Step 1: Installing software

First, we will need to install Docker (all our containers will be Docker ones), Kubectl (our command line for Kubernetes) and Kind (Kubernetes in Docker, in order to synchronize Kubernetes and Docker containers).

Step 2: Creating starting environment

We will create an initial cluster called challenge

```
kind create cluster --name challenge
```

This will:
Download a base image of kubernetes.
Create a Docker container that will do as Master (and worker).
Configure kubectl to work on this cluster.

Then we check that we got at least that Master node

```
asterixdecortes@Kubuntu:~/Descargas$ kubectl get nodes
NAME                        STATUS   ROLES           AGE     VERSION
challenge-control-plane   Ready    control-plane   3m32s   v1.33.1
```

We can also check that the pods needed are created

```
asterixdecortes@Kubuntu:~/Descargas$ kubectl get pods -A
NAMESPACE            NAME                                                READY   STATUS    RESTARTS   AGE
kube-system          coredns-674b8bbfcf-llz72                            1/1     Running   0          4m35s
kube-system          coredns-674b8bbfcf-sjtzw                            1/1     Running   0          4m35s
kube-system          etcd-challenge-control-plane                      1/1     Running   0          4m41s
kube-system          kindnet-qf7c4                                       1/1     Running   0          4m36s
kube-system          kube-apiserver-challenge-control-plane            1/1     Running   0          4m41s
kube-system          kube-controller-manager-challenge-control-plane   1/1     Running   0          4m41s
kube-system          kube-proxy-9c2hd                                    1/1     Running   0          4m36s
kube-system          kube-scheduler-challenge-control-plane            1/1     Running   0          4m41s
local-path-storage   local-path-provisioner-7dc846544d-qsjnh             1/1     Running   0          4m35s
```

If we needed to erase the cluster we do

```
kind delete cluster --name devops-challenge
```

Step 3: Developing Spring app (base for CI/CD)

Go to This page to start a Spring Boot (we will use the latest version and Java)
Add the dependencies
Spring Web
Spring Boot Actuator
Micrometer Prometheus

Then Download the .zip and unzip it, it will be your java project.
on the Application class that is created with a main.

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;

@SpringBootApplication
public class DemoTcsChallengeApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoTcsChallengeApplication.class, args);
    }

    @GetMapping("/hello")
    public String hello() {
        return "Hola desde Spring Boot!";
    }
}
```

Then we will need to expose an endpoint in /actuator/prometheus so that Prometheus can do its scrapping. To do so, on application.properties add these lines

```
management.endpoints.web.exposure.include=*
management.endpoint.prometheus.enabled=true
management.metrics.export.prometheus.enabled=true
```

Step 4: Docker Image

Then we will create a Dockerfile on the root of the project:

```
# First step. compile using Maven
FROM eclipse-temurin:17-jdk-alpine AS builder

WORKDIR /app

# Copy Maven config and source
COPY pom.xml ./
COPY src ./src

# Download dependencies and build jar
RUN apk add --no-cache maven && \
    mvn clean package -DskipTests
```

```
# Second step. run only image
FROM eclipse-temurin:17-jre-alpine

WORKDIR /app

# Copy the compiled JAR from the previous stage
COPY --from=builder /app/target/*.jar app.jar

# Expose port
EXPOSE 8080

# Run the app
ENTRYPOINT ["java", "-jar", "app.jar"]
```

You can try it later using

```
sudo docker build -t challenge_image .
sudo docker run -p 8080:8080 challenge_image
```

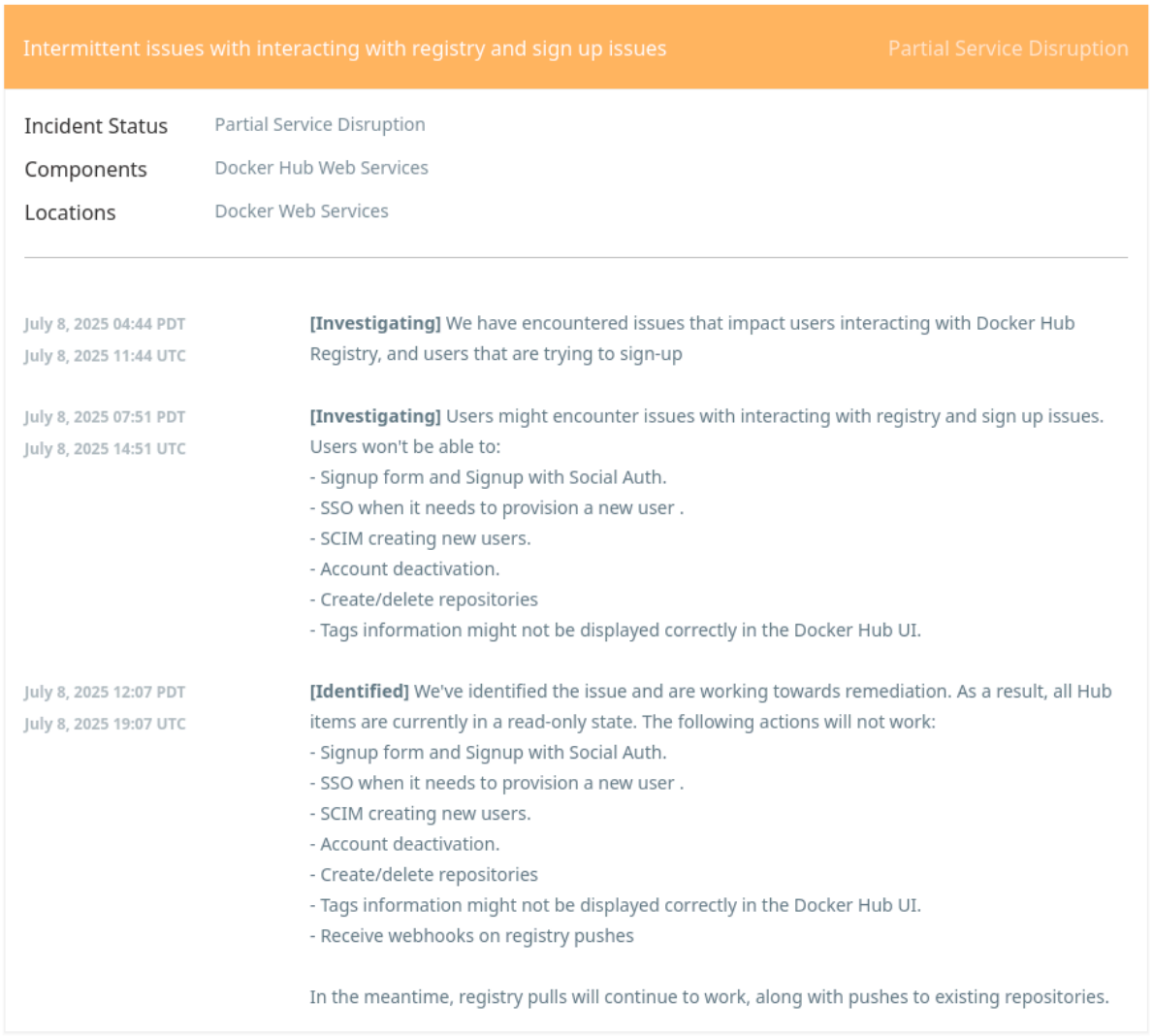
Step 4: Automatize CI

Created a folders in the project called /.github/workflows, inside we create a .yml file where we will have to write all the actions we want github to do on a event

```
name: Build and Push Docker Image

# This will activate only when push on master branch
on:
  push:
    branches:
      - master

# Create a job called build-and-push that will run on the latest ubuntu
version
jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
      # This first step will download the source code from the repo
      - name: Checkout source code
        uses: actions/checkout@v4
```

```
      # Second step that will install java
      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '17'

      # This will store all Maven dependencies
      - name: Cache Maven packages
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: |
            ${{ runner.os }}-m2

      # Next step where it will compile the project
      - name: Build with Maven
        run: mvn clean package -DskipTests

      # Login on Github Container Registry using github secrets
      - name: Log in to Github Container Registry
        run: echo "${{ secrets.GHCR_PAT }}" | docker login ghcr.io -u
${{ github.actor }} --password-stdin

      # Create image using Dockerfile with the name challenge_image
      - name: Build and tag Docker image
        run: docker build -t ghcr.io/${{ github.repository_owner
}}/challenge_image:latest .

      # Upload the Docker image
      - name: Push Docker image
        run: docker push ghcr.io/${{ github.repository_owner
}}/challenge_image:latest
```

Then we can push it to our github.
After it we need to go to settings -> secrets and variables -> Actions -> New repository secret
There we will add the variables used in the .yml (GHCR_PAT), before we had to create the Personal Access Token inside github.
(took this path instead of uploading to dockerhub as it was down at the time)



Intermittent issues with interacting with registry and sign up issues — Partial Service Disruption

Incident Status	Partial Service Disruption
Components	Docker Hub Web Services
Locations	Docker Web Services

July 8, 2025 04:44 PDT
July 8, 2025 11:44 UTC

[Investigating] We have encountered issues that impact users interacting with Docker Hub Registry, and users that are trying to sign-up

July 8, 2025 07:51 PDT
July 8, 2025 14:51 UTC

[Investigating] Users might encounter issues with interacting with registry and sign up issues. Users won't be able to:
- Signup form and Signup with Social Auth.
- SSO when it needs to provision a new user .
- SCIM creating new users.
- Account deactivation.
- Create/delete repositories
- Tags information might not be displayed correctly in the Docker Hub UI.

July 8, 2025 12:07 PDT
July 8, 2025 19:07 UTC

[Identified] We've identified the issue and are working towards remediation. As a result, all Hub items are currently in a read-only state. The following actions will not work:
- Signup form and Signup with Social Auth.
- SSO when it needs to provision a new user .
- SCIM creating new users.
- Account deactivation.
- Create/delete repositories
- Tags information might not be displayed correctly in the Docker Hub UI.
- Receive webhooks on registry pushes

In the meantime, registry pulls will continue to work, along with pushes to existing repositories.

Step 5: Creating K8s files

We will need to create a k8s folder and inside it a deployment.yaml and service.yaml files.
deployment.yaml

```
# This file will create Deployment, it will grant that our app will run
in a image and will always have at least one copy running
apiVersion: apps/v1
kind: Deployment
# Give the object a name
metadata:
  name: demo-app
  labels:
    app: demo-app
spec:
  # Only need 1 instance, we can change it to scalate
```

```
  replicas: 1
  # Which pods need to be part of the Deployment
  selector:
    matchLabels:
      app: demo-app
  # Template of every copy or pod running
  template:
    metadata:
      labels:
        app: demo-app
    # Defines main container
    spec:
      containers:
        - name: demo-app
          image: ghcr.io/asterixdecortes/challenge_image:latest
          ports:
            - containerPort: 8080
```

service.yaml

```
# This file creates a service, allowing the app to communicate
apiVersion: v1
kind: Service
# Name for the service
metadata:
  name: demo-service
# Asign the service with anything matching the label demo-app
spec:
  selector:
    app: demo-app
  # This service will work on tcp, using the port 80 internally but
redirecting to 8080 and exposing to 9090
  ports:
    - protocol: TCP
      port: 80  # internal port for this service
      targetPort: 8080  # listening port for the container
      nodePort: 30080    # exposed port to access
  type: NodePort
```

Then we run

```
kubectl apply -f k8s/deployment.yaml
kubectl apply -f k8s/service.yaml
```

So our app is running now on kubernetes, we can check it with

```
kubectl get pods
```

Step 6: ArgoCD

We need to install ArgoCD in our kubernetes environment

```
kubectl create namespace argocd
kubectl apply -n argocd -f
https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/inst
all.yaml
```

Then check it is installed using

```
kubectl get pods -n argocd
```

Now we can run the web panel using port forwarding

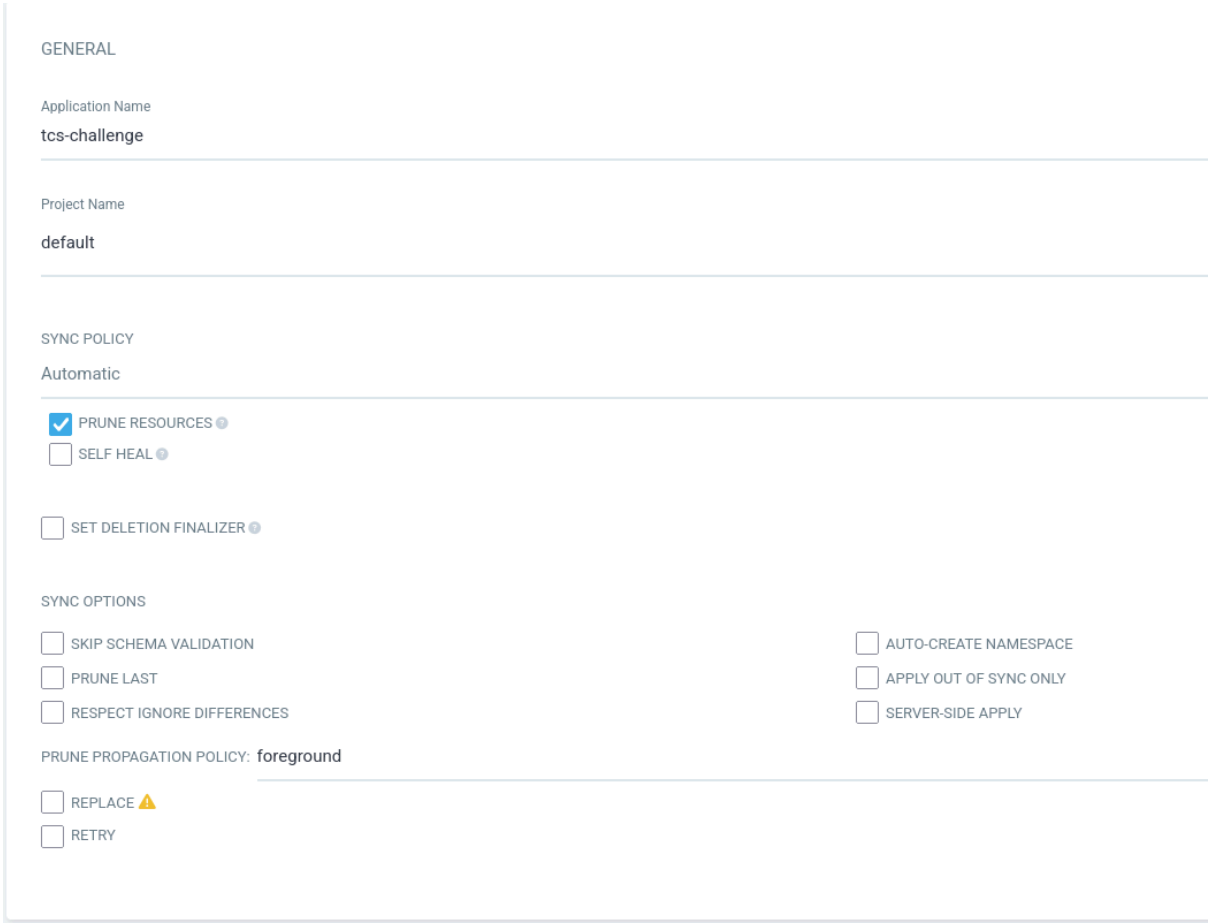
```
kubectl port-forward svc/argocd-server -n argocd 8081:443
```

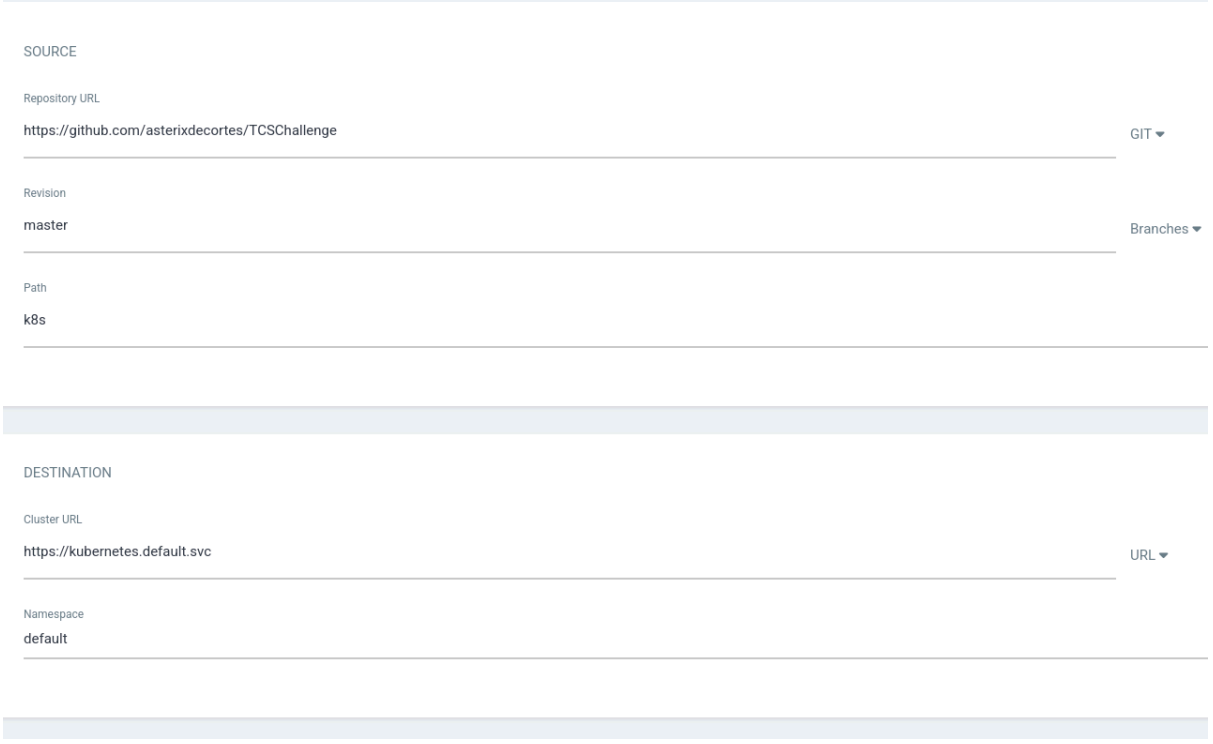
We need to get the admin user and password using (default user is admin)

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

On the web panel, we can configure the CD so that

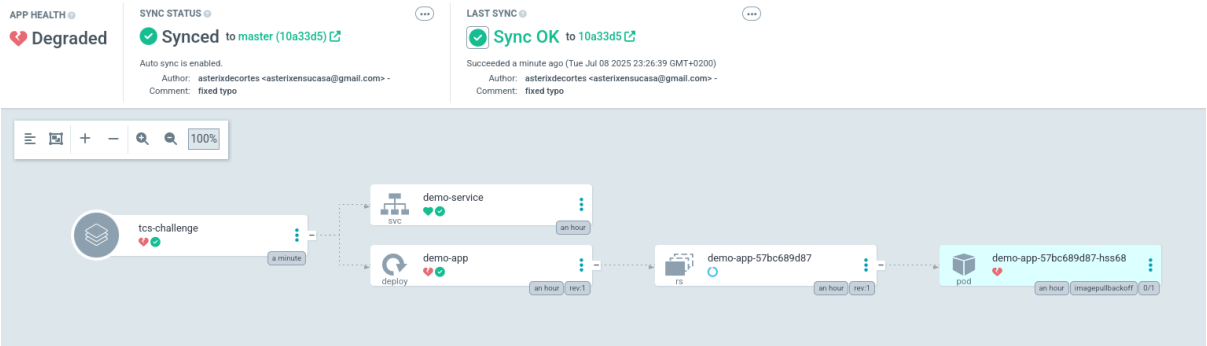
- ArgoCD is always checking my git repo.
- Goes to check my k8s YAMLs.
- Applies them to the k8s cluster inside the namespace.
- If auto Sync is enabled, whenever I push into my GitHub repo, it will update the k8s cluster.





It can also be done configuring a YAML.

Then you can check the tree it creates (CHANGE IMAGE).



Right now it works like this:

- Upload code to GitHub
- GitHub Actions will create and upload a new image to DockerHub
- ArgoCD will see the change on your repo
- ArgoCD will update your kubernetes cluster

Step 7: Monitoring with Prometeus+Grafana

We will need helm to manage apps for kubernetes

```
curl
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |
bash
```

Then we can get prometheus + grafana

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
```

And install them in the cluster

```
helm install kps prometheus-community/kube-prometheus-stack --namespace
monitoring --create-namespace
```

After that we can check that everything is installed

```
kubectl get pods -n monitoring
```

To acces the web panel for grafana we need port forwarding

```
kubectl port-forward svc/kps-grafana -n monitoring 3000:80
```

Then we will need to get the admin password

```
kubectl get secret kps-grafana -n monitoring -o
jsonpath="{.data.admin-password}" | base64 -d && echo
```

If we go to our deployment.yaml and add the following lines, prometheus will target our app and get data from there.

```
metadata:
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: /actuator/prometheus
    prometheus.io/port: "8080"
```

Step 8: Terraform IaC

Create a terraform folder inside the project, there create a main.tf Mine will look like this as it will only need k8s apps but it can be configured to build any cloud instances like AWS EC2.

main.tf

```
terraform {
    # This will tell terraform what plugins it needs to work
    required_providers {
        # To communicate with the cluster
        kubernetes = {
            source  = "hashicorp/kubernetes"
            version = "2.27.0"
        }
        # To install apps needed for the cluster
        helm = {
            source  = "hashicorp/helm"
            version = "2.13.1"
        }
    }
}

# How will terraform connect to the cluster
provider "kubernetes" {
    config_path = "~/.kube/config"
}

# This will tell Terraform to use helm on kubernetes
```

```
provider "helm" {
    kubernetes {
        config_path = "~/.kube/config"
    }
}

# This will install Argocd using Helm inside argocd namespace
resource "helm_release" "argocd" {
    name       = "argocd"
    namespace  = "argocd"
    create_namespace = true

    repository = "https://argoproj.github.io/argo-helm"
    chart      = "argo-cd"
    version    = "5.51.6"

    values = [
        file("${path.module}/argocd-values.yaml")
    ]
}

# This will install Prometheus + Grafana using Helm inside monitoring
namespace
resource "helm_release" "monitoring" {
    name       = "kps"
    namespace  = "monitoring"
    create_namespace = true

    repository = "https://prometheus-community.github.io/helm-charts"
    chart      = "kube-prometheus-stack"
    version    = "57.0.0"

    values = [
        file("${path.module}/monitoring-values.yaml")
    ]
}
```

Then we need
argocd-values.yaml

```
server:
  service:
    type: NodePort
    nodePortHttp: 31080
    nodePortHttps: 31443
```

monitoring-values.yaml

```
grafana:
  service:
    type: NodePort
  adminPassword: admin

prometheus:
  service:
    type: NodePort
```

then we can execute inside the terraform folder

```
terraform init
terraform plan
terraform apply
```