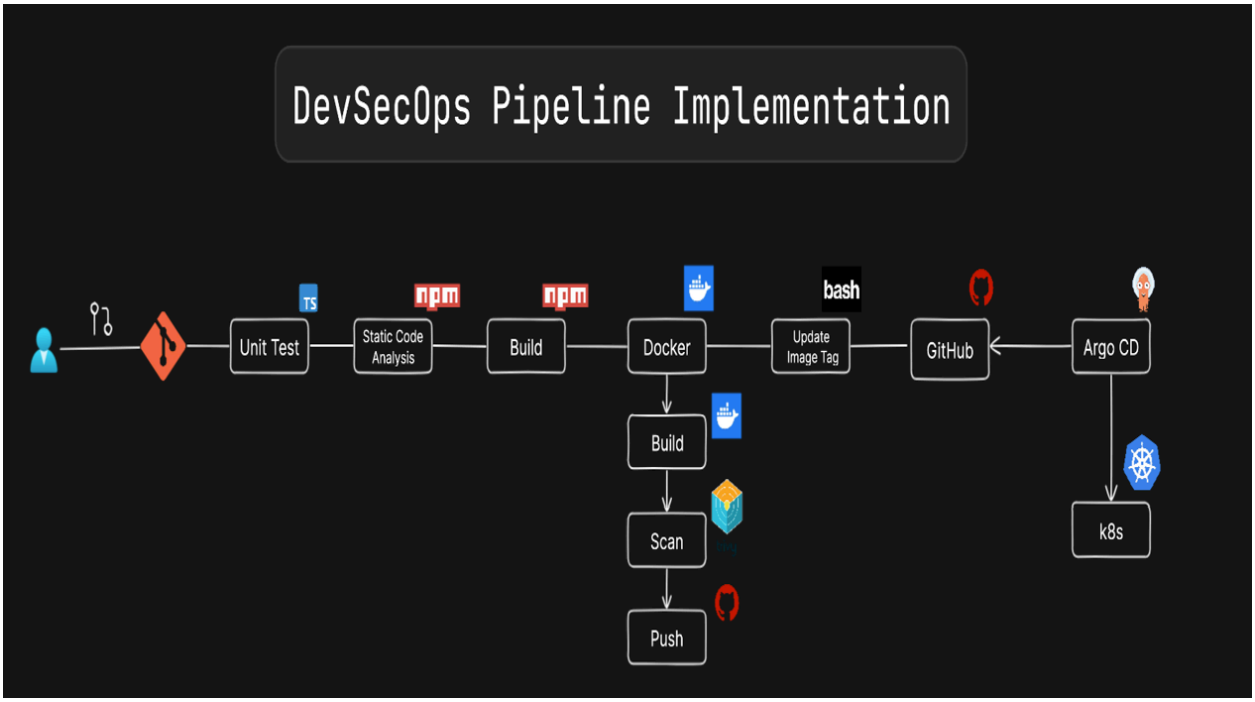
DevSecOps Pipeline CI/CD Implementation - A Complete Guide



Introduction:

In this article, we will implement a complete CI/CD pipeline of a Type Script application which is build using Vite and NPM, following DevSecOps best practices. By integrating security into every phase of development and deployment, we ensure a secure, automated and efficient workflow making this a fundamental part of the software delivery process.

Overview:

1. What is DevSecOps ?
2. Why DevSecOps ?
3. How to Run the pipeline locally ?

4. Containerization
5. DevSecOps Implementation
6. Test and Validate Application Changes
6. Creation of EC2 instance
7. Kubernetes Implementation using Kind

8.1. Kubernetes Implementation using EKS

9. Logging into ArgoCD
8. End to End Implementation Demo

1. What is DevSecOps ?

Many people limit DevSecOps to just security in CI/CD pipelines whereas DevSecOps isn't just about scanning code for vulnerabilities in CI/CD. It is about taking security into every aspect of DevOps, which means integrating security at every stage of the Software Development Life Cycle (SDLC). In simple terms, it extends beyond CI/CD to ensure security in infrastructure provisioning, managing secrets and runtime protection.

Example:

Infrastructure As Code (IAC): Using Hashicorp Vault to store sensitive credentials instead of hardcoding them in terraform scripts.

Kubernetes Security: Storing sensitive configuration in Kubernetes Secrets rather than plaintext file.

2. Why you need to implement DevSecOps ?

There may be many reasons, but here we talk about two primary reasons:

1. **AI generated Code**

AI generated code may contain hardcoded credentials or secrets making the applications vulnerable.

It might suggest outdated dependencies which can lead to potential risks.

2. **Cyber Attacks**

Developers might push code with known CVE (Common Vulnerability Exposure), exposing application to attacks.

Unpatched third party libraires can lead to security flaws.

By implementing DevSecOps, organizations can improve:

- Minimize security risks introduced by AI generated code and third party libraries.
- Detect and fix vulnerability before they are exploited.
- Secure infrastructure and applications at every stage of the DevOps Lifecycle.

Get Started

Before running the code, check the **README** file to get an understanding of the project. In this file, we talk about the features, technologies used, the project structure, logic and getting started guide. If you carefully check the src folder, most of the files are `.tsx` file**,** which means the code is written in TypeScript.

Three things we need to know about the TypeScript file:

1. **TSX**: TypeScript related Source Code files
2. **Vite**: Build tool which is used for building and running the application locally
3. **npm**: It is used for downloading the dependencies

3. How to run the pipeline locally ?

Pre-requisites:

1. Node.js (v14 or higher)
2. npm

Steps:

- Open your terminal
- Clone the repository

```
git clone https://github.com/yourusername/devsecops-demo.git
cd devsecops-demo
```

- Install the dependencies

```
npm install
```

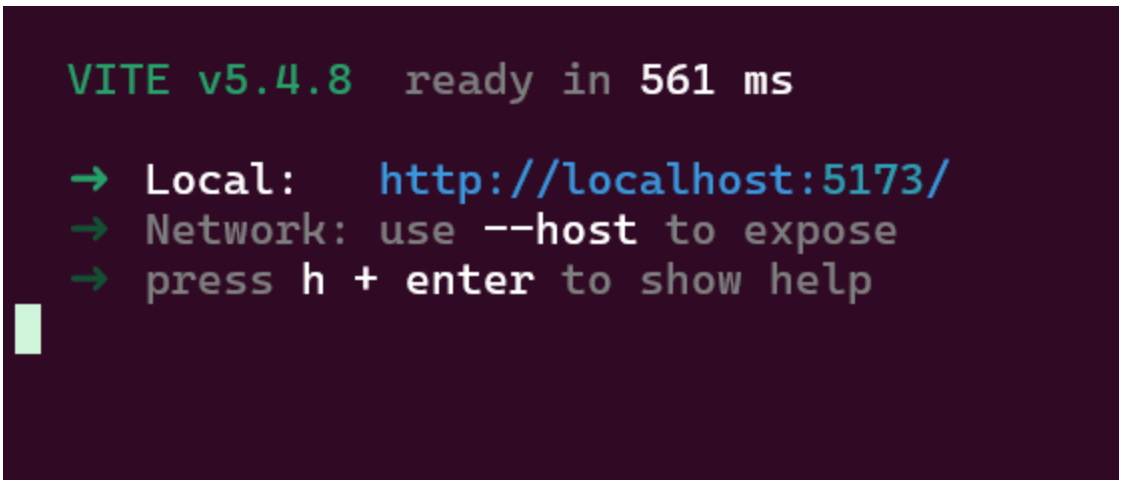
- Build the Application

```
npm run build
```

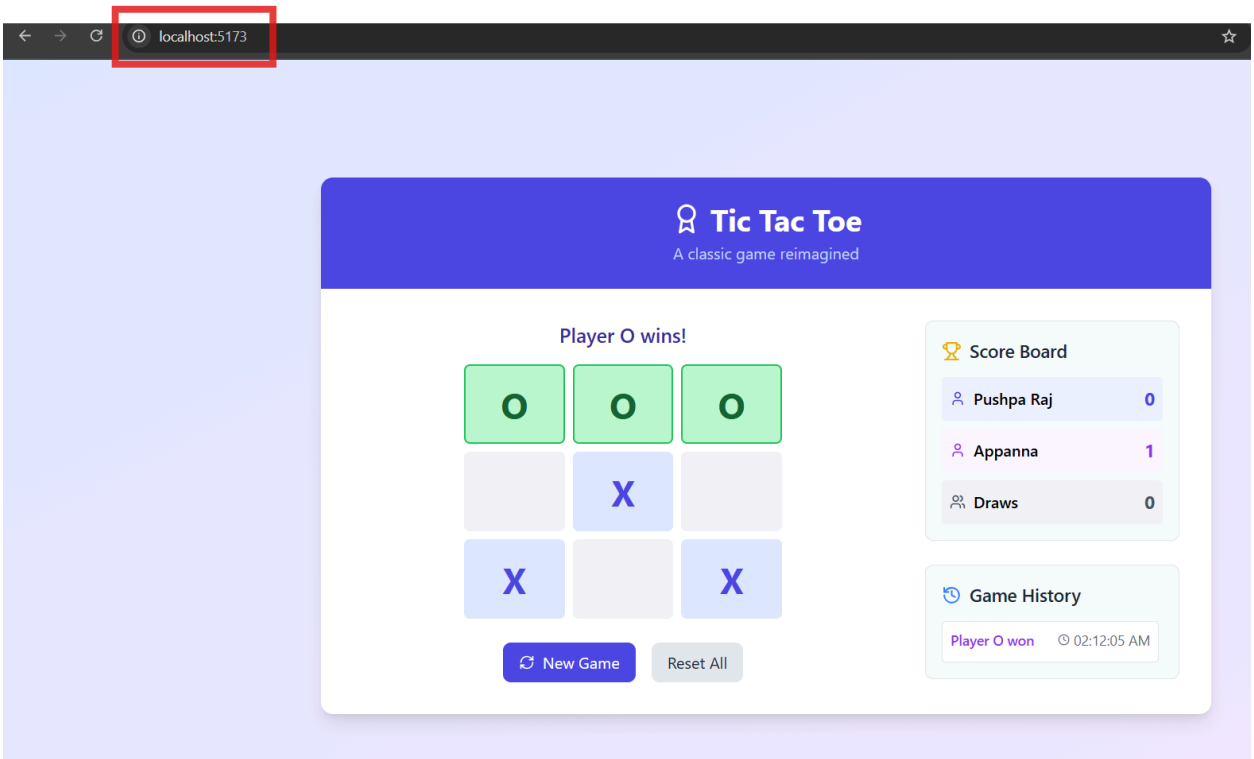
This command packages everything and puts into the `dist` folder which is created newly.

- Run the application locally

```
npm run dev
```



Browse the localhost with port `5137` to see the application running.



4. Containerization

- We will implement a Multi Stage Dockerfile, where in the first stage we will create a working directory and copy the package files inside the working directory. Then install the dependencies and copy the rest of the code to the container and finally run the build. Through this we will get a `dist` folder.
- The reason for installing the dependencies first and later copying the rest of the code is that docker uses layer caching (no need to download again the dependencies and can run directly the application which saves the build time significantly)
- Below we have stage one of the multistage dockerfile

```
# Build stage
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
```

```
COPY . .
RUN npm run build
```

- In the second stage, we will copy the already generated `dist` folder from the first stage and copy it to the nginx location. Expose the port and run the `CMD` command to start the nginx server.

```
# Production stage
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

PS: I have separated the stages to explain each stage, in practical both the stage are written inside one Dockerfile.

- Combining both the stages

```
# Build stage
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Production stage
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

- Open your terminal again in the same location where your project exists.
- Build the Dockerfile

```
docker build -t tiktactoe:v1 .
```

```
bala_pc@Bala:~/devsecops/devsecops-demo$ docker images | head -n2
REPOSITORY                    TAG       IMAGE ID       CREATED          SIZE
tictaktoe                     v1        485d5be0a7c8   2 minutes ago    48.1MB
```

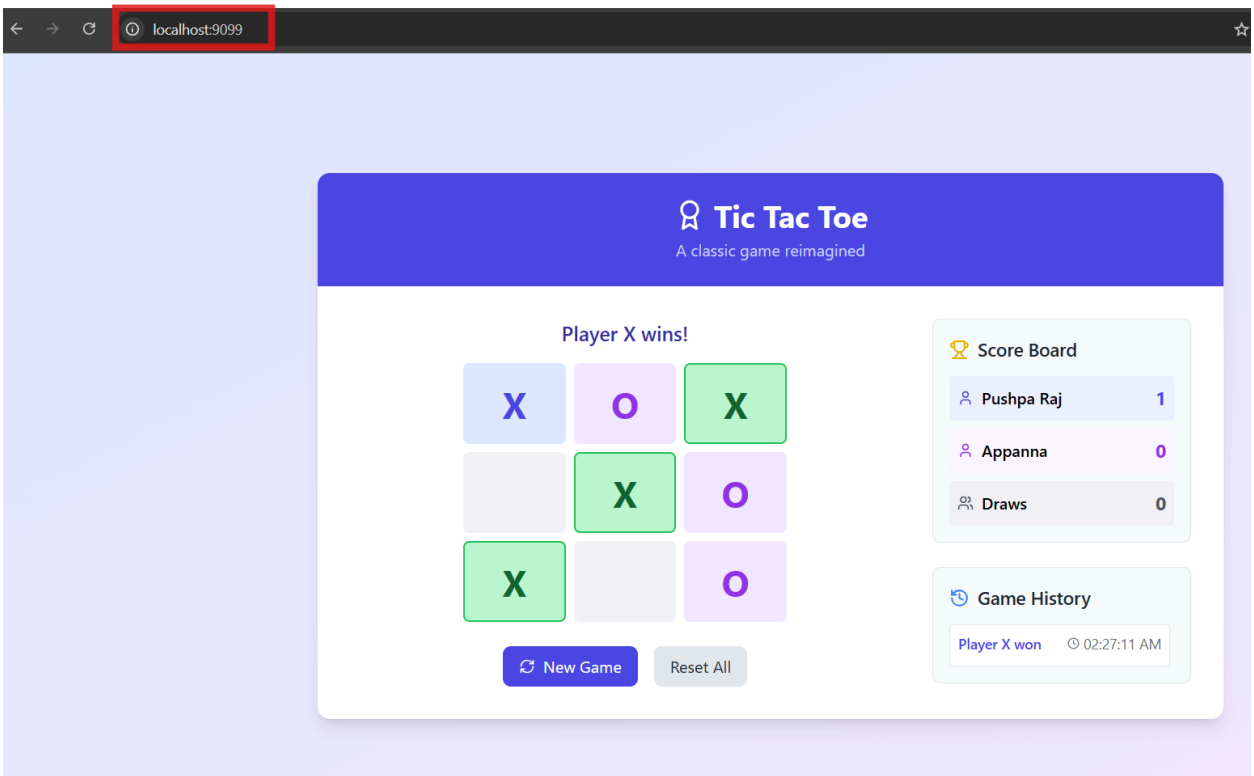
Docker Image is generated.

- Run the Docker image

```
docker run -d -p 9099:80 tiktactoe:v1
```

Docker container is running.

- Browse to the localhost with port 9099 to see the application running.



5. DevSecOps Implementation

What's inside the pipeline ? Stages Explained

Inside the pipeline, there are multiple components such as stating when does this pipeline should trigger to different stages or jobs as we call it in github actions. Below we will explore different stages to understand what exactly the job performs.

Stages/Jobs Explained

1. **Unit Testing**: This step ensures all the tests passes before proceeding.
2. **Static Code Analysis**: This job performs linting to ensure code follows best practices.
3. **Build**: This job compiles the application and generates the necessary artifacts.
4. **Containerization**: This job builds a Docker Image, scans it for vulnerability, and pushes it to GHCR
5. **Update Kubernetes Deployment**: This job updates the kubernetes deployment file and applies the changes.

Let's understand the YAML config file

If you have worked before with Github Workflow, then you can understand that these YAML are self explanatory. Then too, here I will try to explain each stage separately to understand better.

Even before the stages, we have few parts to explain.

name: This is just the name you want to give the action.

on: This is what you want the action to trigger on. Get triggered on push to main branch while ignoring the deployment.yaml changes. Also get triggered on pull request to the main branch.

```
name: CI/CD Pipeline

on:
  push:
    branches: [ main ]
    paths-ignore:
      - 'kubernetes/deployment.yaml'  # Ignore changes to this file to prevent lo
ops
```

```
  pull_request:
    branches: [ main ]
```

1. **Unit testing job**

jobs: There are individual series of steps which you want to execute. You can refer it to a stage, so in this case it is the unit testing stage. You can have as many jobs and steps in a single action file.

In the Github actions file, the two most common used keywords are:

`uses`: Predefined actions, Reusable existing code from Github Actions Marketplace.

`run`: Execute custom shell commands inside the workflow.

In these below steps, it checkout the code from Github, setup node.js with version 20, install the dependencies and run tests.

```
jobs:
  test:
    name: Unit Testing
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test || echo "No tests found, would add tests in a real project"
```

2. **Static Code Analysis job**

In these below steps, it does exactly the same as the previous job only difference is the last step where you run lint.

```
lint:
    name: Static Code Analysis
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run ESLint
        run: npm run lint
```

3. **Build Job**

In this job the `needs` parameter tells that the build job should only run if the test and lint job is completed. Later in the job, the artifacts which is created using the build command is uploaded and the path is mentioned.

```
build:
    name: Build
    runs-on: ubuntu-latest
    needs: [test, lint]
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
```

```
    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '20'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Build project
      run: npm run build

    - name: Upload build artifacts
      uses: actions/upload-artifact@v4
      with:
        name: build-artifacts
        path: dist/
```

4. **Containerization**

In this job, it is responsible for multiple steps such as building, scanning, login into the registry, extracting the tags, pushing to the GHCR. Let me break down each step of this docker job.

```
docker:
  name: Docker Build and Push
  runs-on: ubuntu-latest
  needs: [build]
  env:
    REGISTRY: ghcr.io
    IMAGE_NAME: ${{ github.repository }}
  outputs:
    image_tag: ${{ steps.set_output.outputs.image_tag }}
```

- This job runs on ubuntu machine.

- Depends on the build job, which means it will execute only after the build stage is completed.
- Environment variables are set and output variable are defined which will be used later.

Steps Explained

1️⃣ Checkout Code

```
- name: Checkout code
  uses: actions/checkout@v4
```

It fetches the latest code from the repository.

2️⃣ Download Build artifacts

```
- name: Download build artifacts
  uses: actions/download-artifact@v4
  with:
    name: build-artifacts
    path: dist/
```

Downloads the build artifacts.

3️⃣ Set up Docker Buildx

```
- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v3
```

Setting up docker build, similar to docker build but with advanced features like multi platform build and caching mechanism.

4️⃣ Login to Github Container Registry (GHCR)

```
- name: Login to GitHub Container Registry
  uses: docker/login-action@v3
  with:
    registry: ${{ env.REGISTRY }}
```

```
    username: ${{ github.actor }}
    password: ${{ secrets.TOKEN }}
```

Login into GHCR.

${{ github.actor }} → username.

${{ secrets.TOKEN }} → secret stored in github which is the token.

5️⃣ Extract metadata for Docker

```
- name: Extract metadata for Docker
  id: meta
  uses: docker/metadata-action@v5
  with:
    images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
    tags: |
      type=sha,format=long
      type=ref,event=branch
      latest
```

Generates tags for docker image.

SHA based tags to get the exact version.

Branch based tag to get associated with branch.

Latest will tag the most recent build as latest.

6️⃣ Build Docker Image

```
- name: Build Docker image
  uses: docker/build-push-action@v5
  with:
    context: .
    push: false
    tags: ${{ steps.meta.outputs.tags }}
    labels: ${{ steps.meta.outputs.labels }}
    load: true
```

Builds the docker image but does not push it.

Applies the tags and labels and ensures the image is loaded for scanning.

7️⃣ Run Trivy for Scanning

```
- name: Run Trivy vulnerability scanner
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:sha-${{ github.sh
a }}
    format: 'table'
    exit-code: '1'
    ignore-unfixed: true
    vuln-type: 'os,library'
    severity: 'CRITICAL,HIGH'
```

Keeps the format of the output in a table form.

Scans for OS and library vulnerabilities .

If critical, high severity is found then it fails the workflow (exit-1).

8️⃣ Push Docker Image to GHCR

```
- name: Push Docker image
  uses: docker/build-push-action@v5
  with:
    context: .
    push: true
    tags: ${{ steps.meta.outputs.tags }}
    labels: ${{ steps.meta.outputs.labels }}
```

Pushes the Docker image to the GHCR.

9️⃣ Set image tag output

```
- name: Set image tag output
  id: set_output
```

```
  run: echo "image_tag=$(echo ${{ github.sha }} | cut -c1-7)" >> $GITHUB_OU
TPUT
```

Extracts the first 7 characters of the commit SHA and saves this tag as a output variable.

5. **Update Kubernetes Job**

In this job, it is responsible for modifying the kubernetes deployment file with the latest docker image tag and committing the changes back to the repository.

```
update-k8s:
  name: Update Kubernetes Deployment
  runs-on: ubuntu-latest
  needs: [docker]
  if: github.ref == 'refs/heads/main' && github.event_name == 'push'
```

This job will only execute once the docker build job is completed.

It will run only when the push event occurs on the main branch.

1️⃣ Checkout the code

```
- name: Checkout code
  uses: actions/checkout@v4
  with:
    token: ${{ secrets.TOKEN }}
```

Pulls the latest version and uses secret token as authentication.

2️⃣ Setting up Git Config

```
- name: Setup Git config
  run: |
    git config user.name "GitHub Actions"
    git config user.email "actions@github.com"
```

Configures Git with identity for committing changes.

3️⃣ Update Kubernetes deployment file

```
- name: Update Kubernetes deployment file
  env:
    IMAGE_TAG: sha-${{ github.sha }}
    GITHUB_REPOSITORY: ${{ github.repository }}
    REGISTRY: ghcr.io
  run: |
    # Define the new image with tag
    NEW_IMAGE="${REGISTRY}/${GITHUB_REPOSITORY}:${IMAGE_TAG}"

    # Update the deployment file directly
    sed -i "s|image: ${REGISTRY}/.*|image: ${NEW_IMAGE}|g" kubernetes/dep
loyment.yaml

    # Verify the change
    echo "Updated deployment to use image: ${NEW_IMAGE}"
    grep -A 1 "image:" kubernetes/deployment.yaml
```

Defining env variables

Using sed command to find and replace the old image with the new one

Verifying the update using grep printing with 1 additional line (-A 1)

4️⃣ Commit and Push changes

```
- name: Commit and push changes
  run: |
    git add kubernetes/deployment.yaml
    git commit -m "Update Kubernetes deployment with new image tag: ${{ nee
ds.docker.outputs.image_tag }} [skip ci]" || echo "No changes to commit"
    git push
```

Adds and Commits the changes with new image tag, skips to trigger another CI/CD run by using skip ci. Pushes the changes and if no changes then prints “No changes to commit”.

Combining all stages together

Below is the full Github Actions code.

```
name: CI/CD Pipeline

on:
  push:
    branches: [ main ]
    paths-ignore:
      - 'kubernetes/deployment.yaml'  # Ignore changes to this file to prevent lo
ops
  pull_request:
    branches: [ main ]

jobs:
  test:
    name: Unit Testing
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test || echo "No tests found, would add tests in a real project"

  lint:
```

```
  name: Static Code Analysis
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '20'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run ESLint
      run: npm run lint

  build:
    name: Build
    runs-on: ubuntu-latest
    needs: [test, lint]
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Build project
```

```
      run: npm run build

    - name: Upload build artifacts
      uses: actions/upload-artifact@v4
      with:
        name: build-artifacts
        path: dist/

  docker:
    name: Docker Build and Push
    runs-on: ubuntu-latest
    needs: [build]
    env:
      REGISTRY: ghcr.io
      IMAGE_NAME: ${{ github.repository }}
    outputs:
      image_tag: ${{ steps.set_output.outputs.image_tag }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Download build artifacts
        uses: actions/download-artifact@v4
        with:
          name: build-artifacts
          path: dist/

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login to GitHub Container Registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.TOKEN }}
```

```
      - name: Extract metadata for Docker
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=sha,format=long
            type=ref,event=branch
            latest

      - name: Build Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: false
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          load: true

      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:sha-${{ githu
b.sha }}
          format: 'table'
          exit-code: '1'
          ignore-unfixed: true
          vuln-type: 'os,library'
          severity: 'CRITICAL,HIGH'

      - name: Push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
```

```
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}

      - name: Set image tag output
        id: set_output
        run: echo "image_tag=$(echo ${{ github.sha }} | cut -c1-7)" >> $GITHUB
_OUTPUT

  update-k8s:
    name: Update Kubernetes Deployment
    runs-on: ubuntu-latest
    needs: [docker]
    if: github.ref == 'refs/heads/main' && github.event_name == 'push'
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
        with:
          token: ${{ secrets.TOKEN }}

      - name: Setup Git config
        run: |
          git config user.name "GitHub Actions"
          git config user.email "actions@github.com"

      - name: Update Kubernetes deployment file
        env:
          IMAGE_TAG: sha-${{ github.sha }}
          GITHUB_REPOSITORY: ${{ github.repository }}
          REGISTRY: ghcr.io
        run: |
          # Define the new image with tag
          NEW_IMAGE="${REGISTRY}/${GITHUB_REPOSITORY}:${IMAGE_TAG}"

          # Update the deployment file directly
          sed -i "s|image: ${REGISTRY}/.*|image: ${NEW_IMAGE}|g" kubernetes/
deployment.yaml
```

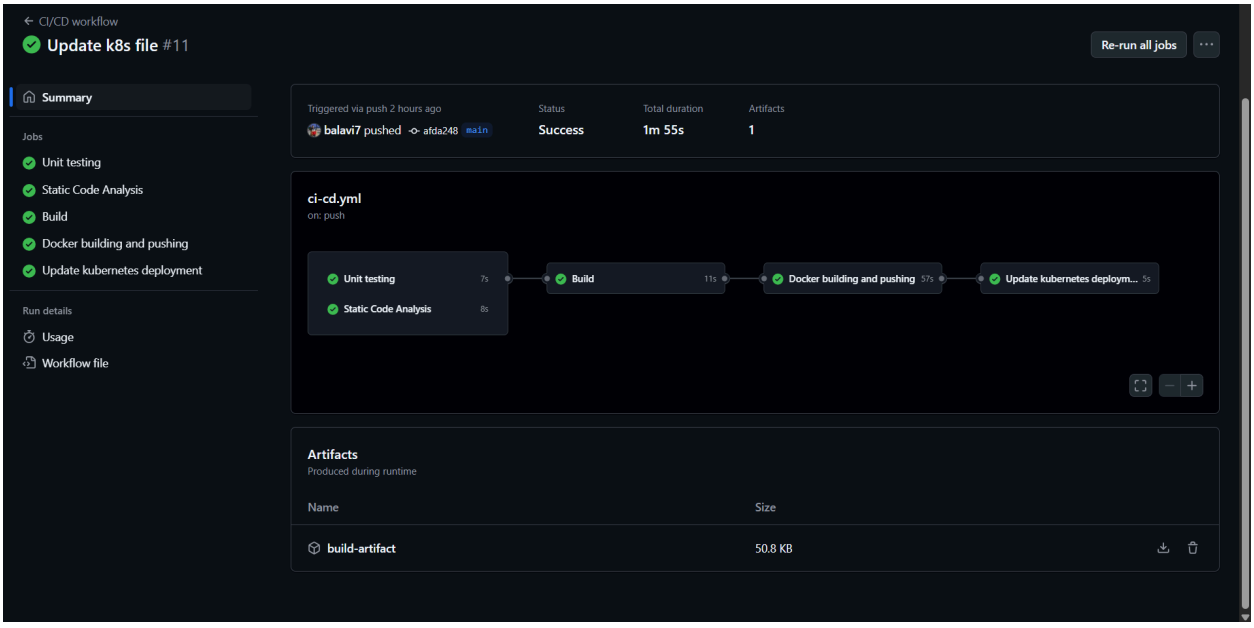
```
        # Verify the change
        echo "Updated deployment to use image: ${NEW_IMAGE}"
        grep -A 1 "image:" kubernetes/deployment.yaml

    - name: Commit and push changes
      run: |
        git add kubernetes/deployment.yaml
        git commit -m "Update Kubernetes deployment with new image tag: ${{ needs.docker.outputs.image_tag }} [skip ci]" || echo "No changes to commit"
        git push
```

Make sure to add the PAT (Personal Access Token) into the repository secret before committing the code.

Now let us commit this code. As soon as you commit the file, the workflow will be triggered as we have set the `on:` parameter to push**. So every time you push a commit to your repo in the main branch, it will build and push a new docker image in the repository.**

Let us check the Actions tab to see the workflow running.



🎊All the jobs have passed successfully.

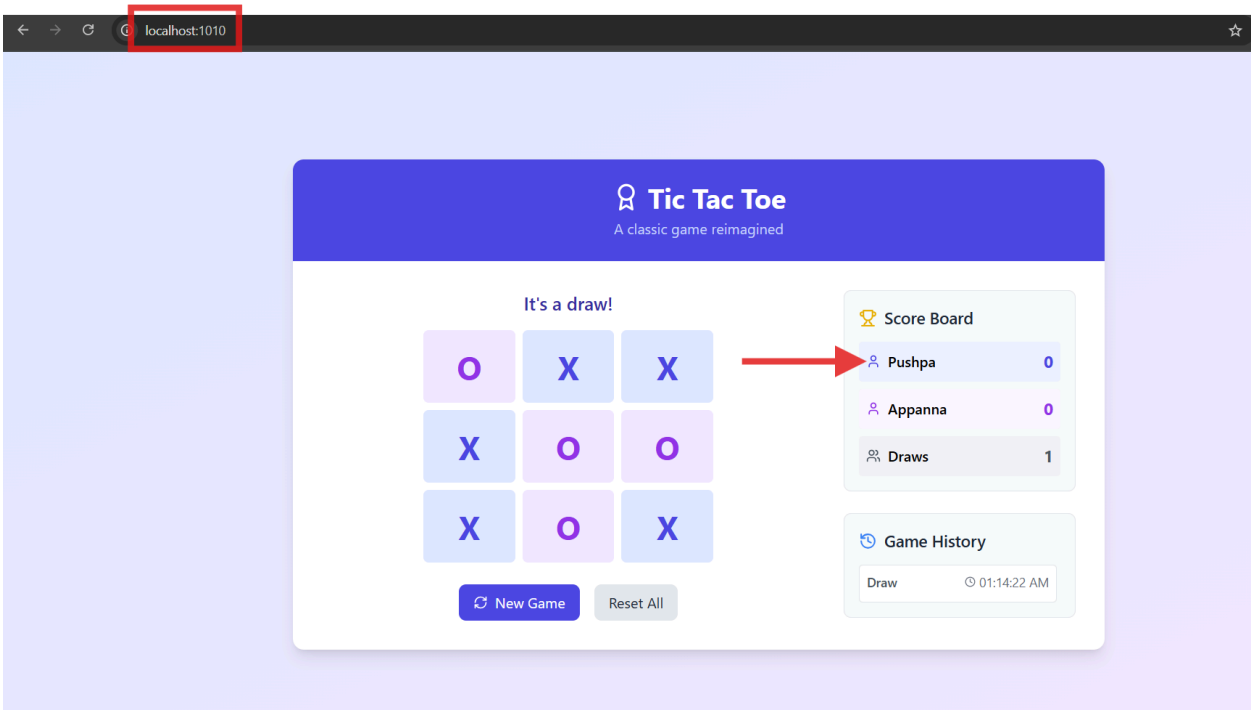
6. Test and Validate Application Changes

- Go to boards.tsx file and change the name to just **pushpa** and commit the change.
- After the jobs are completed, the deployment file must have updated the docker image name. Let us pull the latest image and run it to see the changes.
- Go to the terminal, login into GHCR. Username → dockerhub username and password → TOKEN.
- You will get a message login succeeded.

```
bala_pc@Bala:~/devsecops/devsecops-demo$ docker login ghcr.io
Authenticating with existing credentials...
Login Succeeded
```

- Copy the image name which just got updated the previous step and try to run the container using the below command.

```
docker run -d -p 1010:80 ghcr.io/balavi7/devsecops:sha-24452302c90939ef
d34f786bccfee637cb35b158
```



Perfect! The workflow triggered and the name is updated in our application

7. Creation of EC2 instance

In this step, we are creating an EC2 instance for the kubernetes implementation. Here we will spin up an EKS and kind Kubernetes Cluster on this instance.

Pre-requisites:

- AWS Account
- AWS CLI
- Kubectl
- Docker

Steps:

1. Go to AWS Console, search for **EC2** and click on **Launch Instances.**
2. Select an **ubuntu** OS with **t2.medium** as the type, keeping the rest as default.
3. Once the instance is running, connect to the instance using SSH.
4. We need to update the instance using the below command.

```
sudo apt update
```

5. Authenticate to docker using the below command.

```
docker login ghcr.io
```

username: Give your DockerHub Username

password: Give the generated token from GitHub as the password

You will get a message **Login Succeeded**

8. Kubernetes Implementation (Choose any one method from Kind and EKS)

Using Kind

The last step of the complete CI/CD implementation is to deploy the application on a Kubernetes Cluster.

Steps:

- Download and install Kind using the below code.

```
https://kind.sigs.k8s.io/docs/user/quick-start/#installation
```

- Create a cluster using the below command.

```
kind create cluster --name=devsecops-demo-cluster
```

- Check if the context is set using the below command

```
kubectl config current-context
```

- Check if the nodes are running

```
kubetcl get nodes
```

- Install ArgoCD using the below command.

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

- The above command will install all the dependencies required for the ArgoCD. We will check if all the resources are running then proceed to the next step.

```
kubectl get pods -n argocd
```

All of them are the running state, so now we can proceed to the next step.

- Now we will access the ArgoCD UI. For that you need to enable port forwarding for the ArgoCD server. Run the below command to achieve the same.

```
kubectl port-forward svc/argocd-server 9000:80 -n argocd --address 0.0.0.0
```

Make sure to add port `9000` in the EC2 inbound security group.

- Copy the public IP of the EC2 instance, go to your browser and try to access the ArgoCD page.

8.1. Kubernetes Implementation

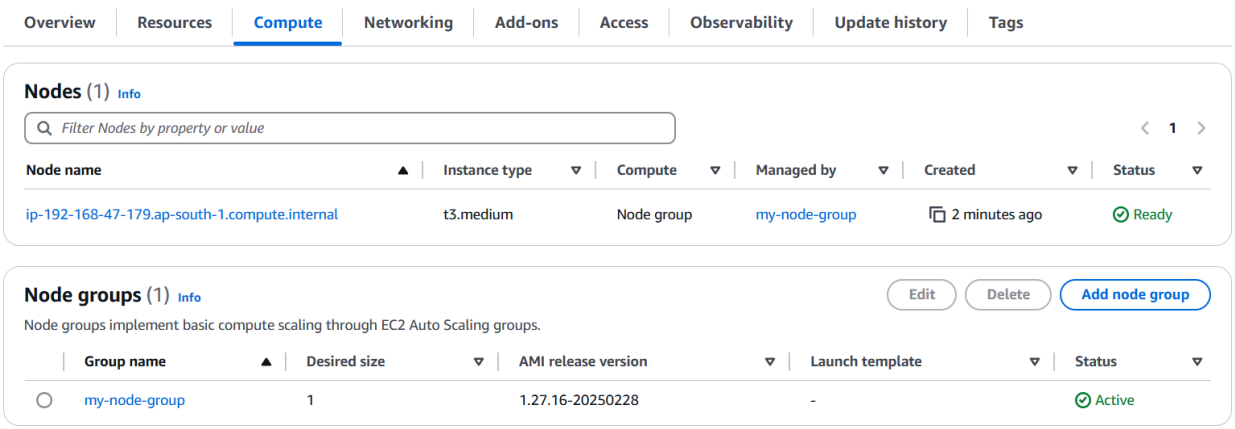
Using EKS

Make sure to install eksctl as a pre-requisites

- Go to your terminal and run the eks installation command.

```
eksctl create cluster --name my-cluster --region ap-south-1
```

- After the creation, go to EKS dashboard → Compute → Add Node group → Create.



- Once the cluster is created, update the context using the below command.

```
aws eks update-kubeconfig --name my-cluster --region ap-south-1
```

- Check if the nodes are running.

```
kubectl get nodes
```

```
ubuntu@ip-172-31-45-158:~$ kubectl get nodes
NAME                                           STATUS   ROLES    AGE     VERSION
ip-192-168-47-179.ap-south-1.compute.internal   Ready    <none>   3m27s   v1.27.16-eks-aeac579
```

- Install argocd on eks.

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

- Check if all the pods are running.

```
kubectl get pods -n argocd
```

```
ubuntu@ip-172-31-45-158:~$ kubectl get pods -n argocd
NAME                                                READY   STATUS    RESTARTS   AGE
argocd-application-controller-0                     1/1     Running   0          38s
argocd-applicationset-controller-85b8885f65-h9pb4   1/1     Running   0          38s
argocd-dex-server-8595586c7c-s9fzn                  1/1     Running   0          38s
argocd-notifications-controller-576696cd86-dzr8x    1/1     Running   0          38s
argocd-redis-56f94b56b9-kp99t                       1/1     Running   0          38s
argocd-repo-server-7996bc9bbc-wbtwm                 1/1     Running   0          38s
argocd-server-9f965d4b9-9459k                       1/1     Running   0          38s
```

- Change the argocd service type to NodePort

```
kubectl edit svc/argocd-server -n argocd
```

Change the service type from ClusterIP → NodePort.

- Verify the service

```
kubectl get svc -n argocd
```

```
ubuntu@ip-172-31-45-158:~$ kubectl get svc -n argocd
NAME                                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
argocd-applicationset-controller      ClusterIP   10.100.167.249   <none>        7000/TCP,8080/TCP            4m39s
argocd-dex-server                     ClusterIP   10.100.90.71     <none>        5556/TCP,5557/TCP,5558/TCP   4m39s
argocd-metrics                        ClusterIP   10.100.207.219   <none>        8082/TCP                     4m39s
argocd-notifications-controller-metrics   ClusterIP   10.100.179.113   <none>        9001/TCP                     4m39s
argocd-redis                          ClusterIP   10.100.36.142    <none>        6379/TCP                     4m39s
argocd-repo-server                    ClusterIP   10.100.138.220   <none>        8081/TCP,8084/TCP            4m39s
argocd-server                         NodePort    10.100.96.165    <none>        80:30900/TCP,443:32698/TCP   4m39s
argocd-server-metrics                 ClusterIP   10.100.117.197   <none>        8083/TCP                     4m38s
```

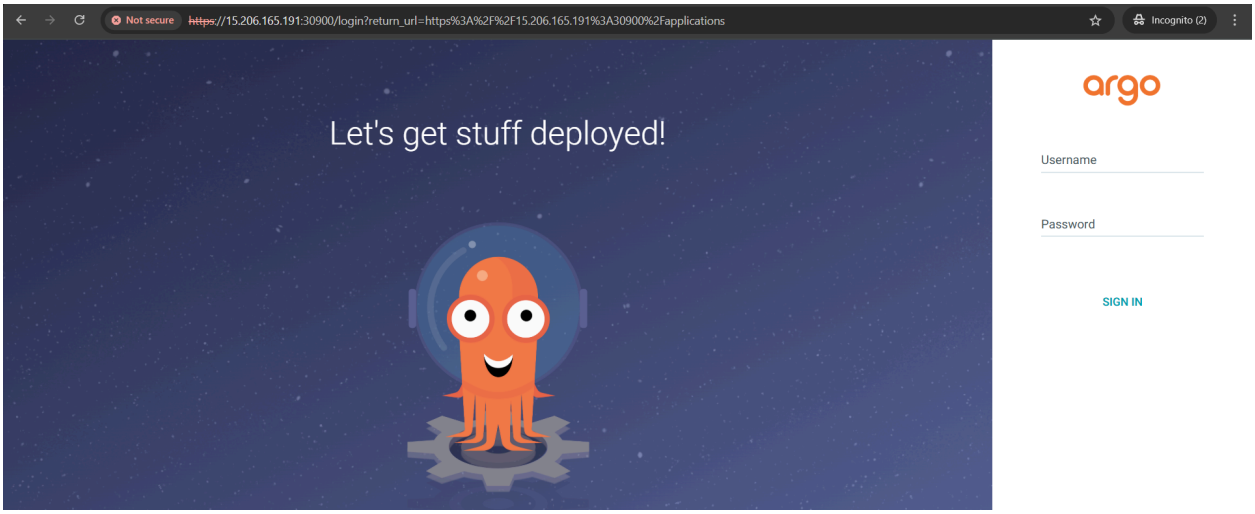
- Find the public IP of your node and access the argocd UI

```
kubectl get nodes -o wide
```

```
ubuntu@ip-172-31-45-158:~$ kubectl get nodes -o wide
NAME                                        STATUS   ROLES    AGE   VERSION               INTERNAL-IP      EXTERNAL-IP      OS-IMAGE         KERNEL-VERSI
ON                  CONTAINER-RUNTIME
ip-192-168-47-179.ap-south-1.compute.internal   Ready    <none>   11m   v1.27.16-eks-aeac579   192.168.47.179   15.206.165.191   Amazon Linux 2   5.10.234-225
.895.amzn2.x86_64   containerd://1.7.25
```

- Access argocd in the browser

```
http://15.206.165.191:30900
```



9. Logging Into ArgoCD

- Login into argocd
- It will ask for username and password. Where username is `admin.`
- Go back to your terminal, the password is available using the below command.

```
kubectl get secrets -n argocd
```

```
ubuntu@ip-172-31-45-158:~$ kubectl get secrets -n argocd
NAME                          TYPE     DATA   AGE
argocd-initial-admin-secret   Opaque   1      10m
argocd-notifications-secret   Opaque   0      10m
argocd-redis                  Opaque   1      10m
argocd-secret                 Opaque   5      10m
```

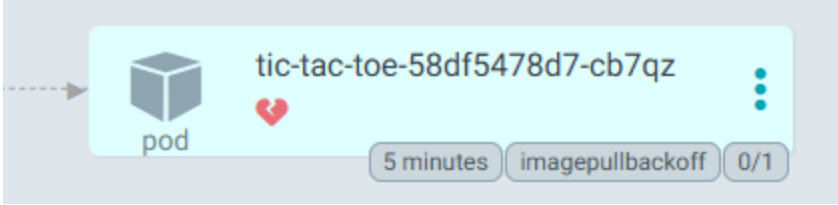
- The password is inside the argocd initial admin secret.

```
kubetctl edit secret argocd-initial-admin-secret -n argocd
```

- Copy the password and exit. This password is base64 encoded, so decode using the below command.

```
echo <password> | base64 --decode
```

- This is the password for the argocd server. Paste it and login into the server.
- To deploy our project, we will create an application inside this argocd server.
- Click on the create application button → give it a name → project name will be default → select automatic sync → in the repository URL paste your github repo URL → give the path name as kubernetes → Cluster URL will be default and finally the namespace will also be default.
- Click on create button on the top.
- You get an error as imagepullbackoff.



- You will notice that the cluster has been failed to deploy. This is because, the **imagePullSecret** has not been configured.

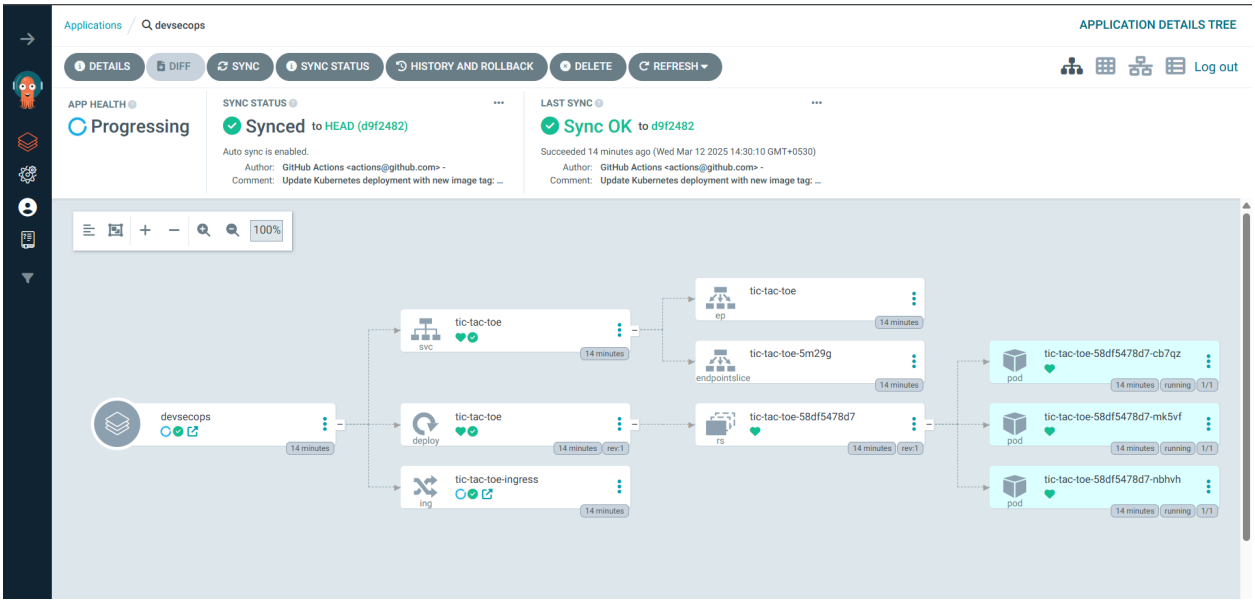
ImagePullSecret: It is a kubernetes object which holds the credentials required to access the private container registry. Since we are using GHCR, we need to

provide the PAT(Personal Access Token) for authentication.

- We have provided the parameter, but we need to set the secrets with the PAT.
- Go to the terminal again, run the below command which will create the secret. Edit the values accordingly.

```
kubectl create secret docker-registry github-container-registry \
  --docker-server=ghcr.io \
  --docker-username=YOUR_GITHUB_USERNAME \
  --docker-password=YOUR_GITHUB_TOKEN \
  --docker-email=YOUR_EMAIL
```

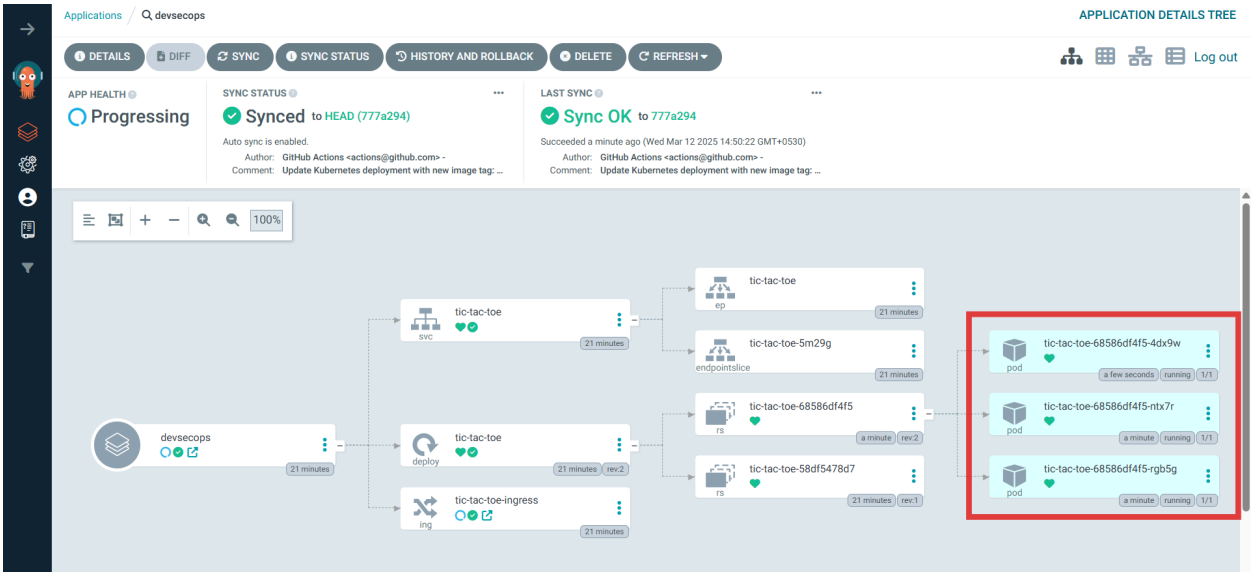
- Check again the pods, this time it will be healthy. This shows our ArgoCD is in sync with the code.

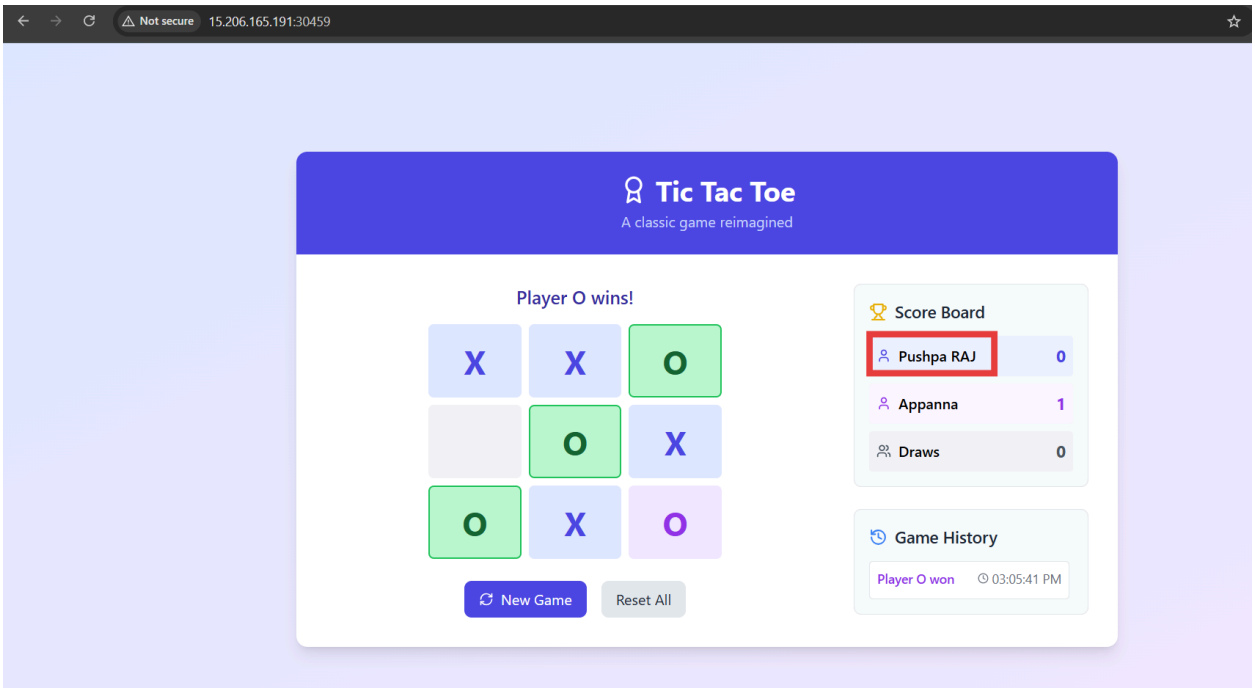


10. End to End Implementation

- Make a small change in the code. Go to the scoreboard.tsx file and change the name from Pushpa to Pushpa RAJ and commit the changes.
- The pipeline triggered automatically. It should update the kubernetes deployment file with the new image.
- Check the deployment file in the repository for the newly created image tag.

- Now check the argocd server, it should have picked up the change and should be in sync with the github repository.





Congratulations 🎉 You have just implemented a complete End to End DevSecOps Pipeline successfully.

Hope you have learnt something from this article!

Conclusion:

In this article, we implemented a complete DevSecOps pipeline from scratch. Whether you are a beginner or a experienced professional this guide provides a practical approach following best practices to achieve fully automated, secure and efficient software delivery process.

If you found this post useful, give it a like👍

Repost♻️

Follow @Bala Vignesh for more such posts 💯🚀