

Project Streamer

A guide/documentation on setting up and deploying a video streaming application using DevOps tools.

Author

H.Ragul

Analyst 1 Infrastructure Services

DXC Technology

Index

- 1. Introduction**
- 2. Tools and Requirements**
- 3. Infrastructure Provisioning**
- 4. Installation and Configuration of Ansible**
- 5. Installation and Configuration of Docker**
- 6. Installation and Configuration of Kubernetes**
- 7. Installation and Configuration of Github Actions**
- 8. Pipeline Creation and Deployment**
- 9. Installation and Configuration of ArgoCD**
- 10. Installation and Configuring of Pi-hole**

1. Introduction

The primary objective of this guide/documentation is to elucidate project deployment utilizing DevOps tools.

About the application

Project Streamer is a video streaming application constructed using MERN stack technologies, incorporating MongoDB (Realm DB), Express.js, React, and Node.js. This application is divided into four separate services, known as microservices.

Note: This application is utilized for deployment with commonly used DevOps tools.

Application High-level Structure:

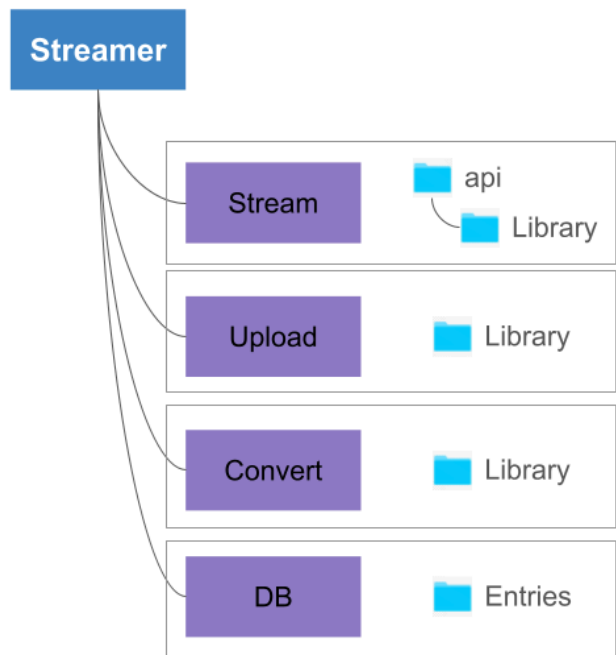
Stream: A frontend server that serves React webpages.

Upload: A backend server that handles the file upload process.

Convert: An internal backend server that converts uploaded videos into m3u8 format.

DB: Realm database (a mobile NoSQL database) used to store details or metadata about the videos.











Note: In this application, all services have a folder named "Library" where the videos are stored.



How does it work?

The application comprises four services. Upon video upload, the "upload service" manages the process and internally triggers the "Convert service" to convert the video to m3u8 format. After conversion, the "Convert service" responds to the upload service, which then internally requests the "DB service" to update entries in the Realm DB. Meanwhile, the "Stream service" serves both the frontend React web application and renders the video.

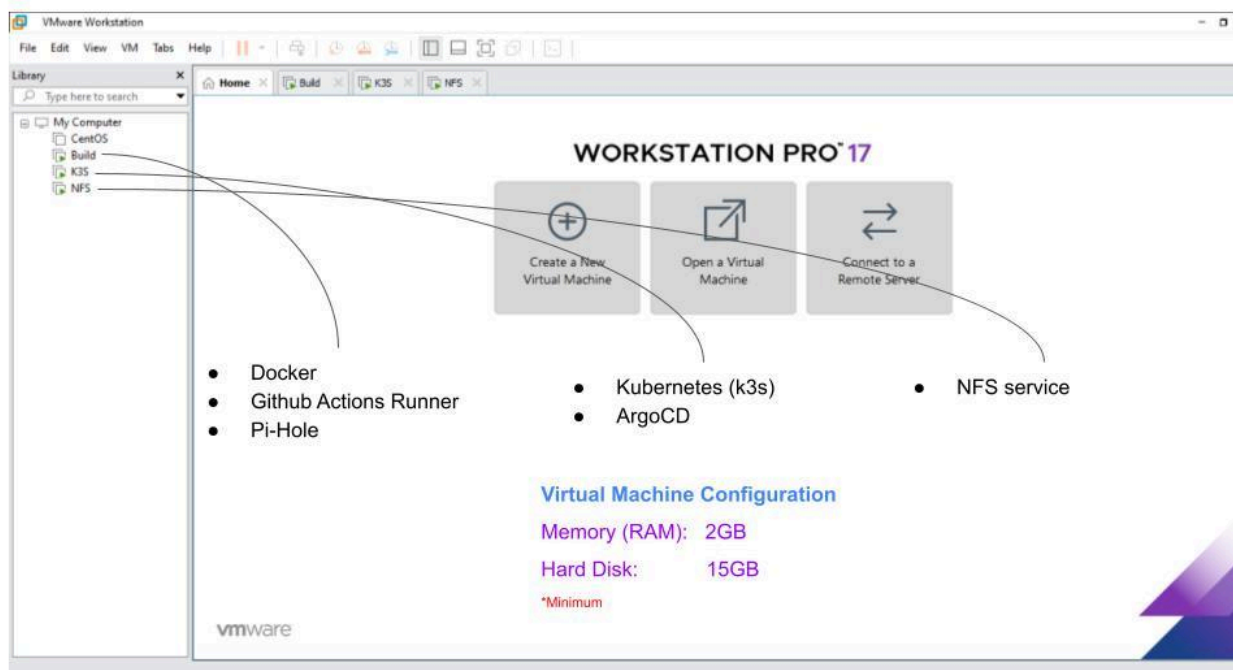
2. Tools and Requirements

VMware Workstation Pro	
Linux (Cent OS)	
Ansible	
Git	
Github	
Github Actions	
ArgoCD	
Docker	
Kubernetes (k3s)	 K3S
Pi-hole	

3. Infrastructure Provisioning

In this project, the lab setup is provisioned using the **VMware Workstation Pro** hypervisor, with virtual machines being created, each running **Linux (CentOS)**.

A total of three virtual machines are mandatory for this setup. The first will serve as a Build server, equipped with Docker, Github Actions Runner, and Pi-hole. The second (k3s) will function as the Kubernetes master server, featuring K3S (a lightweight Kubernetes distribution). Finally, the third (NFS) will serve as an NFS server dedicated to providing storage services.



Vital: Server (VM's) Details

Server Alias Names	Application / Services Running
Build	Docker, Git, Github Actions Runner & Pi-hole
K3S	Kubernetes (k3s) & ArgoCD
NFS	NFS server

4. Installation and Configuration of Ansible

Note: Ansible is installed on the build server, and it will act as the master/controller node for Ansible.

1. Install epel-release

```
# yum install epel-release
```

2. Install ansible

```
# yum install ansible
```

3. Check packages installed

```
# rpm -qa | grep -i ansible
```

4. Include entries for worker nodes in the Ansible inventory hosts file.

File path: /etc/ansible/hosts

```
local    ansible_host=localhost ansible_connection=local
k3s      ansible_host=192.168.199.129    ansible_connection=ssh  ansible_user=root
ansible_ssh_pass=root
nfs      ansible_host=192.168.199.130 ansible_connection=ssh  ansible_user=root
ansible_ssh_pass=root
```

5. Ping command to verify ansible connectivity

```
# ansible all -m ping
```

If the above steps are applied properly you will get response like below

```
local | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3"
  },
  "changed": false,
  "ping": "pong"}
```

5. Installation and Configuration of Docker (using ansible)

Note: Docker is installed on the build server

Create docker.yml file and add below yml script

```
- name: Docker Installation
  hosts: local
  tasks:
    - name: Install Utils
      become: true # You need to become root to install packages
      yum:
        name: yum-utils
        state: present

    - name: Setup Repo
      become: true
      command: yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo

    - name: Install Docker
      become: true
      yum:
        name:
          - docker-ce
          - docker-ce-cli
          - containerd.io
          - docker-buildx-plugin
          - docker-compose-plugin
        state: present
```

Run docker.yml playbook to install docker

```
# ansible-playbook install docker.yml
```

Note: The installation of Docker can be done directly, and whether to use Ansible or not is entirely at the discretion of the administrator.

Direct Installation Steps Link: [Install Docker Engine on CentOS | Docker Docs](#)

5.1 Docker local registry setup

Before setting up a local registry, it's necessary to configure Docker to prevent HTTPS errors when uploading or pushing images to the local registry.

Add the following JSON configuration to the `/etc/docker/daemon.json` file. If the file doesn't exist, you can create it.

```
{
  "insecure-registries": ["<your-ip>:8000"]
}
```

After adding entries in `daemon.json` file restart the docker

```
# systemctl restart docker
```

Ensure that Docker is running

```
# systemctl status docker
```

Steps to run local registry

1. Create directory to store images

```
# mkdir /registry
```

2. Give full permission

```
# chmod 777 /registry
```

3. Ensure permissions

```
# ls -lrt | grep -i registry
```

Note: The directory can exist in any location and under any name. You will need to provide the path to this directory in the docker run command.

4. Run docker registry container

```
# docker run -d -p 8000:5000 --restart=always --name registry -v /registry:/var/lib/registry registry:latest
```

Note: The default port for the Registry container is 5000, and it's mapped to port 8000

5. Ensure registry running

```
# docker ps
```

6. Curl registry endpoint

```
# curl http://<your-ip>:8000/v2/_catalog
```

Sample Output: {"repositories":[]}

Vital:

Source (host)	Destination (container)
Port 8000	Port 5000
/registry	/var/lib/registry

(Optional) steps to check local registry

```
# docker pull nginx
# docker tag nginx <your-ip>:8000/nginx
# docker push <your-ip>:8000/nginx
# docker images
```

If the above commands execute successfully without any errors, everything is functioning correctly.

6. Installation and Configuration of Kubernetes (using ansible)

Note: Kubernetes is installed on the k3s server

Create k3s.yml file and add below yml script

```
name: K3S
hosts: k3s
tasks:
  - name: Install k3s
    shell: curl -sfL https://get.k3s.io | sh -
  - name: Acknowledge
    command: echo "k3s installed successfully"
```

Run k3s.yml playbook to install kubernetes (k3s)

```
# ansible-playbook install k3s.yml
```

Note: The installation of Kubernetes (k3s) can be done directly, and whether to use Ansible or not is entirely at the discretion of the administrator.

Ensure kubernetes running

```
# rpm -qa | grep -i k3s
```

Configure the private registry by adding an entry in the file /etc/rancher/k3s/registries.yaml

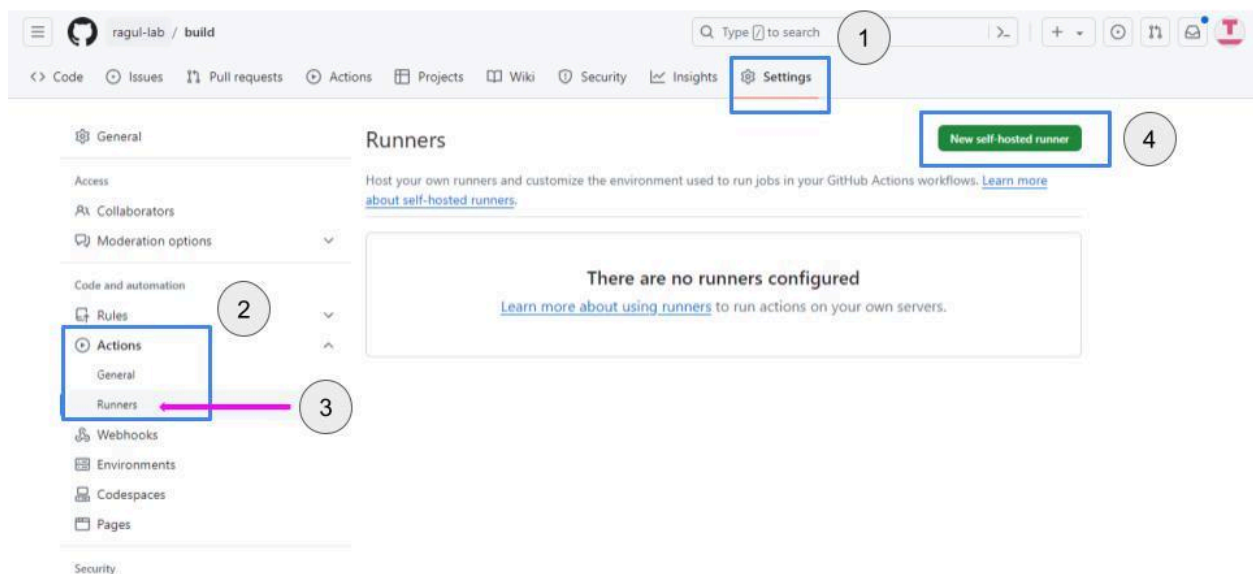
```
mirrors:
  "*":
    endpoint:
      - "https://<your-ip>:8000"
configs:
  "docker.io":
    "*":
      tls:
        insecure_skip_verify: true
```

7. Installation and Configuration of Github Actions (using ansible)

Before proceeding with the installation of the GitHub Actions runner, it is necessary to first create the GitHub repository. In this project, the repository should be created under the name "build". The visibility of the repository, whether it is private or public, is entirely at the discretion of the administrator.

Note: Github actions runner is installed on Build Server

Steps to configure Actions on github



Create Repository → Settings → Actions → Runners → New self-hosted runner

Runner image

☐ macOS ☒ Linux ☐ Windows

Architecture

x64

Runner image → Linux

Architecture → x64

Upon clicking the "New self-hosted runner" button, the subsequent steps will display instructions along with the commands necessary for its installation on the server.

Create a new user and install github actions runner

Note: The newly created user must be added to the Docker group.

1. Create a new user

```
# adduser builder
```

2. Add user to the docker group

```
# usermod -aG docker builder
```

3. Verify user "builder" able to access docker

```
# docker ps
```

4. Create a folder

```
# mkdir actions-runner && cd actions-runner
```

5. Download the latest runner package

```
# curl -o actions-runner-linux-x64-2.316.1.tar.gz -L  
https://github.com/actions/runner/releases/download/v2.316.  
1/actions-runner-linux-x64-2.316.1.tar.gz
```

6. Extract the installer

```
# tar xzf ./actions-runner-linux-x64-2.316.1.tar.gz
```

7. Configure token

```
# ./config.sh --url <your-repo-url> --token <your-token>
```

8. Run

```
# ./run.sh
```

Configure token screenshot

```
[builder@build actions-runner]$ ./config.sh --url https://github.com/ragul-lab/build --token gho_1234567890123456789012345678901234567890

-----
          GitHub Actions
        Self-hosted runner registration
-----

# Authentication

✓ Connected to GitHub

# Runner Registration

Enter the name of the runner group to add this runner to: [press Enter for Default]

Enter the name of runner: [press Enter for build]

This runner will have the following labels: 'self-hosted', 'Linux', 'X64'
Enter any additional labels (ex. label-1,label-2): [press Enter to skip]

✓ Runner successfully added
✓ Runner connection is good

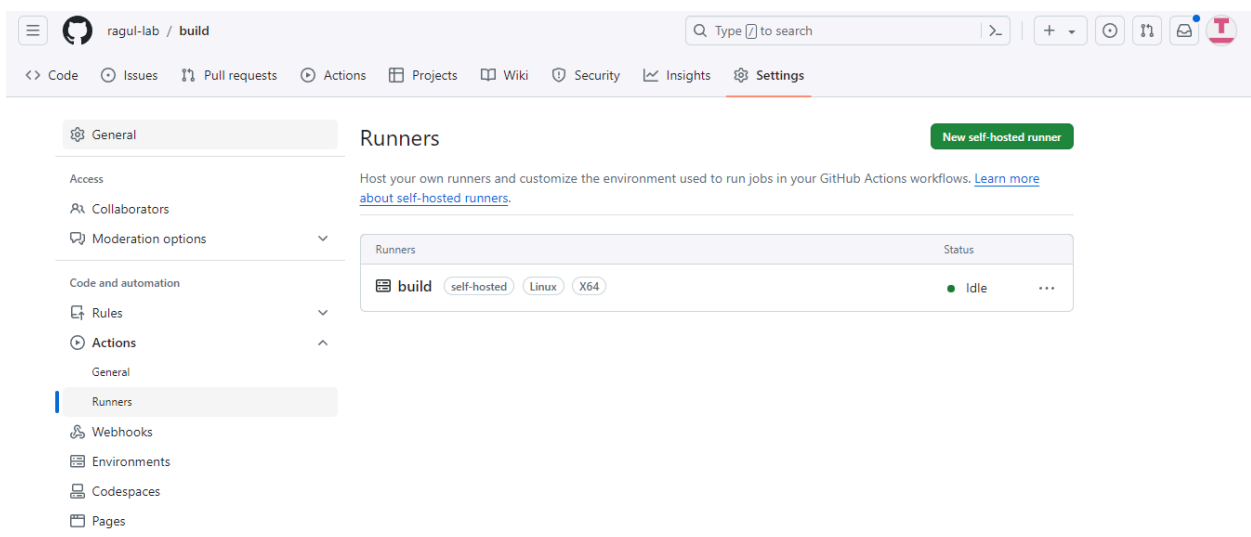
# Runner settings

Enter name of work folder: [press Enter for _work]

✓ Settings Saved.
```

The configuration of the actions runner is completed with default options. If desired, you have the flexibility to modify these defaults, such as the runner group name, the name of the runner itself, and any additional labels for the runner.

After configuring runner



8. Pipeline creation and Deployment

During the configuration stage of the GitHub Action runner, we've already created a "build" repository. Now, within that repository, we need to upload both the application source code and the GitHub Actions Workflow YAML file. This repository will be used for **Continuous Integration (CI)**.

Project Streamer Download Link: <https://github.com/Er-Ragul/streamer.git>

How does it works?

The pipeline creation process is splitted into two stages. Initially, in the first stage, we will conduct **Continuous Integration (CI)**. To facilitate this, we will create a GitHub Actions workflow. When a developer pushes code to the build repository containing the workflow file, the GitHub Actions runner installed and operational on the "build server" will pull the code, generate a Docker image, and push it to our local registry.

In the second stage, we will create a new GitHub repository named "deploy," which will be employed for **Continuous Delivery or Deployment (CD)**. We will configure **ArgoCD** on the Kubernetes (k3s) cluster. When a user pushes Kubernetes YAML scripts to the "deploy" repository, ArgoCD will pull the updates and deploy the application on the k3s cluster.

In the introductory section, the high-level design of the application has already been explained. Therefore, based on that design, we will proceed to create the **Dockerfile** and **GitHub Actions workflow**.

To be continued...!