Argo CD Project Documentation

Summer Internship Project Developed By: Ons **AROURI**

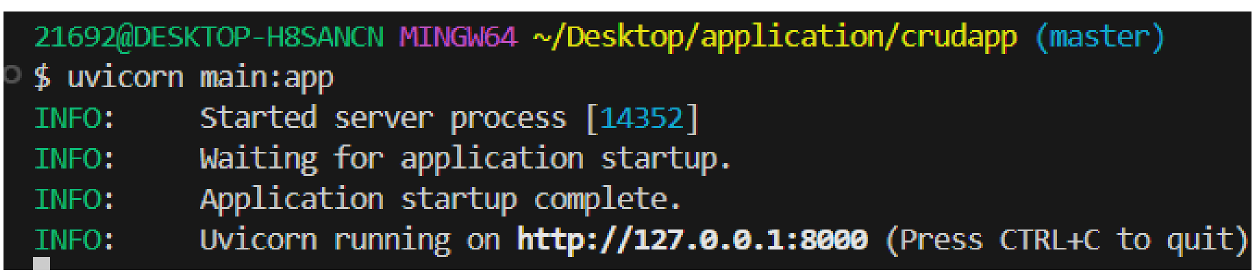
Overview

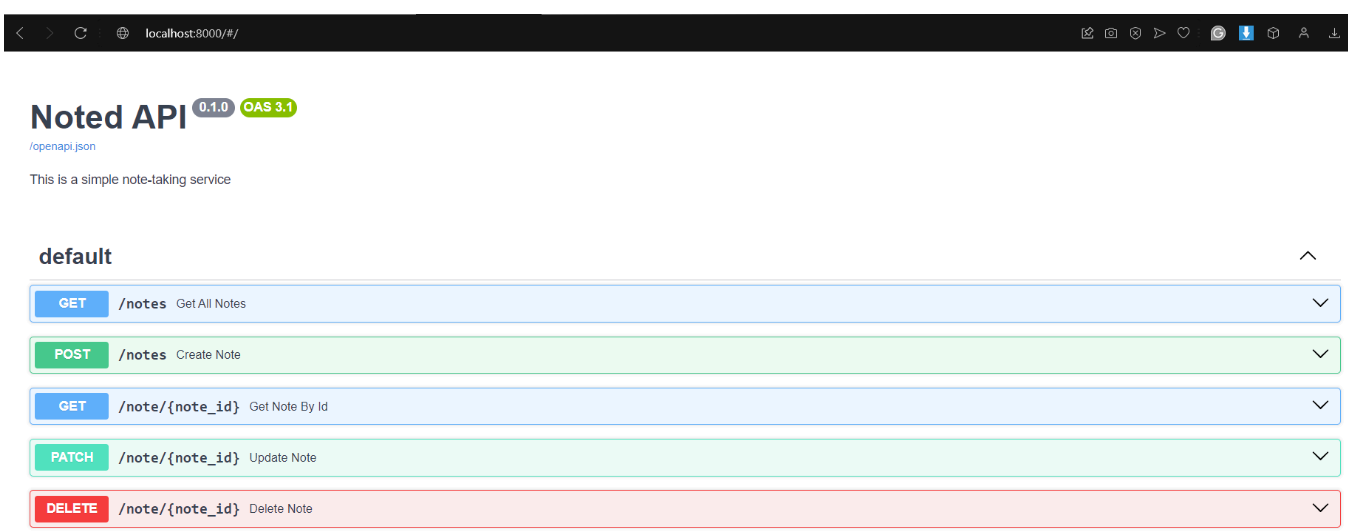
This document will present the journey of developing a FastAPI CRUD application and deploying it to Kubernetes using Docker, Helm, and Argo CD. This project was deployed in a Minikube cluster, with Argo CD for continuous deployment, and notifications set up to monitor the application's status.

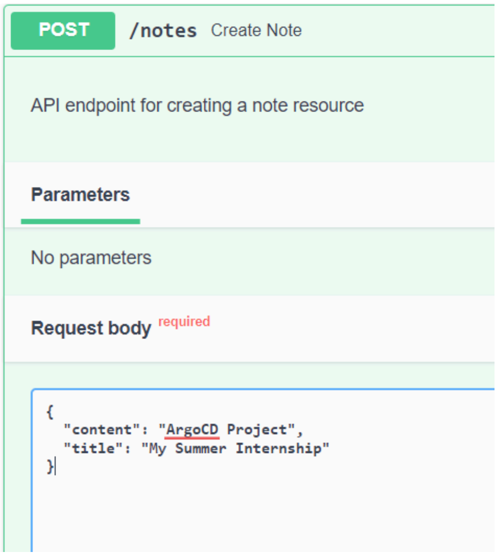
Project Workflow

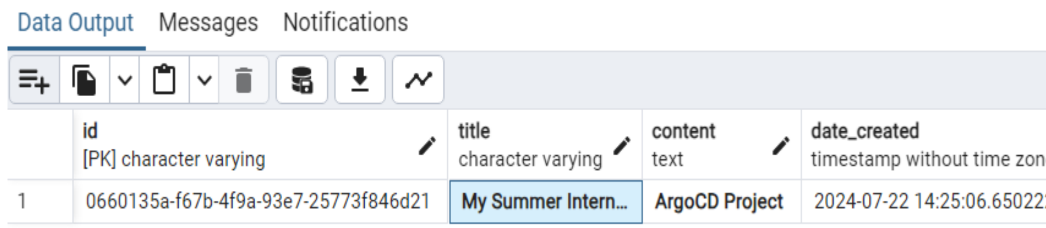
1- Developing the FastAPI CRUD Application

We have created a FastAPI application that handles basic CRUD operations.
Firstly, we developed it locally and tested it to ensure its proper functionality.
This application uses PostgreSQL as a database, and gets, adds, posts, patches, and deletes some notes inside the DB.









→ As shown, the application works as expected

<u>Prerequisites</u>

- Docker
- Minikube
- Kubectl: the command-line tool for interacting with the Kubernetes cluster
- Helm: the package manager for Kubernetes

2- Containerizing the Application with Docker

We have created a Dockerfile to containerize our application.

We've also created a docker-compose file to define and manage a multiservice environment, setting up both the PostgreSQL database and the application, enabling them to work together within the containerized setup.

Therefore, we built the docker image for the application:

```
$ docker build -t myapp-docker .
```

3- Deploying the Application on Minikube Using Helm

Pushing the docker image to Docker Hub ensures that it's publicly accessible for deployment in the Kubernetes cluster

```
$ docker push-t ons.arouri/myapp-docker:latest
```

We also set the context to Minikube to ensure that kubectl commands interact with the Minikube cluster:

```
$ kubectl config use-context minikube
```

Next are the elementary steps to deploy the app on the cluster:

1- Creating a new Helm chart structure

```
$ helm create myapp-docker
```

2- Updating the `values.yaml` file :

It serves as a centralized repository for all the configurable parameters of our application.
→ The centralization of these configurations allows us to deploy the same Helm chart in different environments (development, staging, production) with different settings by simply modifying the `values.yaml.`

We updated it with the image info, the namespace’s name, the port, the database credentials, and replicas.

3- Updating the `deployment.yaml` file :

The deployment for our application, and the database, manages the creation and updating of the application's pods. It ensures that the desired number of replicas is running at any given time.

We provided here:

- **Container Image:** our application runs within a Docker container.
- **Environment Variables:** the application needs to connect to a PostgreSQL database. Therefore, the `DATABASE_URL` environment variable includes the connection string to the database.
- **Startup Command:** the container runs a database initialization script (`create_db.py`) and then starts the FastAPI application using Uvicorn.
- **Ports:** the application listens on port 8000 within the container. This port is exposed to allow internal traffic within the Kubernetes cluster.

And for the PostgreSQL deployment:
- **Container Image:** the official PostgreSQL Docker image (`postgres:latest`).
- **Environment Variables:** `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`

- **Ports:** The database listens on port 5432 within the container.

4- Updating the `service.yaml` file:

The service exposes the application within the Minikube cluster, allowing it to be accessed by other services or users, and the service for PostgreSQL exposes the database to other services within the cluster, such as our application.

We specify inside:

- **Type:** `ClusterIP`, meaning it is only accessible within the cluster, useful for internal communication between services.
- **Port Mapping:** The service maps the internal port (8000) of the application to the same port, allowing the sending of HTTP requests to the application.

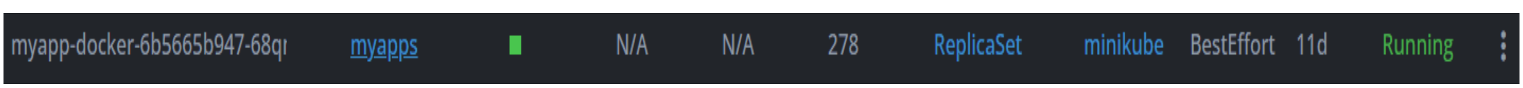
And for the PostgreSQL:

- **Type:** The service is also of type `ClusterIP`, making the database accessible only within the cluster.
- **Port Mapping:** mapping it to 5432 allows the application to connect to the database using the specified connection string.

5- Deploy the Helm Chart:

```
$ helm myapp-docker ./myapp-docker
```

And here is the application deployed successfully on the cluster:



To access the application, we need to:

```
$ kubectl port-forward -n myapps svc/myapp-docker 8000:8000
```

4- Installing and Configuring Argo CD:

We created a namespace called argocd where Argo CD and Argo CD Notifications Controller will both be installed.

```
$ kubectl create namespace argocd kubectl apply -n argocd -f
```

$https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

Pods 11 items Namespaces: argocd, myapps

Name	Namespace	Contai...	CPU	Memory	Restarts	Controlled By	Node	QoS	Age	Status
argocd-application-controller-0	argocd		N/A	N/A	11	StatefulSet	minikube	BestEffort	12d	Running
argocd-applicationset-controller	argocd		N/A	N/A	2	ReplicaSet	minikube	BestEffort	2d2h	Running
argocd-dex-server-55c6d584b5-z	argocd		N/A	N/A	11	ReplicaSet	minikube	BestEffort	12d	Running
argocd-notifications-controller-5	argocd		N/A	N/A	1	ReplicaSet	minikube	BestEffort	29h	Running
argocd-redis-5dcbd87d95-zzllg	argocd		N/A	N/A	11	ReplicaSet	minikube	BestEffort	12d	Running
argocd-repo-server-f9cb564f7-87	argocd		N/A	N/A	8	ReplicaSet	minikube	BestEffort	12d	Running
argocd-server-5d765d79c-kds6l	argocd		N/A	N/A	8	ReplicaSet	minikube	BestEffort	12d	Running

And now we have Argo CD running on our Kubernetes cluster.

To access the UI locally, we need to forward the Argo CD server port:

```
$ kubectl port-forward svc/argocd-server -n argocd 8080:443
```

```
$ kubectl port-forward -n argocd svc/argocd-server 8080:443
Forwarding from [::1]:8080 -> 8080
Handling connection for 8080
Handling connection for 8080
```

The credentials:

To retrieve the password, you can use this command:

```
$ kubectl get secret argocd-initial-admin-secret -n argocd -o yaml
```

```yaml
apiVersion: v1
data:
  password: A Base64 PASSWORD
kind: Secret
metadata:
  creationTimestamp: "2024-08-07T15:58:09Z"
  name: argocd-initial-admin-secret
  namespace: argocd
  resourceVersion: "210691"
  uid: 64f1b713-6814-4a59-9626-739b86d536
type: Opaque
```

Use admin as username.

Decode the password using:

```
$ echo 'your-password-in-base64' | base64 -d
```

The next step was creating the Argo CD application:

Application Manifest Overview:
Here, we specified:

Source: the source of the application's manifests.

- **repoURL**: The URL of the Git repository containing the Helm chart.
- **path**: The directory path within the repository where the Helm chart is located.
- **targetRevision**: The Git revision to deploy, specified as `HEAD` to use the latest commit on the default branch.
- **helm**:
    - **valueFiles**: Specifies the `values.yaml` file to use for customizing the Helm chart.

Destination: specifies where to deploy the application.

- **server**: The Kubernetes API server where the resources will be deployed. `https://kubernetes.default.svc` points to the in-cluster Kubernetes API.
- **namespace**: The Kubernetes namespace where the application will be deployed is `myapps`.

syncPolicy: Defines the synchronization policy for the application.

- **automated**: Enables automated synchronization, which includes:
    - **prune**: Automatically removes resources no longer defined in the Git repository.
    - **selfHeal**: Automatically corrects resources that drift from the desired state defined in the repository. (Any changes will revert to the version in the Git repo.
- **syncOptions**:
    - `CreateNamespace=true`: Ensures that the target namespace `myapps` is automatically created if it doesn't exist.

How it works:

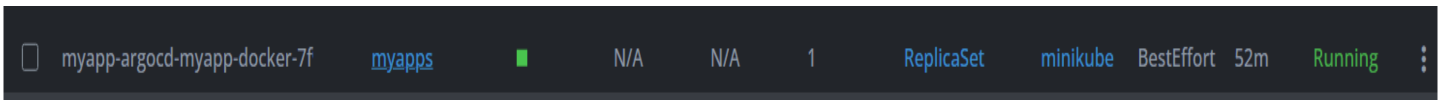
GitOps Workflow: The Argo CD application continuously monitors the specified Git repository. Whenever changes are detected (a new commit), Argo CD syncs these changes to the Kubernetes cluster, ensuring the deployed resources match the desired state defined in the repository.

Automated Deployment: With the sync policy set to automated, Argo CD will automatically deploy any changes without manual intervention. This includes creating the necessary namespace, applying updated configurations, and cleaning up any resources that are no longer needed (prune).

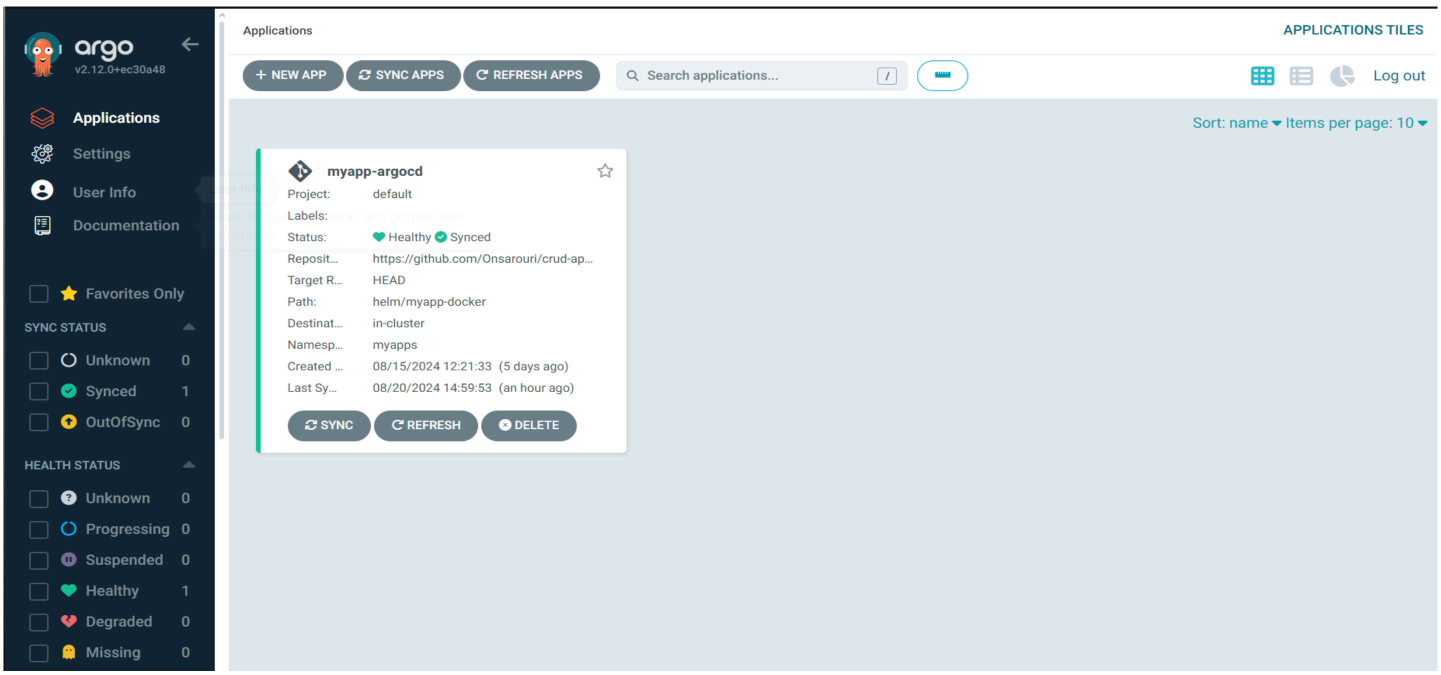
Self-Healing: If any resources in the cluster drift from their desired state (as defined in Git), Argo CD will automatically correct them, maintaining consistency.

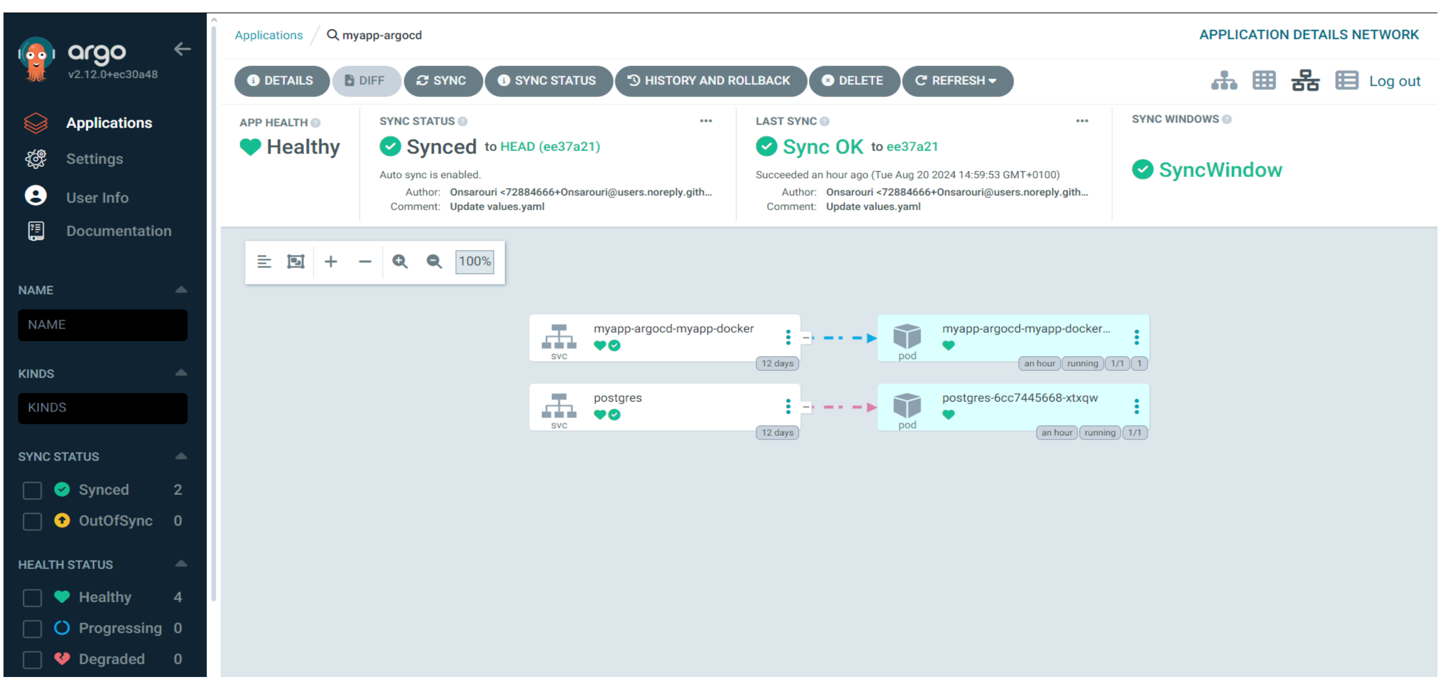
To apply the manifest:

```
$ kubectl apply -f application.yaml
```



And this is the Argo CD UI:





⇒ As shown, making a change inside the Git repository is detected and the Argo CD synchronizes the application inside the cluster to the desired state.

Therefore, we guaranteed that the only source of truth is the Git repository thanks to Argo CD.

5- Setting Up Argo CD Notifications:

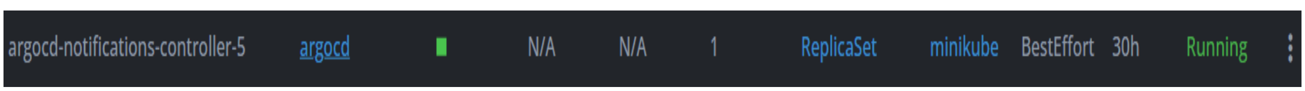
Purpose:

We want to receive an email once the status or health of the application changes, which means, once any changes occur from the Git repository, and the agent is trying to sync the application inside the cluster to the desired state.

1- Installing Argo CD notifications:

```
$ kubectl apply -n argocd -f
https://raw.githubusercontent.com/argoproj-labs/argocd-notifications/
stable/manifests/install.yaml
```

And now it's running on our cluster.



2- Configuring Email Notifications:

Argo CD Notifications is configured using a `ConfigMap` in the `argocd` namespace. This `ConfigMap` defines the triggers for various events and the templates for the email notifications. It sets up the email service details, in our case, using Gmail's SMTP server, and also sets up the subscriptions (the receivers and when they will receive the emails)

- Triggers: when the notification is sent, on which event (sync succeeded, sync failed, etc.)
- Templates: contains the email template according to each event.
- Email Service: contains the details such as the host (in our case SMTP server), port, username, sender's email, and password.

  → note that the password should be an App Password (Enable the 2FA of your Gmail account).

- Subscriptions: contain the recipients' emails, and on which event they will receive the notifications.

We can enter the secrets using this command:

```
$ kubectl create secret generic email-credentials \
  --from-literal=email-username=<email@gmail.com> \
  --from-literal=email-password=<email-app-password> \
  -n argocd
```

3- Set Up Role and RoleBinding:

Role and **RoleBinding** are essential components in Kubernetes for controlling access to resources within a cluster. They are part of Kubernetes' Role-Based Access Control (RBAC) system, which specifies who can do what on different resources.

Cross-Namespace Access:

In this project, the Argo CD Notifications Controller is running in the `argocd` namespace, and it needs to monitor the application deployed in `myapps namespace`. This introduces the need for cross-namespace access.

- Cross-Namespace RoleBinding:
    - To enable cross-namespace access, we bind a `Role` in the target namespace `myapps`, to a Service Account in the `argocd` namespace.
    - The `RoleBinding` is created in `myapps` namespace and refers to the Service Account in the `argocd` namespace. This allows the service account in `argocd` to access resources in `myapps`.
- Role:
    - A `Role` in the `myapps` namespace allows access to `applications` and `configmaps` resources.
    - This is necessary for monitoring application statuses and sending notifications based on those statuses.
- RoleBinding:
    - The `RoleBinding` in the `myapps` namespace binds the `Role` to the `argocd-notifications-controller` service account in the `argocd` namespace.
    - This setup allows the Argo CD Notifications Controller to perform actions such as `get`, `list`, and `watch`, on our application and configmaps in the `myapps` namespace.

Therefore, we created `argocd-notifications-role.yaml` file and it contains:

- **Role** defines a set of permissions on resources within a `myapps` namespace.

It also defines rules that specify:

- **API Groups**
- **Resources**: The specific resources the Role applies to : applications and configmaps.
- **Verbs**: The actions allowed on the resources: get, list, and watch.

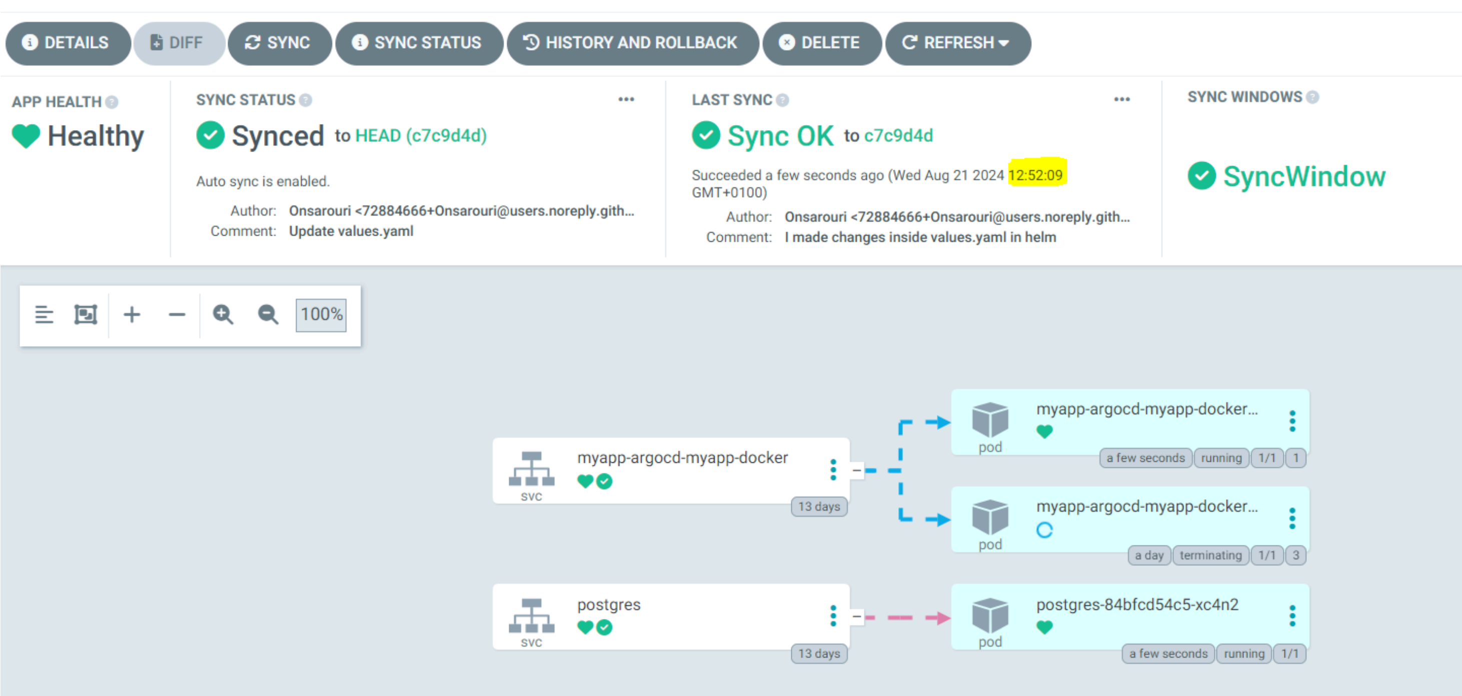
And we also defined the `argocd-notifications-rolebinding.yaml` file:

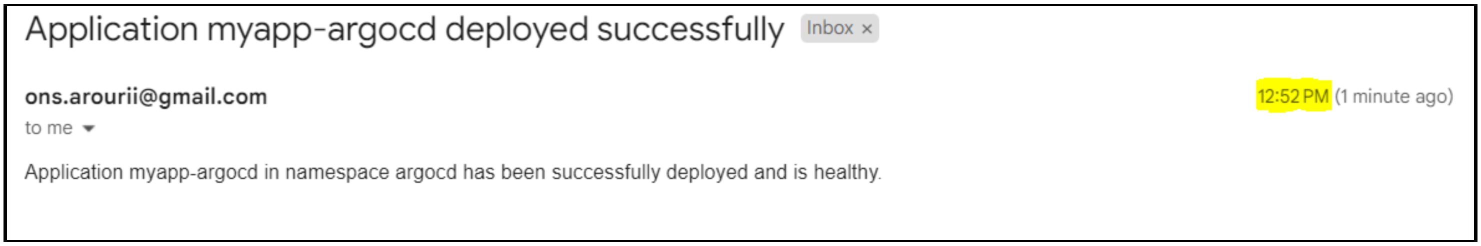
The **roleRef** section specifies that it grants permissions defined in the Role named argocd-notifications-role, which we defined in the myapps namespace.

In the subjects section, we defined the argocd-notifications-controller being granted these permissions, which is in the argocd namespace.

This setup allows the controller from the **argocd** namespace to interact with resources in the **myapps** namespace according to the permissions outlined in the referenced **Role**.

⇒ Now we have the notification controller allowed to access and manage resources across namespaces, and an email is sent once the changes inside the Git Repo are detected by our Argo CD.





Conclusion

This project outlines the deployment of a FastAPI CRUD application using Docker, Helm, and Argo CD in a Kubernetes environment (Minikube).

By containerizing the application, managing deployments with Helm, and automating updates with Argo CD, we've created a scalable and resilient system.

The integration of Argo CD Notifications ensures real-time monitoring, while Kubernetes Secrets securely manages sensitive data. This setup provides an efficient, automated, and secure deployment pipeline, ready for production use and future scaling.