**Lab: Register Your Microservice in Backstage**

**🎯 Goal:**

Add your service (raman-micro) to Backstage’s Software Catalog so it's visible in the portal.

**🧩 1.1 — Clone Your Repo (Optional, if you want to work locally)**

bash

CopyEdit

git clone https://github.com/ramannkhanna2/raman-micro.git

cd raman-micro

**🧩 1.2 — Add catalog-info.yaml**

Create a file named catalog-info.yaml in the root of your repo. This YAML file tells Backstage about the service — its name, owner, type, GitHub location, etc.

📄 **File**: raman-micro/catalog-info.yaml

yaml

CopyEdit

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: raman-micro

description: A Flask microservice with CI/CD and observability

annotations:

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

backstage.io/techdocs-ref: dir:.

prometheus.io/scrape: "true"

loki.io/logs: "true"

spec:

type: service

lifecycle: experimental

owner: platform-team

**Explanation of fields**:

| **Field** | **Description** |
| --- | --- |
| name | Display name in Backstage |
| annotations.github.com/project-slug | Enables GitHub integration |
| techdocs-ref | For TechDocs, we’ll use this later |
| prometheus.io/scrape | Lets Backstage know this service emits metrics |
| loki.io/logs | Indicates the service emits logs |
| owner | Should match a group/user in Backstage (platform-team for now) |

**🧩 1.3 — Commit & Push to GitHub**

bash

CopyEdit

git add catalog-info.yaml

git commit -m "Add Backstage catalog metadata"

git push origin main

✅ After this step, the metadata will be available at:

bash

CopyEdit

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

**🧩 1.4 — Import Service into Backstage UI**

1. Open your Backstage app:  
   🌐 http://localhost:3000 (default)
2. In the sidebar, go to:  
   **Catalog → Register Existing Component**
3. When prompted for the YAML file URL, paste:

bash

CopyEdit

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

1. Click ✅ **Analyze**. Backstage will show metadata preview.
2. Click ➕ **Import** to add it to the catalog.

**🧩 1.5 — Verify Service Is Listed**

* Go to **Catalog → Components**
* You should now see: raman-micro
* Click it to view:
  + Description
  + Metadata
  + Owner
  + GitHub repo link
  + Placeholder for TechDocs

✅ **Success**: You have now added your microservice to the Backstage UI and registered it in the Software Catalog.

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

📌 With TechDocs, developers can view documentation for any service **directly inside the Backstage UI**, powered by [MkDocs](https://www.mkdocs.org/" \t "_new) and Markdown.

**🧠 What We’ll Do**

| **Task** | **Description** |
| --- | --- |
| 1️⃣ | Add mkdocs.yml and docs/ folder to your repo |
| 2️⃣ | Add basic Markdown documentation |
| 3️⃣ | Ensure your catalog-info.yaml includes the TechDocs annotation |
| 4️⃣ | Configure Backstage to generate and serve docs |
| 5️⃣ | Verify in UI via **View TechDocs** |

**📁 Step 2.1: Add Docs Folder + mkdocs.yml to Your Repo**

In your raman-micro GitHub repo, create the following structure:

pgsql

CopyEdit

raman-micro/

├── catalog-info.yaml

├── mkdocs.yml

└── docs/

└── index.md

**📝 mkdocs.yml**

Create this at the root (raman-micro/mkdocs.yml):

yaml

CopyEdit

site\_name: raman-micro

nav:

- Home: index.md

plugins:

- techdocs-core

**📄 docs/index.md**

Create this at raman-micro/docs/index.md:

markdown

CopyEdit

# Raman Microservice

Welcome to the \*\*raman-micro\*\* 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 2.2: Ensure catalog-info.yaml Has TechDocs Annotation**

It should already have this line (you added it in Step 1):

yaml

CopyEdit

annotations:

backstage.io/techdocs-ref: dir:.

✅ If it’s already there, no change needed. If not, add it.

**🔁 Step 2.3: Commit & Push to GitHub**

Run this from the root of your local raman-micro clone:

bash

CopyEdit

git add mkdocs.yml docs/

git commit -m "Add TechDocs documentation"

git push

**⚙️ Step 2.4: Enable TechDocs in Backstage (Optional Check)**

Backstage needs to be configured for **local build** or **CI build** for TechDocs.

You likely already have this by default in app-config.yaml:

yaml

CopyEdit

techdocs:

builder: local # or 'external' if using CI builds

generator:

runIn: docker

publisher:

type: local

And the backend should serve docs at:

yaml

CopyEdit

backend:

baseUrl: http://localhost:7007

If this is present, you're good. Otherwise let me know and I’ll help you configure it.

**✅ Step 2.5: Open Backstage and View TechDocs**

1. Go to **Catalog > raman-micro**
2. Click **“View TechDocs”** on the top-right

You should now see your rendered index.md contents 🎉

**🟢 Summary**

✅ You now have:

* Markdown docs stored in GitHub
* Rendered natively in Backstage via TechDocs
* Future-ready for CI-based doc builds

**Lab : Add Prometheus Metrics & Grafana Dashboards to raman-micro**

**🎯 Goal**

Let’s instrument your raman-micro Flask app to expose metrics in Prometheus format and visualize them in Grafana dashboards — all **linked inside Backstage**.

**🔍 What You’ll Get**

✅ A working /metrics endpoint in Flask  
✅ Prometheus scraping this endpoint  
✅ Grafana showing service metrics  
✅ Dashboard link inside Backstage UI

**📁 Overview of Steps**

| **Step** | **Task** |
| --- | --- |
| 3.1 | Add Prometheus client to Flask |
| 3.2 | Expose metrics on /metrics |
| 3.3 | Add Prometheus config to scrape the service |
| 3.4 | Run Prometheus |
| 3.5 | Add Grafana with dashboard |
| 3.6 | Link Grafana dashboard in Backstage |

**✅ Step 3.1: Add Prometheus Client to Flask**

**📦 1. Install Prometheus Python client**

In your repo (raman-micro):

bash

CopyEdit

pip install prometheus\_client

Also add it to your requirements.txt:

nginx

CopyEdit

prometheus\_client

**🧠 2. Update Flask app**

In app.py or your main Flask file:

python

CopyEdit

from flask import Flask

from prometheus\_client import start\_http\_server, Counter, Histogram, generate\_latest

from prometheus\_client import CONTENT\_TYPE\_LATEST

from flask import Response

import time

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

# Custom metrics

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

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

@app.route("/")

def home():

REQUEST\_COUNT.inc()

with REQUEST\_LATENCY.time():

time.sleep(0.1) # simulate processing

return "Hello from Raman Micro!"

# Expose Prometheus metrics at /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)

✅ Now your service exposes Prometheus-compatible metrics at http://localhost:5000/metrics

**✅ Step 3.2: Run and Test Metrics Endpoint**

bash

CopyEdit

python app.py

In browser or curl:

bash

CopyEdit

curl http://localhost:5000/metrics

You should see output like:

nginx

CopyEdit

# HELP request\_count Total number of requests

# TYPE request\_count counter

request\_count 2.0

...

**🔁 You Need prometheus.yml Because...**

**🧠 prometheus.yml is the configuration for the Prometheus server, which is the metrics collector/scraper.**

This file tells Prometheus:

* What services to scrape (targets)
* How often (scrape\_interval)
* Under what name (job\_name)
* **📦 So, the relationship is:**

| **Component** | **Role** | **Location** |
| --- | --- | --- |
| prometheus\_client in Flask | Exposes /metrics | Inside your app |
| prometheus.yml | Tells Prometheus where and how to scrape | Mounted in Prometheus container |

**✅ Step 3.3: Add Prometheus config :**

Create prometheus.yml config:

root@ip-172-31-14-172:~/raman-micro# cat prometheus.yml

global:

scrape\_interval: 10s

scrape\_configs:

- job\_name: 'raman-micro'

static\_configs:

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

This tells Prometheus to scrape http://localhost:5000/metrics every 10 seconds.

**✅ Step 3.4: Run Prometheus and garafana :**

docker network create observability-net

# Run Prometheus and connect to network

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 and connect to network

docker run -d \

--name grafana \

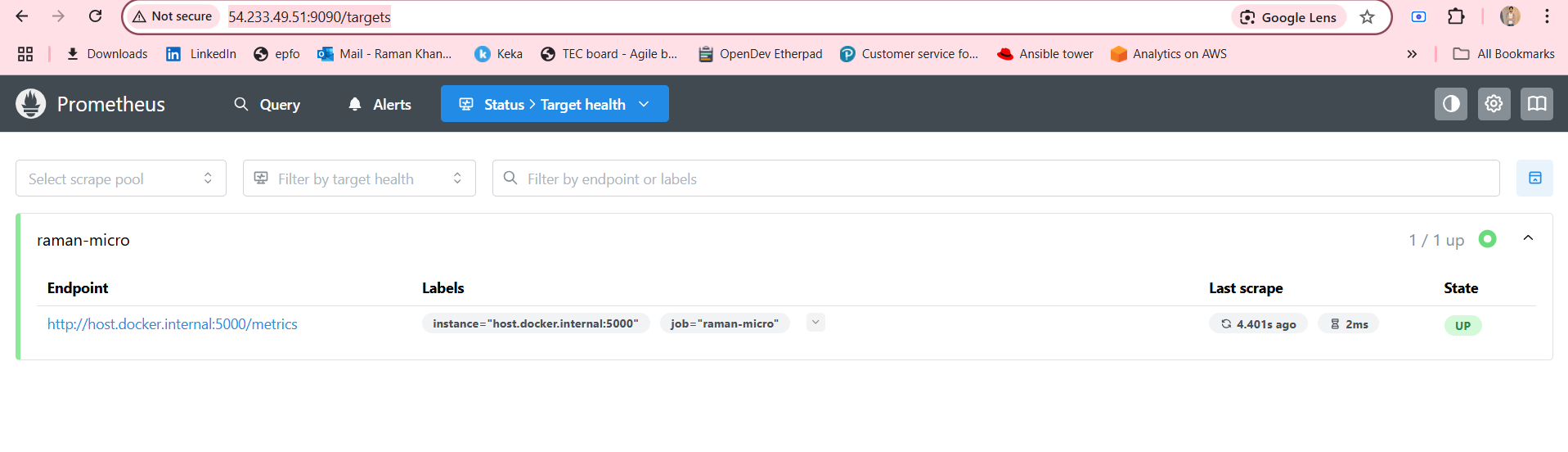
--network observability-net \

-p 3001:3000 \

grafana/Grafana

* Check target health on promethues ui if target point is up or not !!

http://54.233.49.51:9090/targets



You should now see:

* ✅ raman-micro **UP**
* Scrape successful

**📝 Summary**

| **Task** | **Command or Check** |
| --- | --- |
| Flask app running | python3 app/main.py |
| Flask metrics available | curl http://localhost:5000/metrics |
| Prometheus uses host access | --add-host=host.docker.internal:host-gateway |
| Docker network OK | Use same --network observability-net |

1. **Add Prometheus as a data source**:
   * Go to **Gear icon ⚙️ > Data Sources**
   * Click **“Add data source”**
   * Choose **Prometheus**
   * Fill in:

makefile

CopyEdit

Name: Prometheus

URL: http://prometheus:9090

⚠️ Important: This works only because both containers share the observability-net network.

* + Click **Save & Test**

✅ You should see **"Data source is working"**

**🧪 Next: Build a Basic Dashboard**

Let’s visualize your Flask app metrics now.

1. Go to **Dashboards → New → New Dashboard**

**📊 Step 2: Add Metrics Panel**

**🎯 Panel 1: Request Count Over Time**

* **Data source**: Prometheus
* **Query**:

nginx

CopyEdit

request\_count\_total

* **Legend**: Total Requests
* **Visualization**: Time series
* **Title**: Total Requests (request\_count\_total)
* Click **Apply**

**📈 Panel 2: Request Latency**

* Add another panel
* **Query**:

scss

CopyEdit

rate(request\_latency\_seconds\_bucket{le="1.0"}[1m])

* OR visualize just average latency:

css

CopyEdit

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

* **Legend**: Latency
* **Visualization**: Time series
* **Title**: Average Request Latency
* Click **Apply**

**✅ Step 3: Save Dashboard**

* Click the top-right **disk icon**
* Name: raman-micro-dashboard
* Save it

**🧪 Optional: Try Generating Some Traffic**

If you're only seeing request\_count\_total = 1, hit the app a few times:

bash

CopyEdit

curl http://localhost:5000/

Then reload the /metrics and Grafana — you should see request\_count\_total increase, and latency buckets populated.



**Do You Need prometheus\_exporter.py?**

**No — not anymore.**  
You’ve already:  
✅ Integrated Prometheus in your actual Flask microservice  
✅ Exposed /metrics  
✅ Verified Prometheus is scraping real-time app metrics

**✅ You can delete or ignore prometheus\_exporter.py.**

**✅ Final Recommendation**

| **File** | **Keep?** | **Why?** |
| --- | --- | --- |
| app/main.py | ✅ Yes | Your real microservice with embedded metrics |
| prometheus\_exporter.py | ❌ No | Was only for testing/simulation; not needed anymore |

**📝 Summary Table**

| **Component** | **Status** | **Notes** |
| --- | --- | --- |
| Flask app | ✅ | Prometheus client instrumented |
| /metrics | ✅ | Exposes custom & default metrics |
| Prometheus | ✅ | Scrapes every 10s using config |
| Grafana | ✅ | Container running, UI works |
| Data Source | ✅ | Prometheus connected using service name |
| Dashboard | ✅ | Panels for count and latency added |

**Lab: Install the Grafana comunity Plugin on backstage.**

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.

### A typical reason to do this is to handle HTTPS and CORS for internal services.

# 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>**

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 "raman-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: raman-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

**Lab : GitHub Actions Deploy Integration (Click-to-Deploy in Backstage)**

**✅ Goal**

Fully integrate **GitHub Actions plugin** into your Backstage portal such that:

* You can **see GitHub Actions run history**
* You can **trigger deployments manually** (click-to-deploy)
* All developers can inspect, retry, and audit pipelines inside Backstage

**🧩 PART 1: Install GitHub Actions Plugin in Backstage**

**1.1: Install Plugin in Frontend (packages/app)**

bash

CopyEdit

cd ~/backstage-app/raman-app

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

**1.2: Install GitHub Auth in Backend (if not already done)**

bash

CopyEdit

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

Important addition :

And add the following dependency to your backend index file:

(venv) root@ip-172-31-14-172:~/backstage-app/ramanapp/packages/backend/src# vi index.ts :

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

**🔐 PART 2: GitHub Oauth Setup**

**2.1: Create GitHub Oauth App**

Go to: **[https://github.com/settings/developers](https://github.com/settings/developers" \t "_new)** → Oauth Apps → “New Oauth App”

| **Field** | **Value** |
| --- | --- |
| App Name | Backstage Raman |
| Homepage URL | <http://localhost>:3000 |
| Callback URL | <http://localhost>:7007/api/auth/github/handler/frame |

After creating it, copy the **Client ID** and **Client Secret**

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

In ~/backstage-app/raman-app/app-config.yaml:

yaml

CopyEdit

auth:

providers:

github:

development:

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

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

Now export these in your terminal (or .env):

bash

CopyEdit

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

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

**🔗 PART 3: GitHub Integration**

**3.1: Add GitHub integration config in app-config.yaml:**

yaml

CopyEdit

integrations:

github:

- host: github.com

apiBaseUrl: <https://api>.github.com

**🧠 PART 4: Annotate Your Service in Backstage Catalog**

Your service is raman-micro, already in GitHub.

Edit catalog-info.yaml inside your [raman-micro](https://github.com/ramannkhanna2/raman-micro" \t "_new) repo:

yaml

CopyEdit

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: raman-micro

description: Flask microservice

annotations:

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

spec:

type: service

lifecycle: production

owner: user:raman

✅ This annotation is **required**: github.com/project-slug

Push the updated catalog-info.yaml to GitHub.

**🧩 PART 5: Enable Plugin on the Entity Page**

Edit this file:

bash

CopyEdit

~/backstage-app/raman-app/packages/app/src/components/catalog/EntityPage.tsx

Add this import at the top:

tsx

CopyEdit

import {

EntityGithubActionsContent,

isGithubActionsAvailable,

} from ‘@backstage-community/plugin-github-actions’;

Now add a new tab under your EntityLayout:

const serviceEntityPage = (

<EntityLayout>

{/\* other tabs... \*/}

<EntityLayout.Route path=”/github-actions” title=”GitHub Actions” if={isGithubActionsAvailable}>

<EntityGithubActionsContent view=”cards” />

</EntityLayout.Route>

✅ This will show GitHub Actions as a **tab** in your service view.

**🚀 PART 6: Add Deploy Workflow to Your Repo**

In your [raman-micro](https://github.com/ramannkhanna2/raman-micro" \t "_new) repo, create:

bash

CopyEdit

.github/workflows/deploy.yml

Here’s a basic example:

yaml

CopyEdit

name: Deploy to Dev

on:

workflow\_dispatch: # click-to-deploy trigger

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- name: Checkout Code

uses: actions/checkout@v3

- name: Build Docker Image

run: |

echo “Building Docker Image...”

docker build -t raman-micro:latest .

- name: Deploy (Simulated)

run: |

echo “Simulated deployment...”

✅ The key here is workflow\_dispatch – it **enables manual deploy** from Backstage.

**🧪 PART 7: Run & Test It**

**Start the app:**

bash

CopyEdit

cd ~/backstage-app/raman-app

yarn start

# In another terminal

yarn start-backend

OR

root@ip-172-31-14-172:~/backstage-app/ramanapp# yarn start

Starting app, backend

Loaded config from app-config.yaml

<i> [webpack-dev-server] Project is running a …

**✅ PART 8: Final Test**

1. Go to <http://localhost>:3000
2. Login using GitHub
3. Open your registered service: raman-micro
4. Click the tab: **GitHub Actions**
5. See the Deploy to Dev workflow
6. Click Run Workflow ➜ this will trigger deploy from Backstage (click-to-deploy)

**📌 Summary of What We Did**

| **Step** | **Description** | **Status** |
| --- | --- | --- |
| 🔌 Plugin Installed | Frontend & Backend | ✅ |
| 🔐 GitHub Oauth Setup | Auth + Provider Config | ✅ |
| 🏷️ Service Annotation | With github.com/project-slug | ✅ |
| ⚙️ Workflow Created | With workflow\_dispatch | ✅ |
| 🖼️ UI Integration | GitHub Actions Tab in EntityPage | ✅ |
| 🚀 Click-to-Deploy | Workflow triggers from Backstage | ✅ |

**🧩 Selft Lab Activity: Auto-Start Python App → Auto-Commit & Push → Auto-Trigger**

**🎯 Objective**

Use GitHub Actions to:

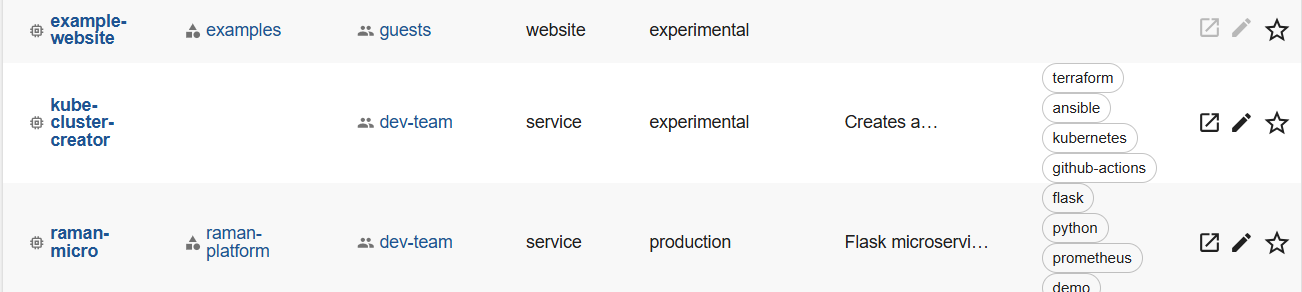
1. Start the Python app (python app/main.py)
2. Automatically stage, commit, and push the updated repo
3. Trigger the same or another CI/CD pipeline from the push

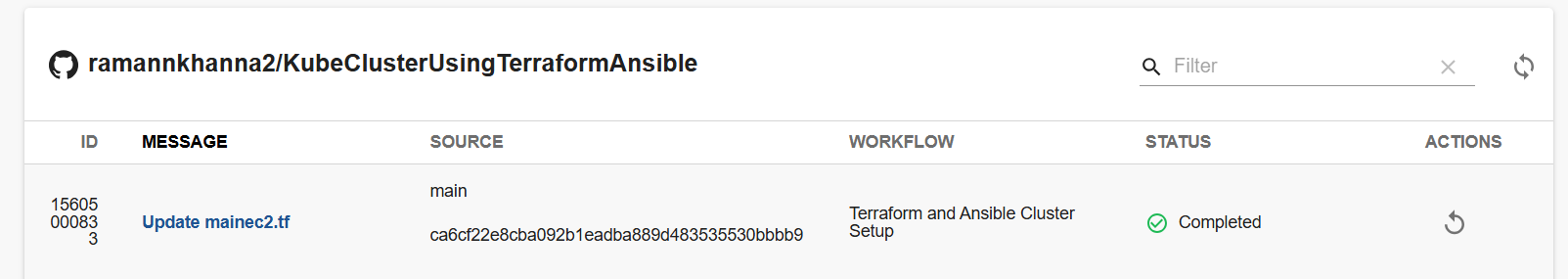
**🧪 What Participants Will Do**

1. Manually trigger the GitHub Actions workflow:  
   ✅ Navigate to **Actions > Auto Run Python App and Push > Run Workflow**
2. The workflow will:
   * Start Python app logic
   * Auto commit and push changes
   * GitHub will trigger ci.yml because of the push

**Lab : Integrate KubeClusterUsingTerraformAnsible into Backstage for One-Click Cluster Creation**

* Already have the setup of kubeadm cluster creation in <https://github.com/ramannkhanna2/KubeClusterUsingTerraformAnsible.git>
* Thers the catalog-info.yaml inside .
* Register it as a component in backstage with below url :
* <https://github.com/ramannkhanna2/KubeClusterUsingTerraformAnsible/blob/main/catalog-info.yaml>
* No u shud see your github actions one click workflow to setup kubeadm 3 node Kubernetes cluster .





**Lab : Adding one click deploy of our ‘terraform-secure-infra’ to backstage.**

* **Upload all content of terraform-secure-infra to remote github repo**

(venv) root@ip-172-31-14-172:~/tf-infra# cat .github/workflows/provision.yml

name: Terraform Infra Provision via Root

on:

workflow\_dispatch:

jobs:

terraform:

runs-on: ubuntu-latest

name: Run Terraform from Remote Server as Root

steps:

- name: Checkout (this repo contains only workflow)

uses: actions/checkout@v3

- name: SSH into Server and Run Terraform as root

uses: appleboy/ssh-action@v1.0.0

with:

host: ${{ secrets.SSH\_HOST }}

username: ${{ secrets.SSH\_USER }} # typically 'ubuntu'

key: ${{ secrets.SSH\_PRIVATE\_KEY }}

script: |

echo "[+] Switching to root user..."

sudo -i <<EOF

echo "[+] Moved to root. Running Terraform..."

cd /root/terraform-secure-infra

terraform init

terraform plan -out=tfplan

terraform apply -auto-approve tfplan

EOF

Add github secrets :

SSH\_HOST :54.39.45.76

SSH\_USER : ubuntu

SSH\_PRIVATE\_KEY :

* Than go to github actions and test.
* Than add destroy automation as well :

(venv) root@ip-172-31-14-172:~/tf-infra# cat .github/workflows/destroy.yml

name: Terraform Infra Destroy

on:

workflow\_dispatch: # Manual trigger only

jobs:

destroy:

runs-on: ubuntu-latest

name: Destroy Infra on Remote via Terraform

steps:

- name: Checkout (optional if needed)

uses: actions/checkout@v3

- name: SSH into server and run Terraform destroy

uses: appleboy/ssh-action@v1.0.0

with:

host: ${{ secrets.SSH\_HOST }}

username: ${{ secrets.SSH\_USER }} # e.g., 'ubuntu'

key: ${{ secrets.SSH\_PRIVATE\_KEY }}

script: |

echo "[+] Switching to root to destroy infra..."

sudo -i <<EOF

cd /root/terraform-secure-infra

terraform init

terraform destroy -auto-approve

EOF

* Test destroy as well
* Now lets add this automation to backstage :

(venv) root@ip-172-31-14-172:~/tf-infra# cat catalog-info.yml

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: tf-infra

description: |

Terraform-based secure infrastructure automation with GitHub Actions.

Provision and destroy infrastructure.

tags:

- terraform

- infrastructure

- github-actions

annotations:

github.com/project-slug: ramannkhanna2/tf-infra

backstage.io/techdocs-ref: dir:.

# Enables GitHub Actions plugin integration

github.com/actions: ramannkhanna2/tf-infra

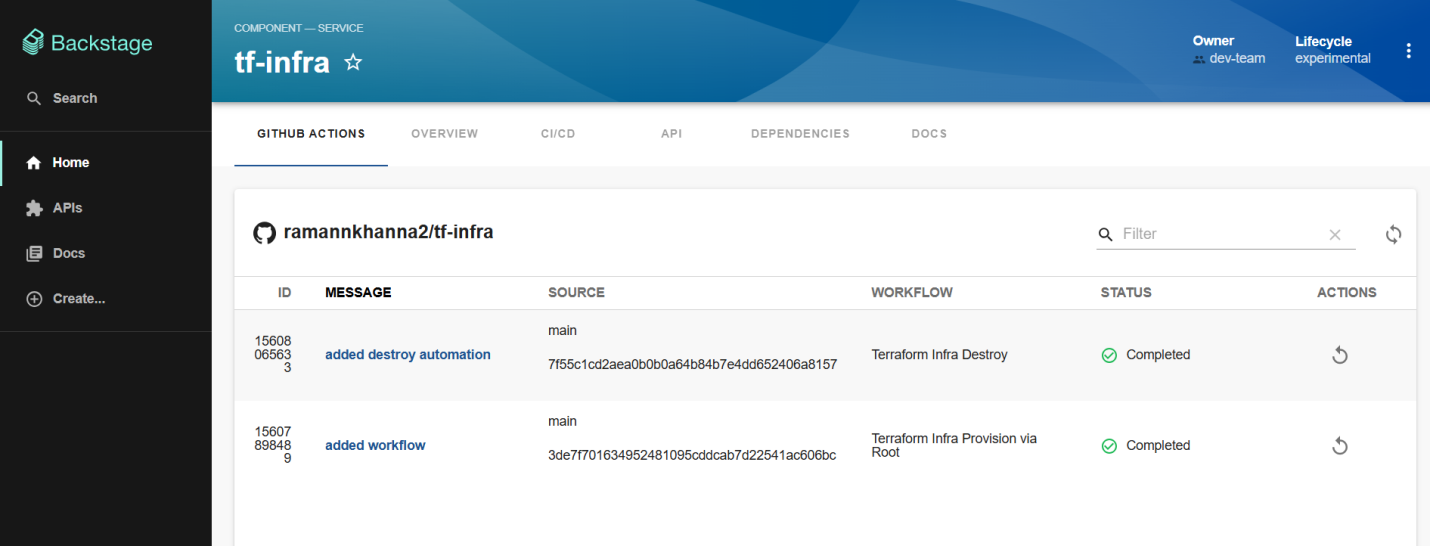
spec:

type: service

lifecycle: experimental

owner: dev-team

* Add above and push to tf-infra and than register to backstage .
* <https://github.com/ramannkhanna2/tf-infra/blob/main/catalog-info.yml>



**Lab 20 : Gatekeeper + GitOps(argoCD) + Backstage Integration**

**📦 Pre-Requirements**

| **Component** | **Status** | **Notes** |
| --- | --- | --- |
| 🧠 Kubeadm cluster | ✅ Ready | Already provisioned by you |
| 🧪 Helm | ✅ Installed | helm version |
| 🔐 OPA Gatekeeper | ✅ We'll install it | Policy enforcement |
| 📁 ArgoCD | ⏳ To be installed | GitOps sync engine |
| 🐙 GitHub Repo | ✅ You have it | Your raman-micro or Helm app |
| 🎛️ Backstage | ✅ Running | We’ll add ArgoCD plugin |

**✅ Phase 1: Enforce Gatekeeper Policies (Cluster Level)**

* Increase capacity of all nodes to 2 vcpus , 4 gb ram atlease for all nodes

**🧱 Step 1.1: Install Gatekeeper**

kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.12/deploy/gatekeeper.yaml

Verify:

bash

CopyEdit

kubectl get pods -n gatekeeper-system

Wait for all pods (controller, audit) to be Running.

* Create a github repo named kube+gatekeeper+argo and clone it to add all files in it

**🚫 Step 1.2: Create ConstraintTemplate: Block Untrusted Image Registry**

root@ip-172-31-27-88:~/kube-gatekeeper-argo# cat template-image-policy.yaml

apiVersion: templates.gatekeeper.sh/v1beta1

kind: ConstraintTemplate

metadata:

name: k8sallowedrepos

spec:

crd:

spec:

names:

kind: K8sAllowedRepos

targets:

- target: admission.k8s.gatekeeper.sh

rego: |

package k8sallowedrepos

violation[{"msg": msg}] {

container := input.review.object.spec.containers[\_]

not startswith(container.image, "docker.io/raman")

msg := sprintf("container image '%v' is not from allowed registry", [container.image])

}

Apply:

bash

CopyEdit

kubectl apply -f template-image-policy.yaml

|  |  |
| --- | --- |
| template-image-policy.yaml | Defines *how* to validate container images |

|  |  |
| --- | --- |
| constraint-image-policy.yaml | Activates enforcement for real objects like Pods |

**⚙️ Step 1.3: Apply Constraint: Only Allow docker.io/raman**

# constraint-image-policy.yaml

apiVersion: constraints.gatekeeper.sh/v1beta1

kind: K8sAllowedRepos

metadata:

name: allowed-docker-raman

spec:

match:

kinds:

- apiGroups: [""]

kinds: ["Pod"]

parameters: {}

bash

CopyEdit

kubectl apply -f constraint-image-policy.yaml

**🔐 You now have:**

**✅ A real-time Kubernetes admission control policy:**

❌ Blocks **all Pods** that do **not** use images from docker.io/raman

**💡 Step 1.4: Add Another Constraint — Require CPU & Memory Limits**

# template-limits-required.yaml

apiVersion: templates.gatekeeper.sh/v1beta1

kind: ConstraintTemplate

metadata:

name: k8srequiredresources

spec:

crd:

spec:

names:

kind: K8sRequiredResources

targets:

- target: admission.k8s.gatekeeper.sh

rego: |

package k8srequiredresources

violation[{"msg": msg}] {

container := input.review.object.spec.containers[\_]

not container.resources.limits.cpu

msg := "Missing CPU limit"

}

violation[{"msg": msg}] {

container := input.review.object.spec.containers[\_]

not container.resources.limits.memory

msg := "Missing memory limit"

}

bash

CopyEdit

kubectl apply -f template-limits-required.yaml

You’ve now successfully created a **second Gatekeeper ConstraintTemplate**:

bash

CopyEdit

✅ constrainttemplate.templates.gatekeeper.sh/k8srequiredresources created

This policy will enforce that **all Pods must define both cpu and memory limits**, which is a **crucial best practice** for Kubernetes multi-tenancy and resource fairness.

Constraint:

yaml

CopyEdit

# constraint-limits.yaml

apiVersion: constraints.gatekeeper.sh/v1beta1

kind: K8sRequiredResources

metadata:

name: require-cpu-memory-limits

spec:

match:

kinds:

- apiGroups: [""]

kinds: ["Pod"]

bash

CopyEdit

kubectl apply -f constraint-limits.yaml

✅ This means your cluster will now **block any Pod that doesn’t define both cpu and memory limits** for every container.

**🧪 Optional Test: Validate Enforcement**

Here’s a **negative test** to confirm Gatekeeper is enforcing the rule:

yaml

CopyEdit

# no-limits.yaml

apiVersion: v1

kind: Pod

metadata:

name: no-limits

spec:

containers:

- name: busybox

image: docker.io/raman/busybox:latest

command: ["sleep", "3600"]

Try to apply:

bash

CopyEdit

kubectl apply -f no-limits.yaml

You should see:

pgsql

CopyEdit

Error from server (Forbidden): admission webhook "validation.gatekeeper.sh" denied the request: [Missing CPU limit, Missing memory limit]

**✅ Current Setup Already Ensures**

| **Layer** | **Tool** | **Enforces Policy?** |
| --- | --- | --- |
| Kubernetes | OPA Gatekeeper | ✅ Denies at admission |

Your cluster will now:

* ❌ Reject any pod without resources.limits
* ❌ Reject any image not starting with docker.io/raman

Which means:

**No manifest can ever be deployed to your cluster if it violates the rules.**

**✅ Goal: GitOps + Backstage Integration for raman-micro**

We’ll now:

1. 🔁 Use **ArgoCD** to GitOps-deploy your k8s/\*.yaml to the cluster
2. 🔍 Hook that into **Backstage** so devs can visually see:
   * Sync status
   * Health
   * Drift

**🚀 Phase 3: GitOps with ArgoCD**

**🔧 Prerequisites Recap**

| **Requirement** | **Status** |
| --- | --- |
| ArgoCD installed (namespace argocd) | ✅ TobeDone |
| raman-micro GitHub repo | ✅ Exists |
| Manifests under k8s/ | ✅ TobeDone |
| Gatekeeper enforcing image + limits | ✅ Active |

Let's create a proper k8s/ directory in your raman-micro GitHub repo with fully compliant Kubernetes manifests that:

* ✅ Use docker.io/raman/raman-micro image
* ✅ Include CPU & memory limits
* ✅ Pass Gatekeeper constraints
* ✅ Are deployable via ArgoCD

**📁 Folder Structure to Add to Your Repo**

markdown

CopyEdit

raman-micro/

└── k8s/

├── deployment.yaml

└── service.yaml

**📄 k8s/deployment.yaml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-micro

labels:

app: raman-micro

spec:

replicas: 1

selector:

matchLabels:

app: raman-micro

template:

metadata:

labels:

app: raman-micro

spec:

containers:

- name: raman-micro

image: docker.io/raman/raman-micro:latest

ports:

- containerPort: 5000

resources:

limits:

cpu: "250m"

memory: "256Mi"

**📄 k8s/service.yaml**

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-micro

spec:

selector:

app: raman-micro

ports:

- protocol: TCP

port: 80

targetPort: 5000

type: NodePort

**✅ What to Do Next**

**🔧 Step 1: Add These Files to Your GitHub Repo raman-micro**

git add .

git commit -m "Add k8s manifests for ArgoCD sync"

git push origin main

* **Install argocd CLI**

Run the following on your Ubuntu (or Amazon Linux) system:

bash

CopyEdit

# Download latest ArgoCD CLI (v2.11.3 as of June 2025)

VERSION=$(curl -s https://api.github.com/repos/argoproj/argo-cd/releases/latest | grep tag\_name | cut -d '"' -f 4)

curl -sSL -o argocd "https://github.com/argoproj/argo-cd/releases/download/${VERSION}/argocd-linux-amd64"

chmod +x argocd

sudo mv argocd /usr/local/bin/

Verify install:

bash

CopyEdit

argocd version

**✅ Yes, the ArgoCD UI runs as a pod in your Kubernetes cluster.**

When you installed ArgoCD using:

kubectl create namespace argocd

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

This deployed all ArgoCD components — including the **UI (argocd-server)** — into your cluster.

**📦 ArgoCD Core Components (All Run as Pods)**

You can check with:

bash

CopyEdit

kubectl get pods -n argocd

* You will notice pods will not get created due to the gatekeeper policy

❌ Problem: Gatekeeper is Blocking ArgoCD Pods

**✅ Disable the policy temporarily ; will activate again after argo cd installation.**

Only if you're still testing.

bash

CopyEdit

Kubectl get constraint

kubectl delete k8srequiredresources.constraints.gatekeeper.sh require-cpu-memory-limits

kubectl delete k8sallowedrepos.constraints.gatekeeper.sh allowed-docker-raman

* Again install argocd :

kubectl delete -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

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

* Kubectl get pods -n argocd
* Now again enable gatekeeper constraints :

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k apply -f constraint-image-policy.yaml

k8sallowedrepos.constraints.gatekeeper.sh/allowed-docker-raman created

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k apply -f constraint-limit.yaml

k8srequiredresources.constraints.gatekeeper.sh/require-cpu-memory-limits created

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k get constraints

NAME ENFORCEMENT-ACTION TOTAL-VIOLATIONS

k8sallowedrepos.constraints.gatekeeper.sh/allowed-docker-raman

NAME ENFORCEMENT-ACTION TOTAL-VIOLATIONS

k8srequiredresources.constraints.gatekeeper.sh/require-cpu-memory-limits

* To expose argocd :

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k edit -n argocd svc argocd-server

service/argocd-server edited

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k get svc -n argocd argocd-server

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

argocd-server NodePort 10.99.164.110 <none> 80:30786/TCP,443:31194/TCP 4m51s

Let’s now help you **log into the UI** — ArgoCD uses:

* **Username:** admin
* **Password:** stored in a Kubernetes **secret** (base64-encoded)

**✅ Step-by-Step: Get ArgoCD Admin Credentials**

**Get the admin password :**

kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d && echo

* **Login argocd using cli as well :**

root@ip-172-31-27-88:~/kube-gatekeeper-argo# argocd login 15.228.172.48:30786 --username admin --password V9kGvXX0SH1Fw7Fn --insecure

'admin:login' logged in successfully

Context '15.228.172.48:30786' updated

**📦 Step 3.1: Create ArgoCD App for raman-micro**

Replace YOUR\_GITHUB\_URL if needed.

bash

CopyEdit

argocd app create raman-micro \

--repo https://github.com/ramannkhanna2/raman-micro.git \

--path k8s \

--dest-server https://kubernetes.default.svc \

--dest-namespace default \

--sync-policy automated \

--self-heal \

--auto-prune

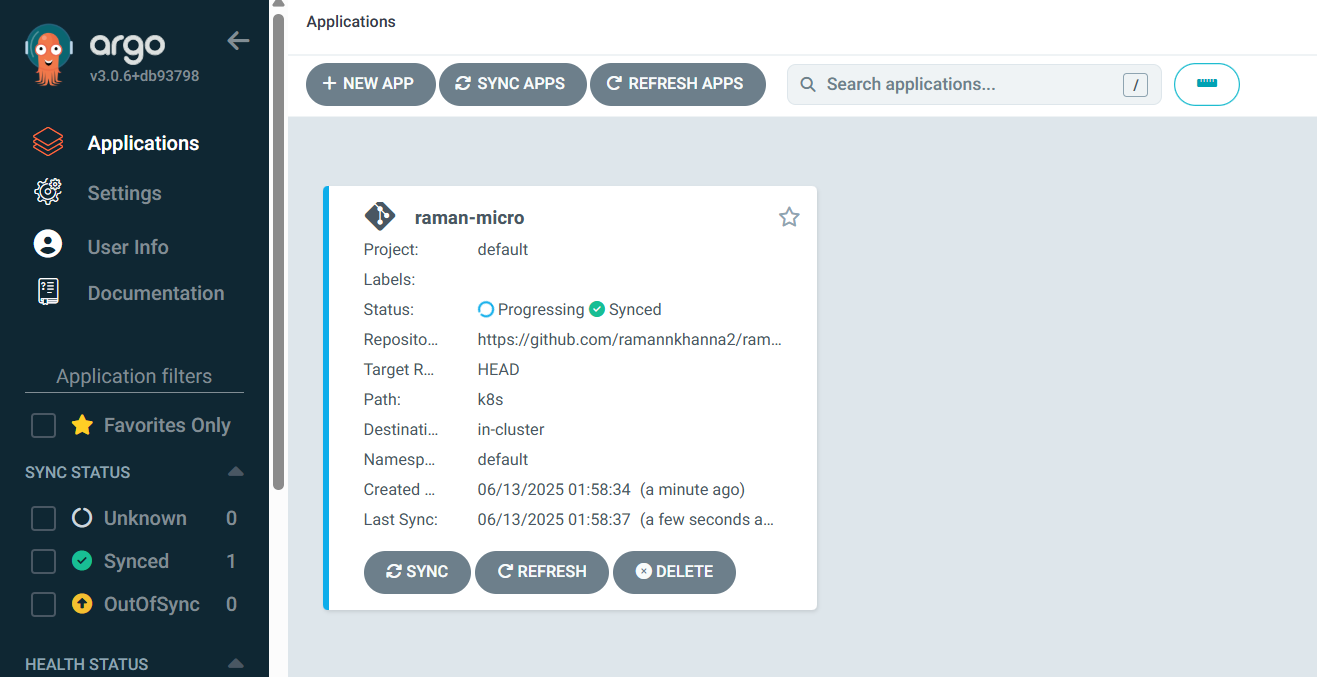
Explanation:

* --path k8s: we’re using plain manifests, not Helm
* --sync-policy automated: auto-deploys when Git changes
* --self-heal: ensures drifted resources are reconciled
* --auto-prune: deletes removed resources from Git

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k get application -n argocd

NAME SYNC STATUS HEALTH STATUS

raman-micro Synced Progressing



**🔄 Step 3.2: Trigger a Sync (optional)**

bash

CopyEdit

argocd app sync raman-micro

* **Check if argo cd setup is working fine by making changes on raman-micro/k8s github repo and se eif its automatically replicating on the kubecluster using argi cd ..**
* **If fails check for the right image from dockerhub and update in raman-micro github repo i.e : ramann123/raman-micro:latest**

**🧭 Phase 4: Backstage + ArgoCD Plugin Integration**

* <https://backstage.io/plugins/>
* https://roadie.io/backstage/plugins/argo-cd/?utm\_source=backstage.io&utm\_medium=marketplace&utm\_campaign=argo-cd

Install the plugin into Backstage.

cd /root/backstage-app/ramanapp/packages/app

yarn add @roadiehq/backstage-plugin-argo-cd

Add proxy config to the app-config.yaml file

proxy:

'/argocd/api':

target: https://<your-argocd-instance>/api/v1/

changeOrigin: **true**

# only if your argocd api has self-signed cert

secure: **false**

headers:

Cookie:

$env: ARGOCD\_AUTH\_TOKEN

For above generate token and save as env :

bash

CopyEdit

argocd account generate-token

Add argoCD widget to your overview page

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

import {

EntityArgoCDOverviewCard,

isArgocdAvailable

} from '@roadiehq/backstage-plugin-argo-cd';

const overviewContent = (

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

...

<EntitySwitch>

<EntitySwitch.Case if={e => Boolean(isArgocdAvailable(e))}>

<Grid item sm={4}>

<EntityArgoCDOverviewCard />

</Grid>

</EntitySwitch.Case>

</EntitySwitch>

...

</Grid>

);

Add annotation to the yaml config file of a component

metadata:

annotations:

argocd/app-name: <your-app-name>

Get and provide **ARGOCD\_AUTH\_TOKEN** as env variable in following format

ARGOCD\_AUTH\_TOKEN='argocd.token=<token>'

**🔄 Step 4.5: Restart Backstage**

bash

CopyEdit

cd ~/backstage-app/raman-app

yarn start

Go to:  
🧭 Backstage → raman-micro → ArgoCD tab

✅ You should now see:

* Sync Status
* Drift
* Health
* Last sync timestamp
* Commit SHA

**🎉 You Now Have:**

| **Capability** | **Tool** | **Status** |
| --- | --- | --- |
| GitOps deployment | ArgoCD | ✅ |
| Admission policy enforcement | Gatekeeper | ✅ |
| Visual deployment status | Backstage | ✅ |
| Real-time sync & drift detection | ArgoCD UI | ✅ |