**Google Kubernetes**

**What is Container Orchestration?**

Container orchestration refers to the automated process of managing, deploying, scaling, networking, and monitoring containers in a distributed environment. It simplifies the management of containers by automating tasks like load balancing, scaling, failover, and resource allocation.

**Why is Container Orchestration Useful?**

1. **Scalability**: Automatically scale containers up or down based on demand.
2. **High Availability**: Ensure application reliability by redistributing workloads if a container fails.
3. **Resource Efficiency**: Optimize resource usage by scheduling containers efficiently across nodes.
4. **Ease of Deployment**: Simplify deploying applications across multiple environments.
5. **Monitoring and Logging**: Centralized tools for monitoring container health and performance.
6. **Load Balancing**: Automatically balance traffic across containers to prevent overloading.

**Example of Container Orchestration**

**Use Case: Scaling a Web Application**

Imagine you have a web application running in containers. During normal traffic, one container instance is enough. But during peak traffic, such as a flash sale, the system needs to handle thousands of users simultaneously.

**Steps Using Orchestration:**

1. **Container Deployment**:
   * Use a container orchestration tool (like Kubernetes) to deploy your container.
2. **Load Balancing**:
   * Automatically distribute incoming traffic across multiple instances of the container.
3. **Scaling**:
   * Scale out (add more container instances) during high traffic and scale back during normal traffic.
4. **Monitoring and Recovery**:
   * Continuously monitor container health and replace failed containers automaticall

**Container Orchestration Example: Microservices with Multiple Instances**

Let’s consider a scenario where:

* **Microservice A** has 10 instances running.
* **Microservice B** has 15 instances running.

These instances are distributed across multiple nodes in a container orchestration system, like Kubernetes.

**Scenario**

**1. Load Balancing**

* Incoming requests are evenly distributed among the 10 instances of Microservice A and 15 instances of Microservice B.
* The orchestration tool (e.g., Kubernetes) ensures no single instance is overloaded by routing requests intelligently.

**2. Scaling**

* If the workload increases, the orchestrator can automatically scale up the number of instances. For example:
  + Microservice A is scaled from 10 to 20 instances.
  + Microservice B is scaled from 15 to 30 instances.
* Conversely, during low traffic, instances are scaled down to save resources.

**3. High Availability**

* If one instance of Microservice A crashes, the orchestrator will detect the failure and replace it with a new instance.
* This ensures that 10 instances of Microservice A and 15 of Microservice B are always running.

**4. Deployment Updates**

* When deploying a new version of Microservice A:
  + The orchestrator performs a **rolling update**, gradually replacing old instances with new ones.
  + This ensures zero downtime.
* Similarly, the same applies to Microservice B.

**5. Service Discovery**

* The orchestrator registers all instances of Microservice A and Microservice B with a service registry.
* Other services and clients can discover the endpoints of Microservice A and Microservice B without worrying about the exact instance IPs.

**Container Orchestration Options in the Market**

1. **Kubernetes**:
   * **Description**: An open-source orchestration platform developed by Google.
   * **Key Features**:
     + Autoscaling
     + Load balancing
     + Rollbacks and rollouts
     + Persistent storage
   * **Use Case**: Large-scale distributed systems.
2. **Docker Swarm**:
   * **Description**: Native container orchestration tool by Docker.
   * **Key Features**:
     + Simple setup and integration with Docker.
     + Networking and scaling built-in.
   * **Use Case**: Small to medium-scale applications.
3. **Amazon ECS (Elastic Container Service)**:
   * **Description**: A fully managed container orchestration service by AWS.
   * **Key Features**:
     + Deep integration with AWS services.
     + Easy-to-use console for scaling.
   * **Use Case**: Applications running on AWS infrastructure.
4. **Amazon EKS (Elastic Kubernetes Service)**:
   * **Description**: Managed Kubernetes service by AWS.
   * **Key Features**:
     + Fully compatible with Kubernetes.
     + Seamless integration with AWS services.
   * **Use Case**: Kubernetes-based workloads on AWS.
5. **Google Kubernetes Engine (GKE)**:
   * **Description**: Managed Kubernetes service by Google Cloud.
   * **Key Features**:
     + Autoscaling
     + High availability
     + Integrated monitoring via Google Cloud Monitoring.
   * **Use Case**: Applications on Google Cloud.
6. **Azure Kubernetes Service (AKS)**:
   * **Description**: Managed Kubernetes service by Microsoft Azure.
   * **Key Features**:
     + Azure DevOps integration.
     + Advanced security features.
   * **Use Case**: Kubernetes workloads on Azure.
7. **Red Hat OpenShift**:
   * **Description**: Enterprise Kubernetes platform by Red Hat.
   * **Key Features**:
     + Built-in CI/CD tools.
     + Enhanced security and support.
   * **Use Case**: Enterprise applications.
8. **Nomad**:
   * **Description**: Lightweight orchestration tool by HashiCorp.
   * **Key Features**:
     + Single binary and lightweight.
     + Can orchestrate non-container workloads.
   * **Use Case**: Hybrid workloads.

**Deploying a Kubernetes Application on Google Kubernetes Engine (GKE)**

**Kubernetes Journey - Create a Deployment and a Service Exploring GKE in GCP Console**

**Main Topics for Notes**

1. **Setting Up the GCP Project and Cluster**
   * Configuring the Google Cloud project.
   * Accessing the Kubernetes cluster.
2. **Deploying the Application**
   * Creating a deployment for the application.
3. **Exposing the Application**
   * Making the deployment accessible externally.
4. **Monitoring Services**
   * Observing the status of services and deployments.
5. **Testing the Application**
   * Accessing the application via the external load balancer.

**Commands and Explanations**

**1. Setting Up the GCP Project and Cluster**

**Command**

gcloud config set project my-kubernetes-project-304910

**Step Name**

Set the Active GCP Project.

**What It Does**

This command sets the active Google Cloud project to my-kubernetes-project-304910. It ensures all subsequent gcloud commands target this project.

Command

gcloud container clusters get-credentials my-cluster --zone us-central1-c --project my-kubernetes-project-304910

**Step Name**

Authenticate Kubernetes Cluster.

**What It Does**

Retrieves the credentials for the Kubernetes cluster my-cluster in the specified zone (us-central1-c) and project. It updates the local kubeconfig file so that kubectl can interact with this cluster

**2. Deploying the Application**

**Command**

kubectl create deployment hello-world-rest-api --image=in28min/hello-world-rest-api:0.0.1.RELEASE

**Step Name**

Create a Kubernetes Deployment.

**What It Does**

Creates a deployment named hello-world-rest-api using the Docker image in28min/hello-world-rest-api:0.0.1.RELEASE. Kubernetes will manage and maintain the desired number of pods based on this deployment.

Command

kubectl get deployment

**Step Name**

View Deployment Status.

**What It Does**

Lists all deployments in the current namespace and shows their status, including the number of replicas, available pods, and their state.

**3. Exposing the Application**

**Command**

kubectl expose deployment hello-world-rest-api --type=LoadBalancer --port=8080

**Step Name**

Expose Deployment as a Service.

**What It Does**

Creates a service of type LoadBalancer for the hello-world-rest-api deployment. The service listens on port 8080 and forwards traffic to the pods created by the deployment.

Command

kubectl get services

**Step Name**

View Services.

**What It Does**

Lists all services in the current namespace, displaying details like service name, type, cluster IP, external IP (if assigned), and ports.

Command

kubectl get services –watch

**Step Name**

Monitor Service Status.

**What It Does**

Continuously monitors the services in the namespace, updating the display in real time as the status of services (e.g., external IP assignment) changes.

**4. Testing the Application**

**Command**

curl 35.184.204.214:8080/hello-world

**Step Name**

Test the Application.

**What It Does**

Sends an HTTP request to the application via its external load balancer's IP (35.184.204.214) and port (8080). If everything is working correctly, it returns a response from the application.

**Summary of Steps:**

1. Configure GCP project and authenticate the Kubernetes cluster.
2. Deploy the application as a Kubernetes deployment.
3. Expose the deployment as a LoadBalancer service to make it accessible externally.
4. Monitor the service to ensure it gets an external IP.
5. Test the application by sending a request to the assigned external IP and port.

**Kubernetes Journey - Scaling Deployments and Resizing Node Pools & Kubernetes Journey - Autoscaling, Config Map and Secrets**

**Main Topics for Notes**

1. **Manual Scaling of Nodes in the Cluster**
   * Adjusting the number of nodes in a Kubernetes cluster.
2. **Enabling Horizontal Pod Autoscaling**
   * Automatically scaling pods based on CPU utilization.
3. **Monitoring Horizontal Pod Autoscalers**
   * Viewing the status of the Horizontal Pod Autoscaler (HPA).

**Commands and Explanations**

**1. Manual Scaling of Nodes in the Cluster**

**Command**

gcloud container clusters resize my-cluster --node-pool default-pool --num-nodes=2 --zone=us-central1-c

**Step Name**

Resize Kubernetes Cluster Nodes.

**What It Does**

Manually resizes the default-pool in the my-cluster Kubernetes cluster to have 2 nodes. This ensures the cluster has enough capacity to handle additional workloads or distribute existing workloads more effectively.

**2. Enabling Horizontal Pod Autoscaling**

**Command**

kubectl autoscale deployment hello-world-rest-api --max=4 --cpu-percent=70

**Step Name**

Enable Horizontal Pod Autoscaling (HPA).

**What It Does**

Configures the hello-world-rest-api deployment to automatically scale its pods. It ensures:

* A maximum of 4 pods can be created.
* Pods will be scaled up or down to maintain an average CPU utilization of 70%.

This helps optimize resource usage and maintain performance under varying loads.

**3. Monitoring Horizontal Pod Autoscalers**

**Command**

kubectl get hpa

**Step Name**

View Horizontal Pod Autoscaler Status.

**What It Does**

Displays the status of the HPA, including:

* The target deployment it manages.
* The desired and current number of pods.
* Current CPU utilization and the threshold set (e.g., 70%).

This helps track the scaling behavior and ensures the autoscaling works as intended.

**Summary of Steps:**

1. Manually scale the number of cluster nodes to meet baseline capacity needs.
2. Configure Horizontal Pod Autoscaling to automatically adjust the number of pods based on CPU utilization.
3. Monitor the HPA to verify that scaling is occurring as expected.

**ConfigMaps and Secrets in Kubernetes**

**Main Topics for Notes**

1. **Creating and Managing ConfigMaps**
   * Storing configuration data in key-value pairs.
2. **Creating and Managing Secrets**
   * Storing sensitive data securely in Kubernetes.

**Commands and Explanations**

**1. Creating and Managing ConfigMaps**

**Command 1**

kubectl create configmap hello-world-config --from-literal=RDS\_DB\_NAME=todos

**Step Name**

Create a ConfigMap.

**What It Does**

Creates a ConfigMap named hello-world-config with a key-value pair:

* Key: RDS\_DB\_NAME
* Value: todos

ConfigMaps are used to decouple configuration details from application code, enabling easier updates without modifying container images.

Command 2

kubectl get configmap

**Step Name**

List All ConfigMaps.

**What It Does**

Displays a list of all ConfigMaps in the current namespace, including their names and statuses

Command 3

kubectl describe configmap hello-world-config

**2. Creating and Managing Secrets**

**Command 1**

kubectl create secret generic hello-world-secrets-1 --from-literal=RDS\_PASSWORD=dummytodos

**Step Name**

Create a Secret.

**What It Does**

Creates a Secret named hello-world-secrets-1 with a key-value pair:

* Key: RDS\_PASSWORD
* Value: dummytodos

Secrets are used to store sensitive data (e.g., passwords, API keys) securely in a Kubernetes cluster.

**Command 2**

kubectl get secret

**Step Name**

View ConfigMap Details.

**Step Name**

List All Secrets.

**What It Does**

Displays a list of all Secrets in the current namespace, including their names and types.

**What It Does**

Shows detailed information about the hello-world-config ConfigMap, including:

* Name, namespace, and creation timestamp.
* Data stored in the ConfigMap (e.g., RDS\_DB\_NAME=todos).

**Command 3**

bash

Copy code

kubectl describe secret hello-world-secrets-1

**Step Name**

View Secret Details.

**What It Does**

Shows detailed information about the hello-world-secrets-1 Secret, including:

* Name, namespace, and creation timestamp.
* Type of Secret (e.g., Opaque).
* Key names (but not the sensitive values)

**Summary of Steps:**

1. Use ConfigMaps to store non-sensitive configuration details like database names.
2. Use Secrets to securely store sensitive information like passwords.
3. Retrieve and inspect ConfigMaps and Secrets using kubectl commands to verify their creation and contents.

**Comparison: Your Project vs. GKE**

| **Feature/Requirement** | **Your Tool** | **Equivalent in GKE** | **Explanation** |
| --- | --- | --- | --- |
| **Centralized Configuration Management** | **Spring Cloud Config Server** | **ConfigMaps & Secrets** | In GKE, ConfigMaps manage configuration data, and Secrets manage sensitive data like credentials. They provide centralized and secure configuration management. |
| **Service Discovery** | **Eureka Naming Server** | **Kubernetes Service + DNS** | Kubernetes services handle service discovery and provide DNS-based resolution for accessing services within the cluster, eliminating the need for external discovery tools. |
| **Load Balancing** | **Spring Cloud Load Balancer** | **Kubernetes Services (LoadBalancer Type)** | Kubernetes provides built-in load balancing using the Service resource (type LoadBalancer or ClusterIP). External services use cloud-native load balancers like Google Load Balancer. |
| **Visibility and Monitoring** | **Zipkin (Distributed Tracing)** | **Stackdriver Monitoring (Cloud Operations) + OpenTelemetry** | GKE integrates with Google Cloud's Stackdriver for monitoring, logging, and tracing. OpenTelemetry can also be integrated for distributed tracing in Kubernetes clusters. |
| **API Gateway** | **Spring Cloud Gateway** | **Ingress + Istio (Service Mesh)** | GKE uses Ingress resources for API gateway functionality and Istio (optional) for advanced traffic management, security, and observability for service-to-service communication. |
| **Fault Tolerance** | **Resilience4j** | **Kubernetes Health Checks + Pod AutoScaling** | Kubernetes uses liveness and readiness probes to detect and recover from faults. Horizontal Pod Autoscaler and Circuit Breaker patterns can be implemented at the service level. |
| **Service Deployment** | **Spring Boot Deployments (WAR/JAR)** | **Kubernetes Deployments** | Kubernetes manages application deployment using Deployment resources, scaling, and updating services seamlessly while maintaining availability. |
| **Container Management** | **Docker** | **Kubernetes Pods + GKE Container Management** | GKE orchestrates containers using Pods and ensures efficient management of workloads. |
| **Scalability** | **Manual Scaling** | **Kubernetes Horizontal Pod Autoscaler (HPA)** | GKE automatically scales pods based on CPU, memory, or custom metrics using the HPA feature. |
| **Resiliency** | **Custom Logic via Resilience4j** | **Kubernetes Self-Healing (Restart Policies)** | GKE ensures pod-level self-healing with automatic restart and rescheduling in case of failures. |
| **Logging** | **Custom Logging Mechanism** | **Stackdriver Logging + Fluentd Integration** | GKE natively integrates with Google Cloud's Stackdriver for centralized logging. |
| **Network Security** | **Custom Spring Security/Firewall Rules** | **Kubernetes Network Policies + Istio Security Policies** | Kubernetes manages network traffic within the cluster using Network Policies, and Istio can enforce additional security controls for inter-service communication. |

**Additional Notes**

1. **ConfigMaps and Secrets**:
   * ConfigMaps are used for non-sensitive configuration.
   * Secrets are encrypted for sensitive data like passwords and API keys.
   * Both replace Spring Cloud Config Server in GKE setups.
2. **Service Discovery**:
   * Kubernetes automatically assigns DNS names to services (<service-name>.<namespace>.svc.cluster.local), simplifying communication.
3. **Ingress as API Gateway**:
   * Ingress acts as an API gateway for external traffic and integrates with Google Cloud Load Balancer.
4. **Distributed Tracing**:
   * You can integrate OpenTelemetry with GKE to replicate Zipkin's functionality.
5. **Fault Tolerance**:
   * Kubernetes ensures resiliency with liveness and readiness probes, ensuring unhealthy pods are restarted or replaced.

**What is a Cluster?**

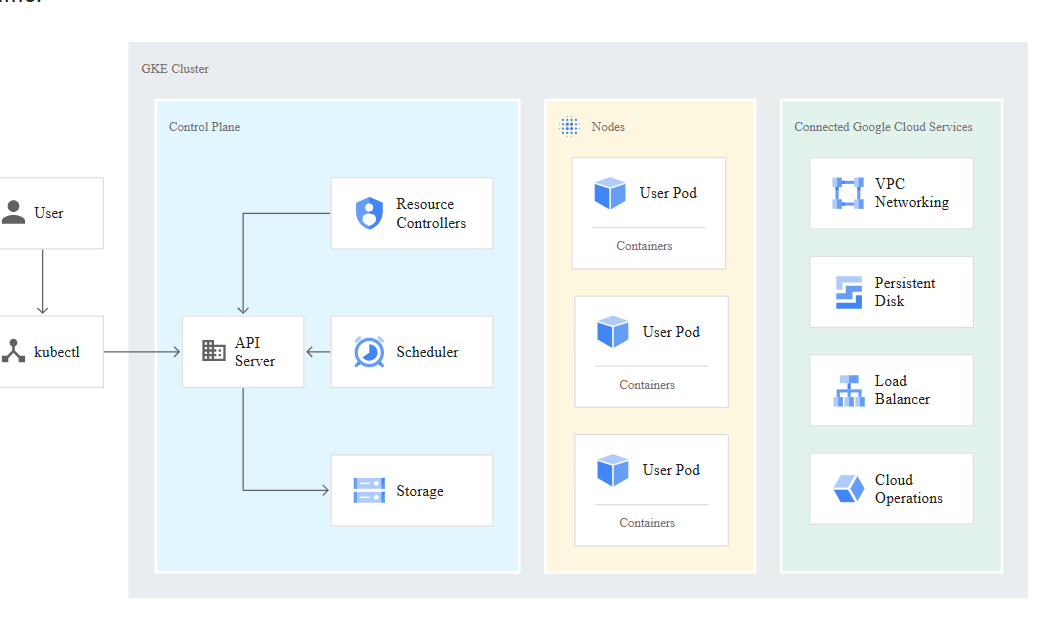
In Kubernetes and GKE (Google Kubernetes Engine), a **Cluster** is a set of machines (called nodes) that work together to run containerized applications. A cluster consists of two main parts:

1. **Control Plane (Master Node)**: Manages the cluster by scheduling workloads, monitoring health, and managing resources.
2. **Worker Nodes**: Runs the actual application workloads (containers).

A **Cluster** acts as a unified environment to deploy, manage, and scale applications.

**GKE Cluster Architecture**

Here’s a simple diagram of a GKE Cluster:

****

**Diagram Explanation**

1. **Control Plane (Master Node)**:
   * **API Server**:
     + Acts as the brain of Kubernetes.
     + Receives commands via kubectl from users or DevOps engineers and coordinates cluster activities.
   * **Scheduler**:
     + Assigns workloads (e.g., pods) to the appropriate worker nodes based on resource requirements.
   * **Resource Controllers**:
     + Ensure the desired state of applications (e.g., maintaining a specific number of pods).
   * **Storage**:
     + Stores cluster state and configuration data using etcd.
2. **Nodes (Worker Nodes)**:
   * Each **Node** runs one or more **Pods**, which contain your application containers.
   * The **Kubelet** runs on each worker node to ensure that the containers in each pod are healthy.
   * **User Pods**:
     + These are the actual running units of your microservices or applications.
3. **Connected Google Cloud Services**:
   * **VPC Networking**:
     + Provides internal and external networking for your applications.
   * **Persistent Disk**:
     + Ensures data persistence for stateful applications.
   * **Load Balancer**:
     + Distributes traffic evenly across nodes or pods.
   * **Cloud Operations**:
     + Monitors and logs cluster activity for troubleshooting and optimization.

**How It Works in Real Life**

Let’s use an **E-commerce website** as an example:

1. **Control Plane**:
   * Decides where to deploy your services like product catalog, payment gateway, and user authentication.
   * Monitors these services and ensures they are always running.
2. **Worker Nodes**:
   * Run the actual application services (e.g., product catalog API in one pod, payment service in another pod).
   * Each node can handle multiple parts of your e-commerce application.
3. **Load Balancer**:
   * When a user visits your website, the load balancer distributes the traffic across worker nodes to ensure no single node is overloaded.
4. **Networking**:
   * Ensures that the product service can communicate with the payment service internally within the cluster.

**What is Service Discovery?**

**Service discovery** is the process by which services in a distributed system (e.g., microservices architecture) dynamically locate each other to communicate and work together. In modern systems with multiple services running across different environments (e.g., containers, virtual machines), service discovery ensures that services can find and connect to one another, even as they scale, move, or restart.

**Why is Service Discovery Important?**

In traditional monolithic systems, all components are within the same application, so no explicit discovery is required. However, in microservices or cloud-native systems:

* Services are **deployed independently** and may run on **dynamic IP addresses**.
* Services may **scale up or down** (new instances are created or destroyed dynamically).
* **Hardcoding service addresses** isn’t feasible due to the dynamic nature of these systems.

Service discovery solves this problem by enabling services to find each other **dynamically**.

**Types of Service Discovery**

1. **Client-Side Service Discovery**:
   * The **client** is responsible for finding the service it needs to communicate with.
   * The client queries a **service registry** (a centralized database that tracks all available services and their locations).
   * Example tools:
     + Eureka (Netflix)
     + Consul (HashiCorp)
2. **Server-Side Service Discovery**:
   * The **load balancer or gateway** sits between the client and the service.
   * The client sends requests to the load balancer, which resolves the actual service instance using the service registry.
   * Example tools:
     + Kubernetes Service (ClusterIP/LoadBalancer)
     + AWS Elastic Load Balancer (ELB)

**Components of Service Discovery**

1. **Service Registry**:
   * A database or system that keeps track of all available services and their addresses.
   * Example: **Eureka, Consul, Zookeeper**
2. **Service Instances**:
   * The actual services (microservices, applications, etc.) that need to be discovered.
3. **Discovery Mechanism**:
   * A way for clients to discover the services (e.g., direct query to the registry or routing through a load balancer).

**How Does It Work? (Real-World Example)**

Imagine a **Food Delivery App** with services like:

* **Order Service**: Manages customer orders.
* **Payment Service**: Handles payments.
* **Delivery Service**: Tracks and assigns deliveries.

**Without Service Discovery:**

* Each service needs the static IP/port of every other service.
* If a service instance changes (e.g., scaling or failure), manual updates are needed.
* This is error-prone and not scalable.

**With Service Discovery:**

1. All services **register themselves** in a **service registry** (e.g., Eureka, Consul).
2. When a service (e.g., Order Service) needs to contact another service (e.g., Payment Service):
   * It queries the service registry for the current IP/port of the Payment Service.
   * The registry responds with the dynamic address of an available Payment Service instance.
3. The client connects to the appropriate instance dynamically.

**Tools for Service Discovery**

* **Eureka (Netflix)**: Widely used in Spring Cloud microservices.
* **Consul (HashiCorp)**: Provides service discovery and configuration.
* **Kubernetes DNS**: Automatically discovers services in Kubernetes clusters.
* **Zookeeper**: Provides service discovery and distributed coordination.

**Service Discovery in Kubernetes: Example with Currency Exchange and Currency Conversion**

In Kubernetes, **service discovery** is handled automatically through its internal **DNS-based mechanism**. When deploying microservices like **Currency Exchange** and **Currency Conversion** in Kubernetes, Kubernetes ensures that services can communicate with each other seamlessly using their **service names** instead of hardcoded IP addresses.

**Example Scenario**

* **Currency Exchange Service**: Provides exchange rates for different currencies (e.g., USD to INR).
* **Currency Conversion Service**: Converts an amount from one currency to another by calling the Currency Exchange service.

**How Kubernetes Handles Service Discovery**

1. **Service Creation**:
   * Each service (Currency Exchange and Currency Conversion) is deployed as a **Kubernetes Deployment**.
   * Kubernetes exposes each Deployment using a **Service** (ClusterIP or LoadBalancer).
   * The Service automatically gets a **DNS name**.

Example:

* + currency-exchange DNS name: currency-exchange.default.svc.cluster.local
  + currency-conversion DNS name: currency-conversion.default.svc.cluster.local

1. **DNS-Based Resolution**:
   * Kubernetes provides an internal DNS server. When **Currency Conversion** wants to call **Currency Exchange**, it uses the Service DNS name instead of hardcoding an IP address.
   * Example:

java

Copy code

String exchangeUrl = "http://currency-exchange.default.svc.cluster.local:8000";

1. **Dynamic Scaling**:
   * If the Currency Exchange service scales up or down (e.g., from 2 instances to 5), the Kubernetes Service automatically updates its backend endpoints. The Currency Conversion service doesn't need to know about these changes.
2. **Health Checks**:
   * Kubernetes ensures that only healthy pods of the Currency Exchange service are part of the DNS resolution. If a pod crashes, Kubernetes removes it from the Service endpoints.

**Step 3: Currency Conversion Calls Currency Exchange**

In the Currency Conversion application code, use the Kubernetes DNS name for the Currency Exchange service:

String exchangeUrl = "http://currency-exchange.default.svc.cluster.local:8000/currency-exchange/from/USD/to/INR";

// Call the Currency Exchange Service using this URL

**Key Benefits of Kubernetes Service Discovery**

1. **No Hardcoding of IP Addresses**:
   * Services communicate using DNS names like currency-exchange.default.svc.cluster.local.
2. **Dynamic Scaling**:
   * If the Currency Exchange service scales up (e.g., more pods added), Kubernetes updates the DNS records automatically.
3. **Fault Tolerance**:
   * Kubernetes removes unhealthy pods from the Service endpoints, ensuring reliable communication.
4. **Namespace Isolation**:
   * Services in different namespaces can coexist without conflict. For example:
     + currency-exchange.dev.svc.cluster.local for the development environment.
     + currency-exchange.prod.svc.cluster.local for production.

**Environment variable concepts**

In Kubernetes, along with the **DNS-based service discovery**, Kubernetes also provides **environment variables** for each Service within the pods where it is used. This mechanism is especially useful when you want to programmatically access service details without relying on DNS.

Let's clarify the concepts:

**How Kubernetes Sets Service Environment Variables**

When a Service is created, Kubernetes automatically sets certain environment variables in the pods of other deployments running in the same namespace.

For example:

* For a service named currency-exchange, Kubernetes will automatically inject environment variables like:

CURRENCY\_EXCHANGE\_SERVICE\_HOST=<Cluster IP of Service>

CURRENCY\_EXCHANGE\_SERVICE\_PORT=<Service Port>

These environment variables are accessible inside any pod running in the same namespace.

CURRENCY\_EXCHANGE\_SERVICE\_HOST=10.96.12.34 # Cluster IP of Service

CURRENCY\_EXCHANGE\_SERVICE\_PORT=8000 # Port exposed by Service

**Accessing the Service Using Environment Variables**

In the **Currency Conversion** service code, you can use the environment variables like this:

String host = System.getenv("CURRENCY\_EXCHANGE\_SERVICE\_HOST");

String port = System.getenv("CURRENCY\_EXCHANGE\_SERVICE\_PORT");

// Construct the URL using the environment variables

String exchangeUrl = "http://" + host + ":" + port + "/currency-exchange/from/USD/to/INR";

This ensures that even if the Service's IP address changes (e.g., due to scaling or redeployment), the application will still be able to find the correct Service.

**Comparing DNS and Environment Variables**

| **Feature** | **DNS-Based Service Discovery** | **Environment Variables** |
| --- | --- | --- |
| **Usage** | Uses DNS names like currency-exchange.default.svc.cluster.local. | Uses environment variables like CURRENCY\_EXCHANGE\_SERVICE\_HOST. |
| **Dynamic Updates** | Automatically updated for scaling or new pods. | Injected when the pod starts (static). |
| **Preferred Use Case** | For most scenarios, especially when calling external namespaces. | For simple cases or legacy applications. |
| **Cross-Namespace Support** | Fully supported with DNS names. | Not supported without custom configuration. |