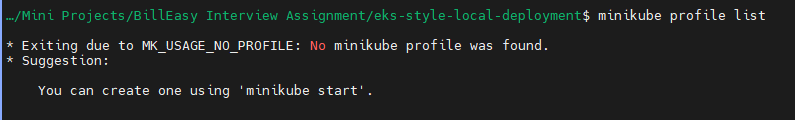
0. Installation and Prerequisites:

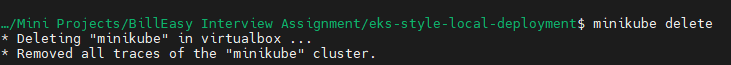
Check for existing minikube cluser:

* minikube profile list



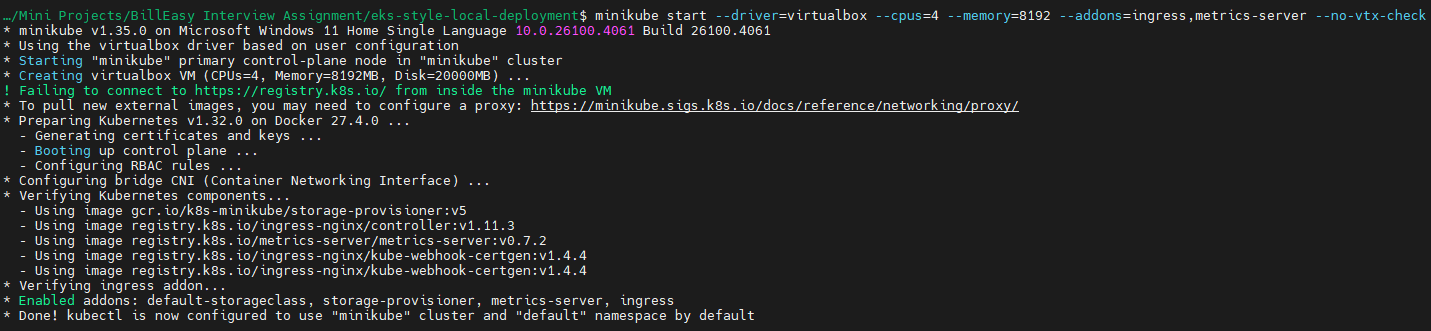
If exists, delete it

* minikube delete



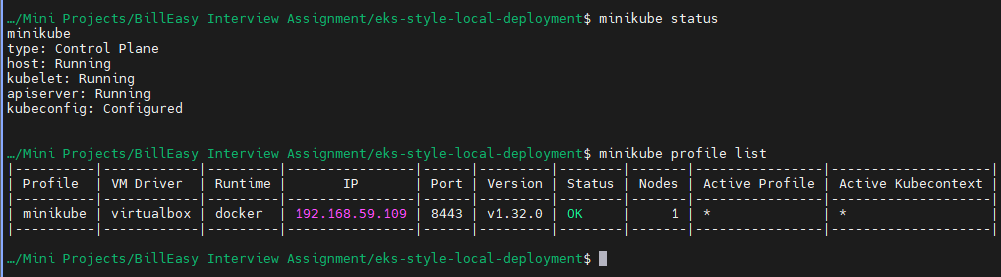
Start your minikube cluster:

* minikube start --driver=virtualbox --cpus=4 --memory=8192 --addons=ingress,metrics-server --no-vtx-check



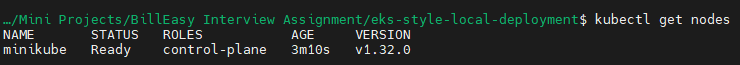
Check the status of minikube cluster:

* minikube status
* minikube profile list

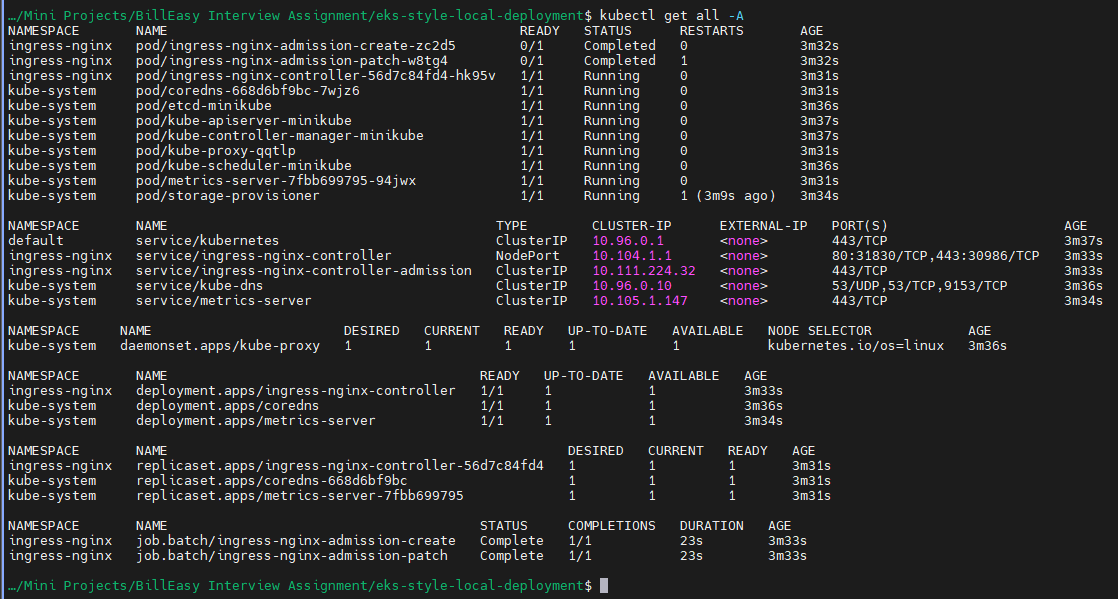


Ensure that your system is ready:

* kubectl get nodes

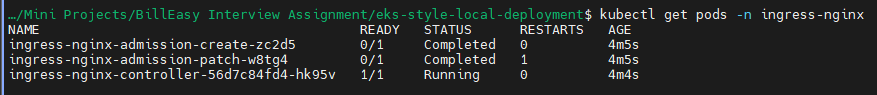


* kubectl get all -A



Verify your ingress:

* kubectl get pods -n ingress-nginx

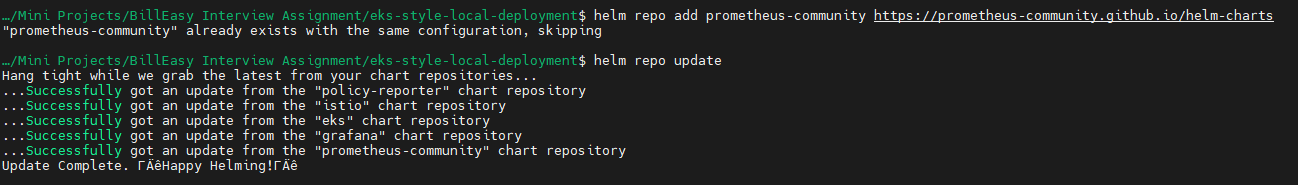


Now we setup Prom and Grafana in a new namespace monitoring by referring: <https://medium.com/@joudwawad/comprehensive-beginners-guide-to-kube-prometheus-in-kubernetes-monitoring-alerts-integration-4ade4fa8fa8c> :

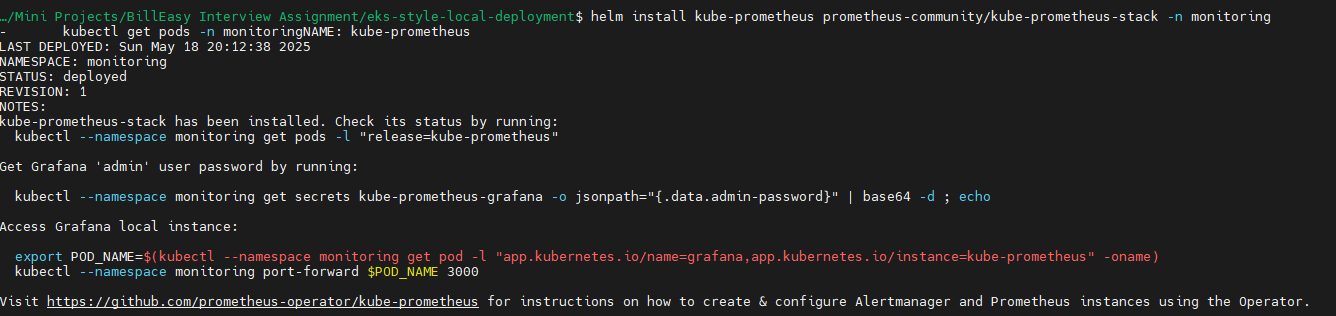
* kubectl create namespace monitoring



* helm repo add prometheus-community <https://prometheus-community.github.io/helm-charts>
* helm repo update

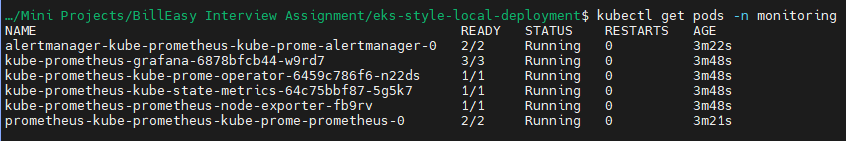


* helm install kube-prometheus prometheus-community/kube-prometheus-stack -n monitoring



Wait for the pods to be in running stage:

* kubectl get pods -n monitoring

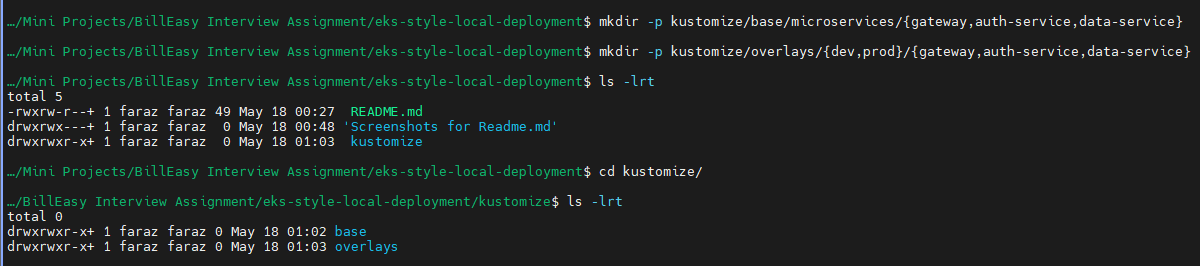


1. Microservice Stack

Refer: <https://subbaramireddyk.medium.com/kustomize-kubernetes-native-configuration-management-f51630d29ac0>

Creating kustomize project structure:

* mkdir -p kustomize/base/microservices/{gateway,auth-service,data-service}
* mkdir -p kustomize/overlays/{dev,prod}/{gateway,auth-service,data-service}

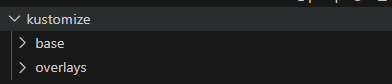


Create kustomization.yaml files:

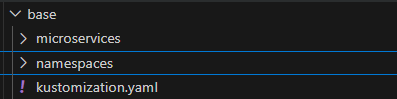


Create rest of the project structure

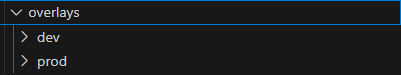
Final project structure:



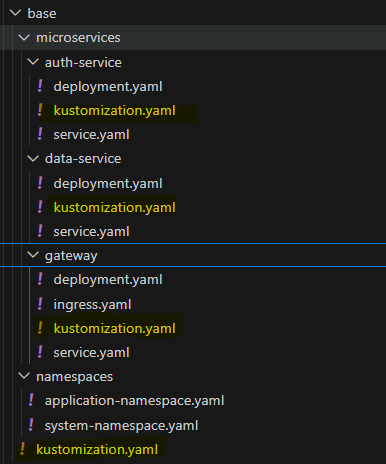
Base:



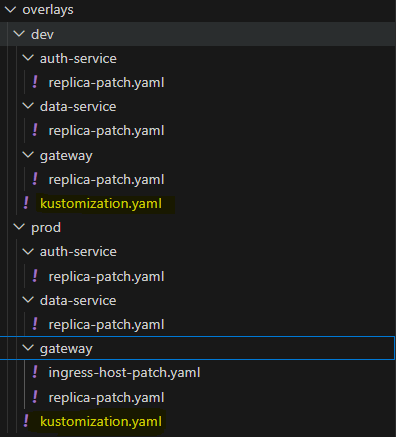
Overlays:



Base project folders:



Overlays project folders:

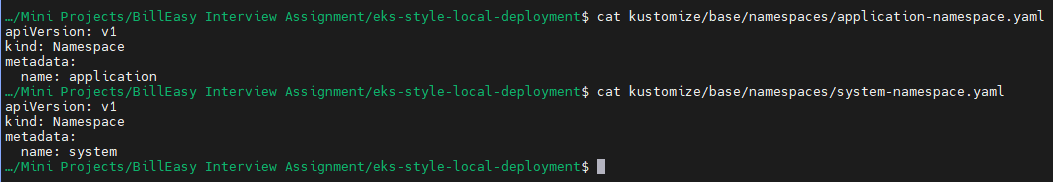


Create namespaces application and system:

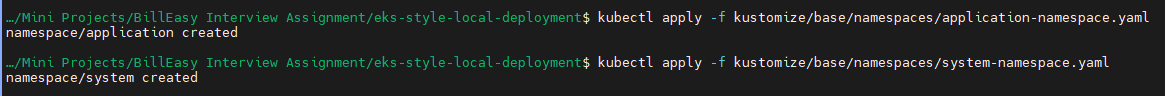
Check the existing namespaces:



Create yaml to create namespaces:



Apply to create namespace:



Verify your namespace creation:



\*\* Assuming that gateway will belong to the application namespace since it needs to be publicly accessible via Ingress, while auth-service and data-service should reside in the system namespace to keep them internal-only.

Now we will deploy the following services:

|  |  |  |
| --- | --- | --- |
| **Service** | **Purpose** | **Docker Image** |
| gateway | API gateway, exposed via ingress | nginxdemos/hello |
| auth-service | Auth logic, logs headers | kennethreitz/httpbin |
| data-service | Mock business logic | hashicorp/http-echo |

Create yaml manifests of gateway, auth-service and data-service with the mentioned requirements

and apply it

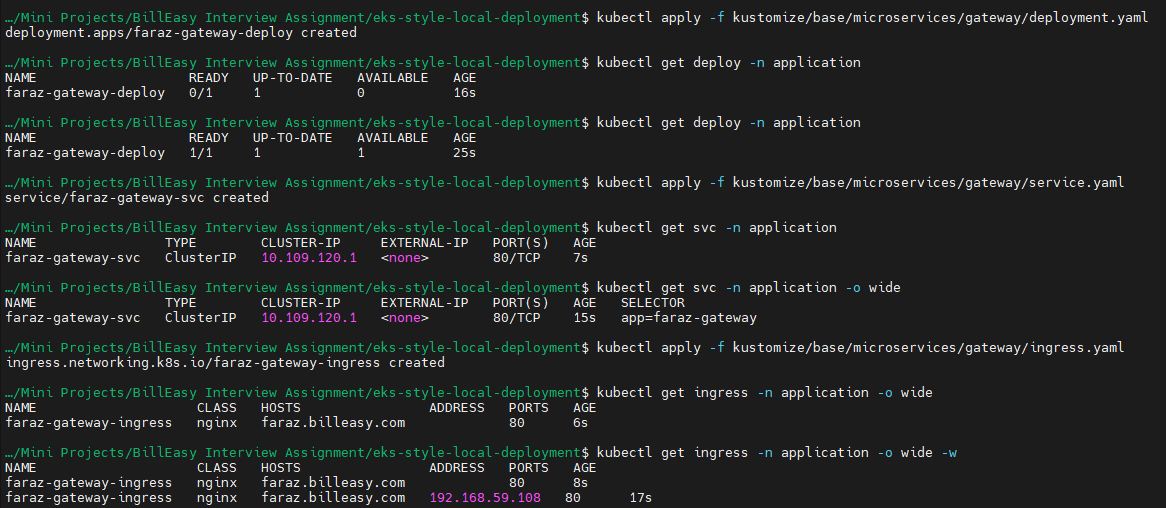
Requirements:

* Containerized deployment using Helm or Kustomize
* Liveness/readiness probes
* Proper resource requests/limits
* gateway should be publicly accessible via ingress
* auth-service and data-service should be internal-only
* Use separate namespaces for system and application

Applying and manually testing if every manifest created is working as expected:

Manually Testing Gateway manifests:

* + kubectl apply -f kustomize/base/microservices/gateway/deployment.yaml
  + kubectl get deploy -n application
  + kubectl apply -f kustomize/base/microservices/gateway/service.yaml
  + kubectl get svc -n application -o wide
  + kubectl apply -f kustomize/base/microservices/gateway/ingress.yaml
  + kubectl get ingress -n application -o wide

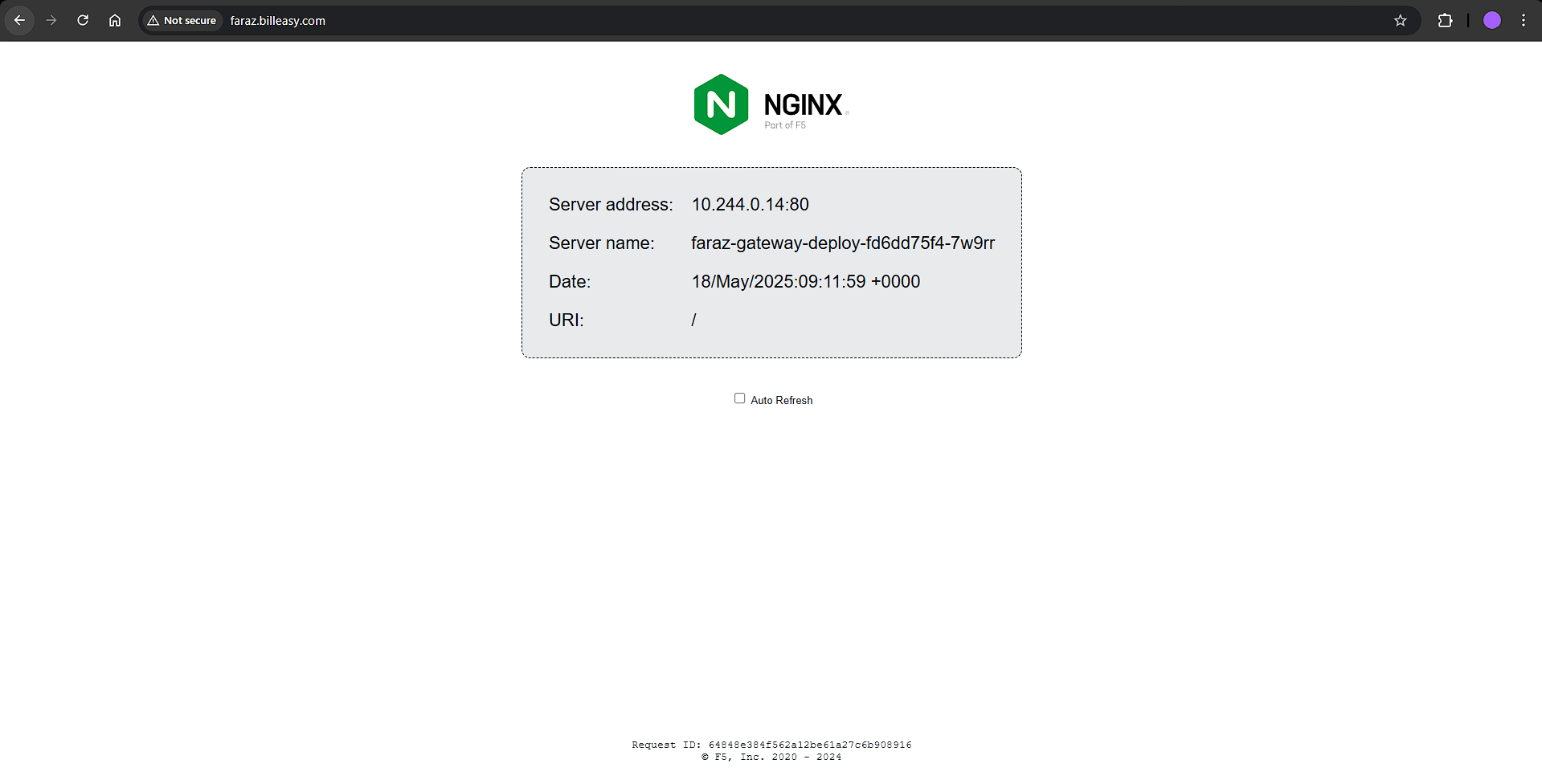


Also add the ingress ip to /etc/hosts

* sudo vim /etc/hosts

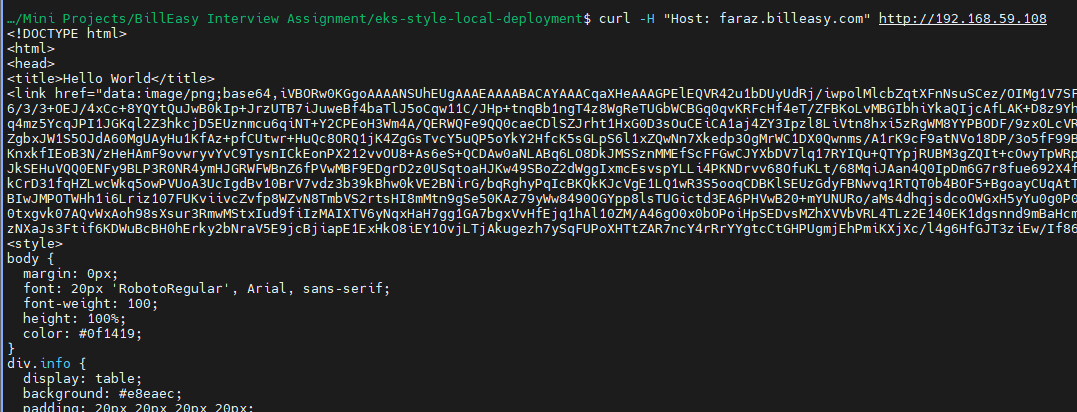


You must now be able to access the application from your browser:



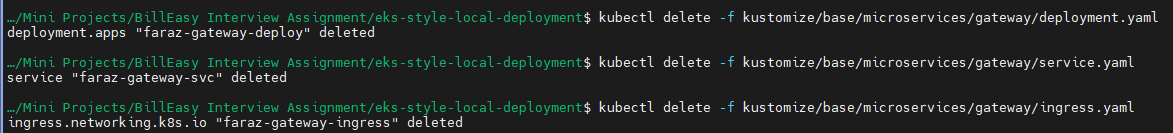
You can also test it using curl:

* curl -H "Host: faraz.billeasy.com" http://192.168.59.108



After testing, delete the created services:

* kubectl delete -f kustomize/base/microservices/gateway/deployment.yaml
* kubectl delete -f kustomize/base/microservices/gateway/service.yaml
* kubectl delete -f kustomize/base/microservices/gateway/ingress.yaml

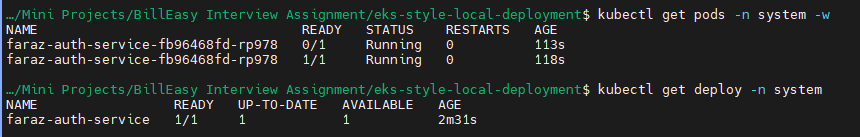


Testing auth-service:

Apply the deployment:

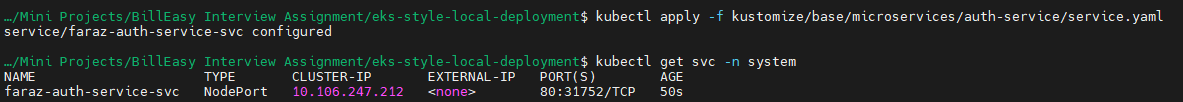
* kubectl apply -f kustomize/base/microservices/auth-service/deployment.yaml
* kubectl get pods -n system
* kubectl get deploy -n system





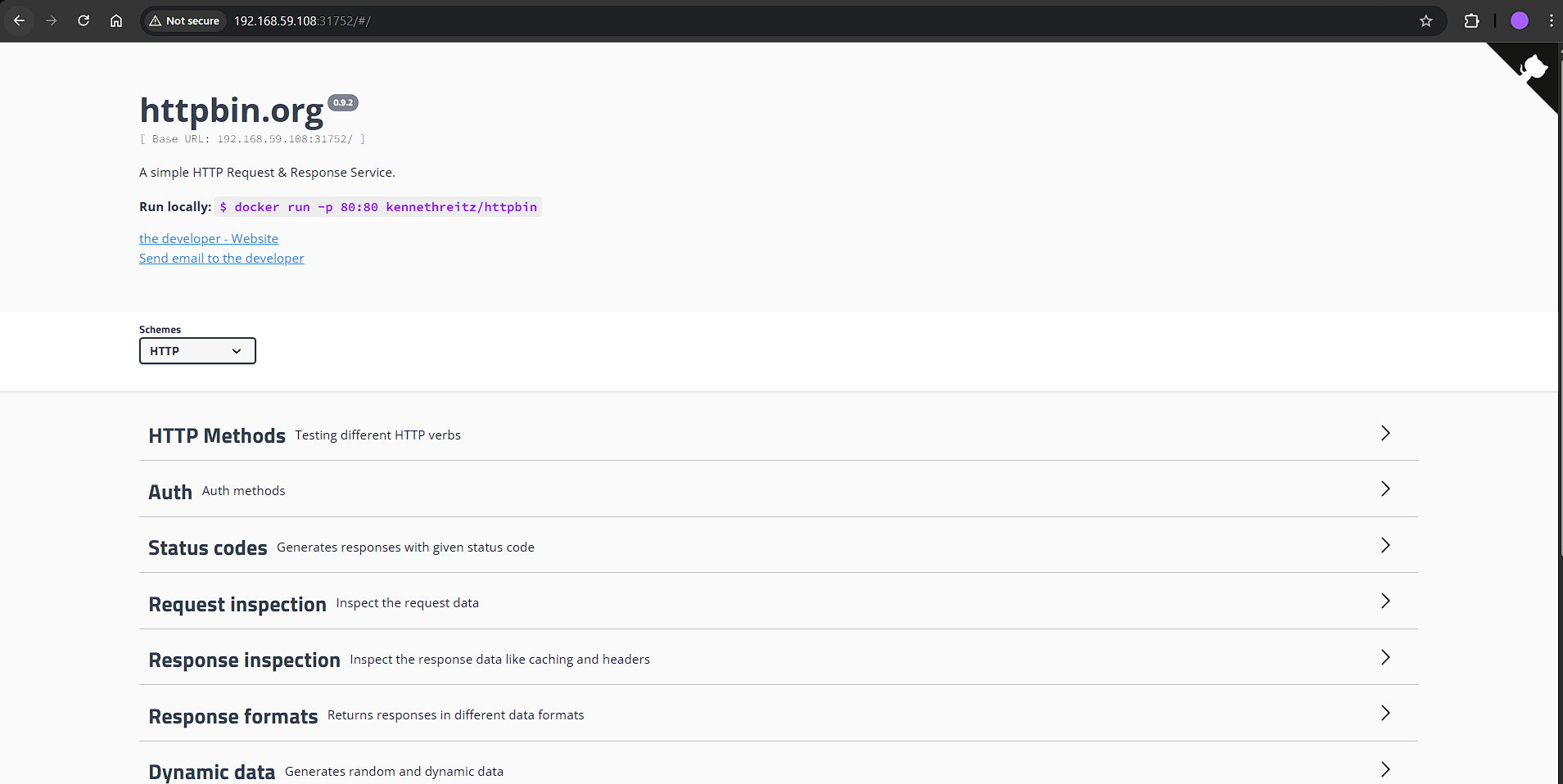
For testing we will apply the service in NodePort mode:

* kubectl apply -f kustomize/base/microservices/auth-service/service.yaml
* kubectl get svc -n system



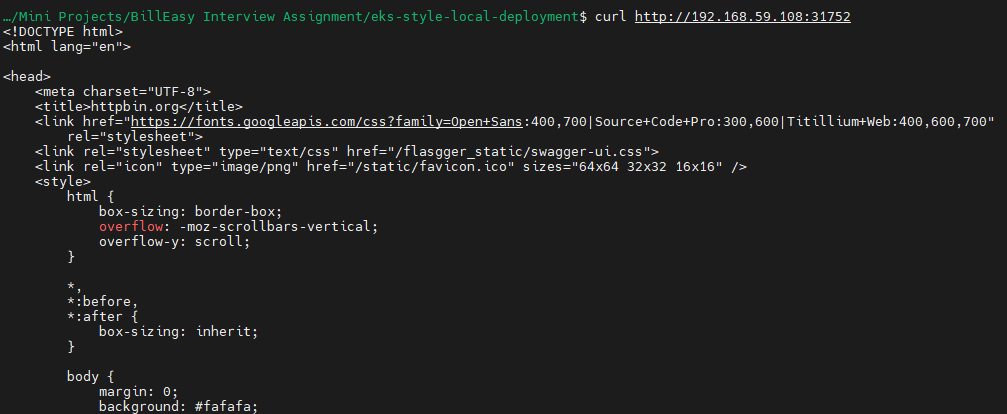
By getting the minikube IP and the Node port:

Verify if it is working:



You can also use curl:

* curl http://192.168.59.108:31752



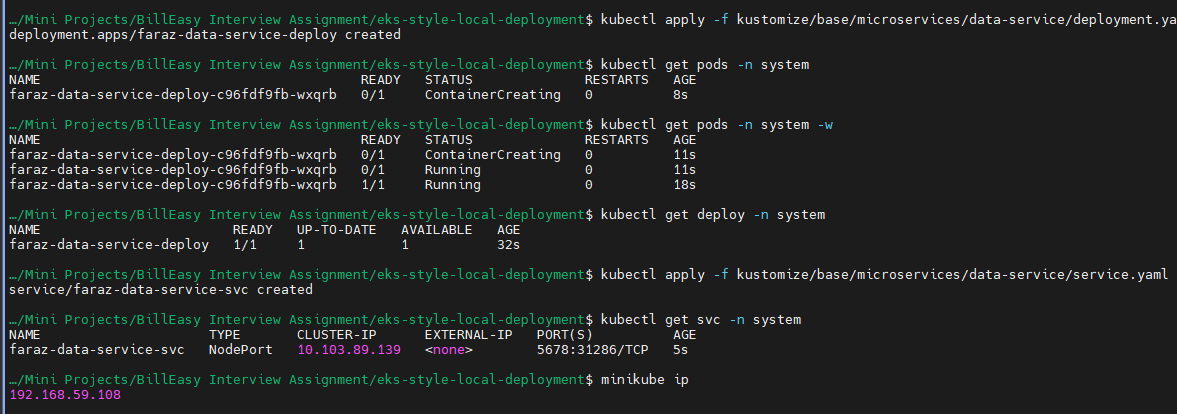
After testing deleting the deployment and service:

* kubectl delete -f kustomize/base/microservices/auth-service/service.yaml
* kubectl delete -f kustomize/base/microservices/auth-service/deployment.yaml



Similarly test data-service:

* kubectl apply -f kustomize/base/microservices/data-service/deployment.yaml
* kubectl get pods -n system
* kubectl get deploy -n system
* kubectl apply -f kustomize/base/microservices/data-service/service.yaml
* kubectl get svc -n system

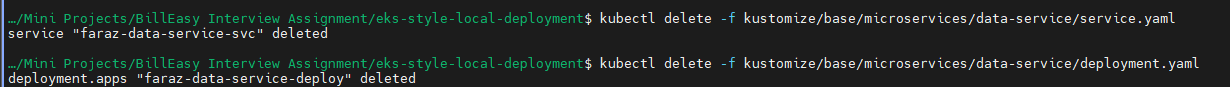


Open it in your browser:



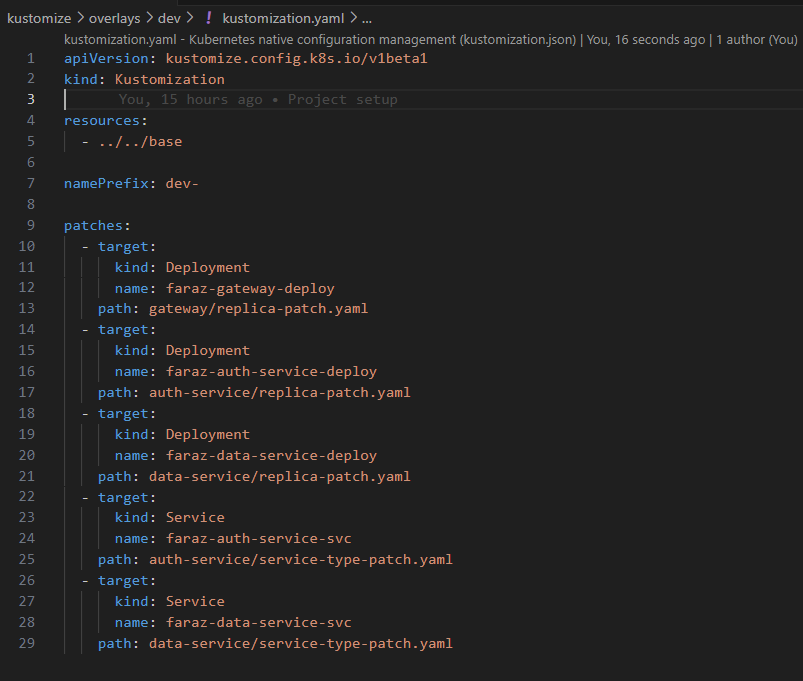
Delete the service and deployment:

* kubectl delete -f kustomize/base/microservices/data-service/service.yaml
* kubectl delete -f kustomize/base/microservices/data-service/deployment.yaml



Now we will use kustomize to test:

Create your kustomization.yaml and create your patches:

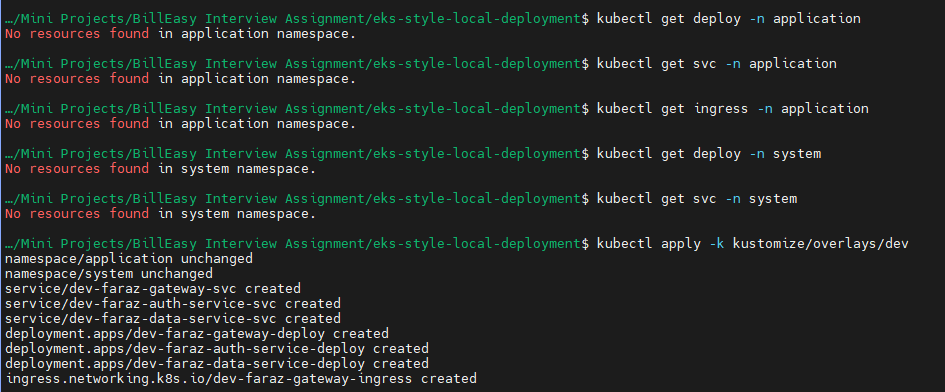


Verify your kustomize using:

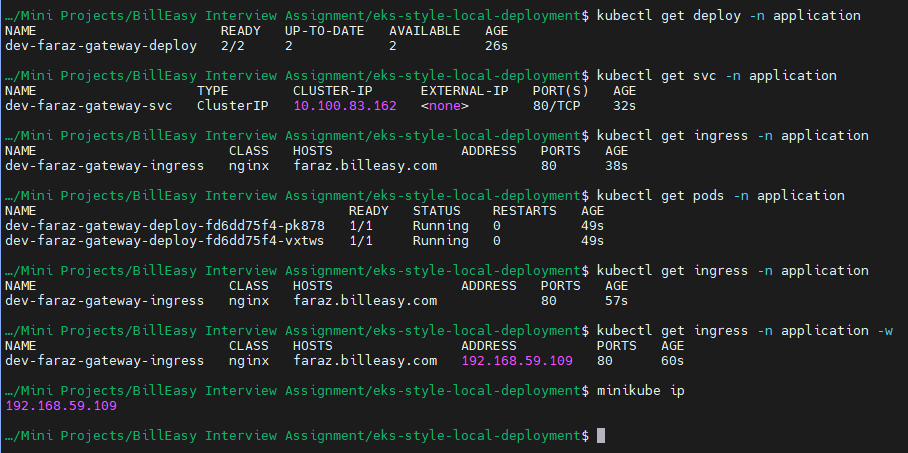
* kubectl kustomize kustomize/overlays/dev/
* kubectl kustomize kustomize/overlays/prod/

Now apply the kustomization:

* kubectl apply -k kustomize/overlays/dev

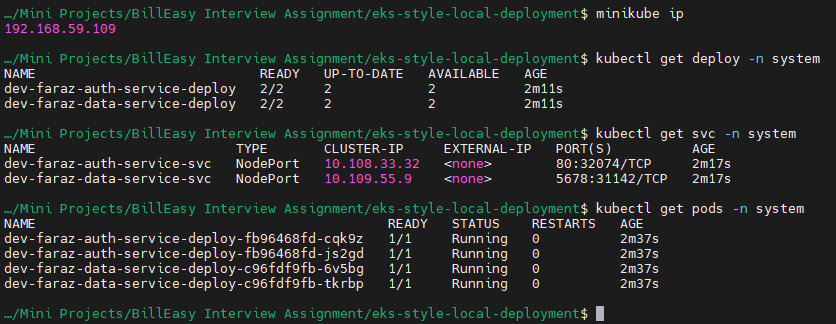


Verify your service, deployment and ingress(only for gateway):

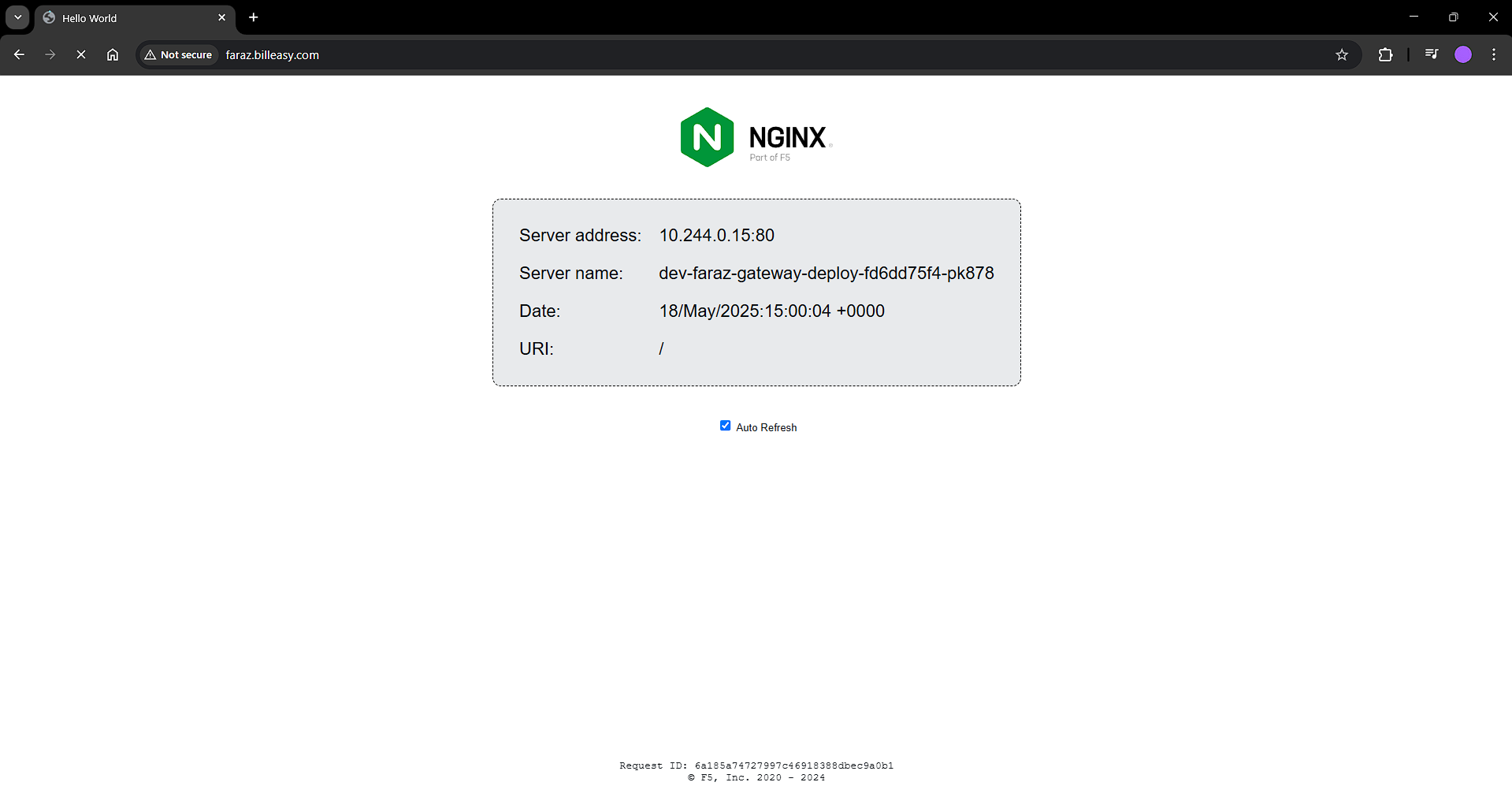


Ensure that you’re mentioning the IP in /etc/hosts

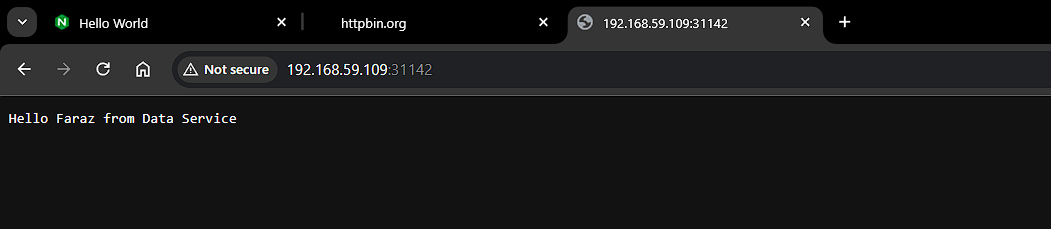
* sudo vim /etc/hosts
* 



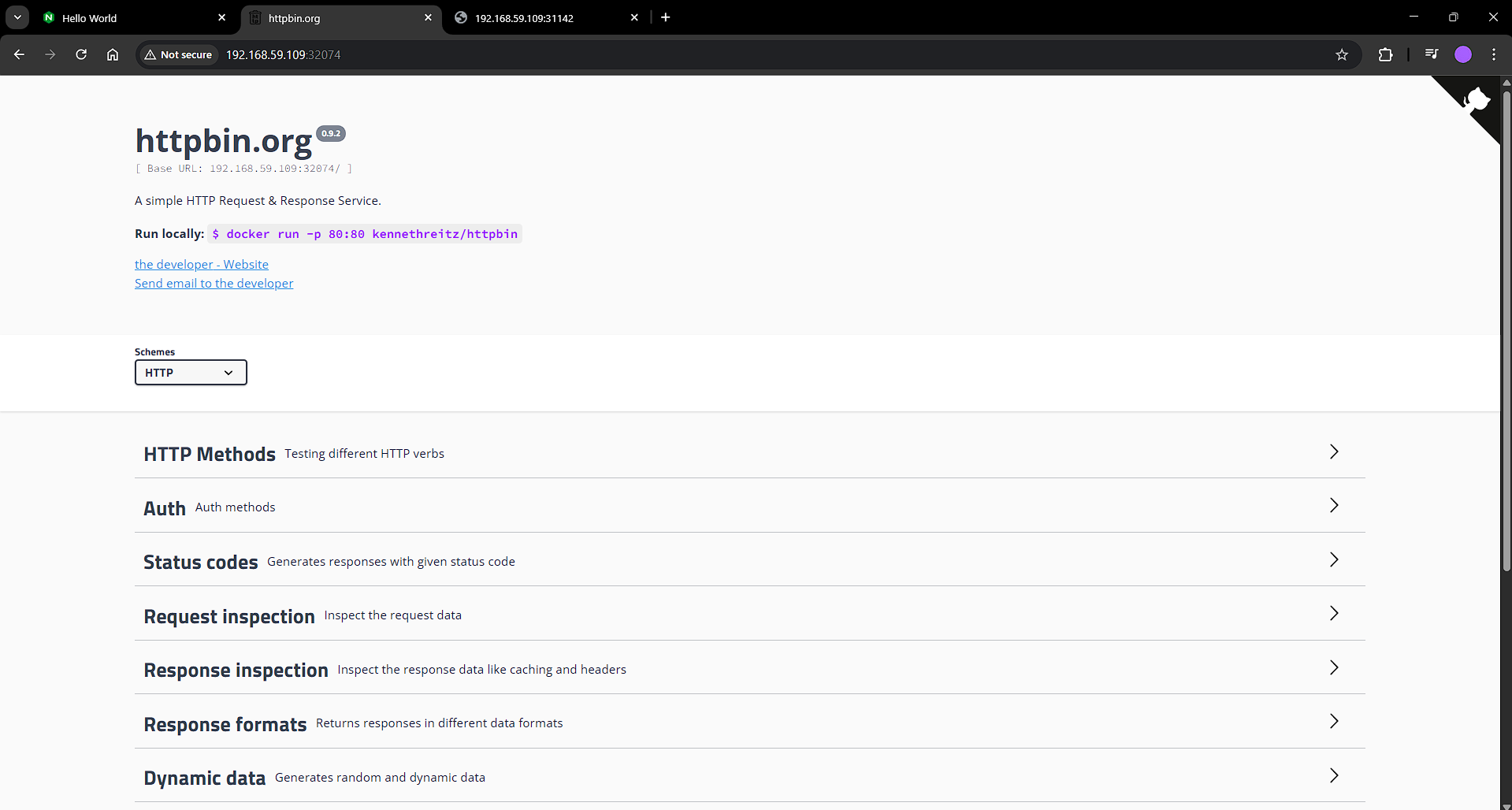
Verify your gateway ingress, svc, deploy on browser:



Verify your data service:

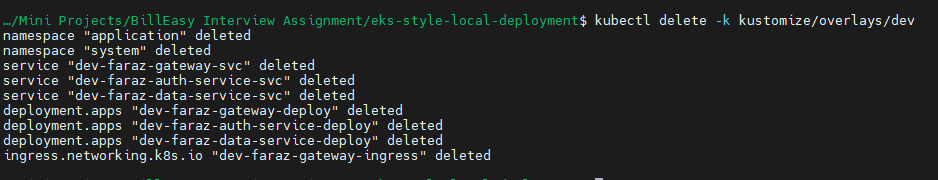


Verify your auth-service:



Deleting the resources:

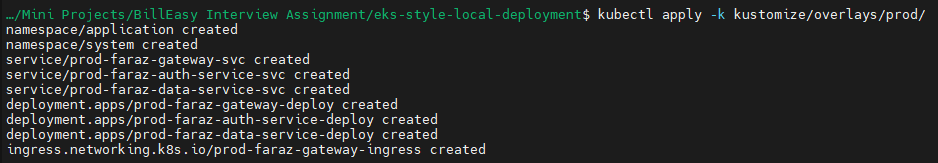
* kubectl delete -k kustomize/overlays/dev



Now similarly, we will verify in prod:

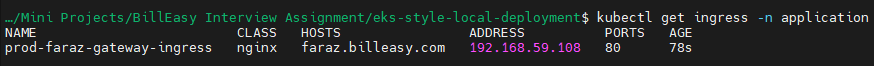
Apply the kustomization:

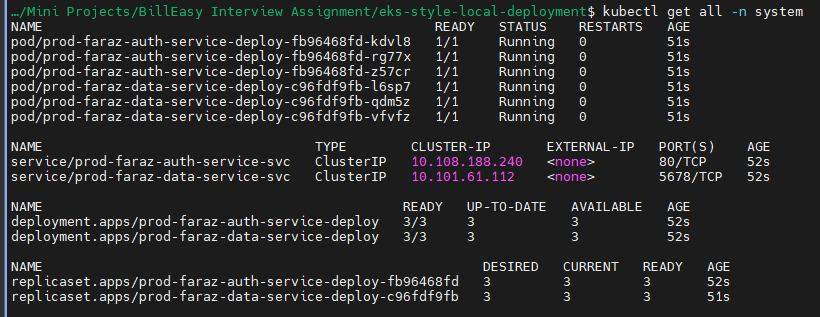
* kubectl apply -k kustomize/overlays/prod/



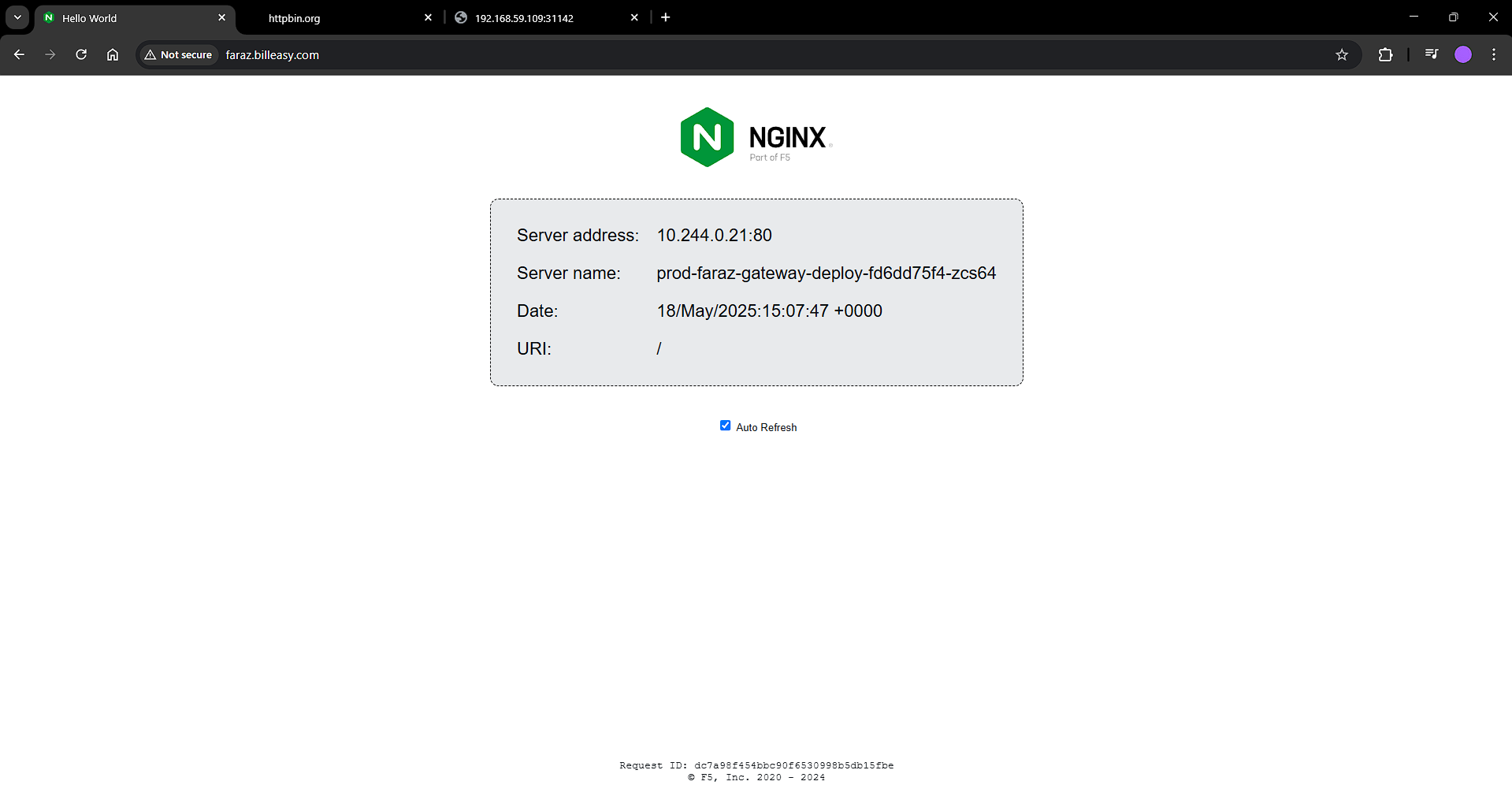
Verify your ingress, svc, deploy, pods:





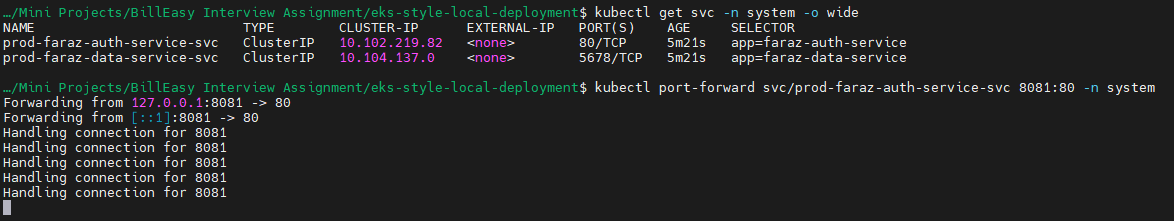


Visible in browser as ingress interacting with service with clusterIP mode and to prod-faraz-gateway-deploy deployment:



Using port forwarding to verify auth-service and data-service:

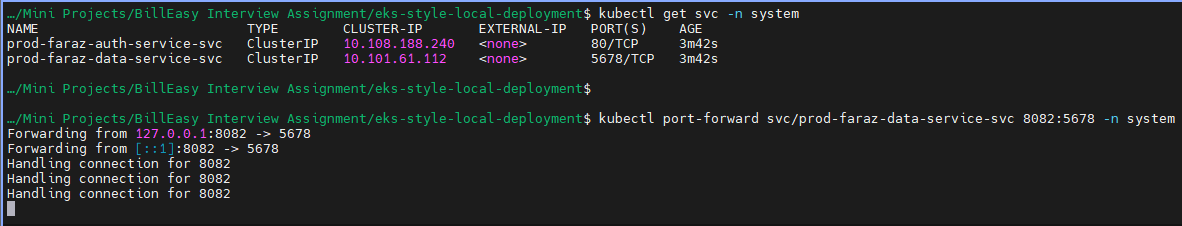
* kubectl port-forward svc/prod-faraz-auth-service-svc 8081:80 -n system



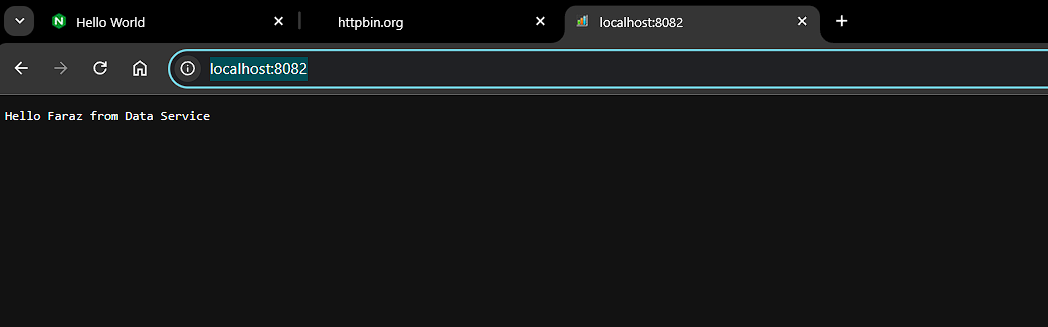
Open: http://localhost:8081/



* kubectl port-forward svc/prod-faraz-data-service-svc 8082:5678 -n system



Open: http://localhost:8082/



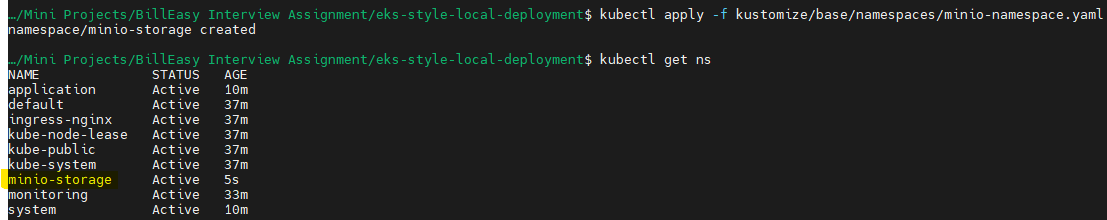
Task 2: Simulate IAM with MinIO

Create a namespace:

* kubectl create namespace minio-storage

or:

* kubectl apply -f kustomize/base/namespaces/minio-namespace.yaml

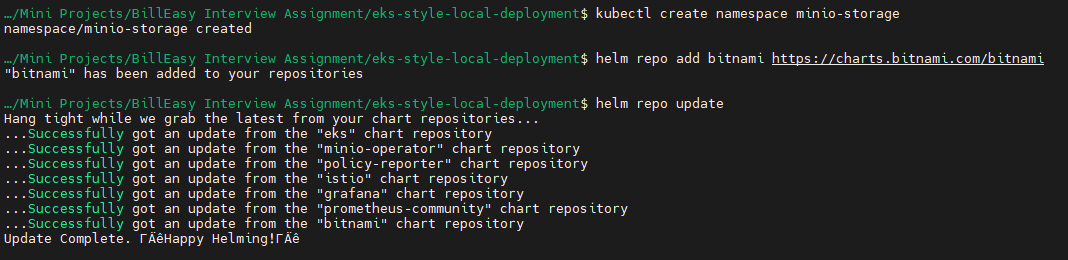


Add the chart:

* helm repo add bitnami <https://charts.bitnami.com/bitnami>

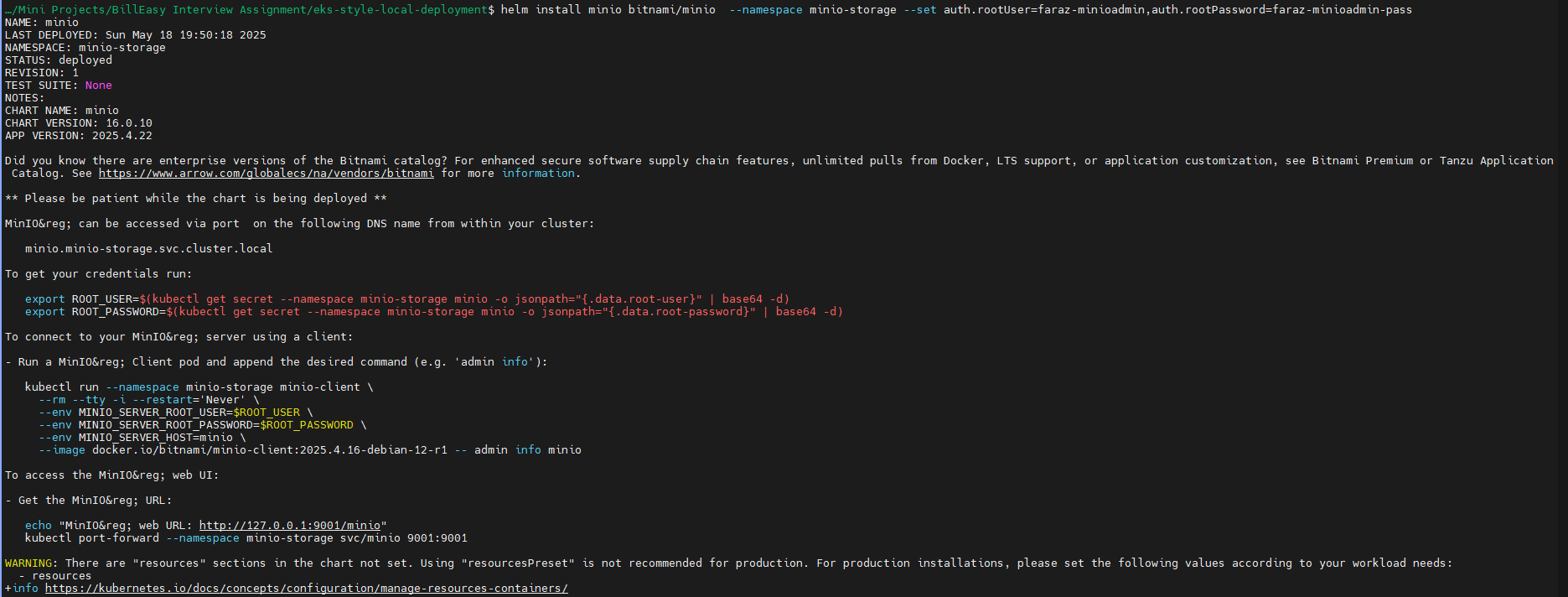
Update the repo:

* helm repo update

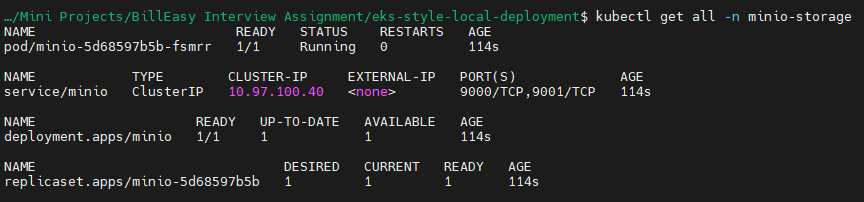


Install:

* helm install minio bitnami/minio --namespace minio-storage --set auth.rootUser=faraz-minioadmin,auth.rootPassword=faraz-minioadmin-pass

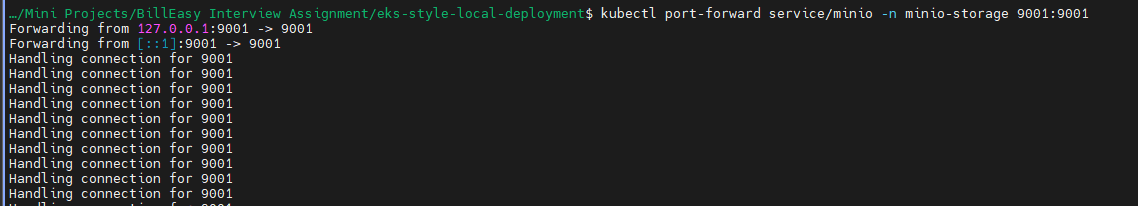


Wait for the pod to be created:



Now since the service is in cluster IP, we can use port forwarding:

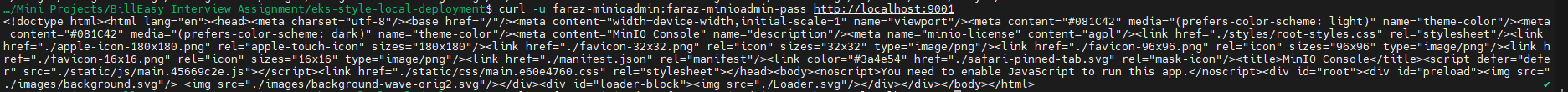
* For UI: kubectl port-forward service/minio -n minio-storage 9001:9001
* For CLI: kubectl port-forward service/minio -n minio-storage 9000:9000



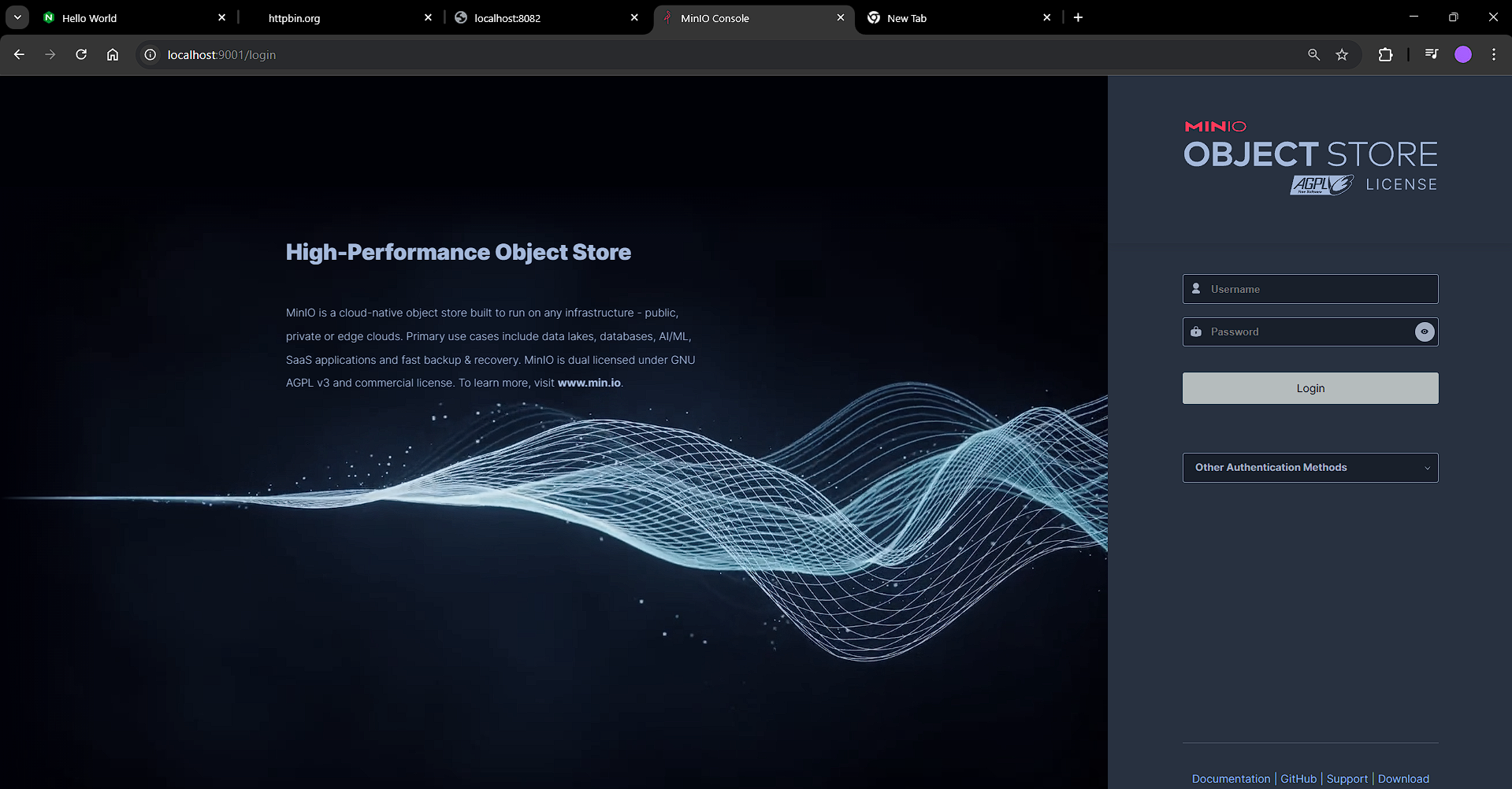
Open in browser: <http://localhost:9001/>

Or you can also curl:

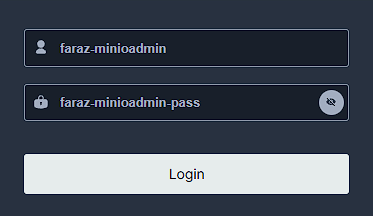
* curl -u faraz-minioadmin:faraz-minioadmin-pass http://localhost:9001



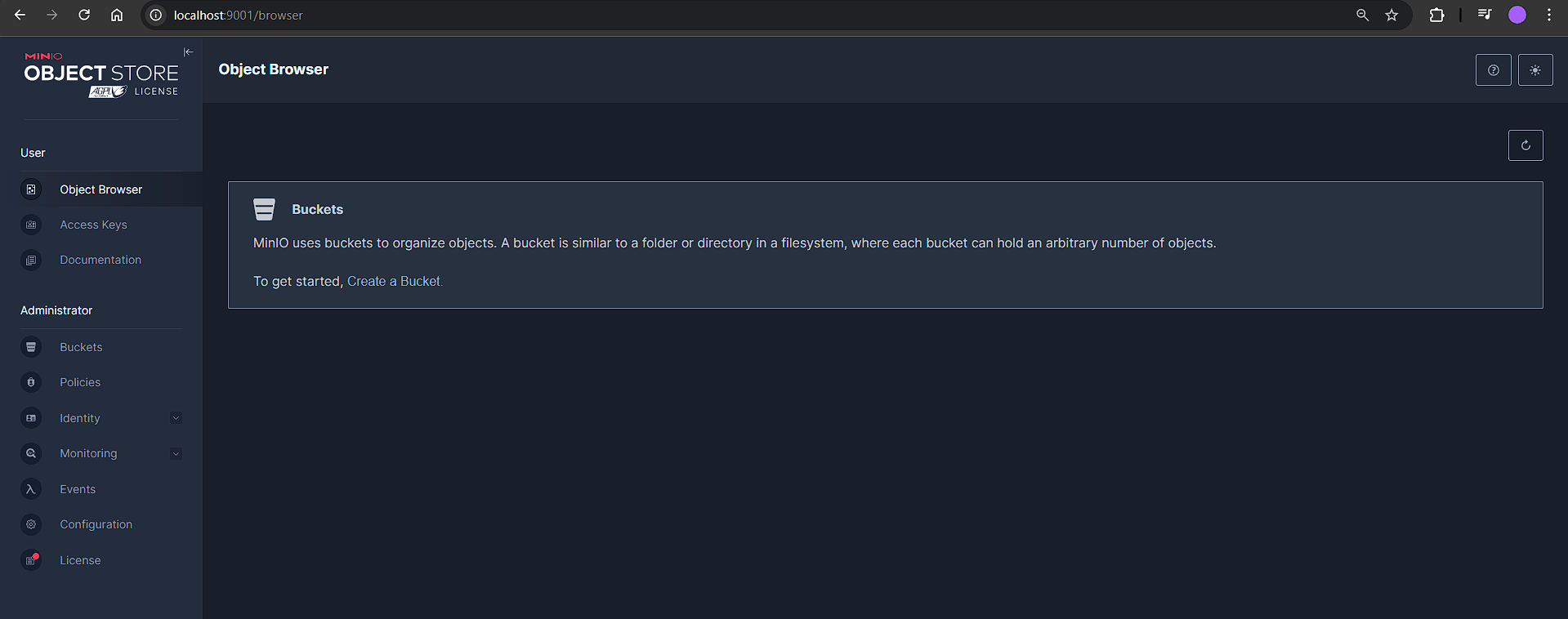
You will get this:



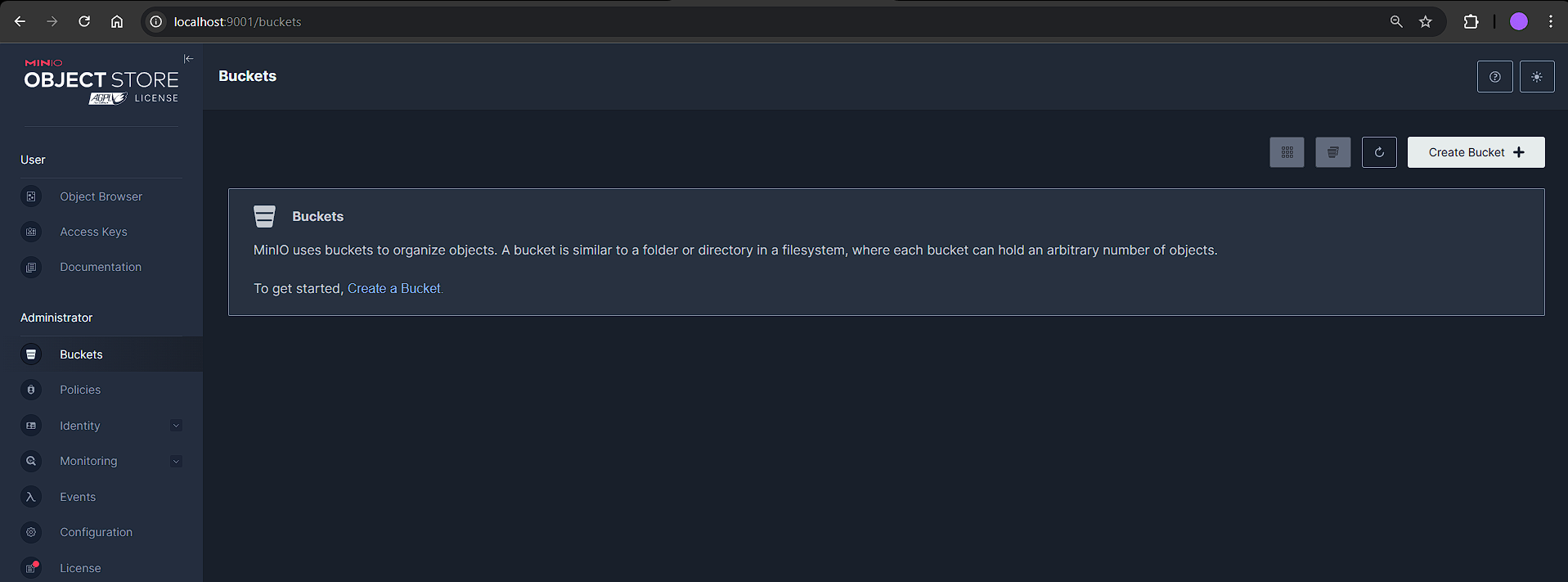
Type in the credentials:



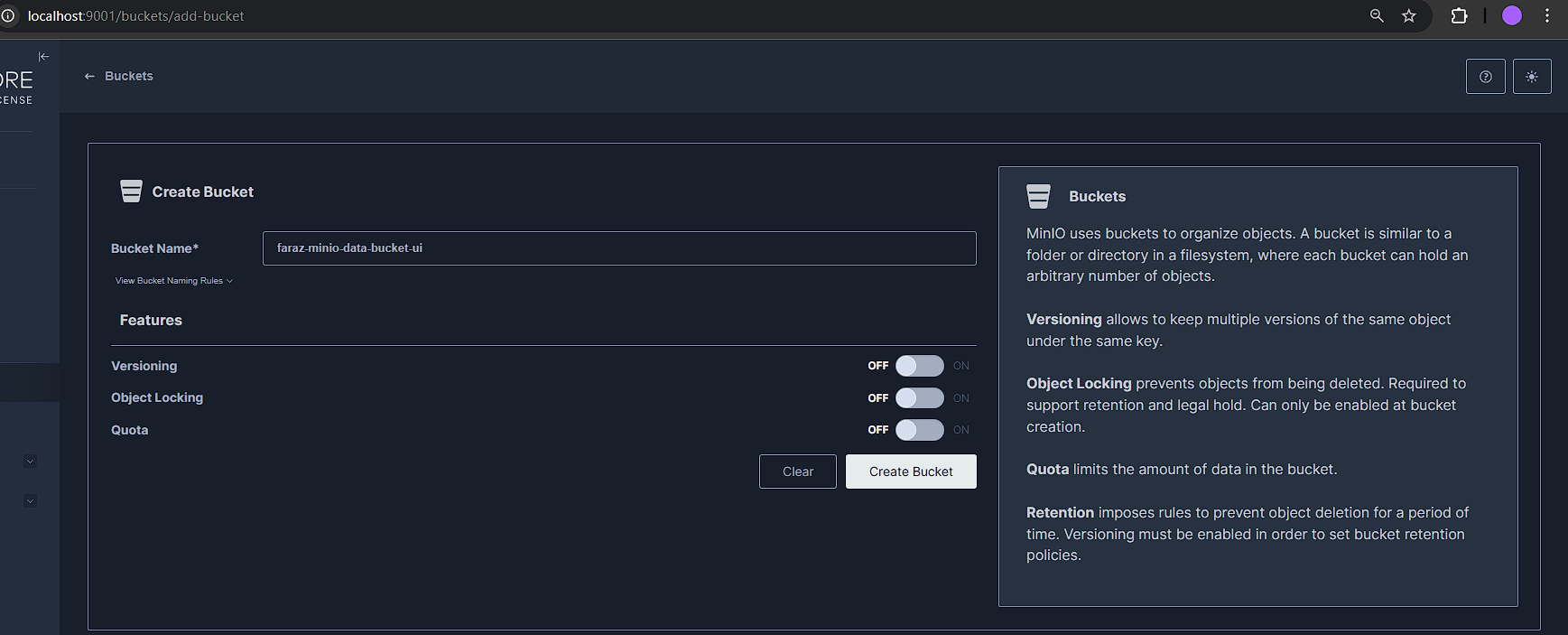
You will be directed to the /browser screen:

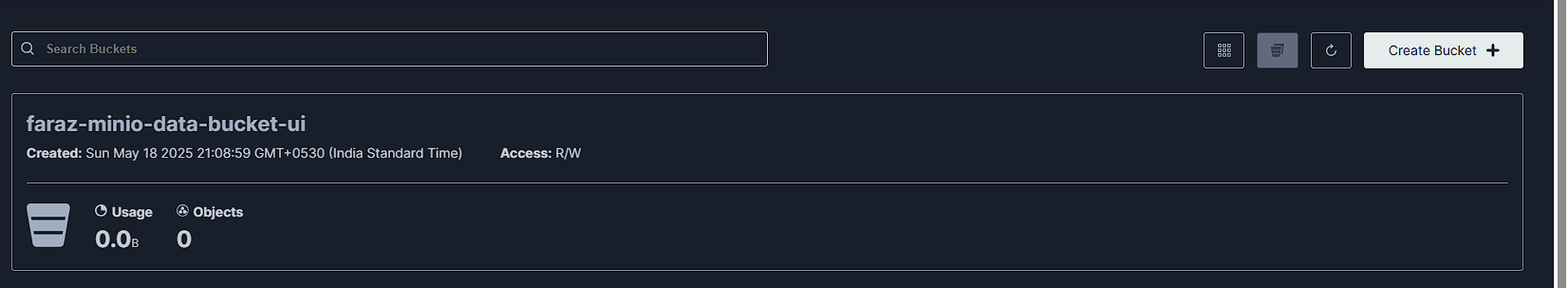


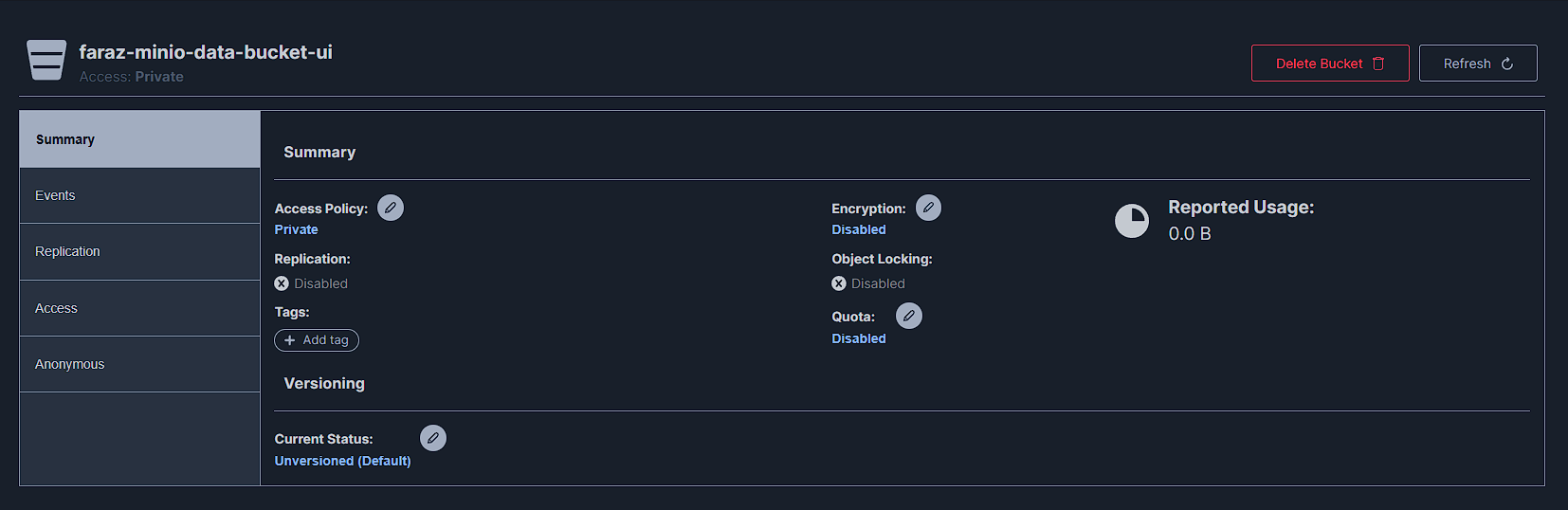
Now create a bucket, by clicking on create a bucket:



Create bucket with name “faraz-minio-data-bucket-ui”:

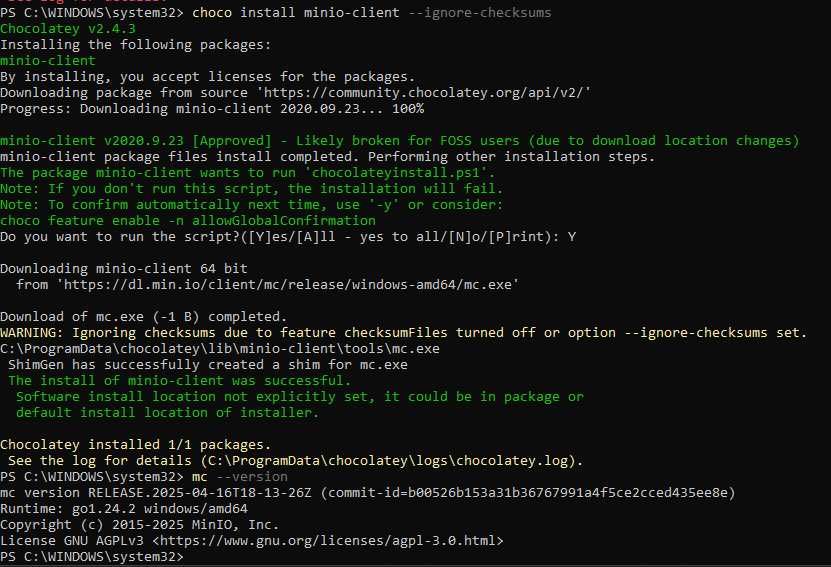






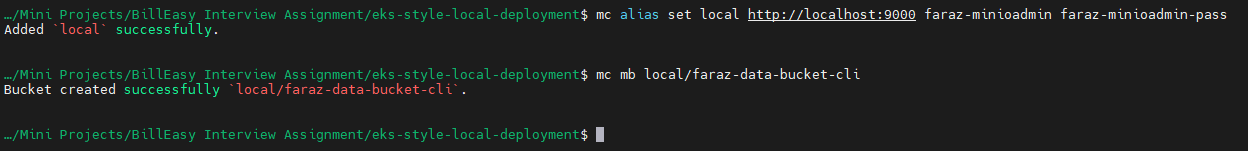
You can also create buckets by installing minio-client:

* Run as admin:
  + choco install minio-client --ignore-checksums

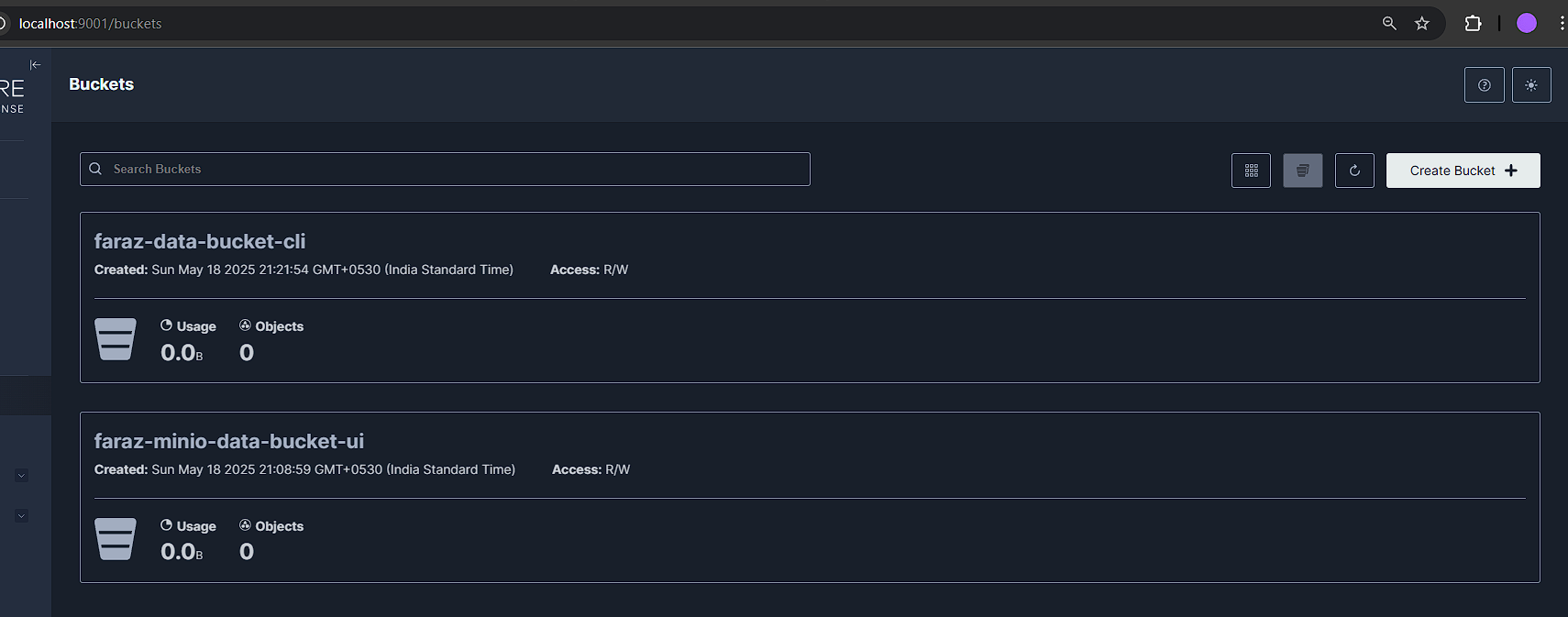


Now to create a bucket from cli:

* kubectl port-forward service/minio -n minio-storage 9000:9000
* mc alias set local http://localhost:9000 faraz-minioadmin faraz-minioadmin-pass
* mc mb local/faraz-data-bucket-cli

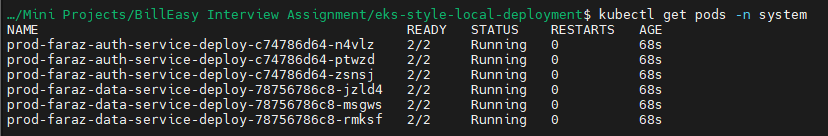


Now you will be able to see a new bucket created from cli on UI:

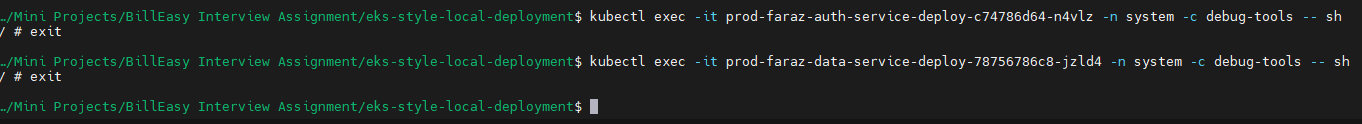


Adding Sidecars to the auth and data service pods deploy so that we can login and verify our connection:

Delete the prod overlay and reapply, you will now notice that there are 2 containers running inside the pod:

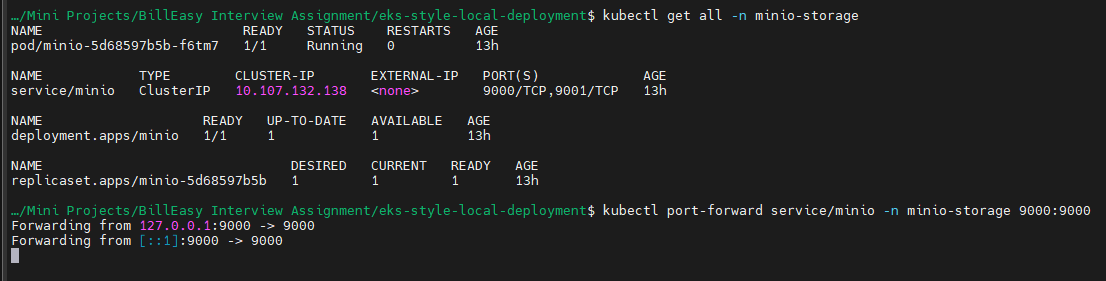


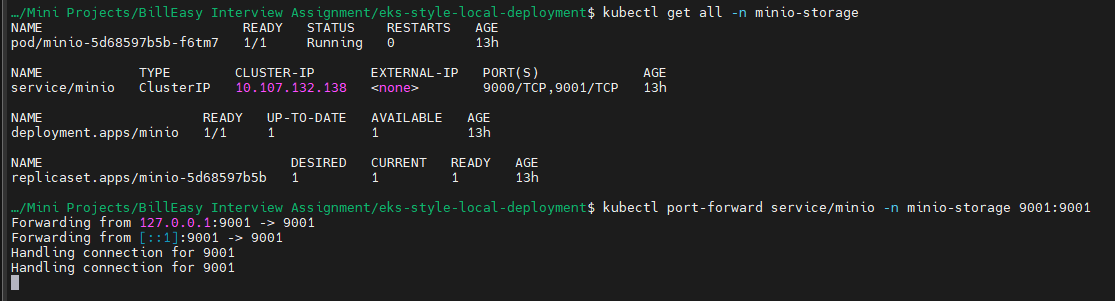
You will now be able to exec and login to the pods:



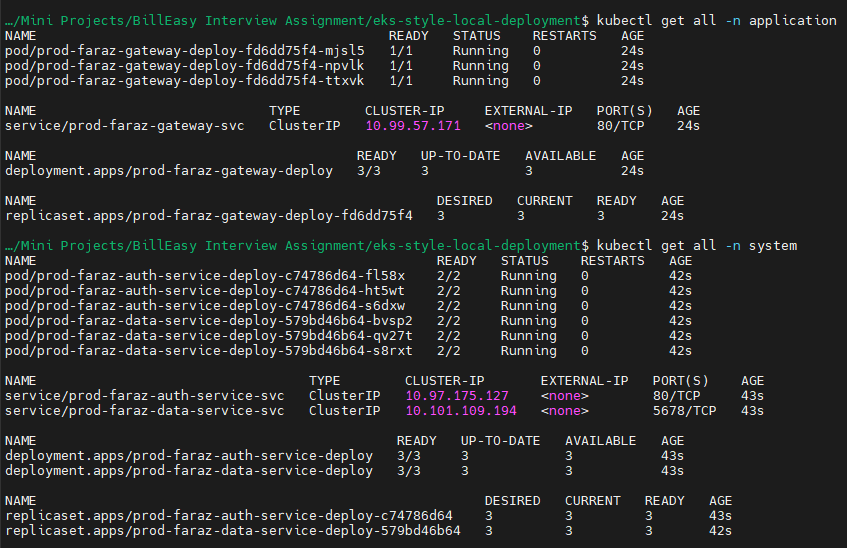
We need this as we will curl to the minio bucket

Ensure that you are port forwarding:

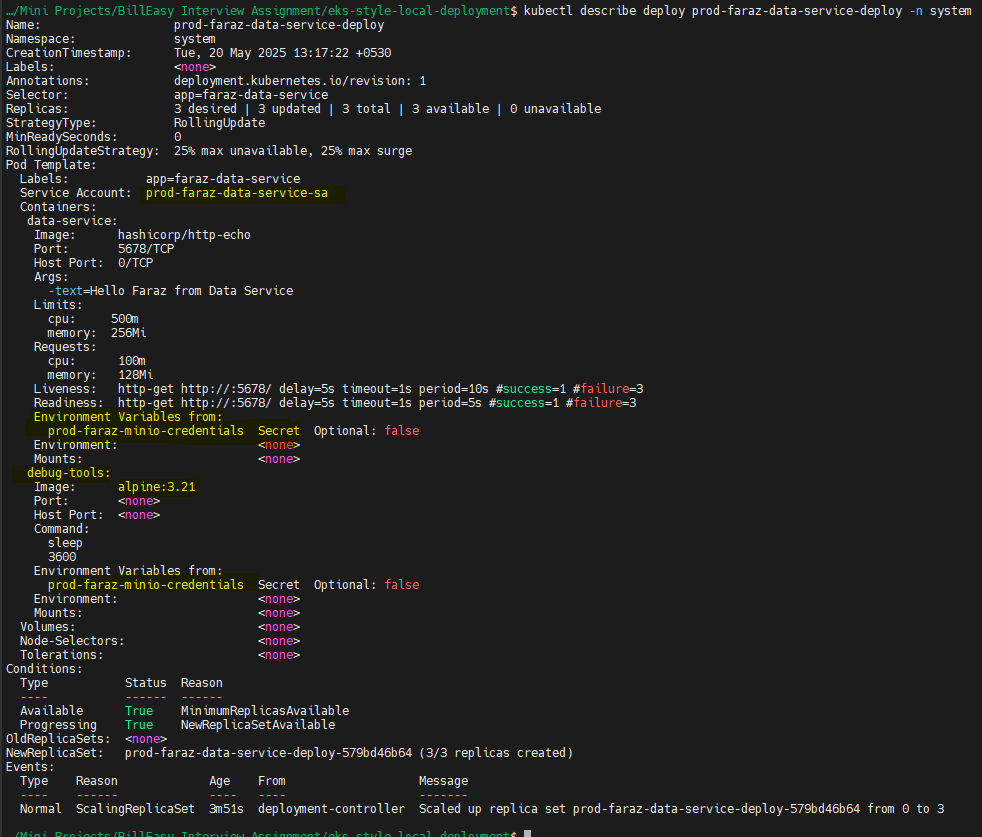




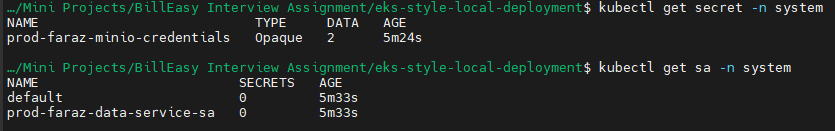
Ensuring that all the resources are up and running:



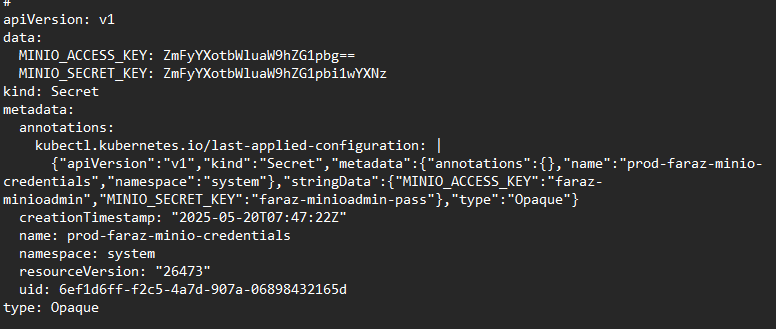
Now if you describe your data-service, you may see that environment variables from :



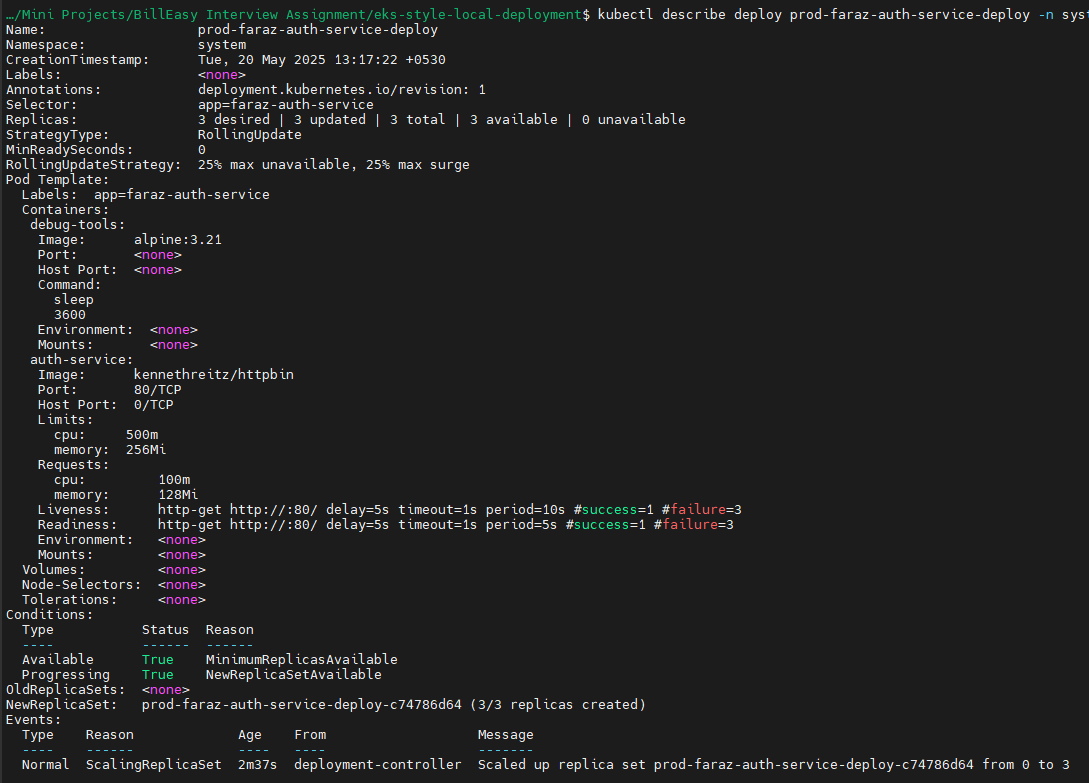
Service account and secret:



Your secret is stored:

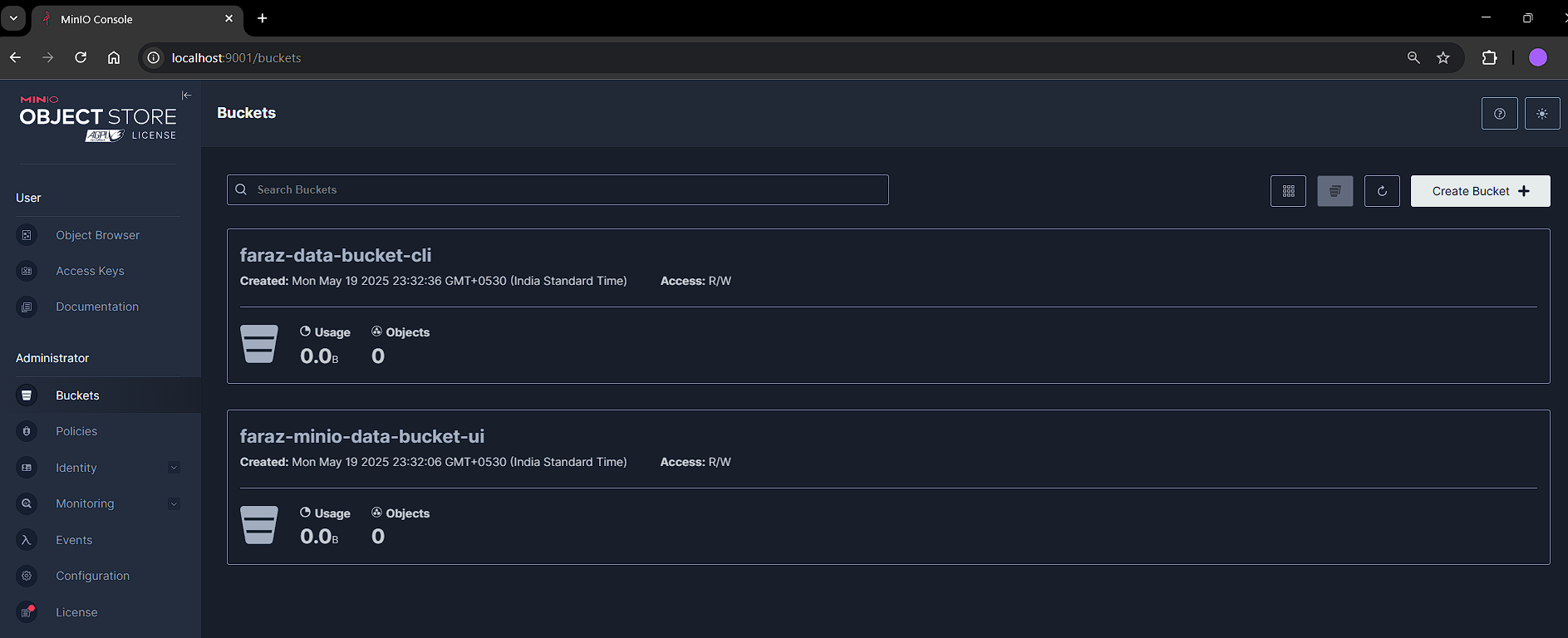


Now if you describe your auth-service, you may not find any environment variables or service account linked:



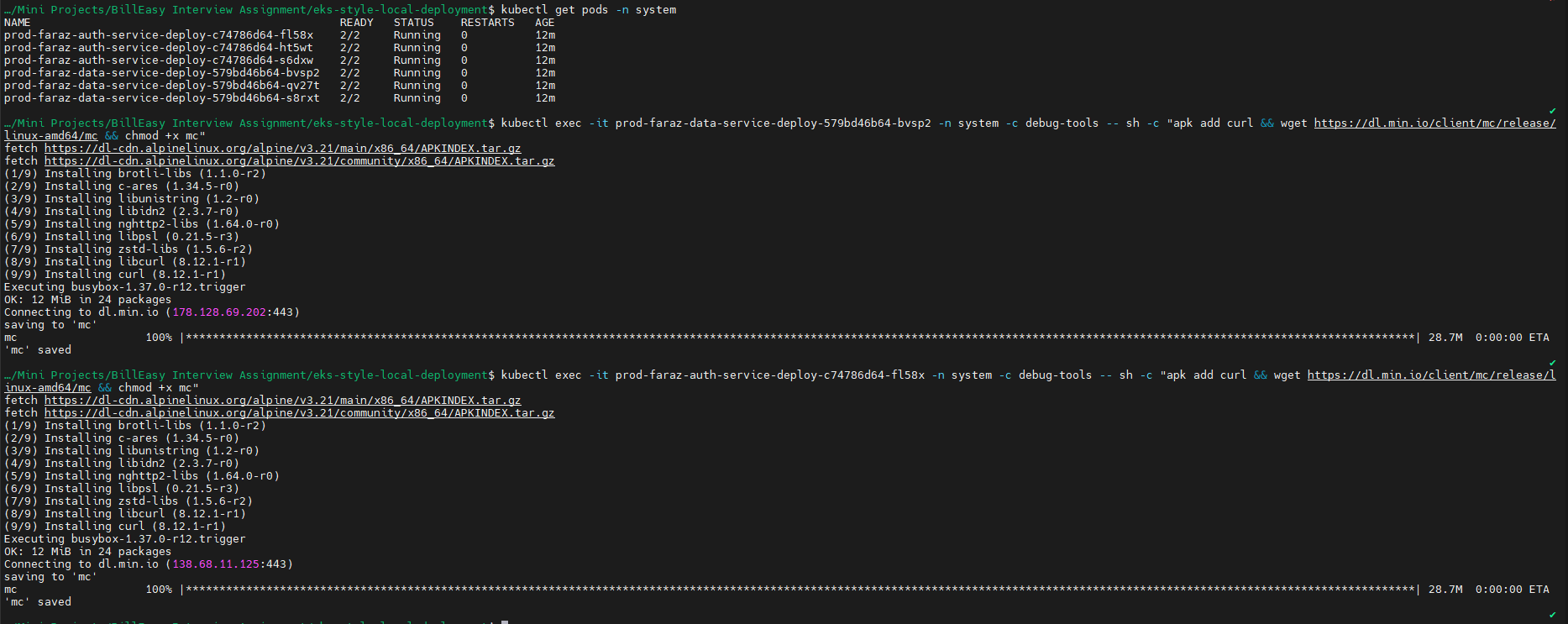
Now let’s test if my data-service has the ability to talk to my minio buckets by listing the buckets:

Currently we have the following buckets:



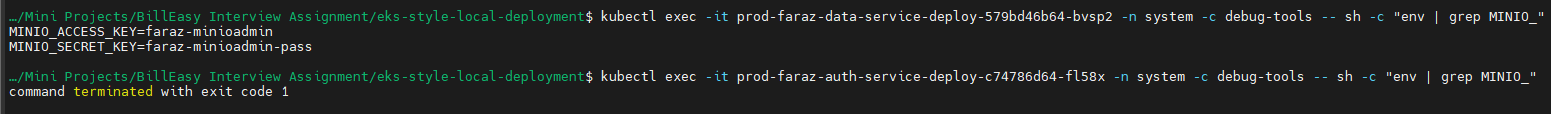
Now lets setup curl, and minio-client so that we have the ability to list the buckets:

* kubectl exec -it prod-faraz-**data-service**-deploy-579bd46b64-bvsp2 -n system -c debug-tools -- sh -c "apk add curl && wget https://dl.min.io/client/mc/release/linux-amd64/mc && chmod +x mc"
* kubectl exec -it prod-faraz-**auth-service-**deploy-c74786d64-fl58x -n system -c debug-tools -- sh -c "apk add curl && wget https://dl.min.io/client/mc/release/linux-amd64/mc && chmod +x mc"



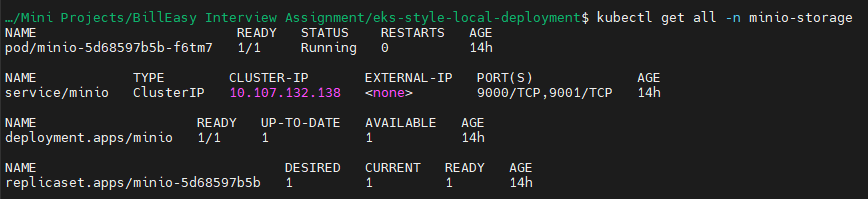
Now try to get the environment variables on from this pods:

* kubectl exec -it prod-faraz-**data-service**-deploy-579bd46b64-bvsp2 -n system -c debug-tools -- sh -c "env | grep MINIO\_"
* kubectl exec -it prod-faraz-**auth-service**-deploy-c74786d64-fl58x -n system -c debug-tools -- sh -c "env | grep MINIO\_"



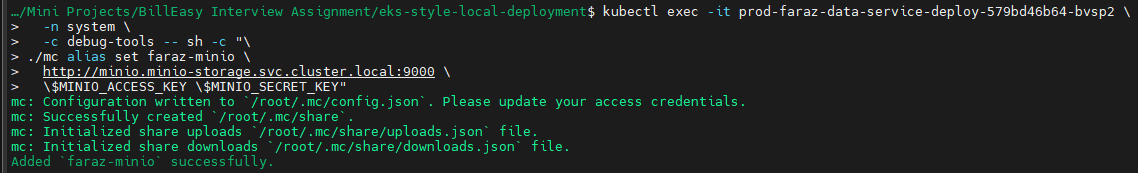
Here you will notice that data-service has the access\_key and secret\_key which we set using secrets and just allowed service account to access this secret. Hence, data-service returned the env secret while auth-service did not return anything.

Now let’s try listing the buckets in minio-storage namespace from data-service in system namespace:



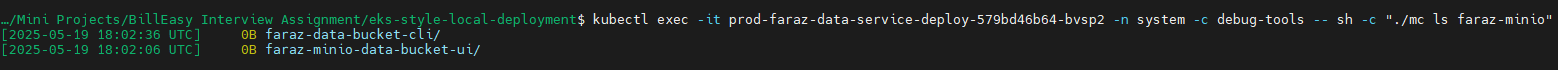
First we need to set alias:

* kubectl exec -it prod-faraz-data-service-deploy-579bd46b64-bvsp2 -n system -c debug-tools -- sh -c "./mc alias set faraz-minio http://minio.minio-storage.svc.cluster.local:9000 \$MINIO\_ACCESS\_KEY \$MINIO\_SECRET\_KEY"



Listing the buckets:

* kubectl exec -it prod-faraz-data-service-deploy-579bd46b64-bvsp2 -n system -c debug-tools -- sh -c "./mc ls faraz-minio"

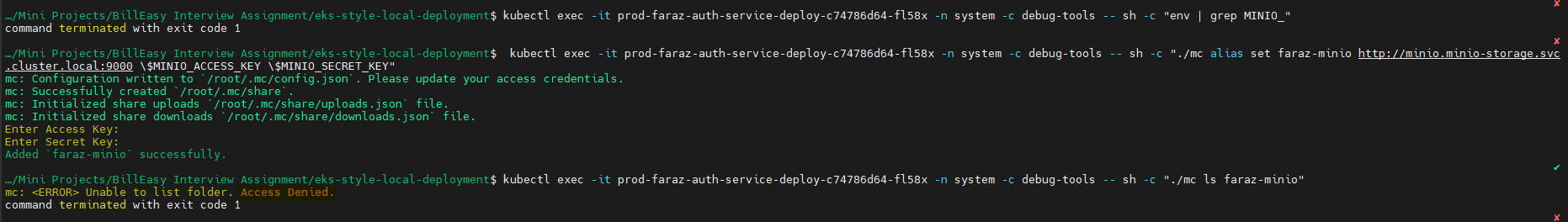


You can also create a test bucket:

* kubectl exec -it prod-faraz-data-service-deploy-579bd46b64-bvsp2 -n system -c debug-tools -- sh -c "./mc mb faraz-minio/testbucket"



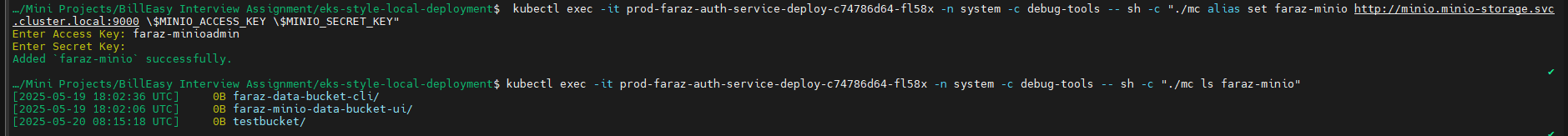
Let’s try to do the same using auth-service:



Here as you can see that the access is denied.

Let’s say somehow I got access to my credentials:

I will be able to access and list my buckets on minio:

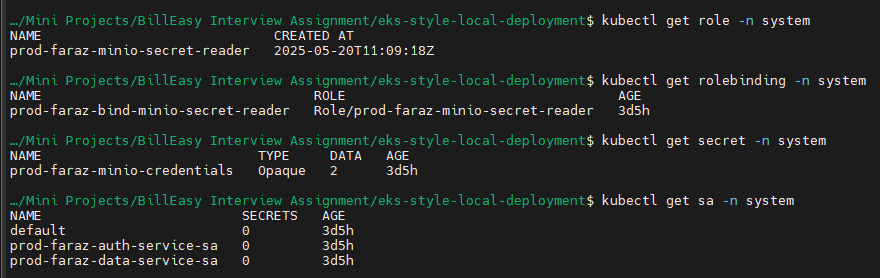


Let’s enforce RBAC and Cluster policies to limit the service accounts to use the minio-credentials:

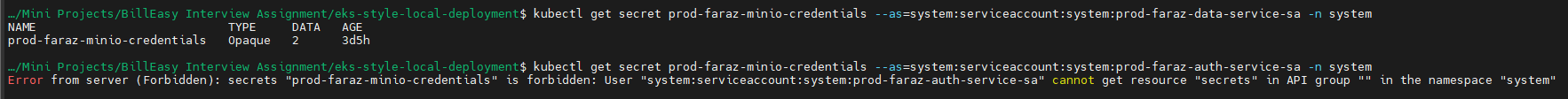
We will practically see how RBAC fails and why do we need cluster policies:

1. Why RBAC Alone Won’t Prevent Unauthorized Secret Mounts

RBAC’s Scope: Role and RoleBinding restrict who can call the Kubernetes API to read or modify a secret. In your manifests, you have defined a Role that grants the “get” and “list” verbs on the secret only to the designated service account (e.g. prod-faraz-data-service-sa). This means that if someone tries to use the API (for example, via kubectl get secret ...) they will only succeed if they’re using an identity that has that permission.

Testing RBAC is working:  


* kubectl get secret prod-faraz-minio-credentials --as=system:serviceaccount:system:prod-faraz-data-service-sa -n system
* kubectl get secret prod-faraz-minio-credentials --as=system:serviceaccount:system:prod-faraz-auth-service-sa -n system



Secret Mounting Behavior: When a pod is created with a reference to a secret (via envFrom or a volume mount), the secret data is automatically injected into the pod by the kubelet. The kubelet does not perform an “authorization check” on the secret reference in the pod spec. In other words, even if the pod is running with a service account that wouldn’t normally have permission to read that secret via an API call, the secret is still mounted as long as the pod manifest explicitly requests it.

Explicit Patching: If someone patches the deployment (or sidecar configuration) of an unauthorized workload (say, the auth-service) to include an envFrom reference to the sensitive secret, the API server will happily accept that patch (if the patching user has permission to change the pod spec). The mounted secret is then delivered to the pod—even though, according to RBAC, that service account wouldn’t be “allowed” to read the secret via a direct API call. In brief, RBAC controls access at the API level, not at the pod runtime level.

2. Enforcing the “Only Data-Service” Rule

To ensure that only pods running with the designated service account (i.e. prod-faraz-data-service-sa) are allowed to mount your MinIO credentials, you must enforce that rule at admission time. With an admission controller (like OPA Gatekeeper or Kyverno), you can write policies that examine every pod submission and reject any pod whose spec tries to mount the secret but which is not using the correct service account.

For example, a Kyverno can be written that:

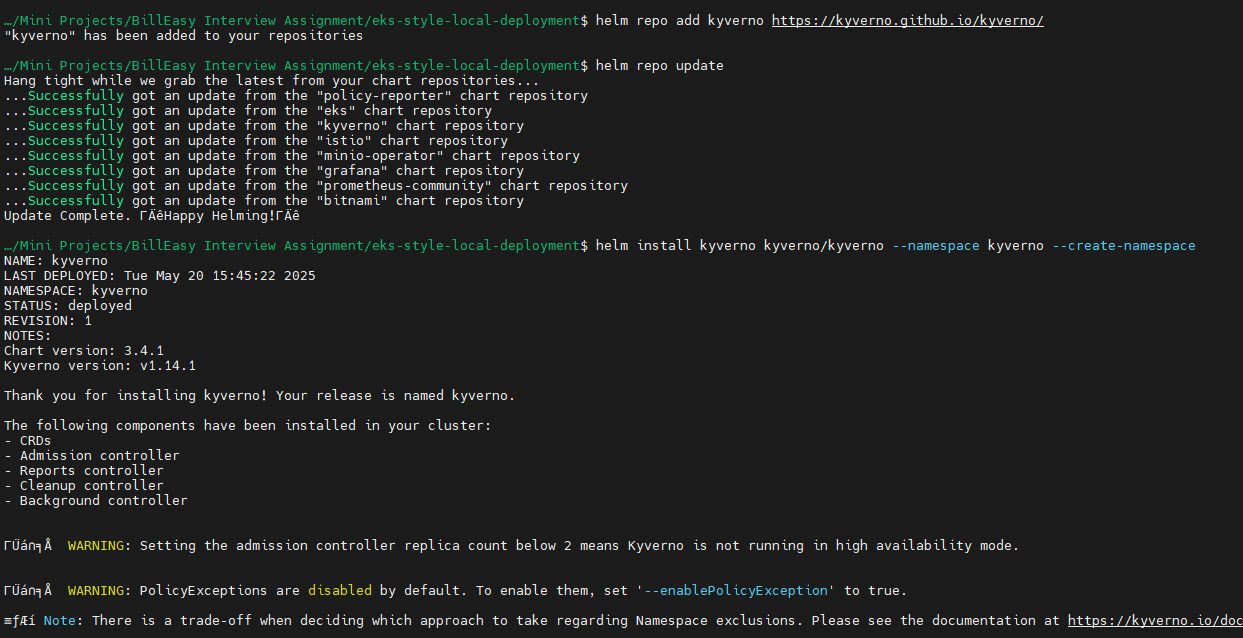
Matches any Pod that references the secret prod-faraz-minio-credentials.

Validates that if the secret is referenced, then the pod’s service account must be prod-faraz-data-service-sa.

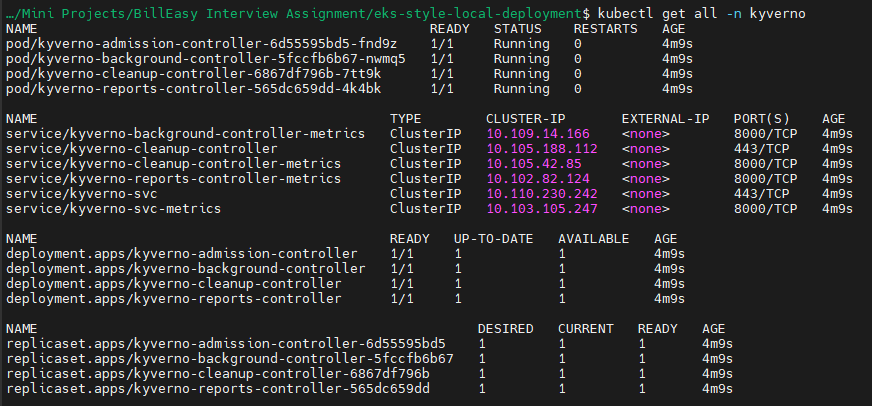
Denies the pod creation for any pod (or patch) that does not meet this condition.

Installing Kyverno:

* helm repo add kyverno <https://kyverno.github.io/kyverno/>
* helm repo update
* helm install kyverno kyverno/kyverno --namespace kyverno --create-namespace

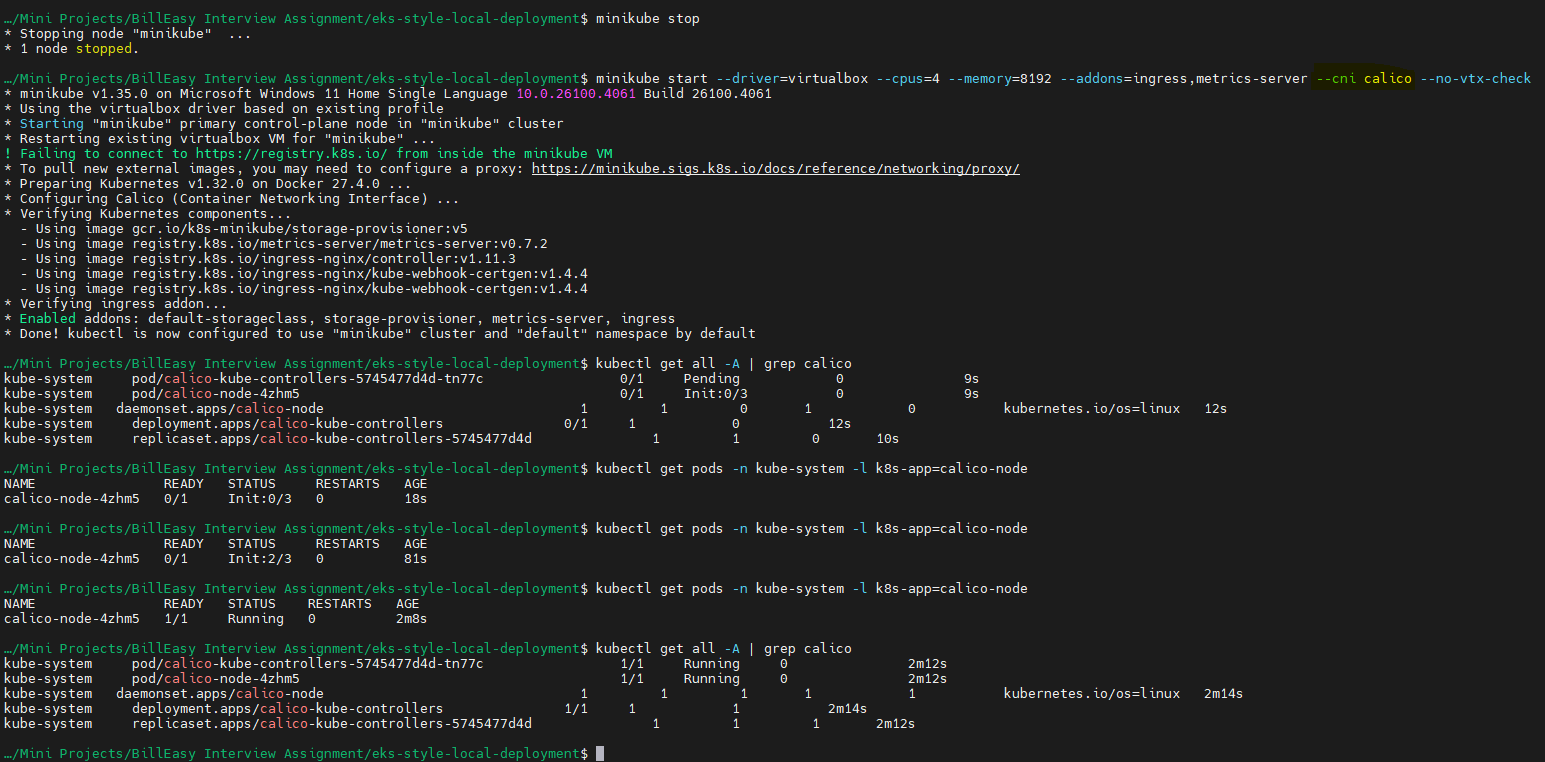


Verify your install:



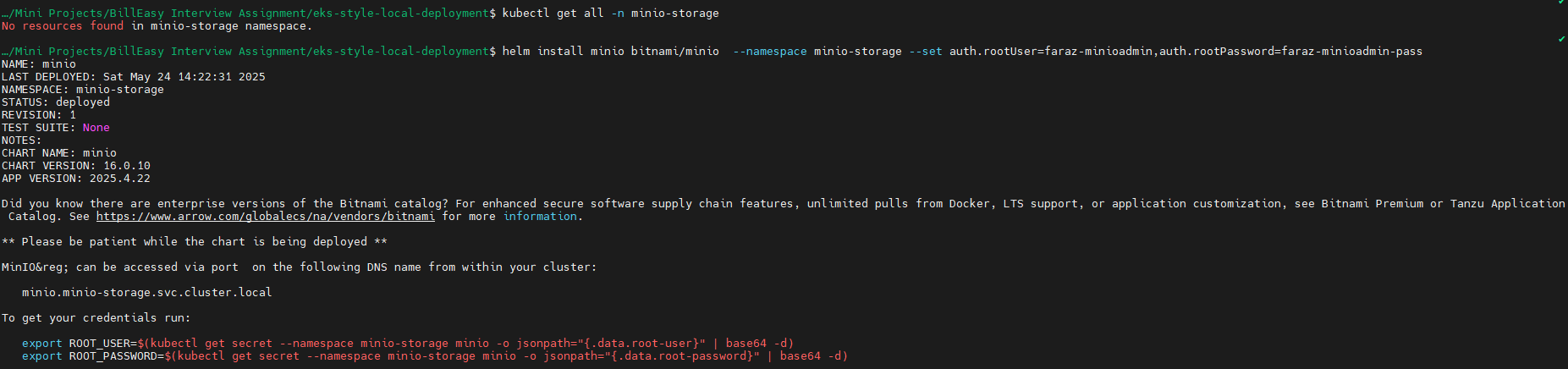
We will use network policies to restrict access to minio-storage namespace:

* minikube stop
* minikube start --driver=virtualbox --cpus=4 --memory=8192 --addons=ingress,metrics-server **--cni calico** --no-vtx-check
* kubectl get pods -n kube-system -l k8s-app=calico-node

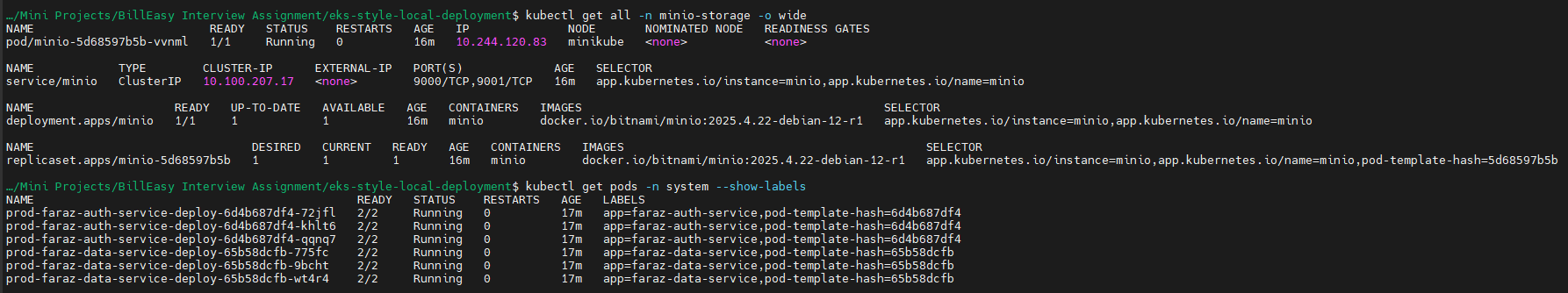


Create the network policy and add it to base kustomization.yaml



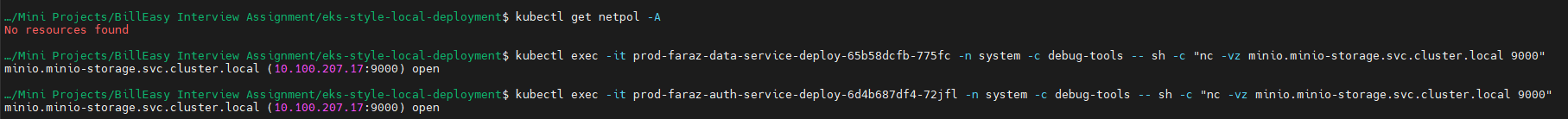
Re-install minio:  


Listing pods for reference:



We are able to connect to minio from the pods when there’s no network policy:

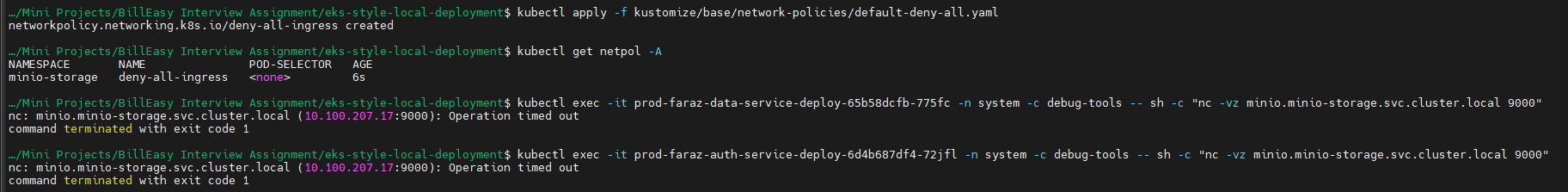
* kubectl exec -it prod-faraz-data-service-deploy-65b58dcfb-775fc -n system -c debug-tools -- sh -c "nc -vz minio.minio-storage.svc.cluster.local 9000"
* kubectl exec -it prod-faraz-auth-service-deploy-6d4b687df4-72jfl -n system -c debug-tools -- sh -c "nc -vz minio.minio-storage.svc.cluster.local 9000"



As we can see that pods are able to connect to minio

Now lets’ test if network policy is working using default deny:

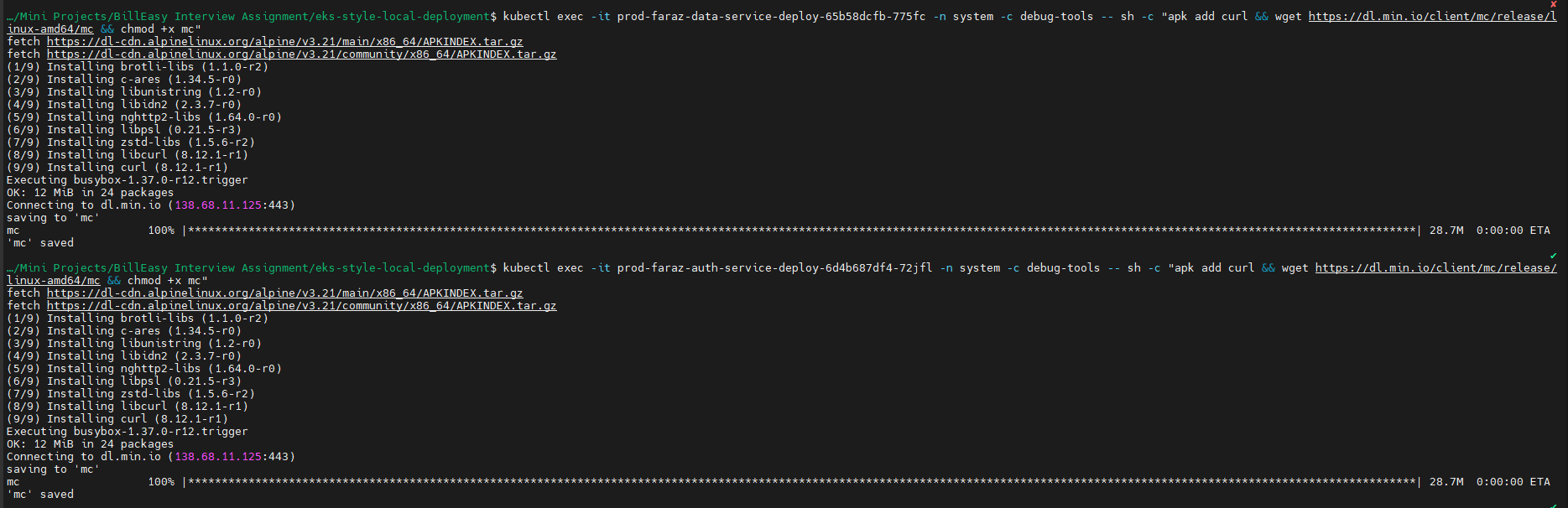
* kubectl apply -f kustomize/base/network-policies/default-deny-all.yaml



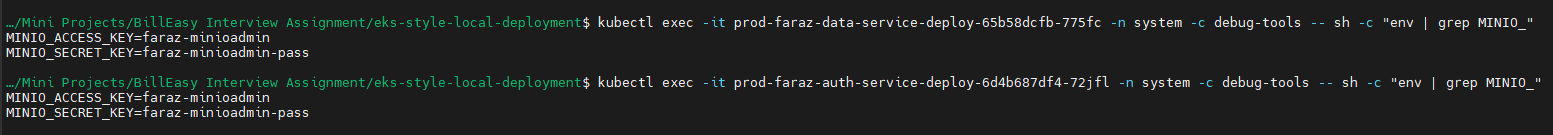
Here we can notice that now the operation is blocked

We can also cross verify by:

* kubectl exec -it prod-faraz-data-service-deploy-65b58dcfb-775fc -n system -c debug-tools -- sh -c "apk add curl && wget https://dl.min.io/client/mc/release/linux-amd64/mc && chmod +x mc"
* kubectl exec -it prod-faraz-auth-service-deploy-6d4b687df4-72jfl -n system -c debug-tools -- sh -c "apk add curl && wget https://dl.min.io/client/mc/release/linux-amd64/mc && chmod +x mc"

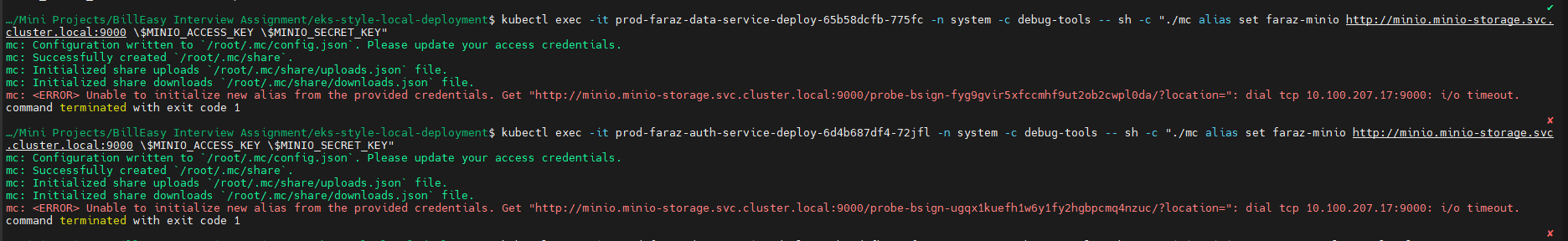


Here you can notice that even though we have credentials:



Due to deny all policy:

* kubectl exec -it prod-faraz-data-service-deploy-65b58dcfb-775fc -n system -c debug-tools -- sh -c "./mc alias set faraz-minio http://minio.minio-storage.svc.cluster.local:9000 \$MINIO\_ACCESS\_KEY \$MINIO\_SECRET\_KEY"
* kubectl exec -it prod-faraz-auth-service-deploy-6d4b687df4-72jfl -n system -c debug-tools -- sh -c "./mc alias set faraz-minio http://minio.minio-storage.svc.cluster.local:9000 \$MINIO\_ACCESS\_KEY \$MINIO\_SECRET\_KEY"

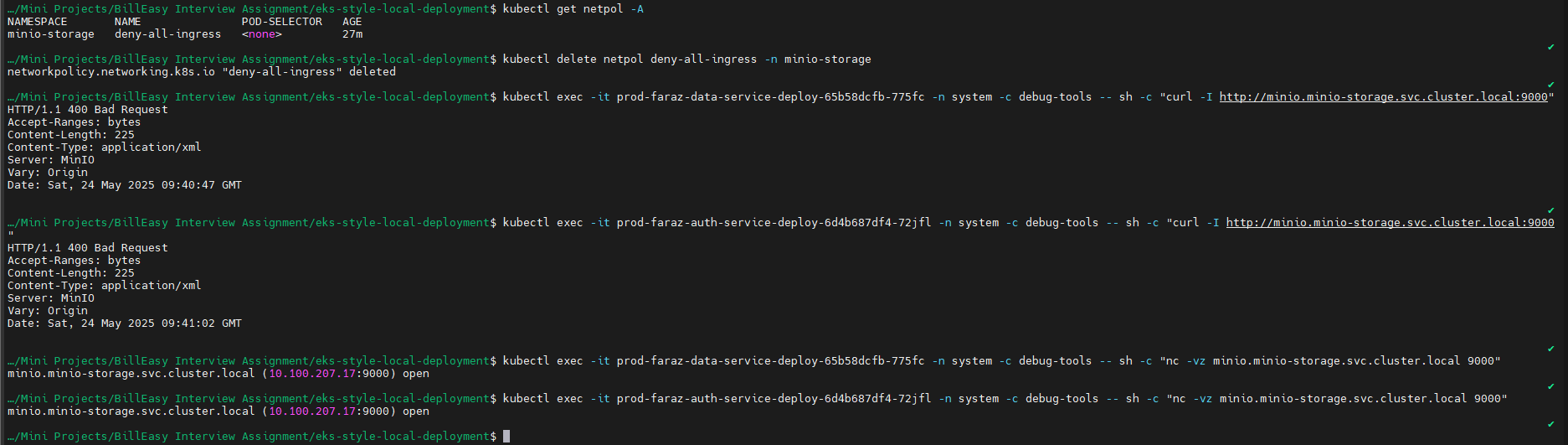


You can also curl and check:

* kubectl exec -it prod-faraz-data-service-deploy-65b58dcfb-775fc -n system -c debug-tools -- sh -c "curl -I <http://minio.minio-storage.svc.cluster.local:9000>"
* kubectl exec -it prod-faraz-auth-service-deploy-6d4b687df4-72jfl -n system -c debug-tools -- sh -c "curl -I <http://minio.minio-storage.svc.cluster.local:9000>"



If you delete your network policy:



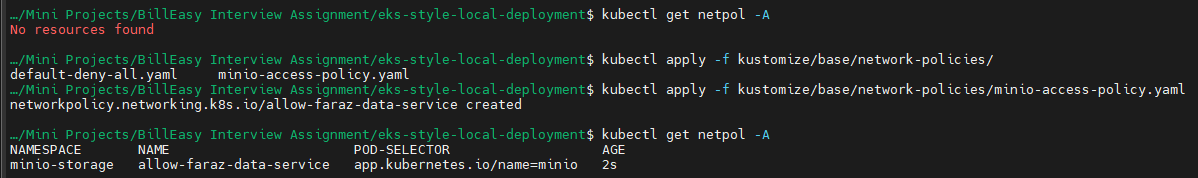
You will observe that your pods can reach out to minio

Hence, proved that Network policy is working

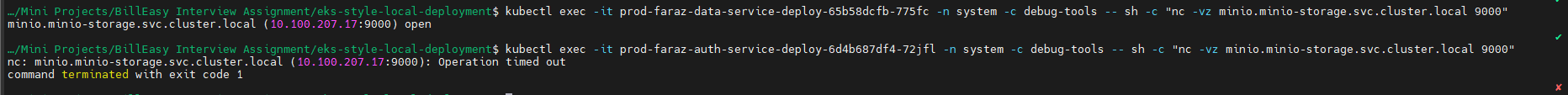
Now let’s only allow faraz-data-service labelled nodes to access minio:



Apply your network policy:

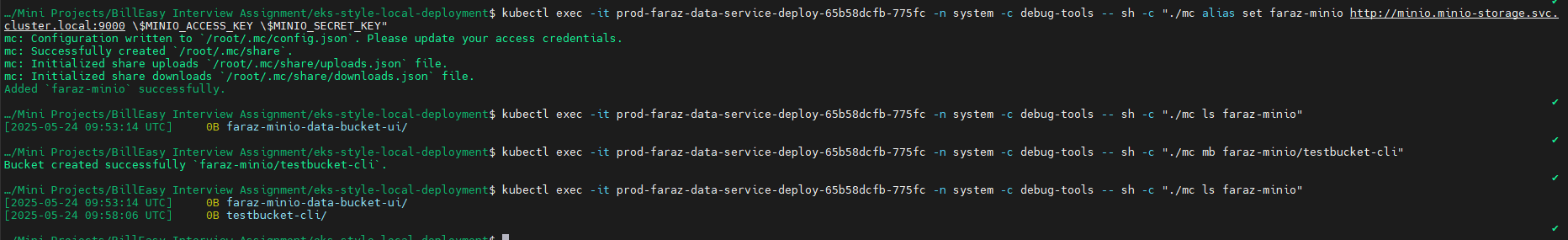


Final testing:

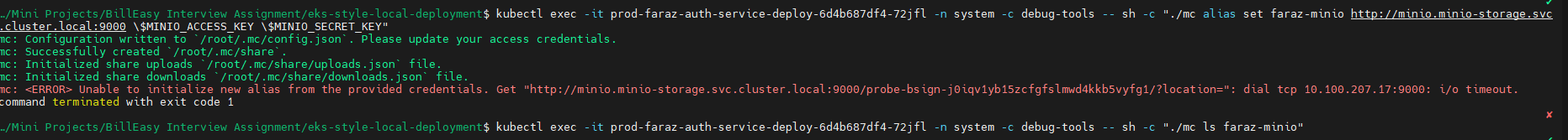


Hence, proved that auth-service cannot connect to minio even if it anyone by mistakenly mounts MINIO secret to it

Here the data-service pods can list and create buckets:



While auth-service pods cannot set alias and connect to minio:

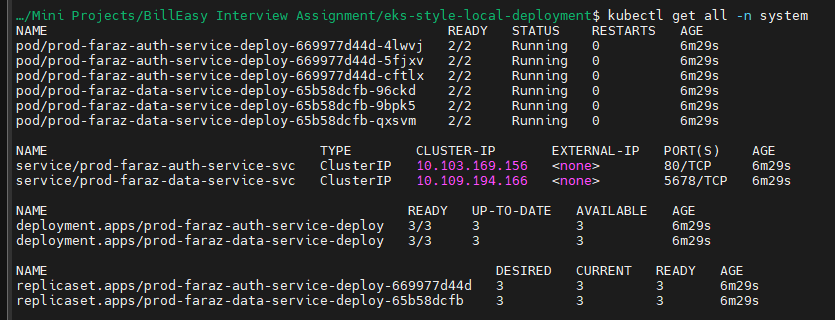


This ensures that even if someone mounts env variables to your unauthorized pods, they cannot communicate to minio as per the network policy set.

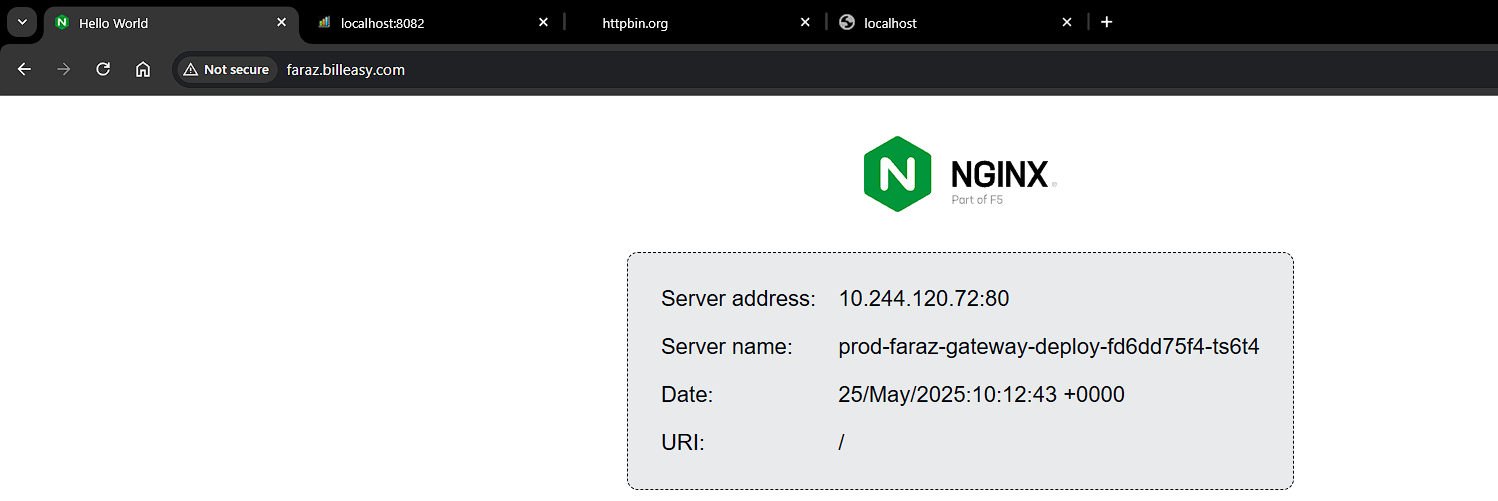
Todo: Enforce Kyverno

Part 3: Security Incident Simulation

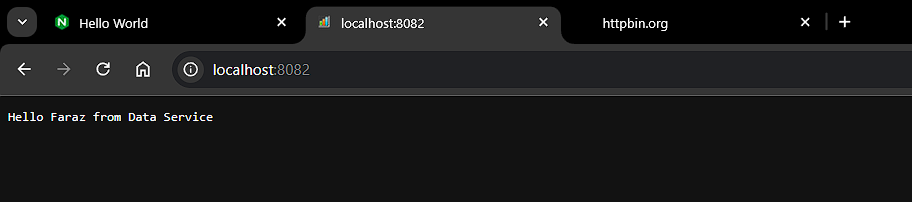
Ensure all your services are up and running:



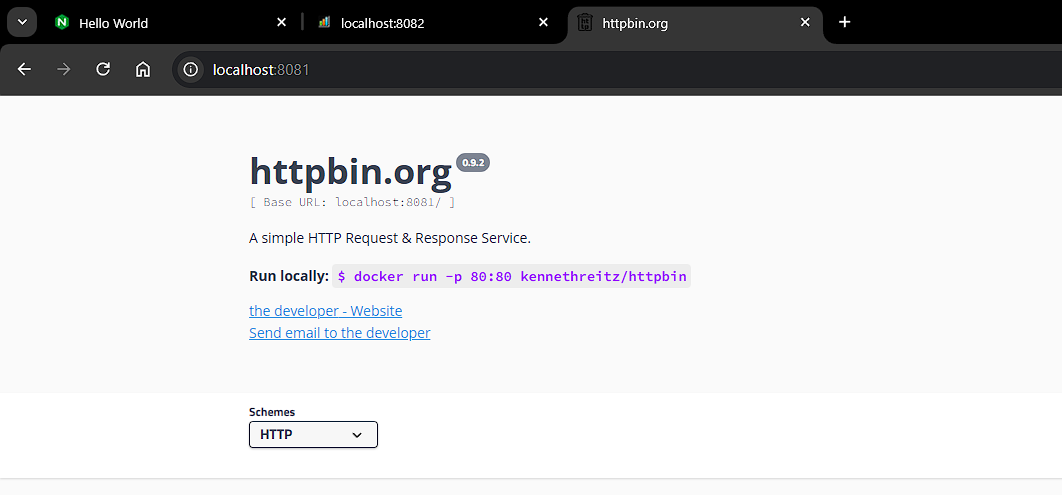
1. Gateway ingress:



2. Data service:

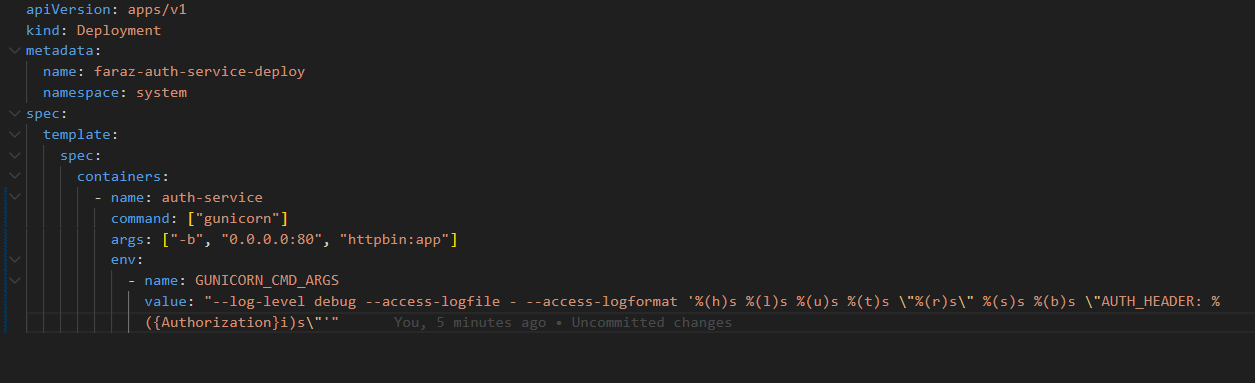


3. Auth service:

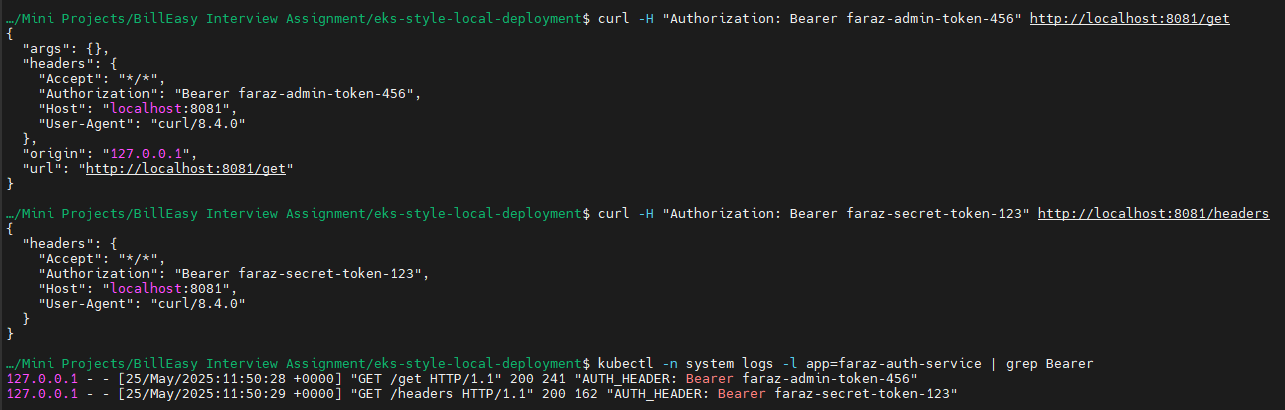


Now to simulate the leak of Authorization headers which will log the Authorization headers:

Add it to deployment via patch:

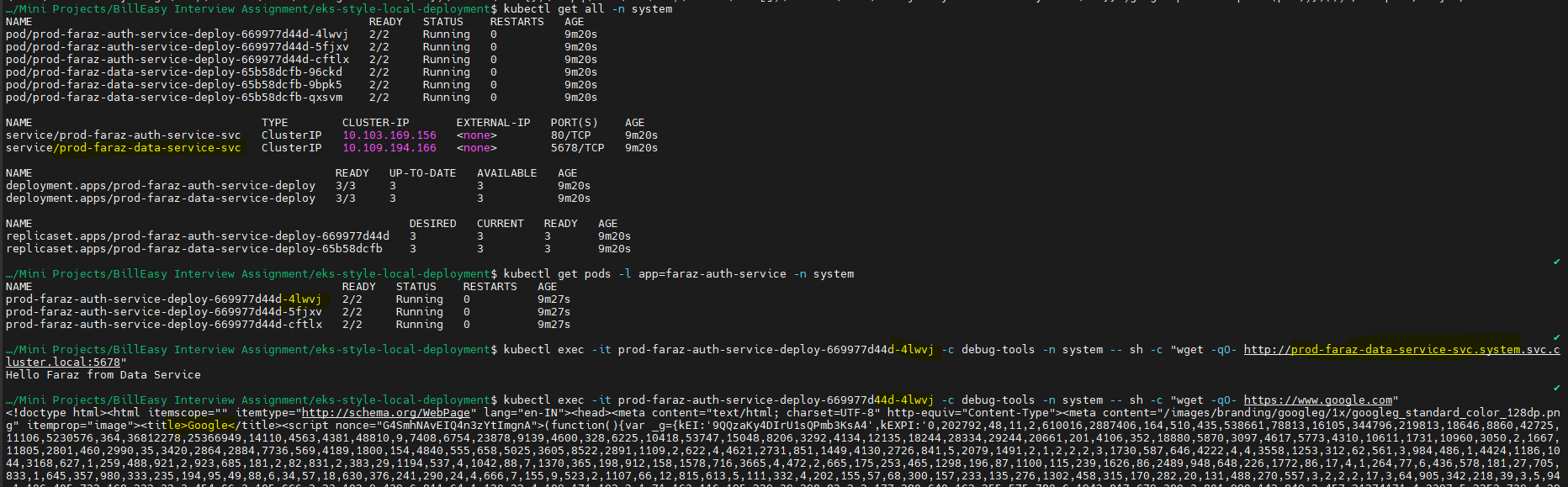


As you can see that Authorization headers is leaking on hitting /headers or /get route:



Also we can notice that it can communicate to data-service and also external service like Google.com:

* kubectl exec -it prod-faraz-auth-service-deploy-669977d44d-4lwvj -c debug-tools -n system -- sh -c "wget -qO- <http://prod-faraz-data-service-svc.system.svc.cluster.local:5678>"
* kubectl exec -it prod-faraz-auth-service-deploy-669977d44d-4lwvj -c debug-tools -n system -- sh -c "wget -qO- <https://www.google.com>"

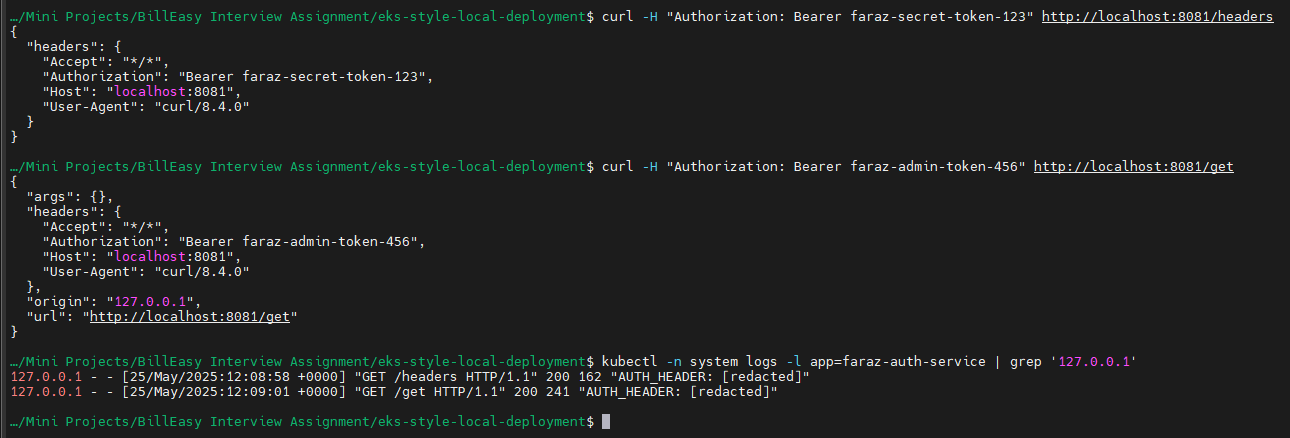


Now to fix the Authorization header leaks:

We will remove %({Authorization}i) from deployment:



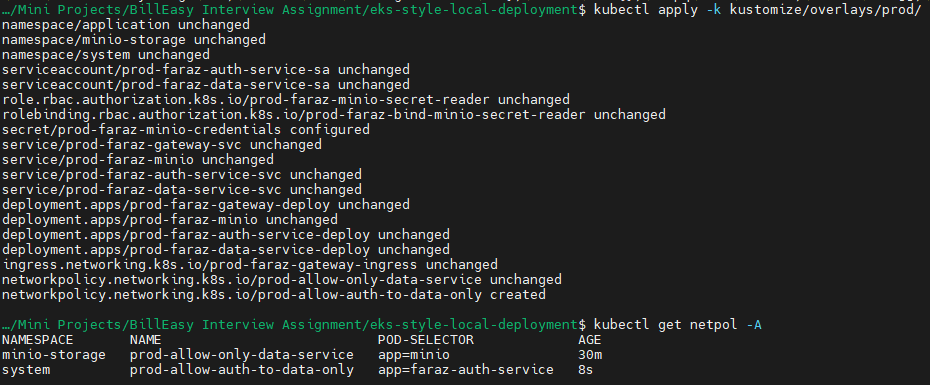
Now if we try:



We can see that the Auth headers are not visible in the logs. Hence solving the leaks

And to now allow auth-service deploy to not communicate to external world we will enforce network policies:

Creating network policy:



The below will not work as it doesn’t have DNS resolution

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: allow-auth-to-data-only

  namespace: system

spec:

  podSelector:

    matchLabels:

      app: faraz-auth-service

  policyTypes:

  - Egress

  egress:

  - to:

    - podSelector:

        matchLabels:

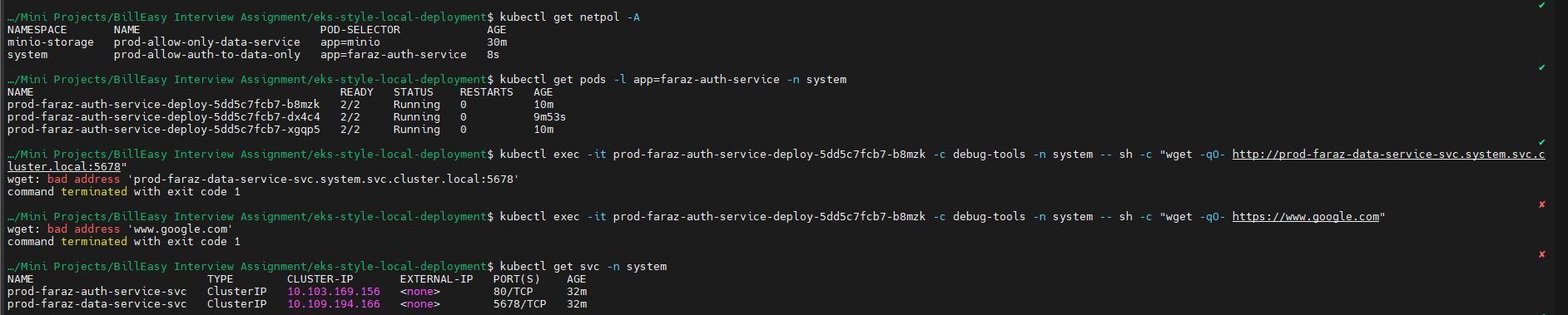
          app: faraz-data-service

    ports:

    - protocol: TCP

      port: 5678

It will shoe bad address:



We may have to use IPs:



Why the Original Policy Failed

1. **DNS Resolution Blocked**

* When you tried to resolve prod-faraz-data-service-svc.system.svc.cluster.local, the auth pod needed to communicate with **Kubernetes DNS (CoreDNS)** to resolve the hostname to an IP address.
* The original policy **blocked all egress traffic except to**app: faraz-data-service**on port 5678**. This included **DNS traffic** (port 53), which is required for resolving hostnames.
* **Result**: The auth pod couldn’t resolve the DNS name, leading to the bad address error.

2. **No External Access**

* The policy also blocked access to external services like https://www.google.com because it only allowed traffic to the data service pods. This is expected behavior but highlights why DNS was critical.

Fixes:

Allows DNS Traffic:

Targets pods labeled k8s-app: kube-dns (CoreDNS) in the kube-system namespace.

Opens port 53 (UDP for DNS queries, TCP for large responses).

Uses namespaceSelector:

Ensures traffic is allowed to the kube-system namespace where CoreDNS runs.

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: allow-auth-to-data-only

  namespace: system

spec:

  podSelector:

    matchLabels:

      app: faraz-auth-service

  policyTypes:

  - Egress

  egress:

  # Allow DNS to kube-dns in kube-system

  - to:

    - namespaceSelector:

        matchLabels:

          kubernetes.io/metadata.name: kube-system  # Targets kube-system namespace

      podSelector:

        matchLabels:

          k8s-app: kube-dns  # Matches CoreDNS pods

    ports:

    - protocol: UDP

      port: 53  # DNS uses UDP port 53

    - protocol: TCP

      port: 53  # Fallback to TCP for large responses

  # Allow traffic to the data service on port 5678

  - to:

    - podSelector:

        matchLabels:

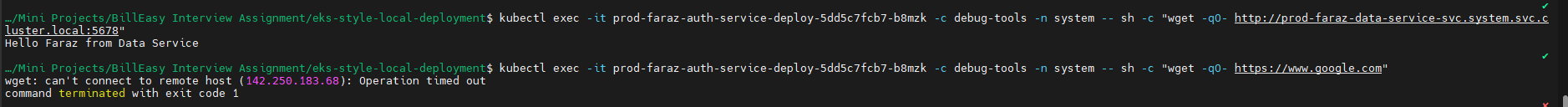
          app: faraz-data-service  # Data service pods in the same (system) namespace

    ports:

    - protocol: TCP

      port: 5678

Now we will be able to see that auth-service-deploy pods are able to only communicate to Data service but not external entities like Google:



Writing Kyverno policy to to block the authorization header leaks:

apiVersion: kyverno.io/v1

kind: ClusterPolicy

metadata:

  name: block-auth-header-logging

  annotations:

    policies.kyverno.io/title: Block Authorization Header Logging

    policies.kyverno.io/category: Security

    policies.kyverno.io/severity: high

spec:

  validationFailureAction: Enforce

  background: false

  rules:

  - name: check-for-auth-header-pattern

    match:

      any:

      - resources:

          kinds:

          - Deployment

    validate:

      message: "Authorization header logging detected. Remove '%({Authorization}i)s' pattern from environment variables."

      cel:

        expressions:

        - expression: >

            !object.spec.template.spec.containers.exists(container,

              container.?env.orValue([]).exists(envVar,

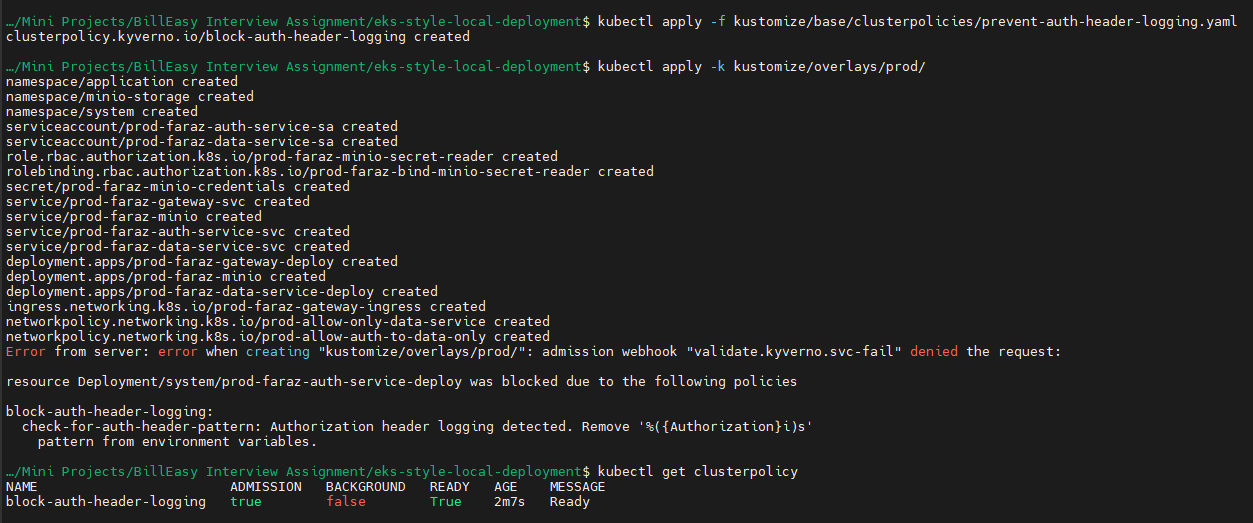
                envVar.?value.orValue('').contains('%({Authorization}i)s')

              )

            )

        - expression: "true"

Apply the cluster policy first the recreate your resources:



As you can see that other deployments work but deployment with leaking Authorization headers failed. Hence, cluster policy if working

Commenting leaky part and trying to create the auth-service deployment again:

If you notice that it created the deployment as we have commented the leaky part:

