**Lab 7 – Test the Golden Path Flask Service Locally with Docker**

**🎯 Objective**

In this lab, you will take the **Flask microservice** scaffolded from your **Golden Path Backstage Template** in **Lab 6**, build a Docker image, run it locally, and validate that:

* The application responds to HTTP requests.
* The /metrics endpoint correctly exposes **Prometheus** metrics.

This ensures the service is production-ready from a containerization standpoint.

**🧠 Why This Matters**

* Confirms that **generated code from Backstage** is **runnable without modification**.
* Validates the **Dockerfile** and **requirements.txt** are correct.
* Ensures **metrics instrumentation** is functional for observability.
* Catches build/runtime issues **before** pushing to a container registry or deploying to Kubernetes.

**📋 Prerequisites**

Before starting, ensure:

1. **Lab 6** is complete:
   * Service scaffolded from Backstage **Golden Path Template**.
   * Code is available locally.
2. **Docker** is installed and running:
   * Test with:

sudo usermod -aG docker ubuntu

newgrp docker

docker --version

1. Your scaffolded service directory contains:

app.py

requirements.txt

Dockerfile

catalog-info.yaml

.github/workflows/ci.yml

**📂 Lab Context**

From Lab 6, your app.py should look similar to:

python

CopyEdit

from flask import Flask

from prometheus\_client import Counter, generate\_latest

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

request\_count = Counter('request\_count', 'Total request count')

@app.route('/')

def home():

request\_count.inc()

return "Hello from {{ values.name }} service!"

@app.route('/metrics')

def metrics():

return generate\_latest(), 200

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

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

💡 {{ values.name }} was replaced during scaffolding with your chosen service name in Lab 6.

**🛠 Step-by-Step Guide**

**Step 1 – Navigate to Your Service Directory**

bash

CopyEdit

cd ~/customer-orders-service

*(Replace customer-orders-service with the folder name generated in Lab 6.)*

**Step 2 – Review Dockerfile**

Check that your Dockerfile matches:

dockerfile

CopyEdit

FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

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

📌 **Key Points**

* python:3.10-slim keeps the image small.
* All dependencies are installed from requirements.txt.
* Application starts with:

python app.py

**Step 3 – Build Docker Image**

bash

CopyEdit

docker build -t my-flask-service .

📌 Replace my-flask-service with your preferred image name.

✅ **Expected Output (excerpt)**:

Successfully built <IMAGE\_ID>

Successfully tagged my-flask-service:latest

**Step 4 – Run Container**

bash

CopyEdit

docker run -d --name flaskcon -p 5000:5000 my-flask-service

📌 **Explanation**:

* -d → Run in detached mode.
* --name flaskcon → Name the container.
* -p 5000:5000 → Map container port 5000 to host port 5000.

✅ **Expected Output**:

<CONTAINER\_ID>

**Step 5 – Verify API Endpoint**

bash

CopyEdit

curl http://PubIP:5000/

✅ **Expected Output**:

csharp

CopyEdit

Hello from customer-orders-service service!

*(The service name will match what you entered in Lab 6.)*

**Step 6 – Verify Metrics Endpoint**

bash

CopyEdit

curl http://PubIP:5000/metrics

✅ **Expected Output (excerpt)**:

nginx

CopyEdit

# HELP request\_count Total request count

# TYPE request\_count counter

request\_count 1.0

📌 This confirms **Prometheus metrics** are available.

**Step 7 – View Running Container**

bash

CopyEdit

docker ps

✅ **Expected Output**:

nginx

CopyEdit

CONTAINER ID IMAGE COMMAND STATUS PORTS NAMES

abc123456789 my-flask-service "python app.py" Up 2 minutes 0.0.0.0:5000->5000/tcp flaskcon

**Step 8 – Stop & Remove Container**

bash

CopyEdit

docker stop flaskcon

docker rm flaskcon

**🔍 Verification Checklist**

✅ Docker image builds without errors.  
✅ API endpoint / responds with correct service name.  
✅ /metrics returns Prometheus metrics.  
✅ Container stops/starts cleanly.

**Lab 8 – GitHub OAuth Authentication with User & Group Control in Backstage**

[**https://backstage.io/docs/auth/github/provider/**](https://backstage.io/docs/auth/github/provider/)

**🎯 Objective**

Configure **GitHub OAuth authentication** in Backstage so that:

* Only **specific GitHub users** (or **members of specific GitHub orgs/teams**) can log in.
* Backstage automatically maps them to **User** and **Group** entities in the **catalog**.
* Unauthorized users are **blocked** from logging in.

**🧠 Why This Matters**

In a real **Internal Developer Platform (IDP)**:

* You **don’t want everyone** with a GitHub account to access your portal.
* You want **identity resolution** so Backstage **knows who is who** and **which team they belong to**.
* **Service ownership** (from Lab 6 Golden Path) should map to **real teams**..

**📋 Prerequisites**

1. Backstage portal running.
2. GitHub Developer OAuth App created:
   * Go to **GitHub → Settings → Developer Settings → OAuth Apps → New OAuth App**
   * Homepage URL: http://<your-server>:7007
   * Authorization Callback URL: http://<your-server>:7007/api/auth/github/handler/frame
   * Copy **Client ID** and **Client Secret**.
3. GitHub **Personal Access Token** (PAT) with repo, workflow, and read:org permissions for Backstage integrations.
4. Your **GitHub Organization** name and list of allowed users/teams.

**🛠 Step-by-Step Guide**

**Step 1 – Store OAuth Credentials in .env**

In your Backstage project root **.env**:

bash

CopyEdit

AUTH\_GITHUB\_CLIENT\_ID=xxxxxxxxxxxxxxxxxxxx

AUTH\_GITHUB\_CLIENT\_SECRET=xxxxxxxxxxxxxxxxxxxx

GITHUB\_TOKEN=ghp\_xxxxxxxxxxxxxxxxxxxxxxxxxxxx

*(Do* ***not*** *commit .env to Git — keep it private)*

**Step 2 – Configure GitHub Auth Provider in app-config.yaml**

Open app-config.yaml and **add/update** the GitHub provider:

auth:

environment: development

providers:

github:

development:

clientId: ${AUTH\_GITHUB\_CLIENT\_ID}

clientSecret: ${AUTH\_GITHUB\_CLIENT\_SECRET}

# Sign-in resolver controls who is allowed in

signIn:

resolvers:  
 - resolver: usernameMatchingUserEntityName

Perform backend installation below if didn’t find plugin in packages folder in bakend : we did this earlier in prev lab maybe :

**Backend Installation**

To add the provider to the backend we will first need to install the package by running this command:

from your Backstage root directory

yarn --cwd packages/backend add @backstage/plugin-auth-backend-module-github-provider

Then we will need to add this line:

in packages/backend/src/index.ts

backend.add(import('@backstage/plugin-auth-backend'));  
backend.add(import('@backstage/plugin-auth-backend-module-github-provider'));

**Next step :**

**Adding the provider to the Backstage frontend : update the file App.tsx : replce it wd below file :**

[https://backstage.io/docs/auth/#sign-in-configuration](https://backstage.io/docs/auth/%23sign-in-configuration)

// packages/app/src/App.tsx

import React from 'react';

import { Navigate, Route } from 'react-router-dom';

import { createApp } from '@backstage/app-defaults';

import { AppRouter, FlatRoutes } from '@backstage/core-app-api';

import {

AlertDisplay,

OAuthRequestDialog,

SignInPage,

} from '@backstage/core-components';

import { githubAuthApiRef } from '@backstage/core-plugin-api';

// Plugins

import { apiDocsPlugin, ApiExplorerPage } from '@backstage/plugin-api-docs';

import {

CatalogEntityPage,

CatalogIndexPage,

catalogPlugin,

} from '@backstage/plugin-catalog';

import {

CatalogImportPage,

catalogImportPlugin,

} from '@backstage/plugin-catalog-import';

import { ScaffolderPage, scaffolderPlugin } from '@backstage/plugin-scaffolder';

import { orgPlugin } from '@backstage/plugin-org';

import { SearchPage } from '@backstage/plugin-search';

import {

TechDocsIndexPage,

techdocsPlugin,

TechDocsReaderPage,

} from '@backstage/plugin-techdocs';

import { TechDocsAddons } from '@backstage/plugin-techdocs-react';

import { ReportIssue } from '@backstage/plugin-techdocs-module-addons-contrib';

import { UserSettingsPage } from '@backstage/plugin-user-settings';

import { CatalogGraphPage } from '@backstage/plugin-catalog-graph';

import { RequirePermission } from '@backstage/plugin-permission-react';

import { catalogEntityCreatePermission } from '@backstage/plugin-catalog-common/alpha';

// Local imports

import { apis } from './apis';

import { entityPage } from './components/catalog/EntityPage';

import { searchPage } from './components/search/SearchPage';

import { Root } from './components/Root';

// Create the Backstage app

const app = createApp({

components: {

SignInPage: props => (

<SignInPage

{...props}

auto

provider={{

id: 'github-auth-provider',

title: 'GitHub',

message: 'Sign in using GitHub',

apiRef: githubAuthApiRef,

}}

/>

),

},

apis,

bindRoutes({ bind }) {

bind(catalogPlugin.externalRoutes, {

createComponent: scaffolderPlugin.routes.root,

viewTechDoc: techdocsPlugin.routes.docRoot,

createFromTemplate: scaffolderPlugin.routes.selectedTemplate,

});

bind(apiDocsPlugin.externalRoutes, {

registerApi: catalogImportPlugin.routes.importPage,

});

bind(scaffolderPlugin.externalRoutes, {

registerComponent: catalogImportPlugin.routes.importPage,

viewTechDoc: techdocsPlugin.routes.docRoot,

});

bind(orgPlugin.externalRoutes, {

catalogIndex: catalogPlugin.routes.catalogIndex,

});

},

});

// Define app routes

const routes = (

<FlatRoutes>

<Route path="/" element={<Navigate to="catalog" />} />

<Route path="/catalog" element={<CatalogIndexPage />} />

<Route path="/catalog/:namespace/:kind/:name" element={<CatalogEntityPage />}>

{entityPage}

</Route>

<Route path="/docs" element={<TechDocsIndexPage />} />

<Route path="/docs/:namespace/:kind/:name/\*" element={<TechDocsReaderPage />}>

<TechDocsAddons>

<ReportIssue />

</TechDocsAddons>

</Route>

<Route path="/create" element={<ScaffolderPage />} />

<Route path="/api-docs" element={<ApiExplorerPage />} />

<Route

path="/catalog-import"

element={

<RequirePermission permission={catalogEntityCreatePermission}>

<CatalogImportPage />

</RequirePermission>

}

/>

<Route path="/search" element={<SearchPage />}>

{searchPage}

</Route>

<Route path="/settings" element={<UserSettingsPage />} />

<Route path="/catalog-graph" element={<CatalogGraphPage />} />

</FlatRoutes>

);

// Export the app root

export default app.createRoot(

<>

<AlertDisplay />

<OAuthRequestDialog />

<AppRouter>

<Root>{routes}</Root>

</AppRouter>

</>,

);

**Step 3 – Create Users & Groups in Backstage**

In catalog/entities/groups.yaml:

apiVersion: backstage.io/v1alpha1

kind: Group

metadata:

name: team-a

description: API & Backend Services Team

spec:

type: team

profile:

email: team-a@company.com

parent: backstage

children: []

In catalog/entities/users.yaml:

apiVersion: backstage.io/v1alpha1

kind: User

metadata:

name: ramannkhanna2

spec:

profile:

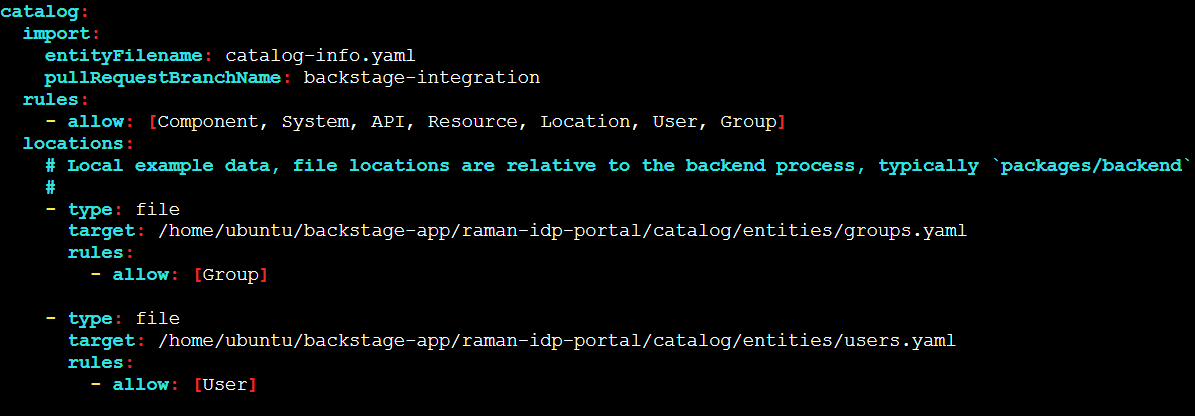
email: raman@company.com

picture: https://avatars.githubusercontent.com/u/89792896?v=4

memberOf: [team-a]

**Step 4 – Register Users & Groups in the Catalog**

In app-config.yaml:



**Step 5 – Restart Backstage**

yarn start

**Step 6 – Test the Login Flow**

1. Go to **http://<your-server>:7007**
2. Click **Sign in with GitHub**
3. If you are **not in allowedUsers/allowedOrganizations** → Login **denied**
4. If you are **in the allowed list** → Login works, and your **User entity** is recognized in Backstage.

**🔍 Verification**

✅ Unauthorized users **cannot** log in.  
✅ Authorized users can log in and are mapped to **User** and **Group** entities.  
✅ In **Catalog → Groups/Users**, you see the correct mapping.  
✅ Ownership in **Lab 6 Golden Path services** now links to **real Backstage teams**.

**Lab 9 – Terraform Secured Infra Module with GitHub Actions (Secret Management using GitHub Secrets)**

**🎯 Objective**

Build a reusable **Backstage Software Template** that scaffolds a Terraform project to provision:

* A **VPC** (custom CIDR, public/private subnets)
* An **S3 bucket with server-side encryption (SSE-S3)**

Integrate with **GitHub Actions** so infrastructure is deployed automatically,  
using **AWS credentials stored in GitHub Secrets**.

**🧠 Why This Matters**

* Developers get **self‑service infra provisioning** via Backstage.
* Infra is deployed in a **secure & auditable** way.
* No AWS credentials hard‑coded — securely stored in GitHub Secrets.
* Terraform code is **reusable** across teams and environments.

**📋 Prerequisites**

1. Backstage running with **GitHub integration** (Day 1 setup).
2. AWS account & IAM user with:
   * AmazonVPCFullAccess
   * AmazonS3FullAccess
   * or **least‑privilege policy** for VPC + S3 provisioning.
3. AWS IAM access key & secret key.
4. GitHub repository access (created by Backstage template).

**🛠 Step-by-Step Guide**

**Step 1 – Create Terraform Module**

Inside your Backstage template’s **skeleton/** folder,  
create Terraform files.

**📄 main.tf**

terraform {

required\_version = ">= 1.3.0"

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~> 5.0"

}

}

}

provider "aws" {

region = "${{ values.aws\_region }}"

}

resource "aws\_vpc" "main" {

cidr\_block = "${{ values.vpc\_cidr }}"

tags = {

Name = "${{ values.project\_name }}-vpc"

}

}

resource "aws\_s3\_bucket" "secure" {

bucket = "${{ values.s3\_bucket\_name }}"

server\_side\_encryption\_configuration {

rule {

apply\_server\_side\_encryption\_by\_default {

sse\_algorithm = "AES256"

}

}

}

tags = {

Name = "${{ values.project\_name }}-bucket"

}

}

**📄 outputs.tf**

output "vpc\_id" {

value = aws\_vpc.main.id

}

output "s3\_bucket\_name" {

value = aws\_s3\_bucket.secure.bucket

}

**Step 2 – Create GitHub Actions Workflow**

Inside **skeleton/.github/workflows/**, create:

📄 **terraform.yml**

name: Terraform Deployment

on:

  push:

    branches:

      - main

jobs:

  terraform:

    runs-on: ubuntu-latest

    env:

      AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

      AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

      AWS\_REGION: ${{ values.aws\_region }}

    steps:

      - name: Verify AWS credentials

        run: |

          echo "AWS\_ACCESS\_KEY\_ID: $AWS\_ACCESS\_KEY\_ID"

      - name: Checkout code

        uses: actions/checkout@v3

      - name: Setup Terraform

        uses: hashicorp/setup-terraform@v2

        with:

          terraform\_version: 1.5.7

      - name: Terraform Init

        run: terraform init

      - name: Terraform Plan

        run: terraform plan -input=false

      - name: Terraform Apply

        run: terraform apply -auto-approve -input=false

**Step 4 – Backstage Template Definition**

📄 **template.yaml**

apiVersion: scaffolder.backstage.io/v1beta3

kind: Template

metadata:

name: terraform-secured-infra

title: Terraform – Secured Infra (VPC, S3)

description: Scaffold a Terraform module for VPC + S3 with encryption.

spec:

owner: platform-team

type: infrastructure

parameters:

- title: Infrastructure Details

required: [project\_name, aws\_region, vpc\_cidr, s3\_bucket\_name]

properties:

project\_name:

type: string

title: Project Name

description: Name prefix for all AWS resources

aws\_region:

type: string

title: AWS Region

default: us-east-1

vpc\_cidr:

type: string

title: VPC CIDR

default: 10.0.0.0/16

s3\_bucket\_name:

type: string

title: S3 Bucket Name

description: Globally unique S3 bucket name

steps:

- id: fetch

name: Fetch Terraform Template

action: fetch:template

input:

url: ./skeleton

values:

project\_name: ${{ parameters.project\_name }}

aws\_region: ${{ parameters.aws\_region }}

vpc\_cidr: ${{ parameters.vpc\_cidr }}

s3\_bucket\_name: ${{ parameters.s3\_bucket\_name }}

- id: publish

name: Publish to GitHub

action: publish:github

input:

repoUrl: github.com?owner=ramannkhanna2&repo=${{ parameters.project\_name }}-infra

repoVisibility: private

output:

links:

- title: GitHub Repository

url: ${{ steps['publish'].output.remoteUrl }}

Create a **catalog-info.yaml**: (inside ./skeleton)

apiVersion: backstage.io/v1alpha1

kind: Resource

metadata:

name: ${{ values.project\_name }}-infra

description: Terraform-managed secured infrastructure

annotations:

github.com/project-slug: ramannkhanna2/${{ values.project\_name }}-infra

spec:

type: infrastructure

owner: platform-team

**Step 5 – Deploy via Backstage**

**Register the template under catalog in app-config.yml; than**

1. In Backstage, go to **Create… → Terraform Secure Infra**.
2. Fill:
   * **Project Name:** demo-secure-infra
   * **VPC CIDR:** 10.1.0.0/16
   * **Repo Name:** terraform-secure-demo
   * S3 Bucket Name: customer-data-secure-raman
3. Click **Create**.

**Step 6 – Verify Deployment**

1. Go to your GitHub repo **Actions** tab — Terraform workflow should run.
2. Check AWS:

bash

CopyEdit

aws ec2 describe-vpcs --region us-east-1

aws s3 ls

1. The **VPC** and **S3 bucket** should exist.

**Step 3 – Store AWS Credentials in GitHub Secrets in new scaffolded repo**

1. Go to **GitHub → Your Repo → Settings → Secrets and Variables.**
2. Add:
   * **Name:** AWS\_ACCESS\_KEY\_ID
     + **Value:** *Your AWS access key*
   * **Name:** AWS\_SECRET\_ACCESS\_KEY
     + **Value:** *Your AWS secret key*

MAKE SURE ADD ACCESSKEY AND SECRETACCESS KEY IN NEW SCAFFOLDED REPO as by default scaffolder will not transfer ur credentials 2 new repo

**Step 7 – Register Infra in Backstage Catalog**

Then:

bash

CopyEdit

# Register via Backstage UI or app-config.yaml

✅ **Result:**  
You now have a **Backstage self‑service Terraform template** that:

* Creates **AWS VPC & S3** with encryption.
* Deploys automatically via **GitHub Actions**.
* Uses **AWS keys stored in GitHub Secrets**.
* Registers infra in the **Backstage Catalog**.

**🔬 Lab 10 – Golden Path CI/CD with SAST, SBOM, and DAST Security Scans**

**🎯 Objective**

In this lab, you will:

* Integrate **Static Application Security Testing (SAST)** using Bandit.
* Generate a **Software Bill of Materials (SBOM)** using CycloneDX.
* Build & push your Docker image to **DockerHub**.
* Run a **Container Security Scan** using Trivy.
* Deploy the container inline in CI/CD and run a **Dynamic Application Security Test (DAST)** using OWASP ZAP.
* Store reports in GitHub for later analysis.

**🧠 Why This Lab Matters (Platform Engineering & IDP Perspective)**

In a **Platform Engineering** setup:

* **Security must be built into the developer workflow**, not bolted on later.
* Every service created via **Backstage Golden Path** inherits **the same CI/CD security checks** automatically.
* Developers **don’t need to learn security tooling** — the platform team bakes it into the service template.
* Security scanning at the **CI/CD stage prevents insecure code and vulnerable images from reaching production**.
* SBOM generation helps with **compliance, audit readiness, and incident response**.
* DAST scans catch **runtime vulnerabilities** that SAST or image scanning might miss.

By embedding this pipeline in the **microservice template** (from Lab 6), **every new service is secure-by-default**.

**📋 Pre-requisites**

Before starting:

1. **Backstage Golden Path Flask Service template** completed from Lab 6.
2. GitHub Actions enabled on your repositories.
3. DockerHub account with:
   * **DOCKERHUB\_USERNAME**
   * **DOCKERHUB\_TOKEN** (personal access token)
4. These GitHub Secrets must be set **in the generated microservice repository** (not the template repository):
   * DOCKERHUB\_USERNAME
   * DOCKERHUB\_TOKEN

**🛠 Step 1 – Add CI/CD Workflow to Template Skeleton**

In your Backstage template repo, inside skeleton/.github/workflows/ci.yml:

yaml

CopyEdit

name: CI Security Pipeline

on:

push:

branches: [ "main" ]

pull\_request:

branches: [ "main" ]

jobs:

lint-test-sast:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- uses: actions/setup-python@v5

with:

python-version: '3.11'

- name: Install Dependencies

run: |

pip install --upgrade pip

pip install -r requirements.txt

pip install bandit cyclonedx-bom pytest

- name: Run Unit Tests

run: pytest tests/ || true

- name: Run Bandit SAST Scan

run: bandit -r app -f json -o bandit-report.json || true

- name: Generate SBOM

run: cyclonedx-py requirements -i requirements.txt -o sbom.json

- name: Upload Reports

uses: actions/upload-artifact@v4

with:

name: sast-sbom-reports

path: |

bandit-report.json

sbom.json

docker-build-trivyscan:

needs: lint-test-sast

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- uses: docker/setup-buildx-action@v3

- uses: docker/login-action@v3

with:

username: ${{ secrets.DOCKERHUB\_USERNAME }}

password: ${{ secrets.DOCKERHUB\_TOKEN }}

- uses: docker/build-push-action@v5

with:

context: .

push: true

tags: ${{ secrets.DOCKERHUB\_USERNAME }}/${{ values.name }}:latest

- name: Install Trivy

run: |

sudo apt-get update

sudo apt-get install -y wget apt-transport-https gnupg lsb-release

wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo apt-key add -

echo deb https://aquasecurity.github.io/trivy-repo/deb $(lsb\_release -sc) main | sudo tee -a /etc/apt/sources.list.d/trivy.list

sudo apt-get update

sudo apt-get install -y trivy

- name: Run Trivy Scan

run: trivy image --format json --output trivy-report.json ${{ secrets.DOCKERHUB\_USERNAME }}/${{ values.name }}:latest

continue-on-error: true

dast-scan:

needs: docker-build-trivyscan

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- uses: docker/setup-buildx-action@v3

- name: Build Docker Image

run: docker build -t ${{ values.name }} .

- name: Run Container

run: docker run -d -p 5000:5000 --name ${{ values.name }} {{ values.name }}

- name: Wait for App

run: |

for i in {1..10}; do

if curl -s http://localhost:5000 | grep "Hello"; then

break

fi

sleep 3

done

- name: Run OWASP ZAP Scan

uses: zaproxy/action-baseline@v0.10.0

with:

target: 'http://localhost:5000'

continue-on-error: true

- name: Stop & Remove Container

if: always()

run: |

docker stop ${{ values.name }} || true

docker rm ${{ values.name }} || true

**🛠 Step 2 – Update template.yaml**

Ensure your Backstage template copies the CI/CD workflow into new services:

yaml

CopyEdit

steps:

- id: fetch

name: Fetch Template Source

action: fetch:template

input:

url: ./skeleton

values:

name: ${{ parameters.name }}

owner: ${{ parameters.owner }}

description: ${{ parameters.description }}

**🛠 Step 3 – Scaffold a New Service**

1. In Backstage, go to **Create → Golden Path Flask Service**.
2. Fill:
   * **Service Name**: flask-micro
   * **Owner**: team-a
   * **Description**: Flask microservice with built-in security checks
3. Click **Create**.
4. Backstage will:
   * Generate your service source code.
   * Publish to a new GitHub repository with the security CI/CD workflow.

**🛠 Step 4 – Configure GitHub Secrets**

In the **new service repo**:

1. Go to **Settings → Secrets and variables → Actions**.
2. Add:
   * DOCKERHUB\_USERNAME
   * DOCKERHUB\_TOKEN
3. Save.

**🛠 Step 5 – Trigger the Pipeline**

Push a commit to main in the new service repo.

1. Go to **Actions tab** in GitHub.
2. Observe the workflow:
   * **SAST**: Bandit runs and uploads a JSON report.
   * **SBOM**: CycloneDX creates sbom.json.
   * **Container Scan**: Trivy scans the built image.
   * **DAST**: OWASP ZAP tests the running container.

**✅ Expected Outcomes**

* Every microservice from Backstage automatically has:
  + SAST scan
  + SBOM generation
  + Container vulnerability scan
  + Runtime DAST scan
* Reports are uploaded as **GitHub Artifacts** for review.
* Developers **don’t need to manually configure security scans** — they’re part of the Golden Path.

**💡 Platform Engineering Best Practice**

* **Centralized**: The workflow lives in the template, controlled by the platform team.
* **Self-Service**: Developers only see results, not security complexity.
* **Governed**: Security scans run on **every commit** to main.
* **Auditable**: Reports are stored automatically for compliance.