# .NET 6 Console Application Containerization Documentation

This document provides an overview of the modifications made to containerize a .NET 6 console project and integrate it with Kubernetes for deployment. The project has been updated to accommodate different environments, support network shared folders, and ensure secure handling of secrets and configurations.

## 1. Configuration Modifications

### 1.1 Local Configuration Files

- When running locally, the application uses XML or JSON configuration files.  
- Previously, these configuration files were stored in Windows directories (e.g., `C:\`). To ensure compatibility within containers, these paths have been modified to use the `/app` folder inside the container.  
- The containerized application reads the configuration files from the `/app` folder, which serves as the root for application-specific configurations.

### 1.2 Network Shared Folders

- To access network shared folders, persistent volume mounts are used in Kubernetes.  
- The Kubernetes PersistentVolume (PV) and PersistentVolumeClaim (PVC) configuration allows the containerized application to read and write to network shared folders seamlessly.  
- The mounted volumes are configured in the Kubernetes deployment YAML to provide access as needed for each environment (e.g., development, production).

## 2. Secret Management

### 2.1 Kubernetes Secrets

- Secrets are managed securely through Kubernetes Secrets and are mounted based on the environment.  
- Monitoring secrets, API keys, database connection strings with passwords, and S3 bucket credentials are stored in Kubernetes Secrets.  
- Secrets are environment-specific and named accordingly, such as `dev\_db\_secrets` for development and `prd\_db\_secrets` for production.  
- These secrets are mounted into the container as environment variables or as files, depending on the requirements of the application.

## 3. Job Scheduling

### 3.1 CronJobs

- All previously scheduled jobs have been converted into Kubernetes CronJobs.  
- Each CronJob is defined in a corresponding `cronjobs.yaml` file.  
- The necessary arguments are passed to the application through the `args` field in the YAML file, allowing parameterization for different tasks.  
- CronJobs are deployed through Jenkins, using a `jenkins.kubernetes` file. Parameters such as CronJob schedules, CPU, and memory limits are adjusted based on the environment (e.g., development, staging, production).

#### 3.2 Demo Code for CronJob

Below is a sample `cronjobs.yaml` file for creating a Kubernetes CronJob:

apiVersion: batch/v1  
kind: CronJob  
metadata:  
 name: sample-cronjob  
spec:  
 schedule: "0 6 \* \* \*" # Runs every day at 6 AM  
 jobTemplate:  
 spec:  
 template:  
 spec:  
 containers:  
 - name: sample-container  
 image: myregistry/sample-dotnet-app:latest  
 args:  
 - "--task=DailyReport"  
 env:  
 - name: CONFIG\_FILE\_LOCATION  
 value: "/app/config/settings.json"  
 - name: DB\_SECRET  
 valueFrom:  
 secretKeyRef:  
 name: dev\_db\_secrets  
 key: db\_connection\_string  
 restartPolicy: OnFailure  
 backoffLimit: 4

## 4. CI/CD Pipeline Integration

### 4.1 Jenkins Integration

- Jenkins is used for automating the deployment of CronJobs and other Kubernetes resources.  
- Environment-specific parameters are passed through Jenkins for efficient and consistent deployment across different environments.  
- Jenkins triggers Kubernetes deployments using the provided `jenkins.kubernetes` file.

#### 4.2 Demo Jenkins Pipeline Script

Below is a sample Jenkins pipeline script for deploying the Kubernetes CronJob:

pipeline {  
 agent any  
 environment {  
 KUBE\_CONFIG = credentials('kubeconfig')  
 }  
 stages {  
 stage('Deploy CronJob') {  
 steps {  
 script {  
 sh 'kubectl apply -f cronjobs.yaml'  
 }  
 }  
 }  
 }  
}