**Hello App Documentation**

While planning the project, firstly the tools to be used were determined. The following tools are used for the DevOps Challenge case.

* Go for the Hello-app application
* Google Cloud Platform for the Kubernetes Cluster
* Docker Hub for Docker image registry
* Github to keep repository
* Helm, Prometheus-Operator, Ingress to install the Autoscale structure, monitor the application and load balancing
* Azure DevOps for CI / CD definitions

First of all, the Hello-App application was written using Go and prometheus was integrated into the project in order to use custom metrics. Normally, kubernetes can implement autoscale using memory and cpu metrics, but prometheus was used because a custom metric like average request was wanted to be autoscale.

A multistage Dockerfile was created to reduce the image size and create a more optimized image. After this process, the image was built with the following command.

*docker build -t pinarbakan/app:v1*

The created image was sent to Dockerhub with the following command.

*docker push pinarbakan/app:v1*

After the image was created, Kubernetes Cluster was created using Google Cloud Platform. In order to keep repo, a repository was created in the Github account and the resource files (Dockerfile and application files) were pushed to Github using Git.

After connecting Google Cloud Platform, two new namespaces have been created to separate both the app and monitoring parts in the Hello-app application. In order to autodiscover and monitor the application, it was deployed the Prometheus-Operator into the monitoring namespace with the following command.

Prometheus-operator has the following dependencies: Grafana, Node-Exporter, Kube-State-Metrics. In order to install the Prometheus-Operator, the necessary dependencies must be installed.

*helm dependency update*

*helm install prometheus prometheus-operator/ -n monitoring*

Both the Prometheus-Operator and Helm were used for the project.

The Prometheus-Operator creates, configures, and manages Prometheus monitoring instances as Kubernetes resources. After installing the prometheus-operator on the application, metrics can be observed with Grafana. The commands below can be followed for this.

First, using port-forward, grafana was run from port 3000.

*kubectl port-forward prometheus-grafana-7b7976476c-6hlkt 3000:3000*

The following command is used to pull the username and password required to login to Grafana.

*kubectl get secret prometheus-grafana -n monitoring*

To encrypt username and password information;

Username:

*kubectl get secret --namespace monitoring prometheus-grafana -o jsonpath="{.data.admin-user}" | base64 --decode ; echo*

Password:

*kubectl get secret --namespace monitoring prometheus-grafana -o jsonpath="{.data.admin-password}" | base64 --decode ; echo*

Prometheus metrics are also exposed in app.go to view Prometheus metrics and installed prometheus-operator for receive metrics. Creating a servicemonitor object for autodiscover and added to servicemonitor.yaml with the selector in the output of the following command via the match selector.

*kubectl -n monitoring get prometheuses.monitoring.coreos.com -oyaml*

If desired, metrics can be displayed by making port-forward at this stage.

Helm is a tool that streamlines installing and managing Kubernetes applications. Helm uses a packaging format called charts. A chart is a collection of files that describe a related set of Kubernetes resources. A single chart might be used to deploy something simple, like a memcached pod, or something complex, like a full web app stack with HTTP servers, databases, caches, and so on. Helm was preferred for many purposes such as implementing the autoscale structure and load balancing.

To be able to configure with Helm, values.yaml was created first and all static values ​​were defined. ingress.yaml was created for load balancing and routing. In Kubernetes, an Ingress is an object that allows access to your Kubernetes services from outside the Kubernetes cluster.

To implement autoscale, HPA, the horizontal pod autoscale, was used and hpa.yaml was created.

In addition to these, deployment.yaml, service.yaml, service-actuator.yaml and service-monitor.yaml required for deployment, monitoring or service configuration were created.

After implementing Autoscale, pods were configured to scale after Cpu and Memory usage reached 10% in hpa.yaml in order to test the structure. The autoscale structure was tested by pressing the load to the application with the following command.

*while true; do wget -q -O-* [*[ingress-ip-adress/]*](https://www.google.com/url?q=http%3A%2F%2F34.120.44.72)*; done*

After these steps were completed, pipeline definitions were made on Azure DevOps to make the CI / CD configuration of the application.

Azure Devops Link:

<https://dev.azure.com/pinarbakan0126/hb-app/>