Step #1: Create a GoLang Program, Dockerize it, and Push to DockerHub

1. Create the GoLang Program (main.go):

Go

```
package main

import (
	"fmt"
	"net/http"
	"time"
)

func handler(w http.ResponseWriter, r *http.Request) {
	currentTime := time.Now().Format(time.RFC1123)
	fmt.Fprintf(w, "Current Date & Time: %s\n", currentTime)
}

func main() {
	http.HandleFunc("/", handler)
	fmt.Println("Server listening on port 8080...")
	http.ListenAndServe(":8080", nil)
}
```

Explanation:

- We import the necessary packages: fmt for formatting and printing, net/http for creating a web server, and time for getting the current date and time.
- The handler function is an HTTP handler that gets the current time using time.Now() and formats it according to time.RFC1123. It then writes this formatted time to the HTTP response writer.
- In the main function, we register the handler function to handle requests to the root path (/).
- We start an HTTP server listening on port 8080.

2. Create a Dockerfile:

Dockerfile

```
FROM golang:1.22-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o main .

FROM alpine:latest
WORKDIR /root
COPY --from=builder /app/main .
EXPOSE 8080
CMD ["./main"]
```

Explanation:

- **FROM golang:1.22-alpine AS builder**: We start with the official Go image with Alpine Linux as the base for building. We name this stage builder.
- **WORKDIR /app**: We set the working directory inside the container to /app.
- **COPY go.mod go.sum ./**: We copy the Go module definition files first to leverage Docker's caching.
- **RUN go mod download**: We download the Go module dependencies.
- **COPY . .**: We copy the rest of the application code.
- **RUN go build -o main .**: We build the Go application and name the output binary main.
- **FROM alpine:latest**: We start with a lightweight Alpine Linux image for the final runtime image.
- **WORKDIR /root**: We set the working directory to /root.
- **COPY --from=builder /app/main .**: We copy the compiled main binary from the builder stage.
- **EXPOSE 8080**: We declare that the container will listen on port 8080.
- **CMD ["./main"]**: We specify the command to run when the container starts.

3. Create go.mod (if you haven't already):

```
module datetime-app

go 1.22
```

Run go mod init datetime-app in your project directory if you don't have this file.

4. Build the Docker Image:

Open your terminal in the directory containing the Dockerfile and main.go and run:

```
Bash

docker build -t <your_dockerhub_username>/datetime-app:latest .
```

Replace <your_dockerhub_username> with your actual DockerHub username.

5. Log in to DockerHub:

```
Bash

docker login
```

Enter your DockerHub username and password.

6. Push the Docker Image to DockerHub:

```
Bash

docker push <your_dockerhub_username>/datetime-app:latest
```

Step #2: Deploy to Kubernetes with 2 Replicas (Declarative Approach)

You'll need access to a Kubernetes cluster (Minikube, K3s, Kind, or a cloud-based cluster).

1. Create a Deployment YAML file (deployment.yaml):

YAML

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: datetime-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: datetime-app
  template:
    metadata:
      labels:
        app: datetime-app
    spec:
      containers:
      - name: datetime-app-container
        image: <your_dockerhub_username>/datetime-app:latest
        ports:
        - containerPort: 8080
```

Explanation:

- **apiVersion: apps/v1**: Specifies the Kubernetes API version for Deployments.
- **kind: Deployment**: Defines the type of Kubernetes object as a Deployment.
- **metadata.name**: Sets the name of the Deployment.
- **spec.replicas: 2**: Ensures that there will always be 2 running instances (pods) of your application.
- **spec.selector.matchLabels**: Defines how the Deployment identifies the Pods it manages (based on the app: datetime-app label).
- **spec.template.metadata.labels**: Labels to be applied to the Pods created by this Deployment.

- **spec.template.spec.containers**: Defines the containers that will run inside the Pods.
  - **name**: Name of the container.
  - **image**: The Docker image to use (replace with your DockerHub image).
  - **ports.containerPort**: The port your application listens on inside the container.

2. Apply the Deployment to your Kubernetes Cluster:

Use kubectl to apply the Deployment:

Bash

```
kubectl apply -f deployment.yaml
```

3. Verify the Deployment:

Bash

```
kubectl get deployments
kubectl get pods
```

You should see the datetime-app-deployment with 2 replicas available and two Pods running with the app: datetime-app label.

Step #3: Expose the App to the Internet (on WAN)

To expose your application to the internet, you'll typically use a Kubernetes Service of type LoadBalancer. This requires a cloud provider with a LoadBalancer implementation (like GCP, AWS, Azure) or a local setup with a tool like MetalLB.

1. Create a Service YAML file (service.yaml):

YAML

```
apiVersion: v1
kind: Service
metadata:
  name: datetime-app-service
spec:
  selector:
    app: datetime-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: LoadBalancer
```

Explanation:

- **apiVersion: v1**: Specifies the Kubernetes API version for Services.
- **kind: Service**: Defines the type of Kubernetes object as a Service.
- **metadata.name**: Sets the name of the Service.
- **spec.selector**: Matches Pods with the label app: datetime-app (the same label used in the Deployment).
- **spec.ports**: Defines how traffic will be routed to the Pods.
  - **protocol: TCP**: Specifies the TCP protocol.
  - **port: 80**: The port on which the Service will be exposed externally.
  - **targetPort: 8080**: The port on which the containers in the Pods are listening.
- **spec.type: LoadBalancer**: Requests that a cloud provider provision a load balancer to expose the Service externally.

2. Apply the Service to your Kubernetes Cluster:

Bash

```
kubectl apply -f service.yaml
```

3. Get the External IP Address (if available):

Bash

```
kubectl get service datetime-app-service
```

Look for the EXTERNAL-IP column. It might take a few minutes for a cloud provider to provision a LoadBalancer and assign an external IP.

Important Considerations for Local Clusters (Minikube, K3s, Kind):

- **Minikube:** You can expose services using minikube service --url datetime-app-service. This will give you a local URL (often using NodePort or a tunnel) to access your application, but it's not directly on the public internet (WAN). To expose externally, you might need to explore options like minikube tunnel or ingress controllers.
- **K3s:** K3s includes Traefik as an ingress controller by default. You would typically use Ingress resources to expose your application externally.
- **Kind:** Kind focuses on local development and doesn't provide a built-in LoadBalancer implementation. You might need to use a local port forwarding mechanism or an Ingress controller.
- **Play with Kubernetes:** This platform often provides a way to expose services via NodePort or Ingress, but a direct LoadBalancer might not always be available.

Using Qwiklabs or other Platforms:

The provided resource hints (Qwiklabs, Play with Kubernetes) will give you a temporary Kubernetes environment where you can experiment with creating Deployments and Services. Follow the specific instructions and tools provided by those platforms to expose your service externally. They might offer different mechanisms than a standard LoadBalancer type, such as NodePort with a public IP or an Ingress controller.

Remember to replace <your_dockerhub_username> in the YAML files with your actual DockerHub username. This comprehensive guide should help you complete the task.