**🔬 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.

**Lab 11 – GitHub Actions Deploy Integration (Click‑to‑Deploy in Backstage)**

**🎯 Objective**

In this lab, you will:

* Integrate **GitHub Actions** into your Backstage developer portal.
* View **CI/CD run history** for each service inside Backstage.
* Trigger deployments manually via **Click‑to‑Deploy** from Backstage.
* Allow developers to **inspect, retry, and audit pipelines** without leaving Backstage.

**🧠 Why This Matters in Platform Engineering**

* **Developer Experience (DevEx):** Developers no longer need to switch to GitHub UI to see CI/CD runs.
* **Self‑Service Pipelines:** Any onboarded service in Backstage can trigger its own build/deploy pipeline.
* **Audit & Compliance:** Centralized visibility for pipeline status, logs, and approvals.
* **Consistency:** Standardized CI/CD integration across all microservices using the **Golden Path** approach from Lab 6.

**📋 Prerequisites**

1. Backstage instance running from Lab 6 setup.
2. GitHub OAuth already configured (Lab 8).
3. Your service (e.g., customer-orders-service) is already in the Backstage catalog with a working CI pipeline (Lab 10).
4. **GitHub Personal Access Token (PAT)** or OAuth credentials.

**🛠 Step‑By‑Step Implementation**

**Part 1 – Install GitHub Actions Plugin**

**1.1 – Install in Frontend**

bash

CopyEdit

cd ~/backstage-app/raman-idp-portal

yarn --cwd packages/app add @backstage-community/plugin-github-actions

**1.2 – Install GitHub Auth in Backend**  
*(Skip if already done in Lab 8)*

bash

CopyEdit

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

**Part 2 – Backend Auth Provider Setup ( skip if dne)**

**2.1 – Enable the GitHub Auth Backend Module**  
Edit:

bash

CopyEdit

vi packages/backend/src/index.ts

Add:

ts

CopyEdit

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

**2.2 – Add Credentials to app-config.yaml**

yaml

CopyEdit

auth:

providers:

github:

development:

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

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

Export in terminal (or .env file):

bash

CopyEdit

export AUTH\_GITHUB\_CLIENT\_ID=your-client-id

export AUTH\_GITHUB\_CLIENT\_SECRET=your-client-secret

**Part 3 – GitHub Integration**

Edit app-config.yaml:

yaml

CopyEdit

integrations:

github:

- host: github.com

apiBaseUrl: https://api.github.com

**Part 4 – Annotate Service for GitHub Actions**

Edit catalog-info.yaml in your service repo (customer-orders-service):

yaml

CopyEdit

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: customer-orders-service

description: Flask microservice

annotations:

github.com/project-slug: ramannkhanna2/customer-orders-service

spec:

type: service

lifecycle: production

owner: user:raman

**Note:**  
github.com/project-slug is mandatory for Backstage to fetch the correct Actions runs.

Push changes to GitHub:

bash

CopyEdit

git add catalog-info.yaml

git commit -m "Add GitHub Actions annotation"

git push origin main

**Part 5 – Enable GitHub Actions Tab in Backstage**

Edit:

bash

CopyEdit

vi packages/app/src/components/catalog/EntityPage.tsx

Add at top:

import {

EntityGithubActionsContent,

isGithubActionsAvailable,

} from '@backstage-community/plugin-github-actions';

Inside your service page definition:

const serviceEntityPage = (

<EntityLayout>

{/\* Existing tabs \*/}

<EntityLayout.Route

path="/github-actions"

title="GitHub Actions"

if={isGithubActionsAvailable}

>

<EntityGithubActionsContent view="cards" />

</EntityLayout.Route>

</EntityLayout>

);

**Part 6 – Test the Integration**

1. Restart Backstage:
2. Yarn start
3. Go to your service in Backstage (customer-orders-service).
4. You should see a **"GitHub Actions"** tab.
5. Click it — you should see:
   * **Run history**
   * **Statuses** (✔️ Passed / ❌ Failed)
   * **Click‑to‑Deploy** button (if workflow supports workflow\_dispatch trigger).

**📌 Extra – Enable Click‑to‑Deploy**

To allow manual deployments from Backstage:

1. In your GitHub Actions workflow (.github/workflows/deploy.yml):

yaml

CopyEdit

on:

workflow\_dispatch: # Enables manual trigger

1. Commit & push.
2. In Backstage, open the **GitHub Actions** tab → click **Run Workflow**.

**✅ Lab Verification**

* GitHub Actions tab visible in Backstage service page.
* Workflow history loads without error.
* Clicking a run opens GitHub logs in a new tab.
* Manual deploy button works (workflow\_dispatch).

**🚀 Outcome**

You’ve now:

* Integrated GitHub Actions into your Backstage portal.
* Enabled pipeline history + click‑to‑deploy for any onboarded service.
* Taken another step towards **self‑service platform engineering**.

**Lab 12: Adding Docs with TechDocs (Backstage-native Docs)**

**🔹 Objective**

By the end of this lab, each trainee will:

* Add service-level documentation using Markdown and MkDocs.
* Configure TechDocs to render and serve these docs within Backstage.
* Understand how documentation becomes a first-class citizen in the developer portal.

**💡 Why This Lab Matters**

**Platform Engineering Perspective:** Documentation is one of the most underappreciated, yet most critical, components of platform engineering. This lab demonstrates how TechDocs enables **self-serve developer experience** by:

* Making documentation visible and accessible *in the same place* where developers discover and operate services.
* Eliminating the need to jump between Confluence, Google Docs, Notion, etc.
* Creating a documentation standard (Markdown + MkDocs).
* Encouraging devs to treat docs like code (version-controlled, PR-reviewed, co-located).

**📁 Lab Setup**

Ensure the following structure exists in your GitHub repo :

Customer-service/

├── catalog-info.yaml

├── mkdocs.yml

└── docs/

└── index.md

**✅ Step 1: Create Documentation Files**

**📄 Create mkdocs.yml in root:**

site\_name: [ssecure-raman-customer-orders-service](https://github.com/ramannkhanna2/ssecure-raman-customer-orders-service)

nav:

- Home: index.md

plugins:

- techdocs-core

**📄 Create docs/index.md**

# Raman Microservice

Welcome to the \*\*[ssecure-raman-customer-orders-service](https://github.com/ramannkhanna2/ssecure-raman-customer-orders-service)\*\* documentation.

## 🔧 Features

- Flask-based microservice

- CI/CD via GitHub Actions

- SAST, DAST, SBOM integrated

- Dockerized and Helm-compatible

- Exposes Prometheus metrics

## 🚀 Deployment

Deployed via GitHub Actions workflow.

## 🧪 Observability

Integrated with:

- Prometheus for metrics

- Loki/ELK for logs

- Grafana dashboards

## 👤 Owner

Maintained by \*\*Raman Khanna\*\*.

**✅ Step 2: Add TechDocs Annotation to Catalog**

Open your catalog-info.yaml and ensure it includes:

metadata:

annotations:

backstage.io/techdocs-ref: dir:.

This tells Backstage to look for docs locally inside the repo.

**✅ Step 3: Commit and Push Changes**

git add mkdocs.yml docs/

git commit -m "Add TechDocs documentation"

git push

**⚙️ Step 4: Confirm TechDocs Configuration in Backstage**

Ensure the following block is in your app-config.yaml:

techdocs:

builder: local

generator:

runIn: docker

publisher:

type: local

backend:

baseUrl: http://localhost:7007

✉️ If you use an external publisher (like GCS or S3), configuration will vary.

**✅ Step 5: View Docs in Backstage UI**

1. Open your Backstage portal
2. Navigate to: Catalog → raman-micro
3. Click on the **"View TechDocs"** button (top-right corner)

You should see your Markdown rendered beautifully inside the UI. 🎉

**📚 Summary for Trainees**

|  |  |
| --- | --- |
| **What You Did** | **Why It Matters** |
| Wrote Markdown Docs | Consistent and readable documentation format |
| Added mkdocs.yml | Defines the structure and metadata for docs |
| Used TechDocs annotation | Connects your repo to Backstage TechDocs |
| Rendered Docs in Backstage | Centralized access in developer portal |

**🔹 Post-Lab Discussion Questions**

1. How does co-locating docs with code improve your dev workflow?
2. Can TechDocs replace existing tools like Confluence?
3. How might you enforce documentation as part of service creation?

**📈 Platform Engineering Takeaways**

* Enables **self-serve documentation** without tribal knowledge.
* Bridges the gap between **code ownership** and **service discoverability**.
* Makes documentation versioned, reviewed, and owned just like code.

**Lab 13 – Add Prometheus Metrics & Grafana Dashboards to customer-order-service**

**🎯 Goal**

Enable your customer-order-service Flask microservice to **expose runtime metrics**, have them **scraped by Prometheus**, visualized in **Grafana dashboards**, and **linked inside Backstage** for developer observability.

**🔍 Why This Lab Matters in Platform Engineering**

As part of an **Internal Developer Platform (IDP)**:

1. **Self-service Observability**  
   Developers should **not** need to manually set up monitoring for every new service. The **Golden Path** should include *instrumentation by default*.
2. **Faster Troubleshooting**  
   Exposing metrics via Prometheus allows:
   * Real-time visibility into service health.
   * Quick detection of anomalies without SSH-ing into servers.
3. **Backstage Integration**  
   Linking **Grafana dashboards** directly inside Backstage means:
   * One-click access to metrics per service.
   * A single pane of glass for developers and SREs.
4. **SLO / SLA Tracking**  
   Metrics collected here can power Service Level Objectives (SLOs) and error budgets.

**📦 Lab Outcomes**

By end of this lab, you will:

* ✅ Have /metrics endpoint in your microservice.
* ✅ See metrics scraped by Prometheus.
* ✅ Build a Grafana dashboard.
* ✅ Link the dashboard in Backstage **catalog-info.yaml**.

**🧩 Steps**

**Step 1 – Add Prometheus Client to Flask**

1. **Install client library**:

sudo apt update

sudo apt install python3-pip -y

apt install python3.12-venv

python3 -m venv venv

# 2. Activate it

source venv/bin/activate

# 3. Install dependencies inside the venv

pip install -r requirements.txt

Add to requirements.txt:

flask

prometheus\_client

1. **Instrument your Flask app** (app.py):

from flask import Flask, Response

from prometheus\_client import Counter, Histogram, generate\_latest, CONTENT\_TYPE\_LATEST

import time

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

# Metrics

REQUEST\_COUNT = Counter("request\_count", "Total number of requests")

REQUEST\_LATENCY = Histogram("request\_latency\_seconds", "Request latency in seconds")

@app.route("/")

def home():

REQUEST\_COUNT.inc()

with REQUEST\_LATENCY.time():

time.sleep(0.1) # Simulate processing

return "Hello from Raman Micro!"

# Expose metrics

@app.route("/metrics")

def metrics():

return Response(generate\_latest(), mimetype=CONTENT\_TYPE\_LATEST)

if \_\_name\_\_ == "\_\_main\_\_":

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

**Step 2 – Run and Test Metrics Endpoint**

Python3 app.py

Check in browser or terminal:

curl http://pubIP:5000/metrics

You should see output like:

# HELP request\_count Total number of requests

# TYPE request\_count counter

request\_count 2.0

**Step 3 – Create Prometheus Config**

Create prometheus.yml:

global:

scrape\_interval: 10s

scrape\_configs:

- job\_name: 'raman-micro'

static\_configs:

- targets: ['host.docker.internal:5000']

This scrapes metrics from your Flask app every 10 seconds.

**Step 4 – Run Prometheus & Grafana in Docker**

Create a shared network:

docker network create observability-net

**Run Prometheus**:

docker run -d \

--name prometheus \

--network observability-net \

--add-host=host.docker.internal:host-gateway \

-v "$(pwd)/prometheus.yml":/etc/prometheus/prometheus.yml \

-p 9090:9090 \

prom/Prometheus

**Run Grafana**:

docker run -d \

--name grafana \

--network observability-net \

-p 3001:3000 \

grafana/grafana

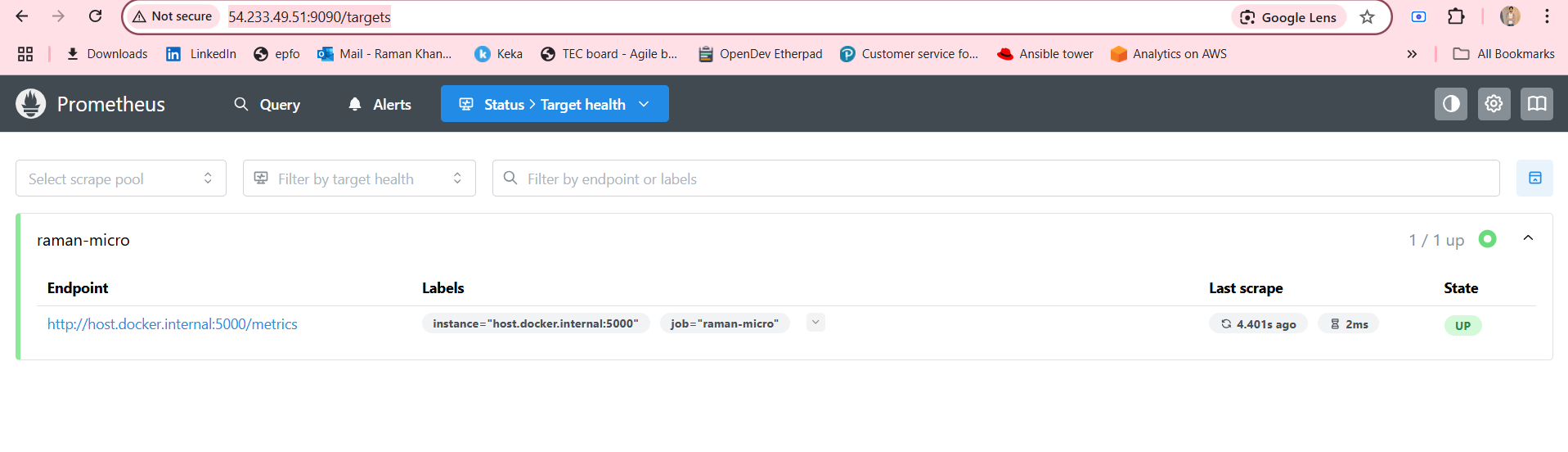
**Step 5 – Check Prometheus Target**

Go to:

http://<your-server-ip>:9090/targets

You should see:

raman-micro – UP



**Step 6 – Configure Grafana**

1. Open Grafana:

http://<your-server-ip>:3001

(Default login: admin/admin)

1. Go to:

⚙️ → Data Sources → Add Data Source → Prometheus

Set URL:

http://prometheus:9090

Click **Save & Test** → Should say “Data source is working”.

**Step 7 – Create Grafana Dashboard**

1. Go to:

sql

CopyEdit

Dashboards → New → New Dashboard

**Panel 1 – Total Requests**:

* Query:

request\_count\_total

* Title: **Total Requests**
* Visualization: Time series

**Panel 2 – Average Latency**:

* Query:

prometheus

CopyEdit

rate(request\_latency\_seconds\_sum[1m]) / rate(request\_latency\_seconds\_count[1m])

* Title: **Average Latency**
* Visualization: Time series

Save as:

CopyEdit

raman-micro-dashboard



**Step 8 – Link Dashboard in Backstage**

Edit catalog-info.yaml of your service:

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: raman-micro

description: Flask microservice with metrics

annotations:

github.com/project-slug: ramannkhanna2/raman-micro

grafana/dashboard-url: http://<your-server-ip>:3001/d/abc123/raman-micro-dashboard

spec:

type: service

owner: user:raman

lifecycle: production

Now, inside Backstage **service page**, you can show a **Grafana link** to view metrics.

**📌 Key Platform Engineering Takeaways**

* **Golden Path for Observability** → Any new microservice should have metrics from day 1.
* **Self-service Monitoring** → Devs can see service health without raising infra tickets.
* **One-Stop Developer Portal** → Backstage becomes the central hub for code, builds, and monitoring.
* **Promotes SRE Practices** → Enables tracking of error budgets, latency SLIs, and uptime SLOs.

**Lab 14 – Integrate Grafana Dashboards into Backstage for customer-order-svc**

**🎯 Goal**

Link your **Grafana dashboards** for the customer-order-svc microservice directly into Backstage so developers, SREs, and platform teams can:

* See service metrics inside the **Backstage service entity page**.
* Open Grafana dashboards with **one click**.
* Easily filter dashboards for the specific service using annotations and tags.

**🔍 Why This Lab Matters in Platform Engineering**

In **Lab 12**, you instrumented your service with **Prometheus metrics** and visualized them in **Grafana**.  
However, without **Backstage integration**, developers have to:

* Remember the Grafana URL.
* Search for the correct dashboard manually.
* Navigate across multiple tools.

By embedding Grafana into **Backstage**:

* **Single Pane of Glass** → Developers see code, builds, deployments, and metrics in one place.
* **Golden Path Observability** → Every scaffolded service can automatically link to its dashboard.
* **Faster Incident Response** → No context switching during production issues.
* **Developer Self-Service** → No tickets needed to access performance data.

This is a key **Platform Engineering principle**:

“Developers should spend time shipping features, not finding dashboards.”

**📦 Lab Outcomes**

By the end of this lab:

* ✅ Grafana community plugin installed in Backstage.
* ✅ Backstage proxy configured for Grafana API access.
* ✅ Service entity page shows **Grafana Dashboard Card**.
* ✅ customer-order-svc dashboard linked via annotations.

Two community versions available:

* Official Backstage community version: **@backstage-community/plugin-grafana**
* Roadie fork by K‑Phoen: **@k-phoen/backstage-plugin-grafana**

📦 For most users, the **community version** is ideal:

<https://github.com/backstage/community-plugins/blob/main/workspaces/grafana/plugins/grafana/docs/setup.md>

root@ip-172-31-14-172:~/backstage-app/ramanapp/packages/app# yarn add @backstage-community/plugin-grafana

➤ YN0000: · Yarn 4.4.1

➤ YN0000: ┌ Resolution step

➤ YN0000: └ Completed in 0s 789ms

➤ YN0000: ┌ Post-resolution validation

➤ YN0060: │ @testing-library/react is listed by your project

* To vierify if plugin installed :

root@ip-172-31-14-172:~/backstage-app/ramanapp/packages/app# cat package.json

* U shud see the plugin there .
* Go to Grafana and generate a service account and token

**🔧 Step 2: Configure Proxy and Grafana URL (if Grafana requires auth or is remote)**

Edit **app-config.yaml** (at project root):

proxy:

### Example for how to add a proxy endpoint for the frontend.

# endpoints:

# '/test':

# target: 'https://example.com'

# changeOrigin: true

'/grafana/api':

target: http://54.233.49.51:3001

headers:

# Only needed if your Grafana API requires an auth token

Authorization: Bearer glsaRxAGZz5qCqX8j8dqflkgbut #grafana sa token

grafana:

domain: http://54.233.49.51:3001

# If you're using Grafana's new unified alerting:

unifiedAlerting: false

* Expose the plugin to Backstage:
* Create a file named plugins.tsx in ~/backstage-app/ramanapp/packages/app/src
* // packages/app/src/plugins.tsx
* // other plugins...
* export { grafanaPlugin } from '@backstage-community/plugin-grafana';
* **for testing if ur able to reach Grafana ui from api :**

curl -H "Authorization: Bearer glsa\_RxAGZz5qCqX8j8dqflnr8oi" http://54.233.49.51:3001/api/search?tag=raman-micro

**🧩 Step 3: Update Service Entity Layout to Include Dashboard Card**

[**https://github.com/backstage/community-plugins/blob/main/workspaces/grafana/plugins/grafana/docs/dashboards-on-component-page.md**](https://github.com/backstage/community-plugins/blob/main/workspaces/grafana/plugins/grafana/docs/dashboards-on-component-page.md)

Edit **packages/app/src/components/catalog/EntityPage.tsx** (or wherever you layout service pages):

**Display dashboards on a component page**

Adding the EntityGrafanaDashboardsCard component to an entity's page will display a list of dashboards related to that entity.

// packages/app/src/components/catalog/EntityPage.tsx

import { EntityGrafanaDashboardsCard } from '@backstage-community/plugin-grafana';

// ...

const overviewContent = (

<Grid container spacing={3} alignItems="stretch">

<Grid item md={6}>

<EntityAboutCard variant="gridItem" />

</Grid>

<Grid item md={6}>

{/\* Grafana alert card start \*/}

<EntityGrafanaDashboardsCard />

{/\* Grafana alert card end \*/}

</Grid>

<Grid item md={4} xs={12}>

<EntityLinksCard />

</Grid>

<Grid item md={8} xs={12}>

<EntityHasSubcomponentsCard variant="gridItem" />

</Grid>

</Grid>

);

* on Grafana ui add the tag to the "count" dashboard "customer-micro" so that backstage plugin can find the dashboard of urs in Grafana ..

**🔁 Step 4: Ensure Your catalog-info.yaml Has the Grafana Annotation**

root@ip-172-31-14-172:~/raman-micro# cat catalog-info.yaml

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: raman-micro

description: Flask microservice demo with Prometheus & Grafana

tags:

- flask

- python

- prometheus

- demo

annotations:

github.com/project-slug: ramannkhanna2/raman-micro

backstage.io/techdocs-ref: dir:.

grafana/dashboard-url: http://54.233.49.51:3001/d/1f416562-37ea-4ef1-8df3-d6d1381b7f89/count

grafana/dashboard-selector: customer-micro

#grafana/dashboard-selector: 'tag=raman-micro'

#grafana/tag-selector: raman-micro

spec:

type: service

lifecycle: production

owner: dev-team

system: raman-platform

This ensures the plugin can pick up the correct dashboard to display.

**🌀 Step 5: Restart Backstage & Refresh Entity**

bash

CopyEdit

cd ~/backstage-app/raman-app

yarn dev

OR

Yarn start

* Than again register ur service on backstage .

https://github.com/ramannkhanna2/raman-micro/blob/main/catalog-info.yaml

→ Go to **Catalog → raman-micro**, then click the “Refresh” button (⋮ menu).

You should see a **“Grafana”** card with your dashboard preview and alerts (if any).

**✅ Summary: What You’ve Set Up**

* 🎯 Installed the **Grafana plugin**
* 🔧 Configured proxy and domain in app-config.yaml
* 🛠️ Added **Grafana cards** to service page layout
* ✅ Provided dashboard URL in your service metadata
* 🔄 Restarted Backstage and refreshed

**📌 Platform Engineering Value Recap**

* **Developer Experience (DX)** → Metrics are part of the developer workflow, not an afterthought.
* **Golden Path Compliance** → Every service follows the same monitoring standard.
* **Self-Service Observability** → Devs don’t rely on ops teams to view metrics.
* **Reduced MTTR** → Troubleshooting happens faster with in-context metrics.
* **IDP Maturity** → This makes your Backstage portal a *true* single entry point for both code & operational insights.