by the platform creator (official clients):
* Kubernetes control plane interaction can be done via the command-line tool kubectl.
* Kubernetes maintains official client libraries for C, .NET, Go, Haskell, Java, JavaScript, Perl, Python, and Ruby.

**3.5.Cluster API:**

An API to construct, configure, and administer Kubernetes clusters programmatically has been defined using the same design concepts. We refer to this as the Cluster API. Using Infrastructure as Software, or the idea that the Kubernetes cluster infrastructure is itself a resource / object that can be handled similarly to any other Kubernetes resource, is a fundamental idea represented in the API. In a similar vein, each machine in the cluster is handled like a Kubernetes resource. The provider implementation and the core API are the two components of the API. The cloud-provider-specific functions in the provider implementation enable Kubernetes to deliver the cluster API in a way that is seamlessly integrated with the cloud-provider's resources and services.

https://en.wikipedia.org/wiki/Kubernetes

**4.Custom resorces:**

* In the Kubernetes API, a resource is an endpoint that holds a collection of a certain kind of API objects; the built-in pods resource, for instance, holds a collection of Pod objects.
* An extension of the Kubernetes API that isn't always included in a standard Kubernetes installation is called a custom resource. It is an instance of customizing a specific Kubernetes installation. But now that a lot of the fundamental features of Kubernetes are constructed with custom resources, Kubernetes is more modular.
* Through dynamic registration, custom resources can appear and disappear in an active cluster, and cluster administrators can change custom resources separately from the cluster. Similar to built-in resources like Pods, users can build and access objects of a custom resource using kubectl once it has been installed.

**4.1.Custom controllers:**

* Custom resources allow you to store and retrieve structured data on their own. A true declarative API is offered by custom resources when paired with custom controllers.
* Separation of duties is enforced by the declarative API of Kubernetes. You state what you would like your resource to be. The Kubernetes controller maintains your defined desired state and the present state of your Kubernetes objects in sync. An imperative API, on the other hand, allows you to tell a server what to do.
* Regardless of the cluster's lifespan, a custom controller can be installed and updated on an operational cluster. Although they can be used with any type of resource, custom controllers perform best when paired with other custom resources. Custom controllers and custom resources are combined in the Operator pattern. Custom controllers can be used to encode domain knowledge into a Kubernetes API extension for applications.

**4.2.Adding custom resources:**

* There are two methods that Kubernetes lets you add unique resources to your cluster:

1. CRD’s are easy to design and don't require programming.
2. Programming is necessary for API aggregation, but it gives you more control over API characteristics like data storage and version conversion.

* These two alternatives are offered by Kubernetes to accommodate varying user needs without sacrificing flexibility or ease of usage.
* Child API servers known as aggregators operate as proxies by hiding behind the primary API server. We refer to this configuration as API Aggregation (AA). The Kubernetes API appears to be comprehensive to users.
* Users can add new resource types to CRDs without having to build additional API servers. Using CRDs does not require an understanding of the Aggregation API.
* The new resources are called Custom Resources to differentiate them from built-in Kubernetes resources (like pods), regardless of how they are installed.

**4.3.Custom Resource definitions:**

* We can define custom resources by using the Custom Resource Definition API resource. A new custom resource with our specified name and schema is created when we define a CRD object. Our custom resource is served and stored via the Kubernetes API. A CRD object's name needs to be a legitimate DNS subdomain name.
* This eliminates the need for us to write our own API server to manage the custom resource, but it limits our flexibility compared to API server aggregation due to its general implementation.

**4.4.API server aggregation:**

* Typically, each Kubernetes API resource needs code to process REST queries and control object persistent storage. In addition to managing built-in resources like pods and services, the primary Kubernetes API server may also handle bespoke resources in a generic manner using CRDs.
* By creating and implementing your own API server, you can use the aggregation layer to offer customized implementations for your unique resources. For the custom resources you manage, the main API server forwards requests to your API server so that all its clients can access them.

**4.5.Selecting an approach for incorporating unique resources:**

* CRDs are simpler to operate. Flexible APIs are aggregates. Select the approach that best suits your requirements.
* CRDs are generally a suitable fit if:

1. You have several different fields.
2. You are incorporating the resource into a modest open-source project or using it within your organization (as opposed to a commercial product)

* Comparison of Ease of Use:

4.5.1.CRD vs. aggregate API:

i. No need for an appointment. A CRD controller lets users select any language. requires building binaries and images as well as programming.

ii. Since the API server manages CRDs, there is no need to launch a separate service. An extra service to develop that might not succeed.

iii. After the CRD is established, there is no further support. Any bug patches are applied as part of routine upgrades to Kubernetes Master. Periodically gathering upstream bug patches, rebuilding, and updating the aggregate API server can be necessary.

iv. There's no need to maintain different versions of your API; for instance, you can update the resource's client in tandem with the API when you have control over it. When you create an extension that you want to make public, for instance, you have to maintain many versions of your API.

4.5.2.Common characteristics:

A feature's function:

1. CRUD: Kubectl and HTTP are used by new endpoints to facilitate basic CRUD activities.
2. Watch: Using HTTP, new endpoints enable Kubernetes Watch operations
3. Discovery: Applications such as kubectl and dashboard offer list, display, and field edit functions for your resources automatically.
4. json-patch: New endpoints are PATCH-compatible with Application/json-patch+json is the content type.
5. merge-patch: PATCH is supported by new endpoints with Application/merge-patch+json is the content type.
6. HTTPS is used by new endpoints.
7. Embedded authentication: To authenticate users, access to the extension leverages the aggregation layer of the underlying API server.
8. Built-in Authorization: The authorization used by the underlying API server, such as RBAC, can be applied to extension access.
9. Finalizers: Refrain from deleting extension resources until after external cleanup has taken place.
10. Webhooks support: this enables the customization and extension of cluster behavior in both scenarios.
11. Kubectl's dashboard has the ability to show extension resources through UI/CLI display.
12. Clients are able to discriminate between fields with zero values and those that are unset.
13. Client library generation: Kubernetes offers both tools for creating type-specific client libraries and general client libraries.
14. Tags and Annotations: Standard metadata for resources, both custom and basic, that tools are able to change.

**4.6.An example of manifest for a host computer using the Component Runtime Descriptor (CRD) might look like this:**

apiVersion: hardware.example.com/v1

kind: Computer

metadata:

name: host-computer

specs:

cpu:

model: Intel i5

cores: 4

memories:

size: 8GB

type: DDR4

storage:

- type: SSD

size: 256GB

networks:

- type: Ethernet

speed: 1Gbps

operating system:

name: Ubuntu

version: "20.04"

* This manifest describes a basic computer with the following specifications:

• An Intel i5 processor with 4 cores.

• 8GB DDR4 RAM memory.

• A 256GB SSD disk.

• A 1Gbps speed Ethernet network connection.

• The installed operating system is Ubuntu 20.04.

* This manifest is only an example and can be adapted according to the specific needs of the system you wish to describe.
* Components may vary depending on the desired configuration and fields may be modified or extended to suit specific requirements.

**5.Operator pattern:**

* Operators are Kubernetes software extensions that handle apps and their components using unique resources. Kubernetes principles—particularly the control loop—are adhered to by operators.
* CoreOS, a firm, first presented the idea of operators in Kubernetes in 2016. 2018 saw the development of the concept and the release of the Operator SDK framework, which made it simpler and more effective for developers to construct and maintain operators for Kubernetes applications. Since then, the idea of operators has gained more traction and grown to be crucial to the administration of intricate applications on the Kubernetes platform.

**5.1.Motivation:**

* The operator model aims to capture the key purpose of a human operator managing a service or set of services. Human operators dealing with specific applications and services have deep knowledge of how the system should behave, how to implement it, and how to react if there are problems.
* People running workloads on Kubernetes often like to use automation to handle repeatable tasks. The operator model captures how you can write code to automate a task beyond what Kubernetes itself provides.

**5.2.An example of operator:**

* An operator can be used to automate the following tasks:

1. launching a program as needed, saving and retrieving backups of its current state, managing updates to the application's code along with associated modifications like database schemas or additional configuration options
2. apps that don't support Kubernetes APIs can find them by publishing a service to them.
3. testing the robustness of your cluster by emulating failure in all or part of it
4. selecting a distributed application's leader without conducting an internal member election process

* What specific appearance might an operator have? As an illustration, consider this:
* You can configure SampleDB, a custom resource, into the cluster.
* a deployment that verifies the operation of a pod containing the operator's controller component.
* An illustration of the operator code in a container.
* Code for the controller that asks the control plane what SampleDB resource configurations are.
* The operator consists primarily of code that instructs the API server on how to match defined resources with reality.
* The operator creates a StatefulSet to execute SampleDB, a Job to manage initial configuration, and PersistentVolumeClaims to offer durable database storage if you add a new SampleDB.
* The operator shoots a picture and ensures that the StatefulSet and Volumes are eliminated when you delete it.
* Additionally, the operator oversees routine database backups. The operator chooses when to establish a Pod that can connect to the database and take backups for each SampleDB resource. These Pods would be dependent on a ConfigMap and/or a Secret containing the credentials and database connection information.
* There would be additional supporting code since the operator wants to provide robust automation for the resource it administers. In this case, the code detects whether the database is using an outdated version and, if it is, generates Job objects that do the upgrade for you.

**5.3.Deploying operators:**

By adding the Custom Resource Definition and its corresponding Controller to your cluster, you can deploy an operator in the most popular manner. Like any other containerized application, the Controller often operates outside of the control plane. As an illustration, you can use the controller as a Deployment in your cluster.

**5.4.Using an operator:**

* After an operator is deployed, you can use it by changing, adding, or removing the resources that the operator requires. Using the operator as an example, you would first set up a deployment, and after that:

kubectl get SampleDB # locate databases that are configured

kubectl manually alter SampleDB/example-database # by changing a few parameters

* That's all there is to it! The operator will handle both implementing the modifications and maintaining the quality of the current service.

**5.5.Writing your own operator:**

* You can write your own code to provide the desired functionality if there isn't an operator in the ecosystem that does it.
* Additionally, you can use any language or runtime that can function as a client for the Kubernetes API to implement an operator, or Controller.
* The libraries and tools listed below can be used to create your own cloud native operator:

1. Operator SDK (Scris in golang)
2. Cadrul Operatorului Charmed (solutie proprietara furnizata de Canonical)
3. Java Operator SDK
4. Kopf (Cadru Pythonic al operatorului Kubernetes)
5. kube-rs
6. kubebuilder
7. KubeOps (SDK operator .NET)
8. KUDO (operator declarativ universal Kubernetes)
9. Catarg
10. Metacontroller împreună cu WebHooks pe care le implementați singur
11. Cadrul operatorului
12. operator-shell

Implementation, using KOPF, of an operator to deploy the url of a wordpress image that uses MariaDB Gallera for storage

**1.Main description of the implementation:**

* To implement a Kubernetes operator to deploy a WordPress image that uses MariaDB Galera for storage, you will need to follow several steps and use different Kubernetes resources.
* Here's a general description of the process you can follow given the uploaded image:
* Creating Docker image for WordPress and MariaDB Galera:
* It starts by creating a custom Docker image for WordPress that can communicate with a MariaDB Galera database.
* You also need a Docker image for the MariaDB Galera cluster if there isn't one already available.
* Configuring Persistent Volumes (PV) and Persistent Volume Claims (PVC):
* Defines PVs to store persistent data for WordPress and database.
* Creates PVCs that will be associated with these PVs. In Kubernetes, PVCs are used by pods to request physical storage.
* Creating a ConfigMap:
* ConfigMap is used to hold and manage application configuration such as configuration files and environment variables.
* Creating a Secret:
* Secret is used to store and manage sensitive information such as passwords and API keys.
* Defining and creating Kubernetes objects for WordPress and the database:
* Deployment: Creates a deployment for WordPress that uses the created Docker image. The deployment configuration will include references to PVCs, ConfigMap and Secret.
* Service: Creates a Kubernetes service to expose WordPress to network traffic.
* StatefulSet or Deployment for MariaDB Galera: Create a StatefulSet (recommended for databases) or Deployment for the MariaDB Galera cluster, ensuring it has persistent storage and is properly configured for replication.
* Operator Implementation:
* Write a Kubernetes operator to manage all these resources and automate WordPress deployment with MariaDB Galera.
* The operator will need to monitor the health of the cluster, perform upgrades, manage backups and provide auto-scaling if necessary.
* To write an operator, you can use the Operator Framework SDK or write your own operator using the Go client for Kubernetes. The operator will need to implement specific logic to manage the lifecycle of the WordPress application and the MariaDB Galera database within the Kubernetes cluster.

**2.Docker image implementation, in docker container for wordpress and PHP FPM, with an exposed port 80:**

To create a Docker image for WordPress running PHP-FPM with port 80 exposed, you can follow the steps below. Deployment is based on creating a Dockerfile that defines how the container should be built.

1. We create a Dockerfile: Open a text editor and create a new file called Dockerfile with the following content:

dockerfile

# We use the official WordPress image with PHP-FPM

FROM wordpress:php7.4-fpm

# We install nginx to serve the pages

RUN apt-get update && apt-get install -y nginx && rm -rf /var/lib/apt/lists/\*

# We configure nginx

COPY default.conf /etc/nginx/conf.d/default.conf

# Copy a script that will run both nginx and PHP-FPM

COPY entrypoint.sh /entrypoint.sh

RUN chmod +x /entrypoint.sh

# We expose port 80

EXHIBITS 80

# We run the entrypoint script

ENTRYPOINT ["/entrypoint.sh"]

2. We create an nginx configuration file: We create a file called default.conf to configure nginx. It will serve as a web server in front of the PHP-FPM process. Example content for default.conf:

nginx

server {

listen 80;

server\_name localhost;

root /var/www/html;

index index.php;

location / {

try\_files $uri $uri/ /index.php?$args;

}

location ~ \.php$ {

fastcgi\_pass wordpress:9000;

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

include fastcgi\_params;

}

}

3. Create an entrypoint script: This script will start both nginx and PHP-FPM. Save it as entrypoint.sh:

bash

#!/bin/bash

# We start PHP-FPM

php-fpm

# Start nginx in the foreground

nginx -g 'daemon off;'

4. Build the Docker image: Open a terminal in the directory where the Dockerfile, default.conf, and entrypoint.sh are located, and run the build command:

bash

docker build -t wordpress-custom .

5. Run the Docker container: After building the image, run the container using:

bash

docker run -d -p 80:80 --name wordpress-server wordpress-custom

You should now have a Docker container running WordPress with PHP-FPM and nginx, with port 80 exposed. You can access the application by navigating to http://localhost in a web browser.

**3.YAML file for Deployment with limited and reserved resources:**

* To create a Kubernetes Deployment YAML file for a WordPress application, you need to specify the container configuration, including resources (requests and limits) for CPU and memory. Below is an example Kubernetes manifest that defines a Deployment for WordPress with the specified resources:

apiVersion: apps/v1

kind: Deployment

metadata:

name: wordpress

tags:

app: wordpress

specs:

replied: 1

selector:

matchLabels:

app: wordpress

templates:

metadata:

tags:

app: wordpress

specs:

containers:

- name: wordpress

image: wordpress:latest

ports:

- containerPort: 80

env:

- name: WORDPRESS\_DB\_HOST

value: wordpress-db

- name: WORDPRESS\_DB\_USER

value: wordpressuser

- name: WORDPRESS\_DB\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: password

- name: WORDPRESS\_DB\_NAME

value: wordpress

resources:

requests:

memory: "512Mi"

cpu: "250m"

limits:

memory: "1024Mi"

cpu: "500m"

volumes:

- name: wordpress-persistent-storage

persistentVolumeClaim:

claimName: wordpress-pvc

---

apiVersion: v1

kind: Service

metadata:

name: wordpress

tags:

app: wordpress

specs:

ports:

- port: 80

selector:

app: wordpress

type: LoadBalancer

* In this example, deploy to a container with the image wordpress:latest. Port 80 of the container is exposed and bound to a LoadBalancer service that will expose the WordPress application to the Internet (in a cloud environment).
* In the resources section, we defined minimum requests of 512Mi of memory and 250m of CPU, which means that Kubernetes will ensure that the container will have these resources available before it is run. We have also defined limits of 1024Mi memory and 500m CPU, which means that the container will not use more resources than these limits.
* For database connection details, environment variables are used, where WORDPRESS\_DB\_PASSWORD is set by a reference to a Kubernetes Secret object that should be created separately to store passwords securely.
* This manifest can be created and applied to the Kubernetes cluster using the following commands:

kubectl create -f wordpress-deployment.yaml

**4.Creating a Persistent Volume Claim (PVC) and a Persistent Volume (PV) to store WordPress and MariaDB data:**

* To create a Persistent Volume Claim (PVC) in Kubernetes, we will need a YAML file that describes the required resources and specifications. It will allow data to be persistent even when the containers running the applications are replaced or refreshed.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: wordpress-pvc

specs:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi

storageClassName: standard

* In this example:

• apiVersion: v1 specifies the API version we are using for this object.

• kind: PersistentVolumeClaim is the type of object we are creating.

• metadata contains metadata about the PVC, such as name: wordpress-pvc, which is the name of the PVC.

• spec contains the specifications for PVC:

o accessModes: - ReadWriteOnce indicates that the volume will be able to be mounted as read-write by a single node. Other options include ReadOnlyMany and ReadWriteMany.

o resources: requests: storage: 10Gi specifies the storage space request for the PVC, in this case 10 GB.

o storageClassName: default refers to the storage class to be used for the PVC. Storage classes are configured by the cluster administrator and can have different policies such as performance, replication, and backup. The standard name is often used for the default storage class in many Kubernetes clusters, but this name may vary depending on your cluster configuration.

* After creating the YAML file, you can apply this manifest to your Kubernetes cluster using the kubectl command:

bash

kubectl apply -f pvc-wordpress.yaml

* Replace pvc-wordpress.yaml with the name of the file where you saved the PVC manifest.
* The PVC created will be managed by Kubernetes, and when a pod requests this PVC by its name, Kubernetes will try to mount it on the pod based on the availability and compliance of the PVs in the cluster. This allows WordPress to keep data on disk across pod reloads and rearrangements.

**5.Implementing Ingress and TLS:**

* To configure an Ingress in Kubernetes to direct traffic to your WordPress application and use TLS to secure the connection, you need two main components:

1. A TLS certificate - you can obtain one from a Certificate Authority (CA) or generate a self-signed certificate for testing.
2. A YAML file for Ingress - which defines traffic routing rules and references the TLS certificate.

* If you use cert-manager to manage certificates in Kubernetes, it can automate the creation and renewal of certificates for Ingress. Below is an example that assumes you have cert-manager installed and configured:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: wordpress-ingress

annotations:

cert-manager.io/cluster-issuer: "letsencrypt-prod"

nginx.ingress.kubernetes.io/force-ssl-redirect: "true"

specs:

pls:

- hosts:

- your-domain.com

secretName: wordpress-tls

rules:

- host: your-domain.com

http:

paths:

- path: /

pathType: Prefix

back end:

service:

name: wordpress

I'm wearing:

number: 80

* In this YAML file:

1. apiVersion: networking.k8s.io/v1 - refers to the version of the networking API used.
2. kind: Ingress - specifies that the object is of type Ingress.
3. metadata - contains metadata about Ingress.
4. annotations - contains various settings:
5. cert-manager.io/cluster-issuer - specifies the issuer for the TLS certificate, in this case letsencrypt-prod, which should be a predefined ClusterIssuer in cert-manager.
6. nginx.ingress.kubernetes.io/force-ssl-redirect - forces redirect traffic from HTTP to HTTPS.
7. spec - describes TLS rules and configuration for Ingress.
8. tls - specifies the TLS configuration:
9. hosts - the list of domains for which the certificate is valid.
10. secretName - the name of the Kubernetes Secret where cert-manager will store the TLS certificate.
11. rules - defines traffic routing rules to backend services.

* To apply this file, save it as wordpress-ingress.yaml and run:

kubectl apply -f wordpress-ingress.yaml

* To create a manual Secret for a TLS certificate, use the command:

kubectl create secret tls wordpress-tls --cert=path/to/tls.crt --key=path/to/tls.key

**6.Service implementation:**

* To expose your WordPress application inside a Kubernetes cluster, you will need to create a Service object. The service will define how other applications in the cluster or external users can access your WordPress application. Here is an example YAML file for a Service that exposes WordPress:

apiVersion: v1

kind: Service

metadata:

name: wordpress-service

tags:

app: wordpress

specs:

type: ClusterIP

ports:

- port: 80

targetPort: 80

protocol: TCP

selector:

app: wordpress

* In this file:

• apiVersion: v1 indicates the version of the API used to create the Service.

• kind: Service defines the Kubernetes object type as a Service.

• metadata contains metadata for the Service, including name and labels.

• spec specifies the Service configuration:

o type: ClusterIP indicates that this is a Service that will be accessible internally within the cluster. Other options may include NodePort or LoadBalancer for external access.

o ports defines the list of ports for which the Service will proxy. port is the port that the Service will expose, and targetPort is the port of the container that the Service will forward traffic to.

a selector that specifies which Pods will be exposed through this Service. Matches labels set on Pods.

* Save the YAML file and apply it to your Kubernetes cluster using the following command:

kubectl apply -f wordpress-service.yaml

* MariaDB Galera - The use of the MariaDB database with Galera for cluster replication is specified:
* To run a MariaDB Galera cluster in Kubernetes, we will need to configure several components. Essentially, they include:

1. A StatefulSet or Deployment for MariaDB Galera nodes.

2. A PersistentVolume (PV) or PersistentVolumeClaim (PVC) for data persistence.

3. A Service to expose and balance connections to MariaDB nodes.

Before we begin, you need to ensure that you have persistent storage configured in your Kubernetes cluster so that data survives container restarts and rescheduling.

Here is a simplified YAML example that configures a MariaDB Galera cluster using StatefulSet in Kubernetes. This example does not include all possible details, but provides a basic configuration to get you started.

apiVersion: v1

kind: Service

metadata:

name: mariadb-galera

specs:

ports:

- port: 3306

clusterIP: None

selector:

app: mariadb-galera

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: mariadb-galera

specs:

serviceName: "mariadb-galera"

replied: 3

selector:

matchLabels:

app: mariadb-galera

templates:

metadata:

tags:

app: mariadb-galera

specs:

containers:

- name: mariadb

image: mariadb:10.4

args:

- "--wsrep-new-cluster"

- "--wsrep-on=ON"

- "--wsrep-sst-method=rsync"

env:

- name: MYSQL\_ROOT\_PASSWORD

valueFrom:

secretKeyRef:

name: mariadb-secret

key: password

- name: MYSQL\_DATABASE

value: wordpressdb

- name: MYSQL\_USER

value: wordpressuser

- name: MYSQL\_PASSWORD

valueFrom:

secretKeyRef:

name: mariadb-secret

key: user-password

ports:

- containerPort: 3306

volumeMounts:

- name: date

mountPath: /var/lib/mysql

volumeClaimTemplates:

- metadata:

name: date

specs:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 10Gi

In this example:

* Un **Service headless** este creat pentru a controla rețeaua între nodurile Galera.
* Un **StatefulSet** este folosit pentru a menține identitatea fiecărui nod Galera. StatefulSet este ideal pentru baze de date și alte aplicații care necesită o stocare persistentă unică pentru fiecare instanță.
* Variabilele de mediu precum MYSQL\_ROOT\_PASSWORD, MYSQL\_DATABASE, MYSQL\_USER și MYSQL\_PASSWORD sunt setate folosind referințe la un Kubernetes Secret care trebuie creat separat pentru a stoca aceste valori în mod sigur.
* Directorul de date al MariaDB este montat la o **PersistentVolume** prin **volumeClaimTemplates** care asigură fiecărui nod din clusterul Galera spațiul său de stocare persistent.

Pentru a crea secretul necesar pentru variabilele de mediu mariadb-secret, folosim comanda:

kubectl create secret generic mariadb-secret \

--from-literal=password='your-root-password' \

--from-literal=user-password='your-user-password'

După ce am configurat YAML-ul și ați creat Secret-ul, puteți aplica fișierul în cluster-ul dvs. Kubernetes folosind:

kubectl apply -f mariadb-galera.yaml

**7.CRDs (namespace) from Mariadb database:**

To implement a custom operator in Kubernetes using KOPF (Kubernetes Operator Pythonic Framework) to manage the deployment of a WordPress application that uses MariaDB Galera for storage, we must first define a Custom Resource Definition (CRD) to describe the specific resource which our operator will manage. In this case, we can create a CRD for a MariaDB Galera database instance.

This CRD will serve as a "template" for creating database instances through our operator:

apiVersion: apiextensions.k8s.io/v1

kind: CustomResourceDefinition

metadata:

name: mariadbgaleraclusters.mycompany.com

specs:

group: mycompany.com

names:

kind: MariaDBGaleraCluster

listKind: MariaDBGaleraClusterList

plural: mariadbgaleraclusters

singular: mariadbgaleracluster

scope: Namespaced

versions:

- name: v1

served: true

storage: true

scheme:

openAPIV3Scheme:

type: object

features:

specs:

type: object

features:

replied:

type: integer

minimum: 3

images:

type: string

databases:

type: object

features:

name:

type: string

user:

type: string

passwordSecret:

type: object

features:

name:

type: string

keys:

type: string

storage:

type: object

features:

size:

type: string

storageClassName:

type: string

required: [replicas, image, database, storage]

required: [spec]

This CRD defines a new resource called MariaDBGaleraCluster in the mycompany.com group with version v1. The schema specifies some essential properties to configure a MariaDB Galera instance, such as the number of replicas, container image, database details (including a password secret), and storage configuration.

After we define the CRD, you need to apply it to your Kubernetes cluster:

kubectl apply -f mariadbgaleracluster-crd.yaml

Once the CRD is created, you can start developing the KOPF handler that will listen for events to create, update, or delete MariaDBGaleraCluster resources and perform the appropriate actions, such as deploying the MariaDB Galera cluster to Kubernetes.

As for the actual implementation of the operator with KOPF, it will involve writing Python code that interacts with the Kubernetes API to manage the lifecycle of MariaDB Galera instances as specified in the MariaDBGaleraCluster resources. This is a more advanced task and requires a solid understanding of Kubernetes APIs as well as the KOPF framework.

**8.The yaml file for User Secret from MariaDB database:**

To manage MariaDB Galera database access within an operator deployment with KOPF, it is essential to store access credentials in a secure manner. Kubernetes provides the Secret object for this purpose, which can be used to store and manage sensitive information such as passwords, OAuth tokens, and ssh keys.

Here is an example YAML file to create a Kubernetes Secret, which contains the user and password for accessing a MariaDB database instance. This Secret can then be referenced in resource definitions (for example, in StatefulSet for MariaDB Galera or in Deployment for WordPress), thus ensuring that sensitive data is not directly exposed in configuration files or source code.

apiVersion: v1

kind: Secret

metadata:

name: mariadb-user-secret

namespace: default

type: Opaque

date:

username: d29yZHByZXNzdXNlcg== # Base64 encoded value for "wordpressuser"

password: cGFzc3dvcmQxMjM= # Base64 encoded value for "password123"

To create this Secret, replace the username and password values ​​with the desired values, encoded in Base64. You can use the command echo -n 'yourtext' | base64 to get the encoded value of your text.

For example, to hardcode the text "wordpressuser", use:

echo -n 'wordpressuser' | base64

The result will be d29yZHByZXNzdXNlcg==, which is the value you need to include in your YAML file.

After we've prepared the YAML file, apply it using kubectl:

kubectl apply -f mariadb-user-secret.yaml

In examples of Kubernetes resources that require database access (such as Deployment for WordPress App or StatefulSet for MariaDB Galera), we will reference this Secret to inject credentials into containers. For example, in Deployment for WordPress, we can use environment variables to specify the username and password:

env:

- name: WORDPRESS\_DB\_USER

valueFrom:

secretKeyRef:

name: mariadb-user-secret

key: username

- name: WORDPRESS\_DB\_PASSWORD

valueFrom:

secretKeyRef:

name: mariadb-user-secret

key: password

This ensures that our application can access the database using securely stored credentials without exposing sensitive information.

**9.Kopf operator configuration that orchestrates and keeps the state of the services:**

Implementing a Kubernetes operator with KOPF to manage the deployment of a WordPress image that uses MariaDB Galera for storage involves creating a Python script to define the operator logic. The handler will listen for specific events related to Kubernetes resources defined by you (such as creating, updating, or deleting a Custom Resource (CR) instance for your WordPress deployment) and perform the necessary operations to manage the lifecycle of these instances as specified .

Step 1: Install KOPF

To get started, you need to install KOPF and the Kubernetes Python clients:

pip install kopf Kubernetes

Step 2: We define the Operator Logic

Creating the wordpress\_operator.py file will contain the operator logic. The handler will respond to events to create the resources needed to run WordPress and MariaDB Galera.

import kopf

from kubernetes import client, config

# Configuring the Kubernetes client

config.load\_kube\_config()

@kopf.on.create('mycompany.com', 'v1', 'wordpresssites')

def create\_fn(spec, \*\*kwargs):

name = kwargs.get('name')

namespace = kwargs.get('namespace', 'default')

# Data for MariaDB

db\_user = spec.get('dbUser')

db\_password = spec.get('dbPassword')

db\_name = spec.get('dbName')

# Data for WordPress

wp\_image = spec.get('wpImage', 'wordpress:latest')

# Create Secret for the database

secret\_data = {

"username": base64.b64encode(db\_user.encode()).decode(),

"password": base64.b64encode(db\_password.encode()).decode(),

}

secret = client.V1Secret(

metadata=client.V1ObjectMeta(name=f"{name}-db-secret"),

date=secret\_date,

type="Opaque",

)

# Creating StatefulSet for MariaDB Galera

create\_mariadb\_statefulset(namespace, name, db\_user, db\_password, db\_name)

# Creating Deployment for WordPress

create\_wordpress\_deployment(namespace, name, wp\_image)

client.CoreV1Api().create\_namespaced\_secret(namespace, secret)

# Logic for creating the other resources (PVC, Deployment, Service, etc.)

def create\_mariadb\_statefulset(namespace, name, db\_user, db\_password, db\_name):

statefulset = V1StatefulSet(

metadata=V1ObjectMeta(

name=f"{name}-mariadb",

namespace=namespace

),

spec=V1StatefulSetSpec(

serviceName=f"{name}-mariadb",

replicas=3,

selector={'matchLabels': {'app': f"{name}-mariadb"}},

template=V1PodTemplateSpec(

metadata=V1ObjectMeta(labels={'app': f"{name}-mariadb"}),

spec={

'containers': [

V1Container(

name="mariadb",

image="mariadb:10.5",

ports=[V1ContainerPort(container\_port=3306)],

env=[

V1EnvVar(name="MYSQL\_ROOT\_PASSWORD", value=db\_password),

V1EnvVar(name="MYSQL\_DATABASE", value=db\_name),

V1EnvVar(name="MYSQL\_USER", value=db\_user),

V1EnvVar(name="MYSQL\_PASSWORD", value=db\_password)

],

volumeMounts=[V1VolumeMount(mount\_path="/var/lib/mysql", name="data")]

)

]

}

),

volumeClaimTemplates=[

V1PersistentVolumeClaim(

metadata=V1ObjectMeta(name="data"),

spec=V1PersistentVolumeClaimSpec(

accessModes=["ReadWriteOnce"],

resources=V1ResourceRequirements(

requests={"storage": "10Gi"}

)

)

)

]

)

)

def create\_wordpress\_deployment(namespace, name, wp\_image, db\_name, db\_user\_secret\_name):

deployment = V1Deployment(

metadata=V1ObjectMeta(name=f"{name}-wordpress", namespace=namespace),

spec=V1DeploymentSpec(

replicas=2,

selector=V1LabelSelector(matchLabels={"app": f"{name}-wordpress"}),

template=V1PodTemplateSpec(

metadata=V1ObjectMeta(labels={"app": f"{name}-wordpress"}),

spec={

'containers': [

V1Container(

name="wordpress",

image=wp\_image,

ports=[V1ContainerPort(container\_port=80)],

env=[

V1EnvVar(

name="WORDPRESS\_DB\_HOST",

value="mariadb-service:3306"

),

V1EnvVar(

name="WORDPRESS\_DB\_NAME",

value=db\_name

),

V1EnvVar(

name="WORDPRESS\_DB\_USER",

valueFrom=V1EnvVarSource(

secretKeyRef=V1SecretKeySelector(

name=db\_user\_secret\_name,

key="username"

)

)

),

V1EnvVar(

name="WORDPRESS\_DB\_PASSWORD",

valueFrom=V1EnvVarSource(

secretKeyRef=V1SecretKeySelector(

name=db\_user\_secret\_name,

key="password"

)

)

),

]

)

]

}

)

)

)

client.AppsV1Api().create\_namespaced\_deployment(namespace=namespace, body=deployment)

return {'message': f"WordPress site '{name}' created with MariaDB Galera!"}

This basic example shows how to respond to creation events for custom resources such as wordpresssites under the mycompany.com group. When such a resource is created, the operator will initiate the creation of a Kubernetes secret to store the database credentials and proceed with the rest of the required resources.

Step 3: Run the Operator

To run the operator, run the script using KOPF:

kopf run wordpress\_operator.py

KOPF will start listening to Kubernetes events for the specified resources and execute the defined functions when the corresponding events are fired.