**CI/CD pipeline for a Node.js-based "Hello World" application using Docker, Backstage, VSCode, and GitHub Hooks**

**1. Project Structure**

Ensure your project structure looks like this:

hello-world-app/

├── src/

│ └── index.js

├── Dockerfile

├── .github/

│ └── workflows/

│ └── ci-cd.yml

├── package.json

└── README.md

**2. index.js (Node.js Hello World Application)**

// src/index.js

const express = require('express');

const app = express();

const port = 3000;

app.get('/', (req, res) => {

res.send('Hello World!');

});

app.listen(port, () => {

console.log(`App listening at http://localhost:${port}`);

});

**3. package.json**

{

"name": "hello-world-app",

"version": "1.0.0",

"description": "A simple Node.js Hello World app",

"main": "src/index.js",

"scripts": {

"start": "node src/index.js",

"test": "echo \"Error: no test specified\" && exit 1"

},

"dependencies": {

"express": "^4.17.1"

}

}

**4. Dockerfile**

# Use the official Node.js image from Docker Hub

FROM node:18

# Set the working directory

WORKDIR /app

# Copy package.json and package-lock.json

COPY package\*.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application code

COPY . .

# Expose the application port

EXPOSE 3000

# Start the application

CMD ["npm", "start"]

**5. GitHub Actions Workflow (.github/workflows/ci-cd.yml)**

This workflow will:

1. Build and test the application on every push to the main branch.
2. Build and push the Docker image to Docker Hub.
3. Trigger a deployment (you can integrate this with Backstage or Kubernetes).

name: CI/CD Pipeline

on:

push:

branches:

- main

jobs:

build-and-test:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Set up Node.js

uses: actions/setup-node@v3

with:

node-version: 16

- name: Install dependencies

run: npm install

- name: Run tests

run: npm test

build-and-push-docker:

runs-on: ubuntu-latest

needs: build-and-test

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Log in to Docker Hub

uses: docker/login-action@v2

with:

username: ${{ secrets.DOCKER\_HUB\_USERNAME }}

password: ${{ secrets.DOCKER\_HUB\_TOKEN }}

- name: Build Docker image

run: docker build -t your-dockerhub-username/hello-world-app:latest .

- name: Push Docker image

run: docker push your-dockerhub-username/hello-world-app:latest

deploy:

runs-on: ubuntu-latest

needs: build-and-push-docker

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Deploy to Backstage (or Kubernetes)

run: |

echo "Deploying to Backstage..."

# Add your deployment script here

# Example: kubectl apply -f deployment.yaml

**6. GitHub Secrets**

Set up the following secrets in your GitHub repository:

* DOCKER\_HUB\_USERNAME: Your Docker Hub username.
* DOCKER\_HUB\_TOKEN: Your Docker Hub access token.

**7. Backstage Integration**

If you’re using Backstage, you can create a catalog-info.yaml file to register your application in Backstage. Example:

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: hello-world-app

description: A simple Node.js Hello World application

tags:

- nodejs

- docker

annotations:

backstage.io/techdocs-ref: dir:.

spec:

type: service

lifecycle: production

owner: default/team

**8. Workflow Steps**

1. **Develop in VSCode**: Write your code and test it locally.
2. **Commit and Push to GitHub**:

git add .

git commit -m "Initial commit"

git push origin main

1. **GitHub Actions**:
   * The CI/CD pipeline will automatically:
     + Build and test the application.
     + Build and push the Docker image to Docker Hub.
     + Trigger a deployment (if configured).

Setting up secrets in GitHub is essential for securely storing sensitive information like API keys, tokens, and credentials. These secrets can then be used in GitHub Actions workflows without exposing them in your code. Here's a step-by-step guide to setting up secrets in GitHub:

**Step 1: Navigate to Your Repository**

1. Go to your GitHub repository.
2. Click on the **Settings** tab in the top navigation bar.

**Step 2: Access the Secrets Section**

1. In the left sidebar, scroll down to the **Security** section.
2. Click on **Secrets and variables**.
3. Select **Actions** (for GitHub Actions secrets).

**Step 3: Add a New Secret**

1. Click the **New repository secret** button.
2. Enter the **Name** of the secret (e.g., DOCKER\_HUB\_USERNAME).
3. Enter the **Value** of the secret (e.g., your Docker Hub username).
4. Click **Add secret** to save it.

**Step 4: Add Multiple Secrets**

Repeat the process to add all the secrets you need. For example:

* DOCKER\_HUB\_USERNAME: Your Docker Hub username.
* DOCKER\_HUB\_TOKEN: Your Docker Hub access token.
* AWS\_ACCESS\_KEY\_ID: AWS access key (if deploying to AWS).
* AWS\_SECRET\_ACCESS\_KEY: AWS secret key.

**Step 5: Use Secrets in GitHub Actions**

In your GitHub Actions workflow file (e.g., .github/workflows/ci-cd.yml), you can reference the secrets using the ${{ secrets.SECRET\_NAME }} syntax. For example:

jobs:

build-and-push-docker:

runs-on: ubuntu-latest

steps:

- name: Log in to Docker Hub

uses: docker/login-action@v2

with:

username: ${{ secrets.DOCKER\_HUB\_USERNAME }}

password: ${{ secrets.DOCKER\_HUB\_TOKEN }}

**Step 6: Verify Secrets**

1. Push a change to your repository to trigger the GitHub Actions workflow.
2. Check the workflow logs to ensure the secrets are being used correctly.
3. Secrets are masked in the logs, so you won't see their actual values.

Integrating **Backstage** with **Harness** for a CI/CD pipeline for a simple Flask web app involves several steps. Below is a comprehensive guide to set up the pipeline using **VSCode**, **Docker**, and **GitHub**.

**Prerequisites**

1. **Backstage** instance set up and running.
2. **Harness** account with a project and pipeline configured.
3. **GitHub** repository for your Flask app.
4. **Docker** installed locally and a Docker Hub account.
5. **VSCode** for local development.

**Step 1: Create a Simple Flask App**

1. Open VSCode and create a new project folder.
2. Inside the folder, create a app.py file for your Flask app:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

return 'Hello, World!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

1. Create a requirements.txt file for dependencies:

Copy

Flask==2.0.1

**Step 2: Dockerize the Flask App**

1. Create a Dockerfile in the project root:

# Use an official Python runtime as a parent image

FROM python:3.9-slim

# Set the working directory

WORKDIR /app

# Copy the requirements file

COPY requirements.txt .

# Install dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Copy the application code

COPY . .

# Expose the application port

EXPOSE 5000

# Run the application

CMD ["python", "app.py"]

1. Build and test the Docker image locally:

docker build -t your-dockerhub-username/flask-app:latest .

docker run -p 5000:5000 your-dockerhub-username/flask-app:latest

1. Push the Docker image to Docker Hub:

docker push your-dockerhub-username/flask-app:latest

**Step 3: Set Up GitHub Repository**

1. Push your Flask app code to a GitHub repository.
2. Add a .github/workflows/ci-cd.yml file for GitHub Actions:

name: CI/CD Pipeline

on:

push:

branches:

- main

jobs:

build-and-test:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.9'

- name: Install dependencies

run: pip install -r requirements.txt

- name: Run tests

run: |

echo "Add your test commands here"

build-and-push-docker:

runs-on: ubuntu-latest

needs: build-and-test

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Log in to Docker Hub

uses: docker/login-action@v2

with:

username: ${{ secrets.DOCKER\_HUB\_USERNAME }}

password: ${{ secrets.DOCKER\_HUB\_TOKEN }}

- name: Build Docker image

run: docker build -t your-dockerhub-username/flask-app:latest .

- name: Push Docker image

run: docker push your-dockerhub-username/flask-app:latest

**Step 4: Set Up Harness for Deployment**

1. Log in to your **Harness** account.
2. Create a new **Pipeline**:
   * Go to **Pipelines** > **Create Pipeline**.
   * Name it Flask App Deployment.
3. Add a **Stage** for deployment:
   * Choose **Deployment** as the stage type.
   * Select your target environment (e.g., Kubernetes, AWS ECS, etc.).
4. Configure the **Service**:
   * Add a new service named flask-app.
   * Use the Docker image you pushed to Docker Hub (your-dockerhub-username/flask-app:latest).
5. Configure the **Infrastructure**:
   * Define the target infrastructure (e.g., Kubernetes cluster, AWS ECS cluster).
6. Add a **Trigger**:
   * Set up a trigger to automatically deploy when a new Docker image is pushed to Docker Hub.

**Step 5: Integrate Backstage with Harness**

1. In **Backstage**, create a catalog-info.yaml file for your Flask app:

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: flask-app

description: A simple Flask web application

tags:

- python

- flask

- docker

spec:

type: service

lifecycle: production

owner: default/team

1. Add a **Harness CI/CD** annotation to the catalog-info.yaml file:

annotations:

harness.io/project-slug: your-harness-project-slug

harness.io/service-id: your-harness-service-id

harness.io/env-id: your-harness-environment-id

1. Register the component in Backstage:
   * Go to the **Catalog** in Backstage.
   * Click **Import** and provide the path to your catalog-info.yaml file.

**Step 6: Test the End-to-End Pipeline**

1. Make a change to your Flask app code in VSCode.
2. Commit and push the changes to GitHub:

git add .

git commit -m "Update Flask app"

git push origin main

1. GitHub Actions will automatically:
   * Build and test the app.
   * Build and push the Docker image to Docker Hub.
2. Harness will automatically:
   * Deploy the new Docker image to your target environment.
3. Verify the deployment by accessing your Flask app.