School of Computer Science

UNIVERSITY OF PETROLEUM AND ENERGY STUDIES

DEHRADUN, UTTARAKHAND

Container Orchestration and Infrastructure Automation

ContactHub: Contacts Management System

Project Report

(2023-2024)

for
5th Semester

SUBMITTED TO:
Ms.Avita Katal
Assistant Professor(SG)
School of Computer Science

SUBMITTED BY:

Sanaya Bhardwaj
SAP ID: 500091835
Roll no.: R2142210690

Part 2: Deployment and Scaling

PART 1 Recap:

ContactHub-The Contact Manager Microservices project leverages the power of Spring Boot, a robust Java-based framework, to create a scalable and highly maintainable solution for contact management.
Microservices are defined with clear purposes and communication methods, justified for scalability. Docker is chosen for containerization, ensuring portability and isolation. Microservices seamlessly interact through APIs, emphasizing integration. Git is effectively used for version control.

Microservices created for the project:

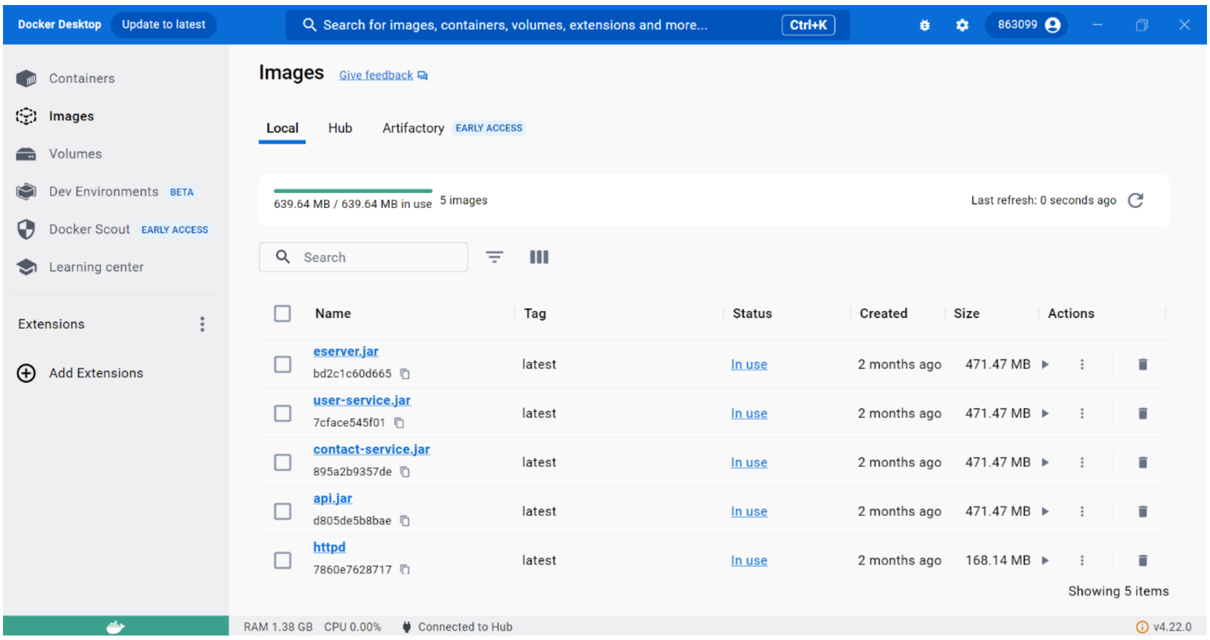
API Gateway: Serves as the entry point for client requests. It routes requests to the appropriate microservices based on the request path or URL.

Eureka (Service Registry): Microservices register themselves with Eureka upon startup. It maintains a registry of all available services and their network locations.

User Service: Manages user-related data. The API Gateway forwards user-related requests to this service.

Contact Service: Handles contact-related data. The API Gateway routes contact-related requests to this service.

Docker images created for Containerization:



Cloud Platform Selection

Justify the choice of a specific cloud platform (e.g., AWS, Azure, Google Cloud) based on its suitability for the project.

For ContactHub, the chosen cloud platform based on its suitability is AWS.

Amazon Web Services (AWS):

- Java Ecosystem Support:

  - AWS has excellent support for Java applications, making it well-suited for a project built on the Spring Boot framework.
  - AWS offers services like AWS Elastic Beanstalk and AWS Lambda, which are particularly useful for deploying and managing Java applications.

- Microservices Architecture:

  - AWS provides a range of services that support microservices architecture, such as Amazon ECS (Elastic Container Service) and AWS Lambda.
  - Integration with AWS API Gateway can help in creating and managing APIs for the microservices.

- Scalability and Flexibility:

  - AWS provides scalable solutions through services like Amazon EC2 for virtual servers and Amazon RDS for managed databases.
  - Auto Scaling features can automatically adjust capacity based on demand.

- Extensive Service Offering:

  - AWS has a comprehensive set of services, including databases (Amazon DynamoDB, Amazon RDS), storage (Amazon S3), and messaging (Amazon SQS), which are crucial for a microservices architecture.

Container Orchestration

Utilize a container orchestration platform (e.g., Kubernetes) to manage and deploy the microservices.

Installing Minikube:

What & Why Minikube?

minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes. All we need is Docker (or similarly compatible) container or a Virtual Machine environment, and Kubernetes is a single command away: minikube start

```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP-PC>minikube start
W1117 00:47:52.297893   18068 main.go:291] Unable to resolve the current Docker CLI context "default": context "default"
: context not found: open C:\Users\HP-PC\.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c4958ba141d5f4133a3
3f0688f\meta.json: The system cannot find the path specified.
* minikube v1.32.0 on Microsoft Windows 11 Home Single Language 10.0.22621.2428 Build 22621.2428
* Automatically selected the docker driver. Other choices: virtualbox, ssh
* Using Docker Desktop driver with root privileges
* Starting control plane node minikube in cluster minikube
* Pulling base image ...
* Downloading Kubernetes v1.28.3 preload ...
    > gcr.io/k8s-minikube/kicbase...:  339.53 MiB / 453.90 MiB  74.80% 9.35 MiB
```

Deploying Microservices with Kubernetes:

- “**docker-k8s.yaml**” for contact service deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: contactservice
spec:
  replicas: 3
  selector:
    matchLabels:
      app: contactservice
  template:
    metadata:
      labels:
        app: contactservice
    spec:
      containers:
      - name: contactservice
        image: 863099/springboot-image:latest
        ports:
        - containerPort: 3000

```

- “**docker-k8s.yaml**” for user service deployment

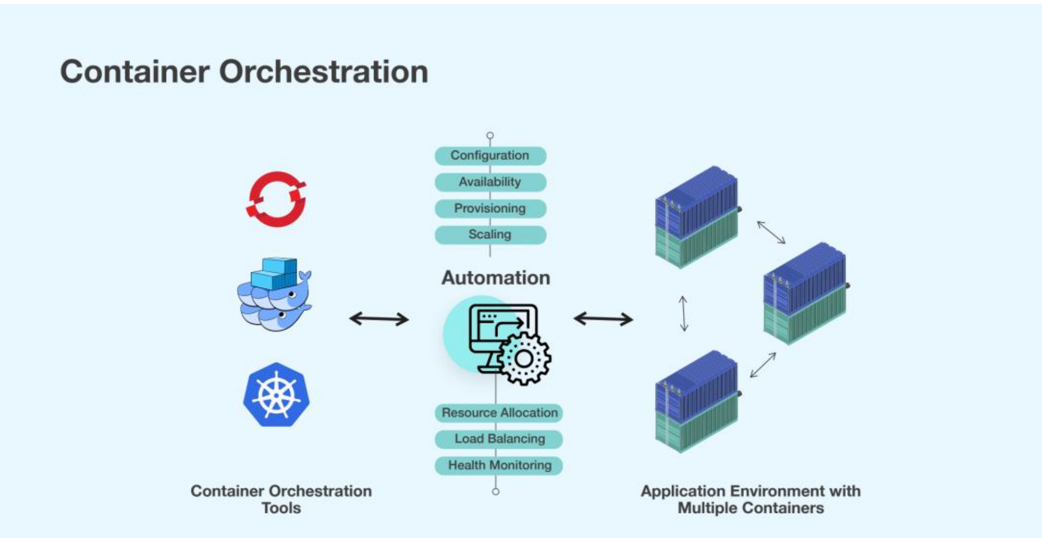
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: userservice
spec:
  replicas: 3
  selector:
    matchLabels:
      app: userservice
  template:
    metadata:
      labels:
        app: userservice
    spec:
      containers:
      - name: contactservice
        image: 863099/springboot-image:latest
        ports:
        - containerPort: 3000
```

In a complete microservices architecture, we would have separate Deployment YAML files for each microservice, allowing us to independently manage and scale each service based on its specific requirements.

DEPLOYMENT STEPS:

- Containerizing Microservices with Docker
- Building Docker Images
- Pushing Docker Images to Container Registry
- Deploying Microservices with Kubernetes
- Creating Kubernetes Deployments using commands:
  - **kubectl apply -f deployment-service1.yaml**
  - **kubectl apply -f deployment-service2.yaml**
- Creating Kubernetes Services using commands:
  - **kubectl apply -f service-service1.yaml**
  - **kubectl apply -f service-service2.yaml**

Describe how container orchestration simplifies deployment, scaling, and management.



Container orchestration is the process of automating the operational effort required to run containerized workloads and services. It automates various aspects of the containers' lifecycle, including provisioning, deployment, scaling, networking, load balancing, traffic routing, and more.

Container orchestration simplifies deployment, scaling, and management by automating the processes of deploying and managing containers at scale. It abstracts complexities, streamlines resource allocation, automates scaling based on demand, and provides centralized management, making it easier to maintain, update, and scale applications in a containerized environment.

Deployment Process

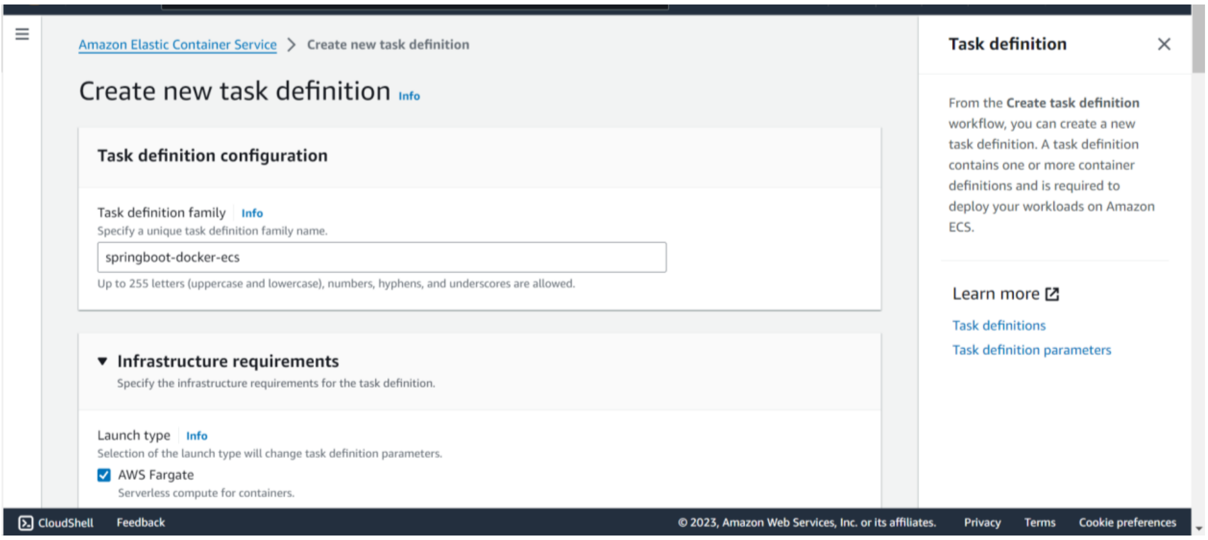
Provide scripts or configurations for deploying the microservices on the chosen cloud platform.

Prerequisites:

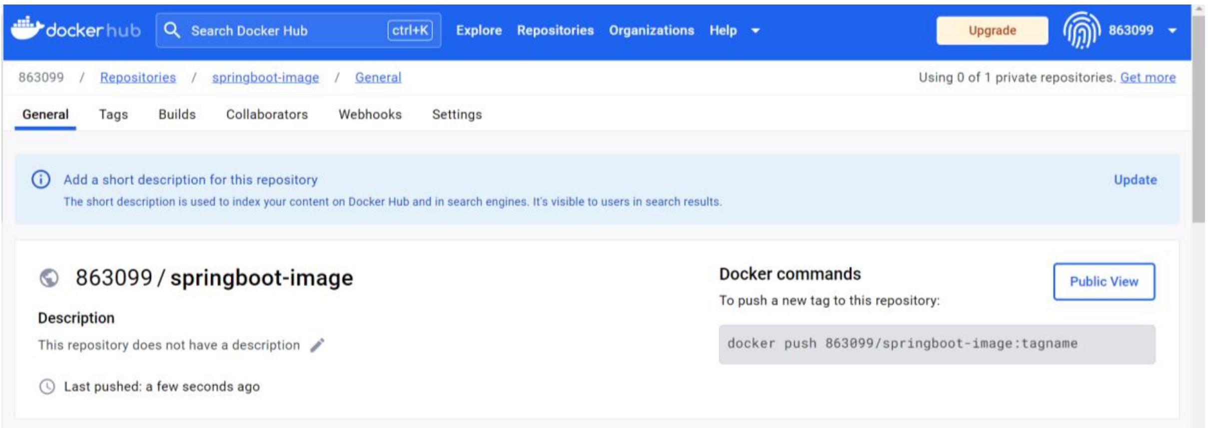
- Docker for Containerization (Already done)
- Kubernetes for Orchestration (Already done)
- Cloud Platform- AWS (Needs to be done)

WORKING ON AWS ECS AND AWS FARGATE:

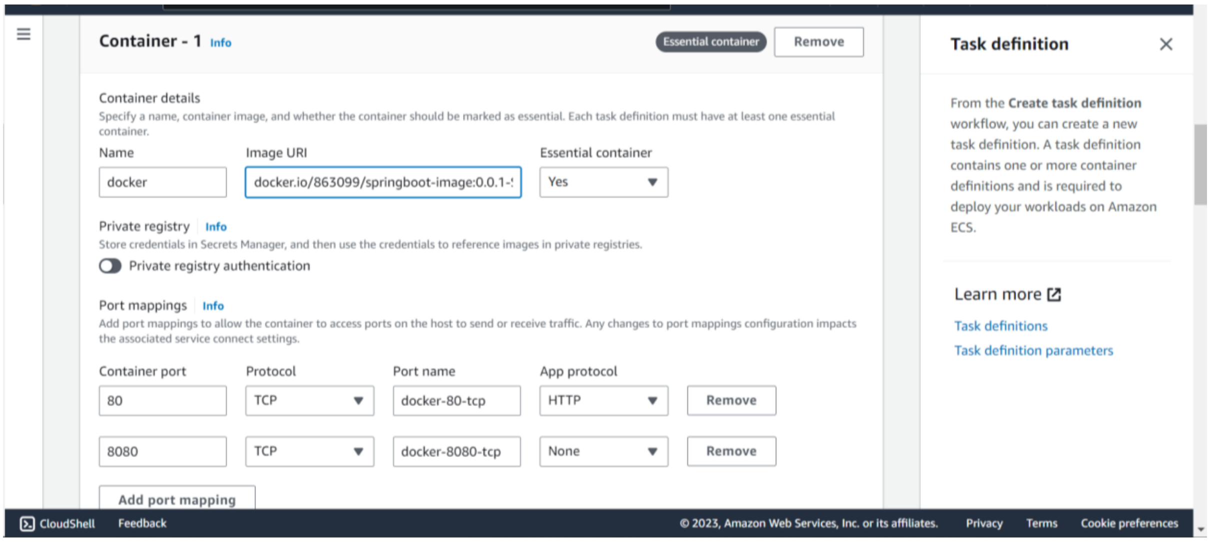
- In AWS ECS, creating task definition (Task definition contains one or more container definitions and is required to deploy our workload on AWS ECS)



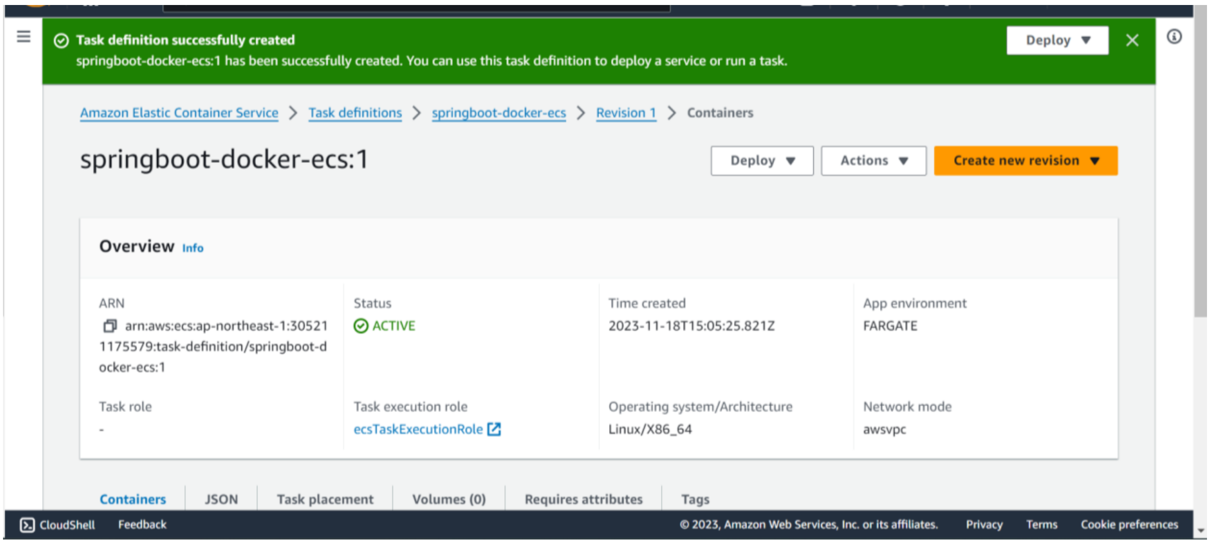
- Docker repository of image created on Docker Hub

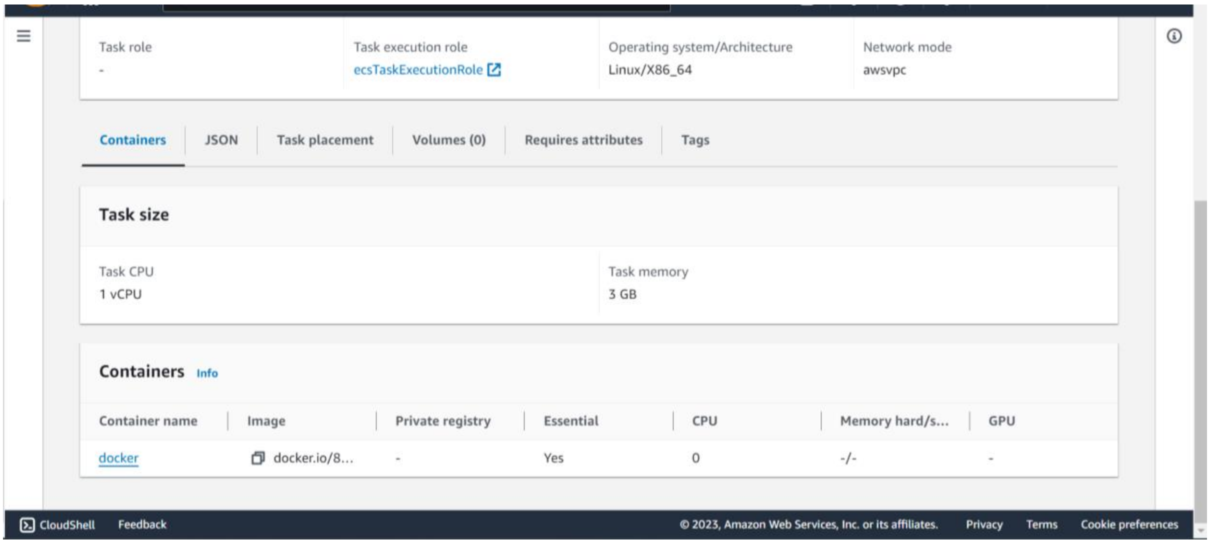


- Adding container and image url from docker hub

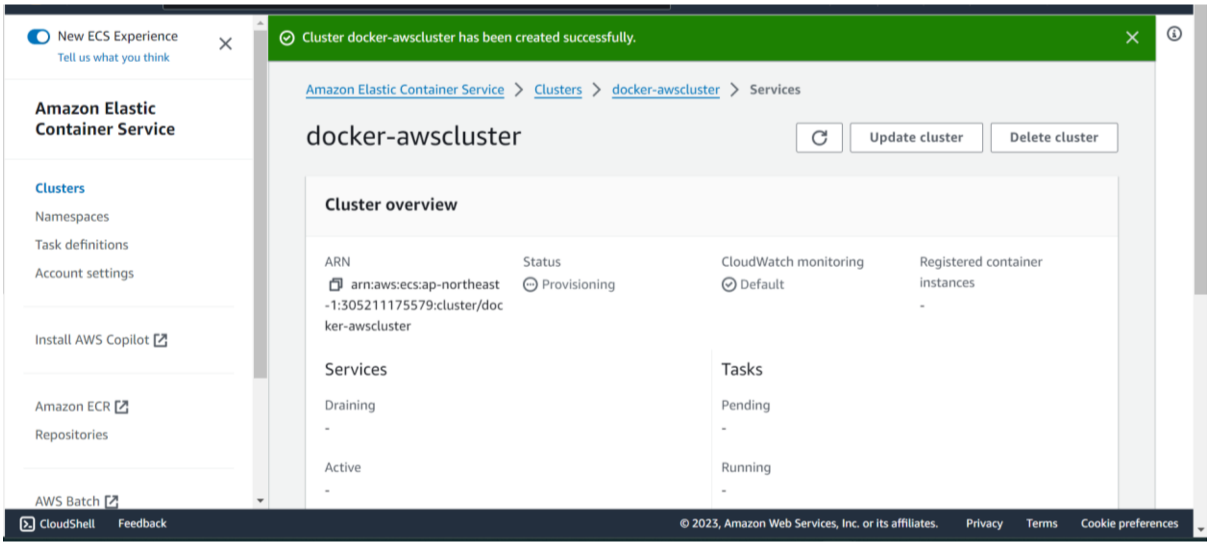


- Task definition created successfully

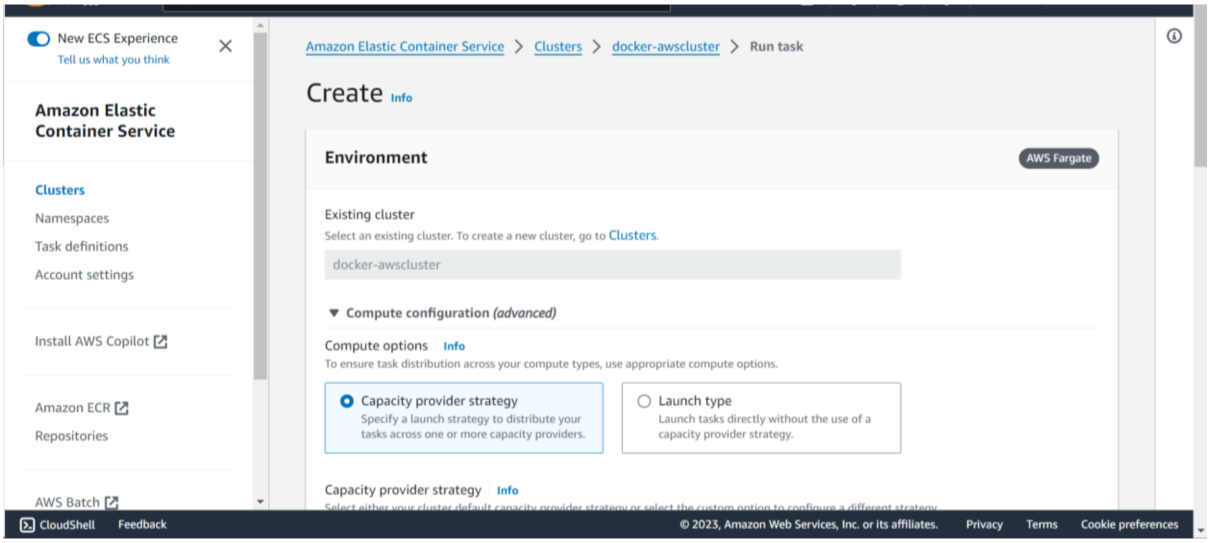




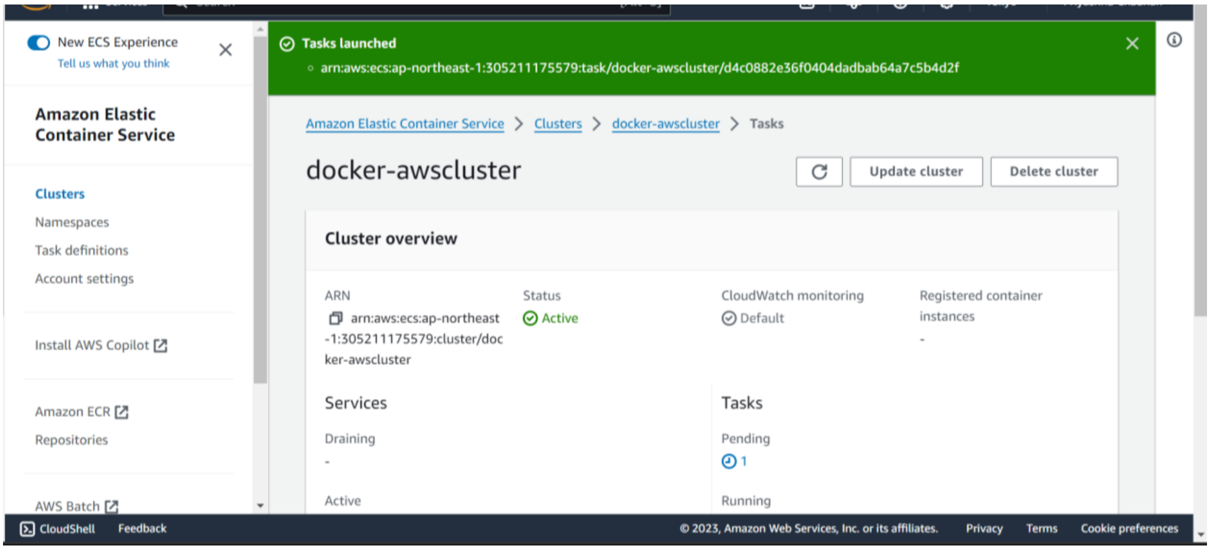
- Successfully Created cluster in AWS ECS.



- Adding Existing cluster to the task definition



- Task Launched successfully



Demonstrate the ability to deploy the application in a cloud environment consistently.

AWS provides a range of services that support microservices architecture:

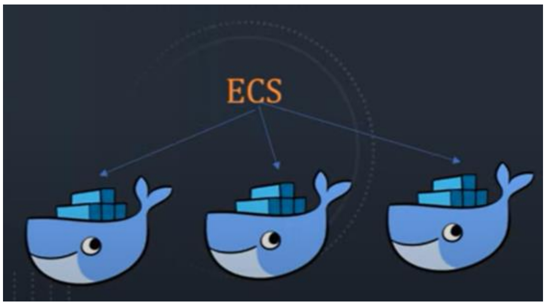
AWS ECS

- Amazon Elastic Container Service.

- ECS is a fully managed container orchestration service.

-Amazon ECS makes it easy to deploy, manage and scale Docker containers running applications, services and batch processes.

-It places containers across our cluster based on our resource needs and is integrated with familiar features like Elastic Load Balancing, EC2 security groups. EBS volumes and IAM roles.



AWS Fargate

-AWS Fargate is a serverless compute engine for containers that works with ECS & it allow you to run containers without having provision, configure & Scaling.

- Eliminate the need of EC2 instance.

- AWS Fargate is compatible with both Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS).

ENTIRE PROCESS:

Step 1: Developing Spring Boot Application- Creating Spring Boot application

Step 2: Dockerizing the Spring Boot Application- Writing a Dockerfile and building Docker images locally

Step 3: Building and Testing Docker Image Locally

Step 4: Pushing Docker Image to Docker Hub

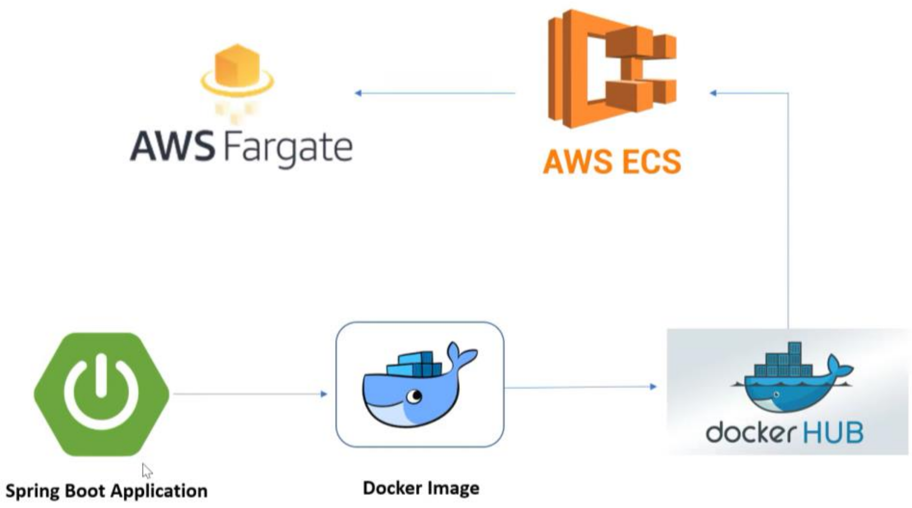
Step 5: Setting up AWS ECS CLUSTER- Creating an ECS Cluster on AWS

Step 6: Creating ECS Task Definition- Defining task definition specifying the Docker image.

Step 7: Creating ECS service- Setting up an ECS service using task definition

Step 8: Configuring AWS Fargate- Choosing Fargate as the launch type.

Step 9: Deploying on AWS Fargate- Deploying the ECS service and Fargate will manage the containers.



Scaling and Load Balancing

Implement automatic scaling mechanisms for the microservices based on resource usage.

Kubernetes provides Horizontal Pod Autoscaling (HPA) for automatically adjusting the number of replicas in a deployment based on observed CPU or memory utilization.

Add Resource Requests and Limits to your Deployment:

Updating Kubernetes deployment YAML files to to include resource requests and limits for CPU and memory:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: contactservice
spec:
  replicas: 3
  selector:
    matchLabels:
      app: contactservice
  template:
    metadata:
      labels:
        app: contactservice
    spec:
      containers:
      - name: contactservice
        image: 863099/springboot-image:latest
        ports:
        - containerPort: 3000
        resources:
          requests:
            memory: "64Mi"
            cpu: "250m"
          limits:
            memory: "128Mi"
            cpu: "500m"
```

Add Horizontal Pod Autoscaler (HPA):

Creating an HPA for each microservice to automatically adjust the number of replicas based on CPU or memory usage.

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: contactservice-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: contactservice
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Applying the updated deployment files and the new HPA files to Kubernetes cluster using commands:

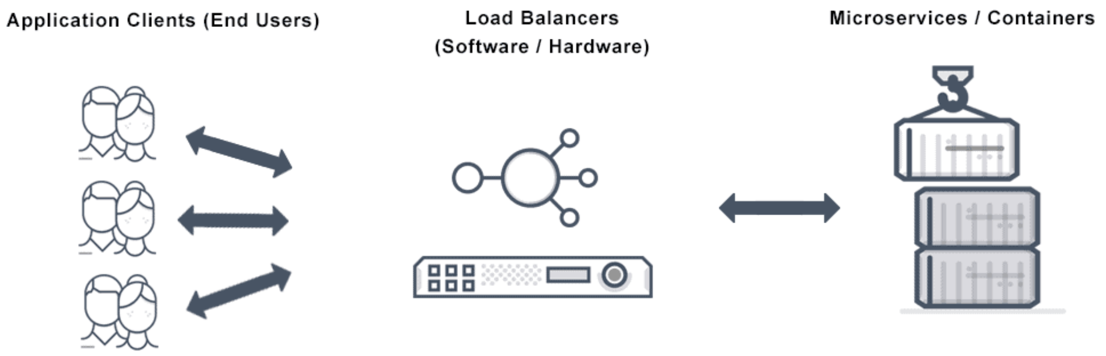
kubectl apply -f deployment-service1.yaml

kubectl apply -f hpa-service1.yaml

Monitoring the scaling activities using the following command:

kubectl get hpa

Explain how load balancing is achieved within the containerized environment.



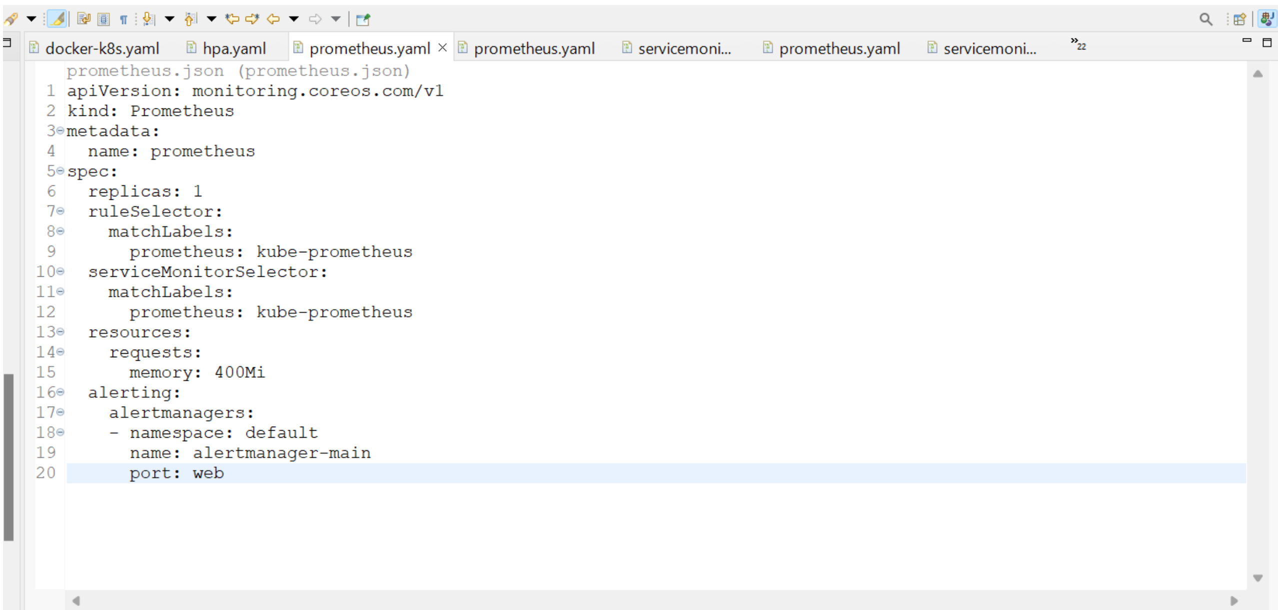
Load balancing is a technique used to distribute incoming network traffic across multiple servers or instances to enhance availability, scalability, and performance. In containerized environments like Kubernetes, load balancing ensures even distribution of requests among containers, optimizing resource utilization and allowing for uninterrupted performance of microservices-based applications. It enables updating or scaling individual microservices without disruptions by distributing traffic effectively across the containerized services. Load balancing is critical for achieving high availability and reliability in distributed systems.

Monitoring and Logging

[Set up monitoring and logging for the deployed microservices.]()

For Setting up monitoring and logging for deployed microservices, used popular tools such as Prometheus for monitoring and Elasticsearch-Fluentd-Kibana (EFK) stack for logging in a Kubernetes environment.

Monitoring with Prometheus:




```yaml
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  replicas: 1
  ruleSelector:
    matchLabels:
      prometheus: kube-prometheus
  serviceMonitorSelector:
    matchLabels:
      prometheus: kube-prometheus
  resources:
    requests:
      memory: 400Mi
  alerting:
    alertmanagers:
    - namespace: default
      name: alertmanager-main
      port: web
```

Applying the configuration using command:

kubectl apply -f prometheus.yaml

Setting Up Service Monitors:

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: contactservice
  labels:
    prometheus: kube-prometheus
spec:
  selector:
    matchLabels:
      app: contactservice
  endpoints:
  - port: metrics
```

Applying the configuration using command:

kubectl apply -f servicemonitor-service1.yaml

Logging with EFK (Elasticsearch, Fluentd, Kibana):

Deploying Elasticsearch:

```yaml
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: elasticsearch
spec:
  version: 7.10.1
  nodeSets:
  - name: default
    count: 1
    config:
      node.store.allow_mmap: false
```

Applying the configuration using command:

kubectl apply -f elasticsearch.yaml

Deploying Fluentd:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluentd
data:
  fluent.conf: |
    <source>
      @type tail
      path /var/log/containers/*.log
      pos_file /var/log/fluentd-containers.log.pos
      time_format %Y-%m-%dT%H:%M:%S.%NZ
      tag kubernetes.*
      format json
      read_from_head true
    </source>
    <match kubernetes.**>
      @type elasticsearch
      host elasticsearch-es-http
      port 9200
      logstash_format true
      include_tag_key true
      tag_key @log_name
      flush_interval 5s
    </match>
```

Applying the configuration using command:

kubectl apply -f fluentd.yaml

Deploying Kibana:

```yaml
apiVersion: kibana.k8s.elastic.co/v1
kind: Kibana
metadata:
  name: kibana
spec:
  version: 7.10.1
  count: 1
  elasticsearchRef:
    name: elasticsearch
```

Applying the configuration using command:

kubectl apply -f kibana.yaml

Showcase the ability to monitor the application's performance and troubleshoot issues.

Monitoring:

- Deploying Prometheus for metrics collection.
- Visualizing metrics using Grafana dashboards.
- Setting up alerts for critical metrics.

Troubleshooting:

- Using `*kubectl logs*` to inspecting container logs.
- Monitoring resource utilization with `*kubectl top*`.
- Implementing health probes for application reliability.
- Leveraging Kubernetes auto-scaling for dynamic scaling.
- Using distributed tracing for end-to-end visibility.
- Regularly audit access controls for security.
- Recreating pods to troubleshoot transient issues.