CS695 : Virtualization and Cloud Computing **Jan-Apr 2024**

End-term Project Report : Building F.a.a.S platform

Team Name: Virtual Vanguards *Team Members: 210050043,210050120*

Abstract

Kubernetes can be used to manage a cluster of nodes to instantiate container-based applications. As part of this work, we are building FaaS platform as a wrapper around K8s. Our FaaS platform supports function registration, trigger registration, trigger dispatch, metrics and more.

1 Introduction

Managing container-based applications efficiently across a cluster of nodes is a pivotal task in modern infrastructure management. **Kubernetes** (K8s) emerges as a robust solution for orchestrating these containers seamlessly. In our project, we delve into harnessing Kubernetes's capabilities to construct a Function as a Service (FaaS) platform, encapsulating the agility and scalability offered by containerization within a versatile wrapper.

Within the realm of FaaS, essential functionalities such as function registration, trigger registration, trigger dispatch are imperative for a seamless operational experience. We aim to integrate these crucial components into the Kubernetes ecosystem, thereby empowering developers with a streamlined approach to deploy and manage serverless applications.

By leveraging Kubernetes as the underlying infrastructure and architecting a FaaS platform atop it, we aim to explore the intersection of container orchestration and serverless computing. Through this project, we aspire to provide an efficient solution for deploying and managing serverless workloads in a containerized environment, catering to the evolving needs of modern application development.

2 Toolset used

Docker

Containerization platform used to package and deploy functions within the FaaS system which enables consistent and portable execution environments for each function.

Kubernetes

Container orchestration platform used to manage the deployment, scaling, and operation of Docker containers. Provides automated scaling, fault tolerance, and resource management for the FaaS system.

Flask

Lightweight web framework for Python used to build and host the web application serving as the entry point for invoking functions. Facilitates the development of RESTful APIs and handling of HTTP requests in the FaaS system.

Interactions

- Requests are received by the Python Flask web application user.html
- Flask routes requests to the appropriate Kubernetes Service.
- Kubernetes orchestrates the execution of the function by scheduling the corresponding Docker container.
- Responses from the function are returned back through the same path.

3 Files and their description

config.yaml

This is a Kubernetes configuration file containing cluster, context, and user information. It specifies a cluster named "kubernetes" with server details and a user named "kubernetes-admin" with client certificate and key data.

In our project, this configuration file is used by Kubernetes clients to authenticate and communicate with the Kubernetes cluster. It provides necessary credentials and connection details for accessing and managing resources within the cluster, ensuring secure and authorized interactions between the client and the cluster.

pv.yaml

In our project, this PersistentVolume is crucial for providing persistent storage for applications like databases. It ensures that data stored by these applications remains intact even if pods are terminated or rescheduled within the Kubernetes cluster. This stability is essential for maintaining data integrity and application reliability.
This YAML configuration creates a PersistentVolume named "database-pv" in Kubernetes. It allocates 10 megabytes of storage on the host's "/data" path.

pvc.yaml

This YAML configuration defines a PersistentVolumeClaim (PVC) named "database-pvc" that requests 10 megabytes of storage from the "database-pv" PersistentVolume described earlier. It specifies the storage class as "default" and allows for read-write access by a single node.
In our project, this PVC acts as a request for storage resources, allowing applications to dynamically bind to the previously defined PersistentVolume, ensuring reliable and persistent storage for database-related data. This configuration establishes a link between the application requiring storage (via the PVC) and the available storage resource (the PV), facilitating efficient management and allocation of storage within the Kubernetes cluster.

nodeport.yaml

This YAML configuration defines a Kubernetes Service named "nodeservice" with a type of LoadBalancer. The Service routes traffic to Pods labeled with "app: server". It exposes two ports:

In our project, this Service enables external access to the server Pods running the application. By exposing multiple ports, different aspects of the server application can be accessed through distinct endpoints, enhancing flexibility and scalability. The nodePort assignments (32000 and 32001) allow traffic from outside the Kubernetes cluster to reach the Service.

clusterip.yaml

This YAML configuration defines a Kubernetes Service named "dindservice" without specifying a service type, implying it's a ClusterIP service, which provides internal-only access within the cluster. It routes traffic to Pods labeled with "app: server".

In our project, this service enables internal communication between different components of the application hosted within the Kubernetes cluster. The ports allow different functionalities or services within the server Pods to be accessed by other components or Pods using the service's internal ClusterIP address.

client.py

- This Python script makes an HTTP POST request to a local server running on port 8000 where server will be listening for data
- It sends the data in myobj as form data and the files in file as multipart/form-data

Dind_Dockerfile

This Dockerfile sets up a Docker image environment for running our server.

- It starts with the Docker docker:23.0-dind image. This specific image was chosen because server runs inside a pod and it should be able to use docker functionalities inside the pod to be able to deploy other pods whihc run the service client asked . Hence dind provides us that privilege
- Installs required packages to run python files inside
- Downloads and installs kubectl, the Kubernetes command-line tool.
- creates working environment and copies required server files inside and exposes ports for server to utilize

/Templates/user.html

This HTML form is part of our FaaS (Function as a Service) project. It allows users to register functions or triggers with our platform.

We can choose to register a function either by uploading files or by specifying a Docker image name. If I choose to register by files, I can upload a file (.tgz or .tar.gz) containing my code, provide a command for starting the container/function/image, and specify any arguments needed. Alternatively, I can register by image name, where I just need to provide the Docker Hub image name along with the command and arguments.

I also have the option to register a trigger by providing a URL and an image name. I can specify the HTTP method for the trigger (GET or POST) and a command to run in the container/function/image after the trigger is activated. Additionally, I can specify files I want in return as an HTTP response.

Once I've filled out the necessary information, I can click the submit button to register the function or trigger with our FaaS platform.

server.yaml

This YAML configuration defines a Kubernetes Deployment named "serverdeployment" that ensures one replica of the specified Pods is running. These Pods are labeled with "app: server", allowing the Deployment to manage them.

The Deployment's Pod specification includes:

- Volumes: It defines a volume named "pvc-storage" that mounts a PersistentVolumeClaim named "database-pvc" into the Pod.
- Containers: The Pod contains one container named "task-pv-container" running the Docker image pvrsaikumar/cs695:dindserver. This container executes commands to start the Docker daemon (dockerd) and run Python scripts (server.py and handler.py).
- Ports: It exposes two ports within the container:
  Port "31000": This port is exposed as "server" and forwards traffic to port 31000 of the server Pods.
  Port "31001": This port is exposed as "handler" and forwards traffic to port 31001 of the server Pods.

server.py

/user.html (GET): Renders the user.html template.
/register (POST): Handles registration of Docker images, building , deploying containers, and registering triggers.
Registration Function: It processes the POST request data to determine the type of registration (new image, existing image, trigger).

- For new image registration: It saves the uploaded image file, builds a Dockerfile dynamically, pushing cont_manager.py, builds and pushes the Docker image to a registry (pvrsaikumar), and deploys it as a Kubernetes deployment. creates a clusterIP service and deploys it to be able to communicate with the pod deployed. It registers the image and its deployment details in JSON files (function.json, trigger.json).
- For existing image registration: It pulls the image from the specified registry, creates a Dockerfile dynamically , pushing cont_manager.py , builds and pushes the Docker image to pvrsaikumar registry (if necessary), and deploys it as a Kubernetes deployment creates a clusterIP service and deploys it to be able to communicate with the pod deployed. It registers the image and its deployment details in JSON files (function.json, trigger.json).
- For trigger registration: It registers the trigger details (URL, type, command, return files) associated with a specific image in the trigger.json file.

handler.py

- It serves as the handler for incoming requests, directing them to the appropriate functions based on predefined triggers.This file acts as the intermediary between incoming requests and the functions that need to be executed, ensuring that each request is handled appropriately and efficiently.
- First, it sets up routes to handle both GET and POST requests for any URL . Then, when a request comes in, it loads trigger and function data from JSON files.
- Next, it iterates through each function in the trigger data, checking if the requested path and method match any triggers associated with that function.
- If a match is found: It sends a request to the corresponding function's container received by cont_manager.py . It sends the appropriate data (form data or query parameters) along with the request .

cont_manager.py

- This Flask application acts as a client-side server within the functio pod . It listens for incoming requests, expecting POST requests with specific parameters. It keeps listening at any URL on its port.
- When it receives a POST request, it extracts the arguments, command, and list of files to return from the request data. It then saves any files attached to the request and executes a command using the received arguments and files.
- After executing the command, it zips the specified return files into a single zip archive and sends it back as a response to the client.
- It allows clients to send requests with arguments and files, executes commands on the server, and returns the specified files as a response. This enables the client container to interact with the server container effectively, contributing to the overall functionality of the application.

4 Execution flow

SETTING UP THE SERVER SIDE INFRASTRUCTURE

To be noted that , in all commands we use the below flag indicateing that we want to use a specific Kubernetes configuration file located at the current working directory ($PWD/config.yaml) to interact with the Kubernetes cluster.

In Kubernetes, the kubectl command-line tool uses a configuration file to specify the cluster, user, and other settings necessary to communicate with the Kubernetes API server. By default, kubectl looks for a configuration file at /.kube/config.

```
--kubeconfig $PWD/config.yaml
```

First we need to issue some persistent storage so the command run is :

```
kubectl --kubeconfig $PWD/config.yaml apply -f pv.yaml
```

We need to borrow some storage to deploy the server container

```
kubectl --kubeconfig $PWD/config.yaml apply -f pvc.yaml
```

Next we build a Docker image as specified in the Dind_Dockerfile

```
docker build -t pvrsaikumar/cs695:dindserver . --file ./Dind_Dockerfile
```

The command below pushes the Docker image tagged as pvrsaikumar/cs695:dindserver to a Docker registry.

```
docker push pvrsaikumar/cs695:dindserver
```

The image built is used to deploy a container using persistent storage borrowed above, with server and handler running inside .

```
kubectl --kubeconfig $PWD/config.yaml apply -f server.yaml
```

To enable communication within the cluster , i.e between server pod (running the server and handler) and its deployed pods , we need to create clusterip service with appropriate port connections

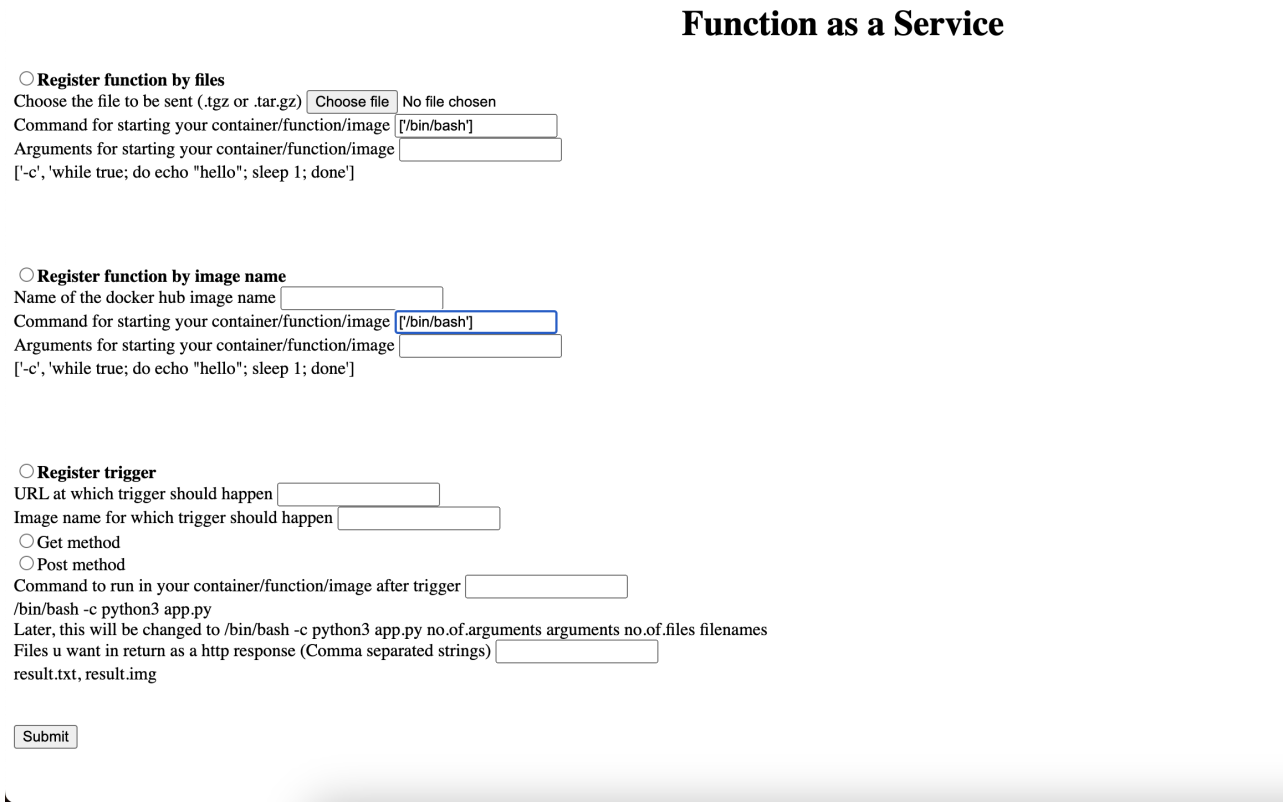
```
kubectl --kubeconfig $PWD/config.yaml apply -f clusterip.yaml
```

For clients to be able to access the web app deployed by the server we deploy Nodeport service exposes them on a port on each node of the cluster. This means the service is accessible externally, outside of the Kubernetes cluster.

```
kubectl --kubeconfig $PWD/config.yaml apply -f nodeport.yaml
```

SHIFTING TO THE CLIENT

- When client wants to register their function , they open the user.html using the ip address of the k8 cluster, where the requests are forwarded to the server running inside the pod by the nodeport service. It is rendered as shown in the image below .



- The client can chose one between register function through file or image name . i.e if already he has an image he wishes to run the function in. We are done with **"FUNCTION REGISTRATION"!!!**
- After he gives an url to which he wishes to send trigger through and sumbits , the focus shifts to the server

SERVER , HANDLER AND TRIGGER REGISTRATION PROCESS

- Now the server receives the trigger registration request and the method of registration .
- It records the function and trigger map accordingly in json files .
  This is where **"TRIGGER REGISTRATION"** is done.
- It then goes on to deploy pods with cont_manager.py and clusterip service to enable communication in future with the pod.
- The handler keeps listening at any addresses in its port for requests .
- If the client send request by hosting his web app filled with appropriate arguments to his function, the handler will receive the data and pass on the data in appropriate format to the appropriate pod referring to the maps in json files , at the port which the pod is listening . We are done with **"TRIGGER DISPATCH"!!**. now the onus is on function pod to execute the function.

FUNCTION EXECUTION IN THE DESIGNATED POD

- Now as cont_manager.py receives the arguments and coupling it with the function data (already present when the pod was deployed), it will begin the execution
- Afteer the result is obtained , it will send the result zipped through the same path back how it received the data, which is redirected to the client by the handler to the client who can access the result of the operation.