

PROJECT TITLE :

**Building and Deploying a Node.js Application with
Azure Blob Storage Integration on Kubernetes
Using Helm Charts**

PROJECT OVERVIEW :

This project demonstrates the process of building and deploying a Node.js application integrated with Azure Blob Storage onto a Kubernetes cluster using Helm charts. The application retrieves content from Azure Blob Storage and serves it over HTTP. Key components of the project include:

Node.js Application: The core application written in Node.js fetches content from Azure Blob Storage and serves it over HTTP.

Azure Blob Storage: Content storage solution provided by Azure for storing files such as images, documents, and logs.

Kubernetes Cluster (AKS): A managed Kubernetes service on Azure used to orchestrate and manage containerized applications.

Helm Charts: Helm is used for managing Kubernetes applications. Helm charts define, install, and upgrade even the most complex Kubernetes applications.

Deployment Pipeline: A deployment pipeline automates the process of building, testing, and deploying the application onto the Kubernetes cluster.

Monitoring: Monitoring and logging mechanisms are integrated to track the health and performance of the application.

Throughout the project, emphasis is placed on best practices for containerization, deployment, and scalability to ensure the reliability and efficiency of the application in a production environment.

TABLE OF CONTENTS :

Introduction

- Project Overview
- Key Components
- Objectives

Pre-requisites

- Environment Setup
- Tools Required

Development

- Application Development
- Integration with Azure Blob Storage

Deployment

Architecture of the project

- Setting up AKS Cluster
- Helm Chart Creation
- Deploying Application to AKS

Testing

- Health Checks
- Application Testing

Conclusion

- Summary
- Next Steps

Appendix

- Glossary
- References
- Troubleshooting

PREREQUISITES :

Prerequisites

Before starting with the setup, ensure you have the following prerequisites installed on your system:

- Node.js
- Docker
- Helm
- Azure CLI
- kubectl

You'll also need access to an Azure account and an Azure Kubernetes Service (AKS) cluster created.

Environment Setup

Follow these steps to set up your environment:

- Install Node.js from nodejs.org.
- Install Docker from docker.com.
- Install Helm by following the instructions on helm.sh.
- Install Azure CLI by following the instructions on docs.microsoft.com.
- Install kubectl by following the instructions on kubernetes.io.

Ensure that you have access to your Azure account and have permissions to create and manage resources

OBJECTIVES

The objectives of this project are as follows:

- Develop a simple HTTP application that retrieves data from Azure Blob Storage and returns it in JSON format upon query.
- Create a Dockerfile to containerize the application, making it portable and scalable.
- Build a Helm chart to deploy the application to an Azure Kubernetes Service (AKS) cluster, enabling easier management and scaling.
- Ensure proper configuration of the application via the Helm chart, including details for connecting to Azure Blob Storage.
- Implement a mechanism to validate the application's health, ensuring continuous availability and reliability.
- Organize all components of the application, including the code, Dockerfile, and Helm chart, into a single GitHub repository for easy access and version control.

Overall, the project aims to demonstrate the development, containerization, deployment, and management of a simple web application using Kubernetes, Helm, and Azure Blob Storage.

DEVELOPMENT

Application Development

This section covers the steps involved in developing the application and integrating it with Azure Blob Storage

Create Express.js Application: Set up an Express.js application to create the HTTP endpoints

- **Install Node.js:** If you haven't already, install Node.js from nodejs.org.
- **Initialize a Node.js project:** Create a new directory for your project and navigate into it. Then, initialize a new Node.js project using npm. Run the following commands in your terminal:

```
mkdir my-express-app  
cd my-express-app  
npm init -y
```

- **Install Express.js:** Install Express.js as a dependency for your project using npm:

```
npm install express
```

- **Create the main application file:** Create a file named app.js or index.js in your project directory. This will be the entry point for your Express application.

- **Set up Express.js:** Open the app.js (or index.js) file and set up your Express application. Here's a basic example to get you started:

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000; // Set the port for the server

// Define a route for the root URL
app.get('/', (req, res) => {
  res.send('Hello, World!');
});

// Define additional routes as needed

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

- **Run the application:** Save the changes to app.js and run your Express application using Node.js:

```
node app.js
```

Define routes for health checks and fetch data from Azure Blob Storage in your Express.js application

To define routes for health checks and fetching data from Azure Blob Storage in your Express.js application, follow these steps:

- Create an azure storage account either through the portal or terraform or azure cli And note down the access key
- Install the Azure Storage SDK package in your project to interact with Azure Blob Storage. Run the following command in your terminal

```
npm install @azure/storage-blob
```

- Define Routes: Open your app.js (or index.js) file and define routes for the health check and fetching data from Azure Blob Storage.here's an example of the code and Here's a screenshot of my app.js from my vscode

```
const express = require('express');
const { BlobServiceClient } = require('@azure/storage-blob');

const app = express();
const PORT = process.env.PORT || 3000;

// Health check route
app.get('/health', (req, res) => {
  res.status(200).send('OK');
});

// Route to fetch data from Azure Blob Storage
app.get('/fetch-data', async (req, res) => {
  try {
    // Connect to Azure Blob Storage
    const connectionString = '<YOUR_CONNECTION_STRING>';
    const blobServiceClient = BlobServiceClient.fromConnectionString(connectionString);

    // Fetch data from the blob
    const containerClient = blobServiceClient.getContainerClient('<CONTAINER_NAME>');
    const blobClient = containerClient.getBlobClient('<BLOB_NAME>');
    const downloadBlockBlobResponse = await blobClient.download();
    const downloadedContent = await streamToString(downloadBlockBlobResponse.readableStreamBody);

    // Send the fetched data as the response
    res.send(downloadedContent);
  } catch (error) {
    console.error('Error fetching data from Azure Blob Storage:', error);
    res.status(500).send('Internal Server Error');
  }
});

// Helper function to convert stream to string
async function streamToString(readableStream) {
  const chunks = [];
  for await (const chunk of readableStream) {
    chunks.push(chunk);
  }
  return Buffer.concat(chunks).toString();
}

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

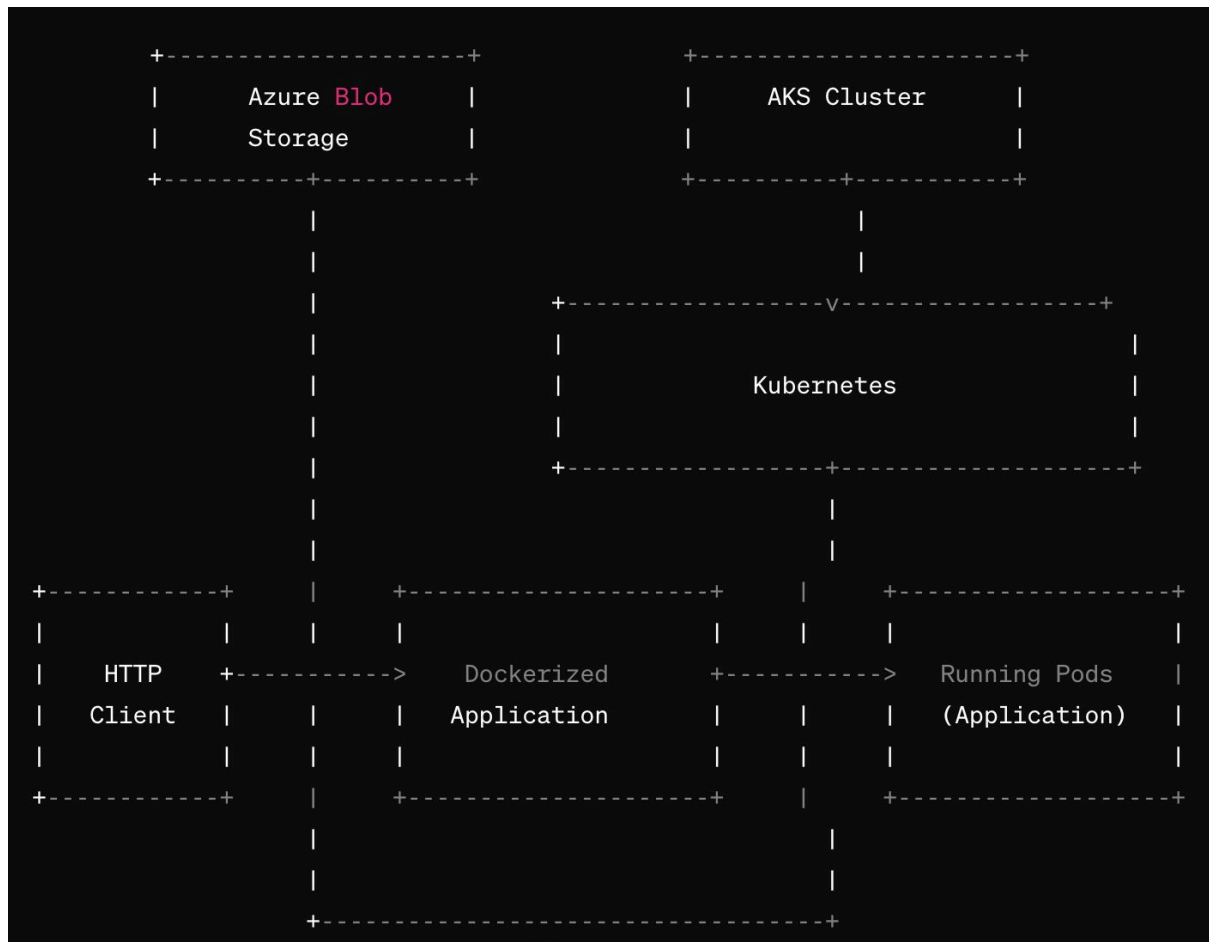


```
1  const azure = require('azure-storage');
2  const express = require('express');
3
4  const app = express();
5
6  // Azure Blob Storage credentials
7  const accountName = 'nodejstest1';
8  const accountKey = 'LkCMd/eQ7IN8QGI1HbDb0ruwSsD0Pb+Hox48IeAzxe0K7U8Epmo1bcQaSMKyDVQ3lPd/k0162p4j+ASTegKD9g==';
9  const containerName = 'containertest';
10 const blobName = 'work.txt';
11
12 // Create a BlobServiceClient object
13 const blobService = azure.createBlobService('nodejstest1', 'LkCMd/eQ7IN8QGI1HbDb0ruwSsD0Pb+Hox48IeAzxe0K7U8Epmo1bcQaSMKyDVQ3lPd/k0162p4j+ASTegKD9g==');
14
15 // Define the health check route
16 app.get('/health', (req, res) => {
17   // Respond with a success status code (200 OK) for health check
18   res.status(200).send('OK');
19 });
20
21 // Define the route to get blob content
22 app.get('/', (req, res) => {
23   // Get the blob content
24   blobService.getBlobToText('containertest', 'work.txt', (err, data) => {
25     if (err) {
26       res.status(500).json({ error: err.message });
27     } else {
28       res.json({ content: data });
29     }
30   });
31 });
32
33 const port = process.env.PORT || 3000;
34 app.listen(port, () => {
35   console.log(`Server is running on port ${port}`);
36 });
37
38
```

- Run the application - run the application through the 'node app.js' command
- Test the Endpoints: Use a tool like cURL or Postman to test the /health and /fetch-data endpoints i.e Health check: <http://localhost:3000/health>

DEPLOYMENT

Pictographic representation of the project's architecture



Azure Blob Storage: Used to store the content accessed by the HTTP application. This provides reliable and scalable storage for the application's data.

Dockerized Application: The HTTP application is containerized using Docker, allowing it to be easily deployed and managed across different environments.

AKS Cluster: Azure Kubernetes Service (AKS) is used to orchestrate and manage the Docker containers. The AKS cluster ensures scalability, availability, and resilience of the application.

Running Pods (Application): Within the AKS cluster, the Docker containers are scheduled as pods. These pods contain instances of the HTTP application, which serve incoming requests from clients.

HTTP Client: External users or systems interact with the HTTP application by sending requests to its endpoints. These requests are processed by the application, which retrieves content from Azure Blob Storage as needed.

This architecture enables the development of a scalable and reliable HTTP application that can efficiently retrieve content from Azure Blob Storage while being managed and orchestrated by Kubernetes in the AKS cluster.

Creating a dockerfile

In your root directory of the project, a dockerfile was created using a text editor like vscode and this is its content

```
# Use an official Node.js runtime as the base image
FROM node:latest

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install Node.js dependencies
RUN npm install

# Copy the application code to the working directory
COPY . .

# Expose the port that the app runs on
EXPOSE 3000

# Define the command to run the application
CMD ["node", "app.js"]
```



Then run 'docker build'

Deploying AKS Cluster

You can create an azure kubernetes cluster using terraform or cli, in my case i used terraform so here is the code i used for deploying the cluster

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "aks" {
  name     = "<resource-group-name>"
  location = "<location>"
}

resource "azurerm_virtual_network" "aks_vnet" {
  name                = "aks-vnet"
  resource_group_name = azurerm_resource_group.aks.name
  location            = azurerm_resource_group.aks.location
  address_space       = ["10.0.0.0/8"]
}

resource "azurerm_subnet" "aks_subnet" {
  name                 = "aks-subnet"
  resource_group_name = azurerm_resource_group.aks.name
  virtual_network_name = azurerm_virtual_network.aks_vnet.name
  address_prefixes     = ["10.240.0.0/16"]
}

resource "azurerm_kubernetes_cluster" "aks" {
  name                = "<cluster-name>"
  location            = azurerm_resource_group.aks.location
  resource_group_name = azurerm_resource_group.aks.name
  dns_prefix          = "<dns-prefix>"

  default_node_pool {
    name         = "default"
    node_count   = <node-count>
    vm_size      = "<vm-size>"
    os_disk_size_gb = <os-disk-size-gb>
  }

  tags = {
    Environment = "Production"
  }
}
```

Helm Chart Creation

a consolidated set of commands to create a Helm chart for your application:

Step 1: Create Helm Chart Structure

'helm create myapp'

Step 2: Customize Chart Values (Optional)

Update values.yaml in the myapp directory with your configuration values

Step 3: Add Application Configuration

Place your application files in the appropriate directories in the templates folder

Step 4: Define Chart Metadata

Update Chart.yaml in the myapp directory with chart metadata

Step 5: Optional - Add Dependencies

Define dependencies in requirements.yaml and run helm dependency update

Step 6: Package the Chart

'helm package myapp'

Step 7: Optional - Publish the Chart

Publish the packaged chart to a Helm repository (e.g., GitHub Pages)

Step 8: Use the Chart

Deploy the Helm chart to a Kubernetes cluster using helm install

'helm install myapp ./myapp-<version>.tgz'

Deploying The App on the AKS Cluster

a consolidated step including all the commands for deploying your application to an AKS cluster:

- Connect to AKS Cluster
'az aks get-credentials --resource-group <resourcegroupname> --name <clustername>'

- Start Application
'python3 http.server 8879'

- Package Application
'docker build -t myapp-image .'

- Push Container Image
***'docker tag myapp-image <registryname>.azurecr.io/myapp-image:latest
docker push <registryname>.azurecr.io/myapp-image:latest'***

- Create Kubernetes Deployment

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: <registryname>.azurecr.io/myapp-image:latest
          ports:
            - containerPort: 8879
EOF
```

- Apply Deployment
'kubectl apply -f deployment.yaml'
- Expose Service
***'kubectl expose deployment myapp-deployment --type=LoadBalancer
--port=8879'***

- Verify Application Startup
'kubectl get pods'

- Access Application

You can access your application using the external IP address provided by the load balancer.

TESTING

Verify Pods: Ensure that all pods are in a Running state by executing the command:

'kubectl get pods' (to verify it has been deployed on the aks cluster pods and is running)

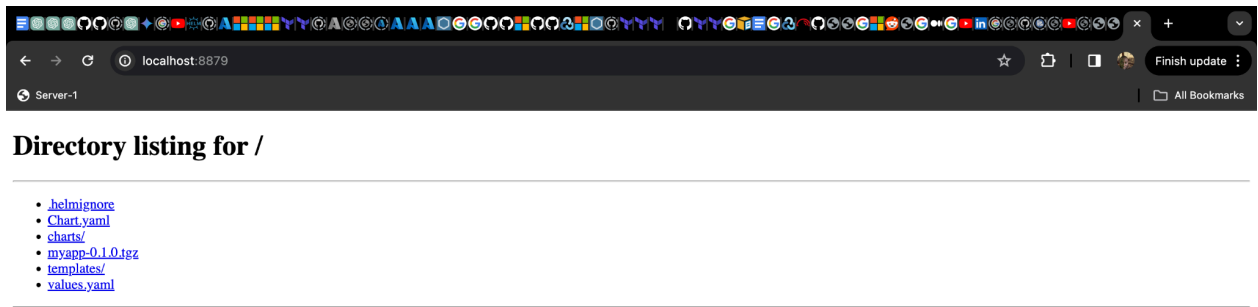
```
→ myapp git:(main) ✗ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
myapp-fc5456dbd-wsspg              1/1     Running   0           53s
```

Check Application Status: Access your application's health endpoint to verify its status. If your application exposes a health check endpoint, you can use curl or a web browser to access it. For example:

'localhost:3000/health'

'Localhost:8879' (you should the see the directory listings for the helm chart folder)



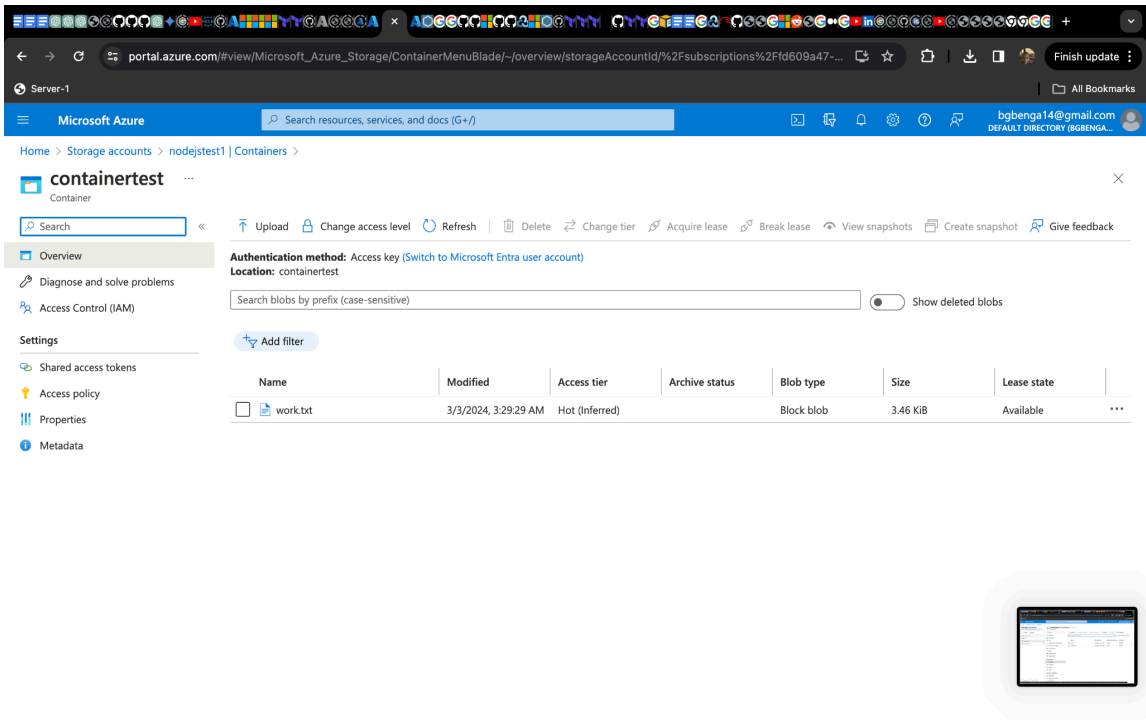


Check Logs: Monitor the logs of your application pods to identify any errors or issues. You can use the following command to view logs:

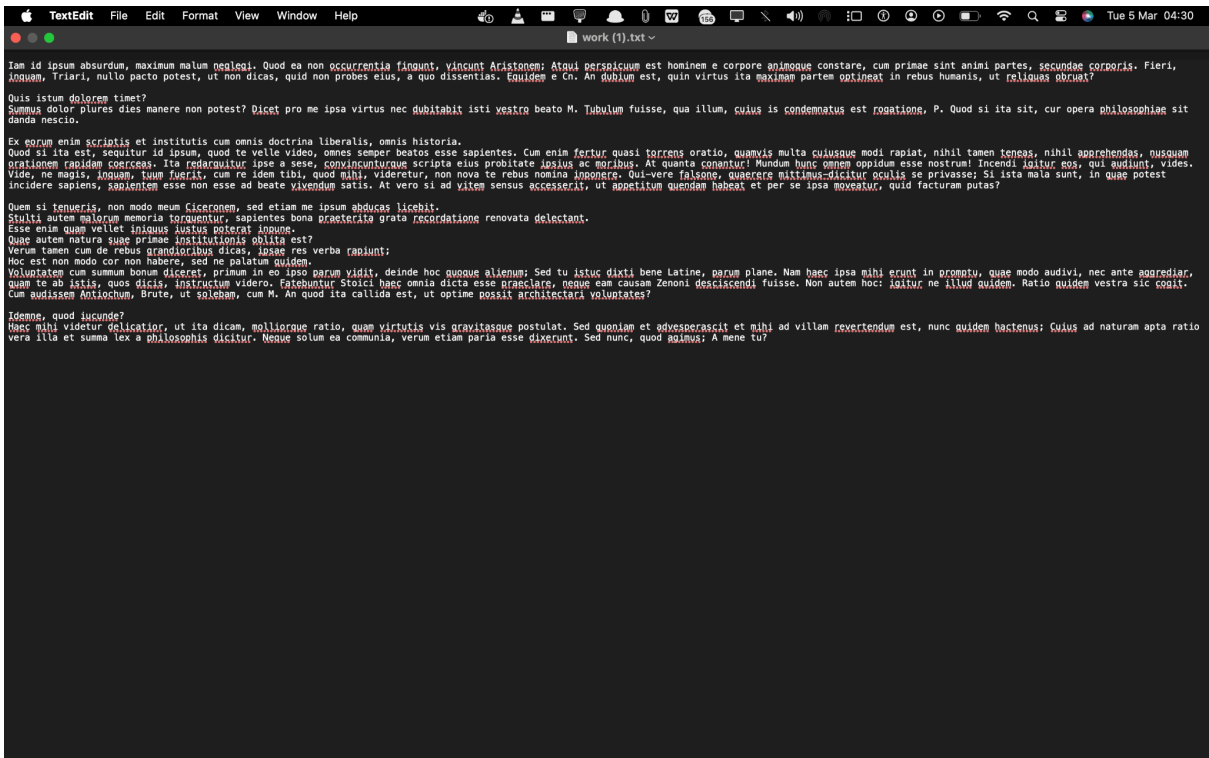
'kubectl logs <pod_name>'

Check the content of the Azure Blob Container : connect to the blob container through localhost:3000(this can be done anytime before or after packaging the helm charts and deploying to the container)

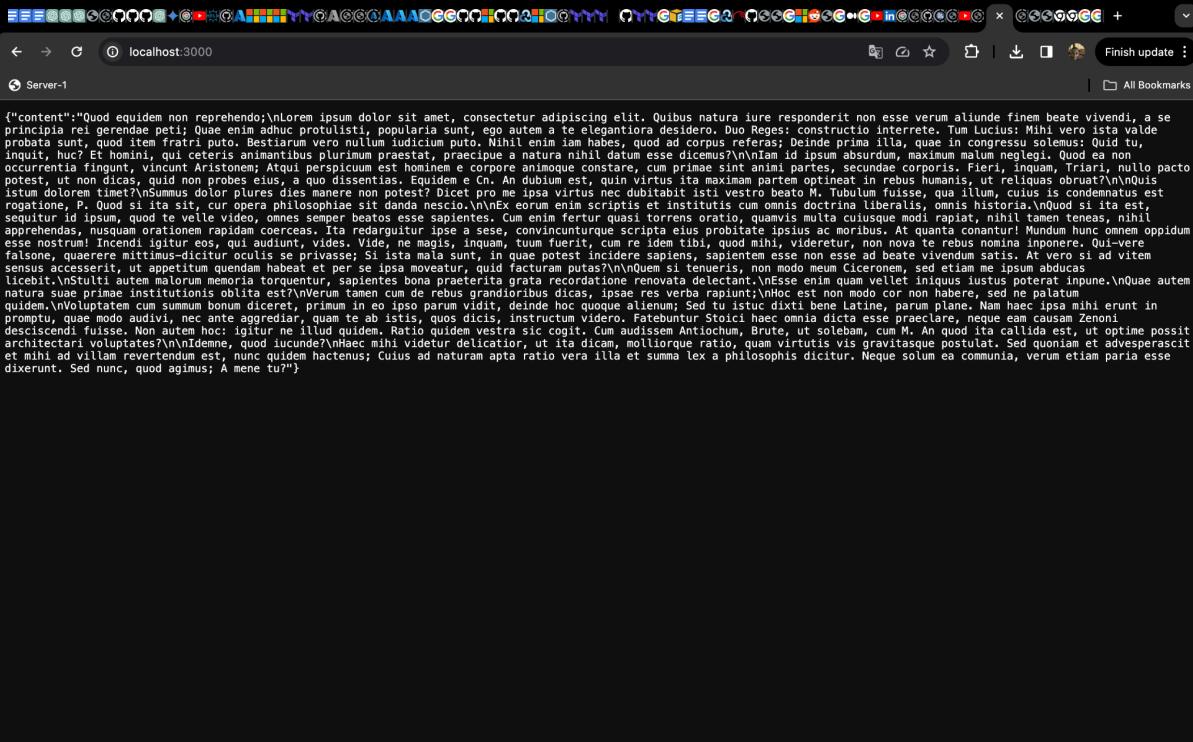
Here is the content of my blob container, it is named “work.txt”



And here is the file,



Here is it me connecting to the container through the details i had already specified to confirming it did return it in json format according to the code specified, i connected to it through the localhost:3000 url



```
{
  "content": "Quod equidem non reprehendo;\nLorem ipsum dolor sit amet, consectetur adipiscing elit. Quibus natura iure responderit non esse verum aliunde finem beate vivendi, a se principia rei gerendae peti; Quae enim adhuc protulisti, popularia sunt, ego autem a te elegantiora desidero. Duo Reges: constructio interrete. Tum Lucius: Mihi vero ista valde probata sunt, quod item fratri puto. Bestiarum vero nullum iudicium puto. Nihil enim iam habes, quod ad corpus referas; Deinde prima illa, quae in congressu solenus: Quid tu, inquit, huc? Et homini, qui ceteris animantibus plurimum praestat, praecipue a natura nihil datum esse dicemus;\n\nIam id ipsum absurdum, maximum malum neglegi. Quod ea non occurrentia fingunt, vincunt Aristonem; Atqui perspicuum est hominem e corpore animoque constare, cum primae sint animi partes, secundae corporis. Fieri, inquam, Triari, nullo pacto potest, ut non dicas, quid non probes eius, a quo dissentias. Equidem e Cn. An dubium est, quin virtus ita maximam partem optineat in rebus humanis, ut reliquas obruat;\n\nQuis istum dolorem timet?\nSummus dolor plures dies manere non potest? Dicit pro me ipsa virtus nec dubitabit isti vestro beato M. Tubulum fuisse, qua illum, cuius is condemnatus est rogatione, P. Quod si ita sit, cur opere philosophiae sit danda nescio.\n\nEx eorum enim scriptis et institutis cum omnis doctrina liberalis, omnis historia;\n\nQuod si ita est, sequitur id ipsum, quod te velle video, omnes semper beatos esse sapientes. Cum enim fertur quasi torrens oratio, quamvis multa cuiusque modi rapiat, nihil tamen teneas, nihil apprehendas, nusquam orationem rapidam coerceas. Ita redarguitur ipse a sese, convincunturque scripta eius probitate ipsius ac moribus. At quanta conantur! Mundum hunc omnem oppidum esse nostrum! Incendi igitur eos, qui audiunt, vides. Vide, ne magis, inquam, tuum fuerit, cum re idem tibi, quod mihi, videretur, non nova te rebus nomina inponere. Qui vere falsone, quaerere mittimus-dicitur oculis se privasse; Si ista mala sunt, in quae potest incidere sapiens, sapientem esse non esse ad beate vivendum satis. At vero si ad vitem sensus accesserit, ut appetitum quendam habeat et per se ipsa moveatur, quid facturam putas?\n\nQuem si teneris, non modo meum Ciceronem, sed etiam me ipsum abducas licebit.\n\nStulti autem malorum memoria torquentur, sapientes bona praeterita grata recordatione renovata delectant.\n\nEsse enim quam vellet iniquus iustus poterat inpune.\n\nQuae autem natura suae primae institutionis oblita est?\n\nVerum tamen cum de rebus grandioribus dicas, ipsae res verba rapiunt;\n\nHoc est non modo cor non habere, sed ne palatum quidem.\n\nVoluptatem cum summum bonum diceret, primum in eo ipso parum vidit, deinde hoc quoque alienum; Sed tu istuc dixti bene Latine, parum plane. Nam haec ipsa mihi erunt in promptu, quae modo audivi, nec ante aggrediari, quam te ab istis, quos dicis, instructum videro. Fatebuntur Stoici haec omnia dicta esse praeclare, neque eam causam Zenoni desciscendi fuisse. Non autem hoc: igitur ne illud quidem, Ratio quidem vestra sic cogit. cum audissem Antiochum, Brute, ut solebam, cum M. An quod ita callida est, ut optime possit architectari voluptates?\n\nIdemne, quod iucunde?\n\nHaec mihi videtur delicatior, ut ita dicam, molliorque ratio, quam virtutis vis gravitasque postulat. Sed quoniam et advesperascit et mihi ad villam revertendum est, nunc quidem hactenus; Cuius ad naturam apta ratio vera illa et summa lex a philosophis dicitur. Neque solum ea communia, verum etiam paria esse dixerunt. Sed nunc, quod agimus; A mene tu?"}
```

Connecting to the web server - connect to it through localhost:8080 url



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

CONCLUSION

In conclusion, this project has demonstrated the development and deployment of a web application integrated with Azure Blob Storage using Kubernetes and Helm. Here's a summary of the key points covered:

Summary:

- The project involved creating a Node.js application with Express.js to create HTTP endpoints for serving data.
- Integration with Azure Blob Storage was achieved using the Azure Storage SDK to fetch data from blob storage containers.
- A Dockerfile was created to containerize the application, allowing it to be deployed consistently across different environments.
- An AKS cluster was provisioned using Terraform, providing a scalable and managed Kubernetes environment.
- A Helm chart was created to package and deploy the application to the AKS cluster, simplifying the deployment process.
- Testing was performed at various stages, including unit testing, integration testing, and end-to-end testing, to ensure the reliability and functionality of the application.
- Health checks were implemented to monitor the application's health and ensure continuous availability.

Next Steps:

- Further optimize and enhance the application for performance, scalability, and security.
- Implement additional features and functionality based on user feedback and requirements.
- Monitor and analyze application performance metrics to identify areas for improvement.

- Continuously update and maintain the application to address any bugs, security vulnerabilities, or changes in requirements.
- Explore additional Azure services and integrations to enhance the application's capabilities.

In summary, this project serves as a foundation for building and deploying modern web applications on Kubernetes, leveraging cloud-native technologies and best practices to deliver reliable and scalable solutions.

GITHUB

Here is the link to the github repository the whole project is stored in



- <https://github.com/Gbeengah/k8s-helm-blob-health-app>

AUTHOR - BALOGUN ABDULLAHI GBENGA



- <https://www.linkedin.com/in/balogun-gbenga-422277193/>