Research Project - Software Engineering 2

Marco Molè

AA 2022/2023

Contents

1 Introduction

Kubernetes, also known as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, and management of

containerized applications. Originally developed by Google and later donated to the Cloud Native Computing Foundation, Kubernetes is now widely used in cloud computing environments to manage the deployment of containerized applications across a cluster of nodes.

At its core, Kubernetes operates by organizing containers into logical units called pods, which are then deployed and managed as a single entity. These pods are designed to be highly portable and can be deployed across a range of cloud and on-premises environments, enabling organizations to build and deploy applications in a way that is both scalable and cost-effective.

One of the key benefits of Kubernetes is its ability to automate many of the tasks associated with managing containerized applications, including load balancing, scaling, and failover. This automation not only helps to reduce the operational overhead of managing large-scale container deployments but also ensures that applications are always available and running smoothly.

Overall, Kubernetes represents a significant advancement in the field of container orchestration, providing developers and operations teams with a powerful tool for managing containerized applications at scale. As such, it has become an essential technology for organizations looking to leverage the benefits of containerization and cloud computing to build more agile, scalable, and resilient applications.

The Kubernetes configuration language is expressed in YAML or JSON format, and it consists of a set of declarative statements that describe the desired state of the Kubernetes resources. These statements include specifications for the containers that run within the resources, as well as other settings such as environment variables, ports, and volumes.

In this report I'll explore how the declarative language is used to define the main kubernetes objects. Then I'll propose a mechanism for simplifying the process of writing correct configurations of Pods.

1.1 How kubernetes works

At its core, Kubernetes is built around the concept of a cluster, which is a group of machines (called nodes) that work together to run containerized applications. The Kubernetes control plane, which is responsible for managing the cluster, consists of several components that work together to provide a robust and scalable platform for deploying and managing containerized applications.

One of the key concepts in Kubernetes is the pod, which is the smallest deployable unit in the system. A pod is a logical host for one or more containers, and it provides a shared environment for those containers to run in. Each pod has a unique IP address and hostname, which makes it easy to communicate with other pods in the same cluster.

The Kubernetes control plane consists of several components, including the API server, etcd, the scheduler, and the controller manager. The API server provides a RESTful API that users can use to interact with the Kubernetes cluster. Etcd is a distributed key-value store that stores the configuration data for the cluster, such as the desired state of the system. The scheduler is responsible for assigning pods to nodes in the cluster based on various criteria, such as resource availability and affinity. Finally, the controller manager is responsible for ensuring that the current state of the cluster matches the desired state.

Kubernetes also provides several key features that make it easy to manage containerized applications, such as service discovery, load balancing, and auto-scaling. Service discovery allows applications to easily find and communicate with other services in the same cluster. Load balancing ensures that traffic is distributed evenly across all instances of a service, while auto-scaling makes it easy to automatically scale the number of instances up or down based on demand.

2 Main components

In this section I'll provide a more detailed explanation of the most important components.

2.1 Pods

In Kubernetes, a pod is the smallest deployable unit that can be created, scheduled, and managed. Pods can contain one or more containers, which share the same network namespace and can communicate with each other using inter-process communication (IPC). The declarative language of Kubernetes is used to define pods in a Kubernetes manifest file, which is a YAML or JSON file that describes the desired state of the pod.

To define a pod using Kubernetes' declarative language, the manifest file must include the following information:

1. metadata: This includes information such as the name and labels of the pod.
2. spec: This includes information about the desired state of the pod, such as the containers it should contain, the image(s) to use, and the commands to run. In this section, you can also specify the ports to expose, any environment variables to set, and any volumes to mount.

For example, the following YAML manifest file defines a pod with one container:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
spec:
  containers:
  - name: my-container
    image: nginx
    ports:
    - containerPort: 80
```

In this example, the manifest file specifies that the pod should be named "my-pod" and labeled with "app: my-app". It also specifies that the pod should contain one container named "my-container" that runs the Nginx image and exposes port 80.

Once the manifest file has been created, it can be applied to the Kubernetes cluster using the *kubectl apply* command. Kubernetes will then compare the desired state specified in the manifest file with the actual state of the cluster and make any necessary changes to ensure that the desired state is achieved.

2.2 Deployments

A deployment is an object that defines the desired state of a set of replicated pods. Deployments are a higher-level abstraction that enables declarative updates to the desired state of the pod set, allowing for easier management of the underlying resources.

A Kubernetes deployment is responsible for creating and managing a ReplicaSet, which is a Kubernetes object that ensures a specified number of

identical replicas of a pod are running at any given time. Deployments provide a way to manage the scaling and rolling updates of pods, by automating the creation and deletion of replica sets as necessary. The deployment object also provides rollback and pause/resume functionality, allowing for more fine-grained control over the lifecycle of the pod set.

When creating a Kubernetes deployment, a user specifies the desired number of replicas, the container images and configurations, and the update strategy. The deployment controller then uses this information to ensure that the desired state of the pod set is achieved and maintained. If the actual state of the pod set deviates from the desired state, the deployment controller automatically performs rolling updates, scaling, or deletion of pods as necessary.

Deployments are a critical component of the Kubernetes platform, as they enable users to manage containerized applications at scale with ease. By abstracting away the complexity of managing individual pods, Kubernetes deployments provide a higher-level interface that is more intuitive for users and reduces the potential for human error. With Kubernetes deployments, users can confidently manage their applications in a highly available and fault-tolerant manner.

To define a deployment using Kubernetes' declarative language, the manifest file must include the following information:

1. metadata: This includes information such as the name and labels of the deployment.
2. spec: This includes information about the desired state of the deployment, such as the number of replicas, the selector that identifies the pods managed by the deployment, and the configuration of the containers. For example, the following YAML manifest file defines a deployment with 3 replicas:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
```

```
template:
  metadata:
    labels:
      app: my-app
  spec:
    containers:
    - name: my-container
      image: nginx
      ports:
      - containerPort: 80
```

In this example, the manifest file specifies that the deployment should be named "my-deployment" and should include three replicas. It also specifies that the deployment should manage pods labeled with "app: my-app" and should create pods with one container running the Nginx image and exposing port 80.

2.3 Services

A Service is an abstraction that enables a stable IP address and DNS name to be assigned to a set of Pods. In essence, it serves as a mechanism for defining a logical grouping of Pods and providing a single point of access to them, regardless of the specific Pod that a client may be communicating with at any given time. Services are typically used to enable load balancing and horizontal scaling of application components running within a Kubernetes cluster.

A Service is defined through a declarative YAML configuration file, which specifies the selector label that matches the Pods to be included in the Service, as well as the port(s) that the Service should expose. When a Service is created, Kubernetes automatically assigns it a cluster-unique IP address and creates an associated DNS entry. Requests sent to the Service's IP address and port are then automatically routed to one of the available Pods that match the Service's selector, based on a simple round-robin algorithm. This allows clients to access the application components running within a Kubernetes cluster in a consistent and scalable manner, without needing to know the specific IP addresses or hostnames of individual Pods.

Service can be configured in several ways depending on the requirements of the application or workload. Some of the common configurations include:

1. ClusterIP: This is the default configuration for a Service and provides

a stable IP address and DNS name within the cluster. The Service is only accessible from within the cluster and is not exposed to the external network.

2. NodePort: This configuration exposes the Service on a static port on each node in the cluster. This allows the Service to be accessed from outside the cluster using the node's IP address and the static port number.

3. LoadBalancer: This configuration creates a load balancer in a cloud provider's network, which distributes incoming traffic to the Service across multiple nodes. This is typically used when external traffic needs to be distributed across multiple nodes or when a high degree of availability is required.

4. ExternalName: This configuration provides a way to create a Service that simply maps to an external DNS name. This is useful when an application needs to access an external resource, such as a database or web service, using a stable DNS name.

In addition to these basic configurations, Services can also be combined with other Kubernetes features such as selectors, labels, and endpoints to provide more advanced functionality. For example, Services can be used in conjunction with Ingress controllers to provide an HTTP(S) load balancer that routes traffic based on the URL path or hostname of incoming requests.

2.4 Volumes

A volume is a directory accessible to containers in a Pod. A volume can be used to store data that needs to be shared between containers, or to persist data beyond the lifetime of a container. Volumes in Kubernetes are defined using a declarative configuration file in YAML format.

There are several types of volumes that can be used in Kubernetes, including hostPath volumes, emptyDir volumes, configMap volumes, secret volumes, and persistent volumes. HostPath volumes mount a directory from the host into the container, while emptyDir volumes are created when a Pod is scheduled on a node and deleted when the Pod is terminated. ConfigMap volumes and secret volumes provide a way to inject configuration data or sensitive information into a container, respectively. Persistent volumes are used to store data beyond the lifetime of a Pod, and can be dynamically provisioned or statically created by an administrator.

To define a volume in Kubernetes, the YAML configuration file would include a volumes section with the desired volume type and configuration options. For example, to define a hostPath volume that mounts the /data directory from the host into a container, the YAML file would look like:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - name: my-volume
      mountPath: /data
  volumes:
  - name: my-volume
    hostPath:
      path: /data
```

This configuration file specifies that a hostPath volume named *my-volume* should be mounted at */data* inside the container. The *volumeMounts* field in the container specification specifies which volume to mount, and where to mount it.

3 My contribution

3.1 The Problem

When writing the specification of a Pod one can specify the container images to run and from which registry pull it from. Container registries are software repositories that store and distribute container images. A container registry is typically used by developers and DevOps teams to store and manage container images, which can be easily shared across different environments, such as development, staging, and production. Container registries provide features like versioning, access control, and image scanning to ensure that container images are secure, reliable, and up-to-date.

Most often, images require configuration through the use of environment

variables. The documentation of these variables is generally found on the registry itself, to be consulted by the developer when writing the specification of the Pod.

Configuration errors could be caused by human error, either neglecting to read the documentation, or by misinterpreting it, or by the lack of documentation of the image itself. Misconfiguration could lead to the application not working as expected, or not working at all. It also could be a security risk, since a misconfiguration of security parameters could lead to the use of default credentials, or to the use of a non secure protocol.

An erroneous configuration of a container is hard to debug, since the point of failure may not be immediately recognizable due to the highly level of coupling of microservices architecture. There is also an additional element of complexity due to the fact that the application is running inside a virtualized environment, and the failure could be caused by a misconfiguration of the container itself.

The manifestations of these errors are mainly two: the application does not work as expected, or it does not work at all. The first case happens when the misconfiguration is not critical, and the application is still able to run, but with some functionalities not working as expected. For example, the application could be running with some default configuration that does not expose all the functionalities of the application.

The second case happens when the misconfiguration is critical, and the application is not able to run at all. For example, no credentials are provided to the application, and the application is not able to connect to the database.

In both cases, the developer has to debug the application to find the cause of the error, and then fix it. This is a time consuming process, since the developer has to find the cause of the error, and then fix it.

It could be also a potentially costly error, since most of K8s clusters exist on a pay-per-use cloud environment.

There is no automatic mechanism that inform the developer if the configuration they have written is correct, or at least if it satisfies some minimum configuration requirements.

3.2 A Possible Solution

This mechanism could be included in the development environment, for example as a LSP language server. The solution I propose is to provide a mechanism that allows the developer to check if the configuration they have written satisfies some minimum configuration requirements, defined by the maintainer of the image. This minimum configuration schema is stored in the registry, and is accessible by the developer's IDE when writing the specification of the Pod. The developer can then check if the configuration they have written satisfies the minimum configuration requirements, and if not, the IDE will inform the developer of the missing configuration options, and the type of the configuration options.

The Language Server Protocol (LSP)[1] is a communication protocol that enables the integration of programming language analyzers, such as code editors, with language servers that provide language-specific features such as code completion, error detection, and refactoring. The protocol standardizes the exchange of information between the language server and the client, allowing for interoperability between different code editors and programming languages. This can help developers work more efficiently by providing consistent, language-specific tools across different editing environments.

3.2.1 Schema of the configuration options

The schema of the configuration options is a JSON object that specifies the configuration options of the image, and the type of each configuration option. The schema should contain information about the mandatory configuration options, and the facultative configuration options. The schema should also contain information about the default value of the facultative configuration options.

[1] https://microsoft.github.io/language-server-protocol/

The format of the schema could be generalized to this structure:

```
{
  "image" : "name of the image:version",
  "example_of_mandatory_configuration_option" : {
    "mandatory" : true,
    "type":"type of the configuration option",
  },
  "example_of_facultative_configuration_option" : {
    "mandatory" : false,
    "type":"type of the configuration option",
    "default": "default value of the configuration option"
  },
}
```

The type of the configuration option could be one of the following: string, number, path (ie a string that represents a path), boolean etc. More types definition could be added if needed.

An interesting case is the case of the default value of a facultative configuration option. The default value could be a string, or it could be the value of another configuration option. In the latter case, the value of the default value should be a string that represents the name of the configuration option, preceded by the *$* symbol. The *$* symbol is used to express that the default value is the value of another configuration option.

This could be useful in the case of a configuration option that is facultative, but that has a default value that depends on the value of another configuration option. This is the case of the configuration options of the postgres image, where the default value of the POSTGRES_DB configuration option is the value of the POSTGRES_USER configuration option.

```
{
  "image": "postgres:latest",
  "POSTGRES_PASSWORD" : {
    "mandatory" : true,
    "type":"string"
  },
  "POSTGRES_USER" : {
    "mandatory" : false,
    "type":"string",
    "default": "postgres"
```

```
  },
  "POSTGRES_DB" : {
    "mandatory" : false,
    "type":"string",
    "default": "$POSTGRES_USER"
  }
}
```

Another case is when the default value of a facultative configuration option is a random and auto-generated value. In this case, the value of the default value is written as "!random". An example of this is the DOCKER_INFLUXDB_INIT_ADMIN_TOKEN variable, which can be found in the example section.

3.2.2 Protocol for the communication between the IDE and the registry

The registries should offer an HTTP endpoint that replies with the configuration schema of the requested image, to be consumed by the IDE.

The flow of the interaction between the parties is shown in the following sequence diagram.

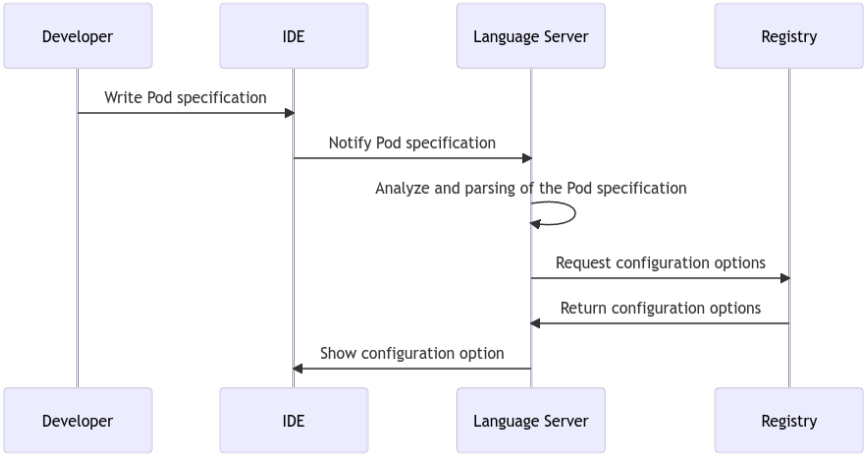

Figure 1: Sequence Diagram of the interaction between the parties

The developer writes the specification of the Pod in a YAML file, and the IDE sends a request to the registry to retrieve the configuration options of the specified image.

This solution provides very little overhead on the registry, since the registry only has to reply to the request of the IDE with the configuration schema of the requested image. The registry does not have to store any additional information, since the configuration schema is stored in the registry as a JSON object.

This information is then checked against the PodSpec.containers.env object, which is the object where all the environment variables of the specific containers are specified. The results of this analysis are then shown to the user as warnings or error marks in the IDE.

Recall how kubernetes objects are structured. The PodSpec.containers.env object is a list of key-value pairs. The key is the name of the environment variable, and the value is the value of the environment variable. Environment variables must be defined as a string.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:  # PodSpec
  containers: # PodSpec.containers
  - name: nginx
    image: nginx
    env: # PodSpec.containers.env
    - name: ENVIRONMENT
      value: "production"
    - name: LOG_LEVEL
      value: "INFO"
```

3.2.3 Generation of the configuration schema

The current status of documentation of the registries is that it is not machine readable. The documentation is written in a human readable format, and it is not possible to automatically generate the configuration file from the documentation. There is also a lack of standardization of the documentation format, since each registry has its own format, and there is no standard way to retrieve the documentation of an image. Sometimes the documentation is in the image description of the registry, sometimes it's on the website of the application, sometimes it is in the README of the image repository.

A possible way to ease the generation of the configuration schema is to ask

the image maintainer to provide the configuration schema when they push the image to the registry. This could be done by providing a standard way to specify the configuration schema in the image description of the registry.

3.2.4 Conclusion

The advantages of implementing this "type checking" mechanism in the the development environment are:

1. Highly portable to multiple types of development environment, due to the high adoption of the LSP protocol.
2. Easily extendible with more code analysis features and eventually adaptable to future changes in the language. An possible code analysis feature could be the check of existence of Pods with a certain label when defining a Service.
3. Flexible, since the declarative language is only one of the ways to configure kubernetes objects (Secrets and ConfigMap can be stored on external providers). The LSP server could be configured to analyze codebases that rely on third party External Secret Store providers.

In the last decade we have seen the rise of a new figure in the software engineering space: the DevOps engineer. Its role is to implement and maintain the ever more complex systems development life cycle and deployment. The one who deploys software is not the one who wrote the application. This solution could facilitate the knowledge exchange between the two parties and reduce the possibilities of errors when deploying a big system with a lot of different components.

3.3 Examples

These are examples based on the most popular images taken from the public registry of Docker[2].

3.3.1 mongoDB

```
{
  "image": "mongo:latest",
  "MONGO_INITDB_ROOT_USERNAME" : {
    "mandatory" : true,
```

[2] https://hub.docker.com

```
    "type":"string"
  },
  "MONGO_INITDB_ROOT_PASSWORD" : {
    "mandatory" : true,
    "type":"string"
  }
}
```

3.3.2 nginx

```
{
  "image": "nginx:latest",
  "NGINX_ENVSUBST_TEMPLATE_DIR " : {
    "mandatory" : false,
    "type":"path",
    "default" : "/etc/nginx/templates"
  },
  "NGINX_ENVSUBST_TEMPLATE_SUFFIX " : {
    "mandatory" : false,
    "type":"string",
    "default": ".template"
  },
  "NGINX_ENVSUBST_OUTPUT_DIR " : {
    "mandatory" : false,
    "type":"path",
    "default": "/etc/nginx/conf.d"
  },

}
```

3.3.3 InfluxDB

```
{
  "image": "influxDB:latest",
  "DOCKER_INFLUXDB_INIT_USERNAME " : {
    "mandatory" : true,
    "type":"string",

  },
  "DOCKER_INFLUXDB_INIT_PASSWORD " : {
```

```
    "mandatory" : true,
    "type":"string",

  },
  "DOCKER_INFLUXDB_INIT_ORG " : {
    "mandatory" : true,
    "type":"string",
  },
  "DOCKER_INFLUXDB_INIT_BUCKET " : {
    "mandatory" : true,
    "type":"string",
  },
   "DOCKER_INFLUXDB_INIT_RETENTION " : {
    "mandatory" : false,
    "type":"int",
    "default": "-1"
  },
   "DOCKER_INFLUXDB_INIT_ADMIN_TOKEN " : {
    "mandatory" : false,
    "type":"string",
    "default": "!random" // random string
  },

}
```