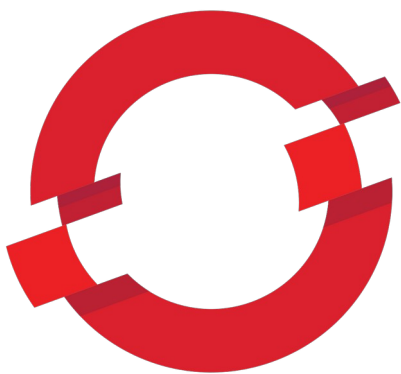


Proyecto Despliegue Continuo Openshift



OPENS SHIFT

X



Índice

Fundamentos teóricos.....	3
¿Que es Openshift?.....	3
¿Que es Tekton?.....	3
¿ A que nos referimos con “Despliegue continua ?”	4
Conceptos específicos.....	4
Pipelines.....	4
Tareas.....	5
Git y webhooks.....	5
Escenario necesario para la realización del proyecto.....	5
Instalaciones y configuraciones.....	6
Comenzar en Openshift.....	6
.....	9
Backend.....	9
Dockerfile.....	9
Deployment.....	10
Service.....	11
Frontend.....	12
Dockerfile.....	12
Deployment.....	13
Route.....	14
Tasks.....	15
Pipeline.....	19
Triggers.....	25
Trigger template.....	25
Trigger binding.....	26
Trigger interceptor.....	27
EventListener.....	28
Git Webhook.....	29
Comprobación despliegue continuo con webhook.....	31
Tolerancia a fallos, Escalabilidad y Balanceo de carga.....	34
Autoescalado.....	37
Rollback.....	39
SonarQube.....	41
Conclusiones y propuestas.....	50
ArgoCD.....	50
SonarQube.....	50
Terraform.....	50

Conclusión.....	51
Dificultades e inconvenientes.....	51
Limitaciones.....	51
Poca documentación.....	51
Bibliografía.....	52

Fundamentos teóricos

¿Que es Openshift?

OpenShift es una plataforma de contenedores de código abierto que permite a los desarrolladores crear, implementar y administrar aplicaciones en contenedores. Es desarrollado por Red Hat y se basa en Kubernetes, un sistema de orquestación de contenedores de código abierto.

OpenShift facilita la implementación y la gestión de aplicaciones en contenedores al proporcionar herramientas integradas para la construcción, el despliegue, la escalabilidad y la recuperación de aplicaciones. También proporciona una interfaz de usuario web y una CLI (interfaz de línea de comandos) para la gestión de aplicaciones y la configuración de la infraestructura.

¿Que es Tekton?

Tekton es un marco de trabajo de código abierto para automatizar la construcción, prueba y despliegue de aplicaciones en contenedores. Fue desarrollado por Google, Red Hat y otros contribuyentes de la comunidad de software libre. Tekton se basa en la ejecución de tareas y pipelines, las tareas se describen en archivos

YAML y se ejecutan en un entorno aislado (Contenedores). Las tareas son las unidades básicas de trabajo en Tekton y se pueden definir para realizar acciones como la construcción de contenedores, la ejecución de pruebas, la implementación y el despliegue de aplicaciones. En resumen, Tekton es un marco de trabajo de automatización de CI/CD de código abierto que se basa en la ejecución de tareas y tuberías de construcción de forma declarativa. Permite la integración continua y la entrega continua de aplicaciones en contenedores de forma escalable y flexible.

¿ A que nos referimos con “Despliegue continuo ?”

El Despliegue Continuo (CD) es una práctica de desarrollo de software que implica la entrega de software a producción de manera rápida y automática.

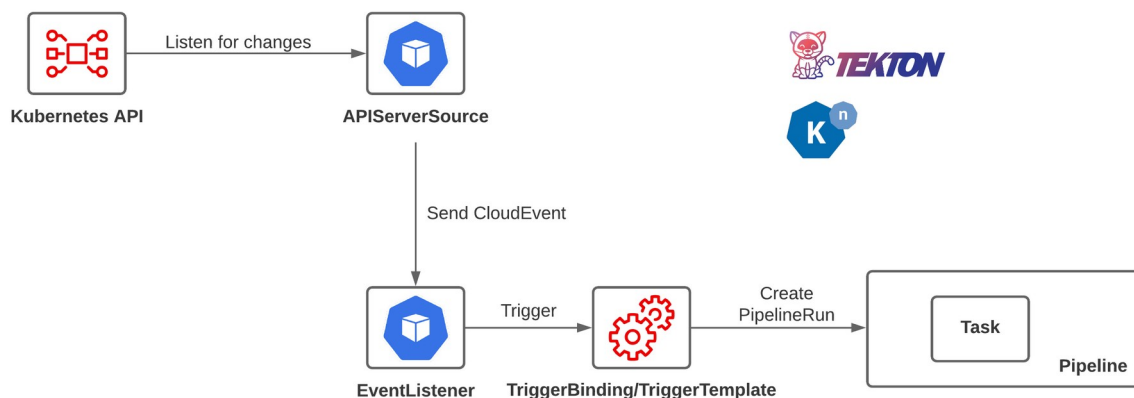
Básicamente, el Despliegue Continuo se refiere a la automatización de todo el proceso de entrega de software, desde la integración del código fuente hasta la implementación en producción. El objetivo es entregar nuevas funcionalidades y correcciones de errores de manera rápida y confiable a los usuarios finales.

El Despliegue Continuo se basa en la Integración Continua (CI), que garantiza que el código fuente esté siempre en un estado de trabajo y que se pruebe y verifique de manera continua. Una vez que se realiza la integración y se ejecutan las pruebas, el código fuente se implementa automáticamente en producción.

Conceptos específicos

Pipelines

Un pipeline es un conjunto de tareas que se realizan para hacer un despliegue de una aplicación. Las tareas más comunes son (git clones, construcción de imagen, testeo de la app, despliegue de la app ...) Los pipelines están pensados para que actúen como disparadores, por lo que tendremos que configurar (en caso de Tekton) una serie de triggers para qe cuando hagamos un push, el pipeline se ejecute en nuestro Openshift



Tareas

Como he comentado antes, varias tareas conforman un pipeline. Las tareas o bien las podemos crear nosotros, o podemos utilizar las tareas que vienen instaladas predeterminadas en Tekton o las que crea la comunidad.

Las tareas predeterminadas más utilizadas son:

- s2i (Source to image)
- bulildah (Construir imagen a partir de un Dockerfile)
- fetch-repository (Clonar repositorio para contruir la imagen)

Git y webhooks

Git es una herramienta de control de versiones muy popular que se utiliza en muchos proyectos de software para gestionar y controlar el código fuente. En conjunto con OpenShift, Git puede ser utilizado para implementar el Despliegue Continuo.

Se puede configurar un webhook en el repositorio Git para notificar a OpenShift cuando se realice un cambio en el repositorio. Esto permitirá que OpenShift inicie automáticamente la compilación y el despliegue del código fuente en los contenedores de la aplicación.

Escenario necesario para la realización del proyecto

Como tal no tengo un escenario físico ya que como he comentado en varias ocasiones, mi versión gratuita de Openshift está en la nube.

Por otra parte me gustaría indicar la estructura que tengo en git para la realización de este proyecto

Repositorio backend de mi app: <https://github.com/apalgar24/pipelines-vote-api>

Aquí debemos de almacenar el código fuente del backend, el Dockerfile y el deployment, service y route necesarios.

Repositorio frontend de mi app: <https://github.com/apalgar24/pipelines-vote-api>

Aquí debemos de almacenar el código fuente del frontend el Dockerfile y el deployment, service y route necesarios.

Repositorio donde tengo almacenado mis pipelines y triggers :
https://github.com/apalgar24/pipelines-triggers/tree/master/01_pipeline

Instalaciones y configuraciones

Comenzar en Openshift

En mi caso, como ya adelanté en mi Pre-Proyecto, usaré una prueba gratuita que ofrece Openshift con las siguientes características:

Ofrece un entorno de desarrollo integrado (IDE) basado en EclipseChe, plantillas Charts de Helm, imágenes del compilador de Red Hat, acceso a Git, la herramienta de diseño S2I.

Para poder conseguir esta edición de pruebas de Openshift, debemos de dirigirnos a la siguiente URL <https://www.redhat.com/es/technologies/cloud-computing/openshift/try-it>

Debemos de tener una cuenta registrada en Red Hat y estar logueados para poder proceder con la obtención de la edición gratuita.

Clickamos en el botón “Comience Prueba”

Espacio aislado para los desarrolladores

Acceso instantáneo a su propio entorno mínimo configurado previamente para desarrollar y probar aplicaciones

Comience la prueba

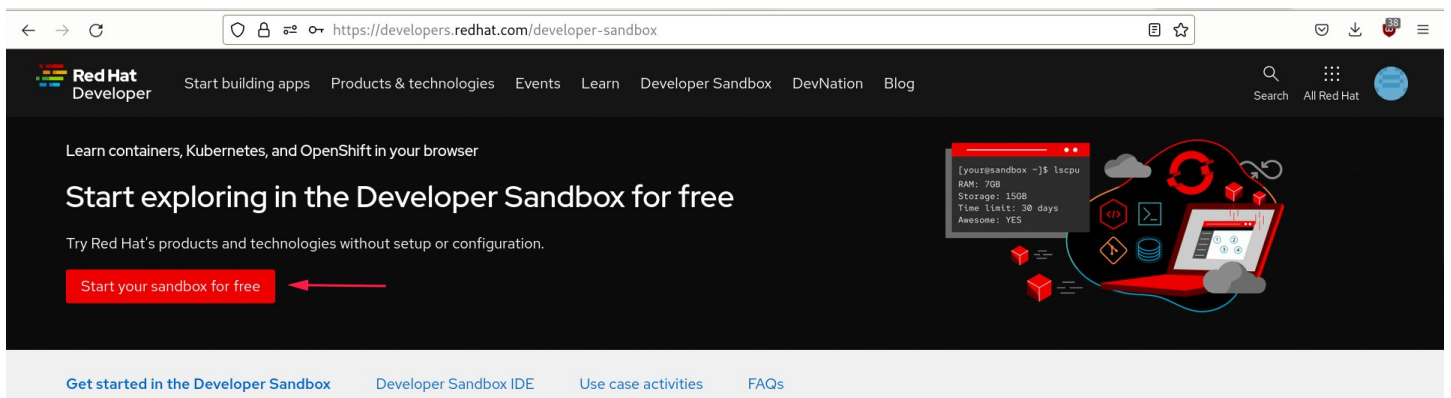
Costo: gratuito

Características y aspectos destacados:

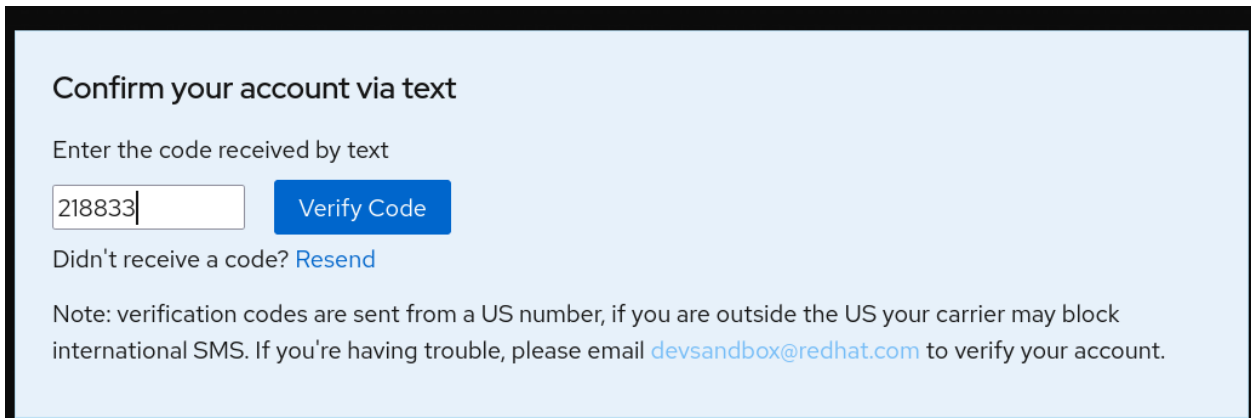
Además de estar diseñado para los desarrolladores, ofrece un entorno de desarrollo integrado (IDE) basado en Eclipse Che, plantillas Charts de Helm, imágenes del compilador de Red Hat, acceso a Git, la herramienta de diseño

Nos redirigirá a otra página de Red Hat

Luego clickamos en el botón donde dice “Start your sandbox for free”

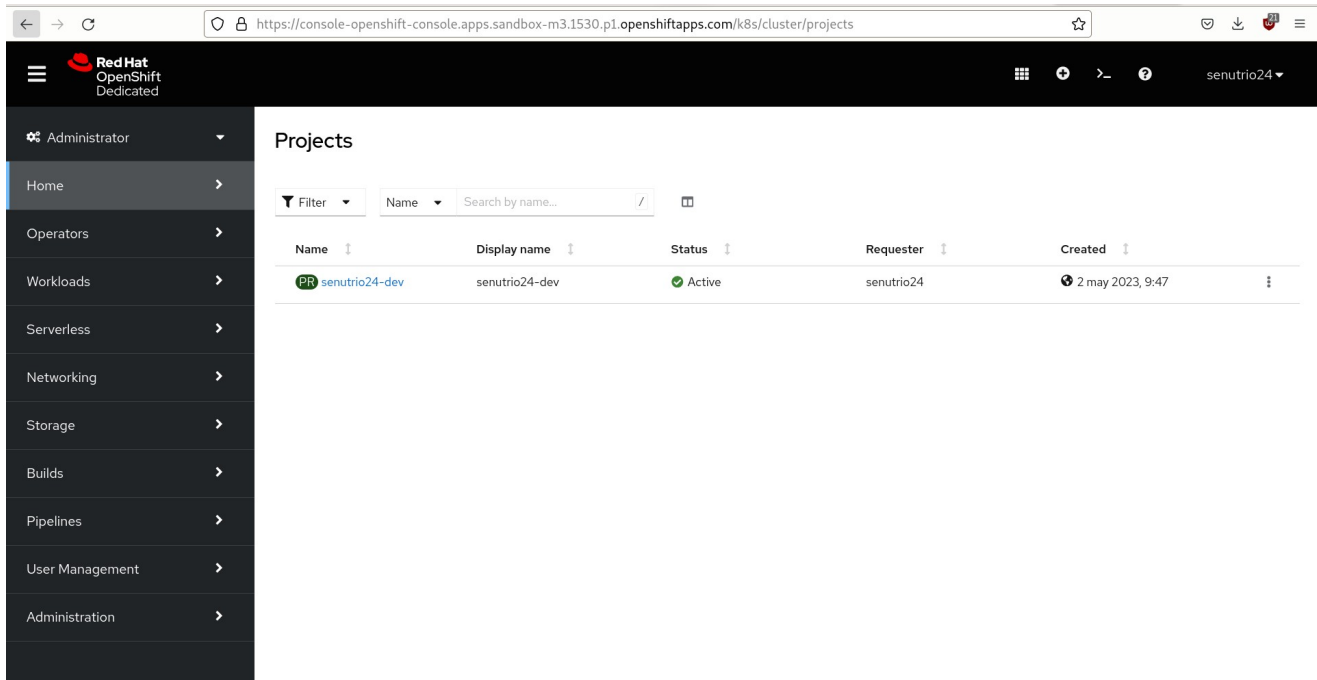


Por último, ponemos un código de verificación que nos enviarán al correo de nuestra cuenta registrada en su plataforma web.



The screenshot shows a light blue rectangular form with a black border. At the top left, the heading "Confirm your account via text" is displayed. Below it, the instruction "Enter the code received by text" is shown. A text input field contains the number "218833" with a vertical cursor at the end. To the right of the input field is a blue button with the text "Verify Code". Below the input field, there is a link: "Didn't receive a code? [Resend](#)". At the bottom of the form, a note reads: "Note: verification codes are sent from a US number, if you are outside the US your carrier may block international SMS. If you're having trouble, please email devsandbox@redhat.com to verify your account."

Una vez introduzcamos el código de verificación, por fin tendremos Openshift listo para usar



Backend

Antes de comenzar con lo bueno, debemos de preparar los dos repositorios de nuestra aplicación para su posterior despliegue.

Todo el contenido backend lo tendré en el siguiente repositorio:

<https://github.com/apalgar24/pipelines-vote-api.git>

Al tratarse de un backend, no necesitaremos un route para exponer la aplicación.

Dockerfile

Lo primero es tener un Dockerfile en la raíz de nuestro repositorio.

```
FROM  
image-registry.openshift-image-registry.svc:5000/openshift/golang:  
latest as builder
```

```
WORKDIR /build
```

```
ADD . /build/
```

```
RUN mkdir /tmp/cache
```

```
RUN CGO_ENABLED=0 GOCACHE=/tmp/cache go build -mod=vendor -v -o /tmp/api-server .
```

```
FROM scratch
```

```
WORKDIR /app
```

```
COPY --from=builder /tmp/api-server /app/api-server
```

```
CMD [ "/app/api-server" ]
```

Deployment

Dentro de nuestra carpeta k8s almacenaremos nuestro deployment, service del backend.

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  labels:
```

```
    app: pipelines-vote-api
```

```
    name: pipelines-vote-api
```

```
spec:
```

```
  replicas: 1
```

```
  selector:
```

```
  matchLabels:
    app: pipelines-vote-api
template:
  metadata:
    labels:
      app: pipelines-vote-api
  spec:
    containers:
      - image: quay.io/openshift-pipeline/vote-api:latest
        imagePullPolicy: Always
        name: pipelines-vote-api
        ports:
          - containerPort: 9000
            protocol: TCP
```

Hay que tener en cuenta que demos de poner el mismo nombre de la imagen que vamos a definir luego en la ejecución de nuestro pipeline, al igual que el nombre del deployment

Service

Ahora crearemos un service que servirá para que el backend se comuniquen con el frontend

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: pipelines-vote-api
    name: pipelines-vote-api
spec:
  type: ClusterIP
```

```
ports:
  - protocol: TCP
    port: 9000
    targetPort: 9000
selector:
  app: pipelines-vote-api
```

Frontend

El repositorio del frontend de mi app será : <https://github.com/apalgar24/pipelines-vote-ui>

Dockerfile

```
# Using official python runtime base image
FROM
image-registry.openshift-image-registry.svc:5000/openshift/python:
latest

# Install our requirements.txt
ADD requirements.txt /opt/app-root/src/requirements.txt
RUN pip install -r requirements.txt

# Copy our code from the current folder to /app inside the
container
ADD . /opt/app-root/src

# Make port 80 available for links and/or publish
EXPOSE 8080
```

```
# Define our command to be run when launching the container
#CMD ["gunicorn", "app:app", "-b", "0.0.0.0:8080", "--log-file",
"-", "--access-logfile", "-", "--workers", "4", "--keep-alive",
"0"]
CMD ["python", "./app.py"]
```

Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: pipelines-vote-ui
  name: pipelines-vote-ui
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pipelines-vote-ui
  template:
    metadata:
      labels:
        app: pipelines-vote-ui
    spec:
      containers:
        - image: quay.io/openshift-pipeline/vote-ui:latest
          imagePullPolicy: Always
          name: pipelines-vote-ui
          ports:
```

```
- containerPort: 8080
  protocol: TCP
- containerPort: 9090
  protocol: TCP
env:
- name: VOTING_API_SERVICE_HOST
  value: pipelines-vote-api
- name: VOTING_API_SERVICE_PORT
  value: "9000"
```

Muy importante indicar el mismo puerto que usa el backend para que se comuniquen, en este caso 9000

Route

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  labels:
    app: pipelines-vote-ui
    name: pipelines-vote-ui
spec:
  port:
    targetPort: 8080-tcp
  to:
    kind: Service
    name: pipelines-vote-ui
    weight: 100
```

En este caso si necesitaremos de un route para mostrar la aplicación ya que se trata del frontend

Tasks

Como ya sabemos, Tekton ya viene instalado sobre OpenShift por lo cual no hace falta explicar la instalación de Tekton.

Hay dos maneras para trabajar con Tekton, desde la interfaz gráfica desde la terminal con el CLI de Tekton “tkn”. Personalmente prefiero trabajar con la terminal y monitorizar la ejecución del pipeline por interfaz gráfica

Nuestro pipeline se compondrá de momento de 4 tareas (1º Git-clone 2º Construcción de imagen 3º aplicar despliegue 4º En caso de nueva versión, actualizar despliegue). Las dos últimas tareas tendremos que crearlas nosotros ya que las dos primeras son predeterminadas de Tekton

Comenzaremos creando dos tareas que servirán para hacer el despliegue de nuestra aplicación, es decir empezaremos creando las últimas dos tareas de nuestro pipeline

```
  apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: apply-manifests
spec:
  workspaces:
    - name: source
  params:
    - name: manifest_dir
      description: The directory in source that contains yaml
      manifests
      type: string
      default: "k8s"
  steps:
    - name: apply
```

```

    image:
image-registry.openshift-image-registry.svc:5000/openshift/cli:latest
    workingDir: /workspace/source
    command: ["/bin/bash", "-c"]
    args:
      - |-
        echo Applying manifests in $(inputs.params.manifest_dir)
directory
        oc apply -f $(inputs.params.manifest_dir)
        echo -----

```

En mi caso crearé la tarea desde mi terminal ya que tengo los ficheros subidos en un repositorio de github.

```
oc create -f
https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/01\_pipeline/01\_apply\_manifest\_task.yaml
```

Seguimos creando la segunda tarea

```

apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: update-deployment
spec:
  params:
    - name: deployment
      description: The name of the deployment patch the image
      type: string
    - name: IMAGE

```



```

    description: Location of image to be patched with
    type: string
  steps:
  - name: patch
    image:
image-registry.openshift-image-registry.svc:5000/openshift/cli:latest
    command: ["/bin/bash", "-c"]
    args:
    - |-
      oc patch deployment $(inputs.params.deployment) --
patch='{"spec":{"template":{"spec":{"
  "containers":[{"
    "name": "$(inputs.params.deployment)",
    "image":"$(inputs.params.IMAGE)"
  }]}
}}}'

    # issue: https://issues.redhat.com/browse/SRVKP-2387
    # images are deployed with tag. on rebuild of the image
tags are not updated, hence redeploy is not happening
    # as a workaround update a label in template, which
triggers redeploy pods
    # target label:
"spec.template.metadata.labels.patched_at"

    # NOTE: this workaround works only if the pod spec has
imagePullPolicy: Always
    patched_at_timestamp=`date +%s`
    oc patch deployment $(inputs.params.deployment) --
patch='{"spec":{"template":{"metadata":{"
  "labels":{"

```

```
        "patched_at": '\ "$patched_at_timestamp\ "'
    }
}}}}'
```

```
oc create -f
https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/01_pipeline/02_update_deployment_task.yaml
```

Ahora podremos listar las tareas que tenemos definidas en nuestro Tekton

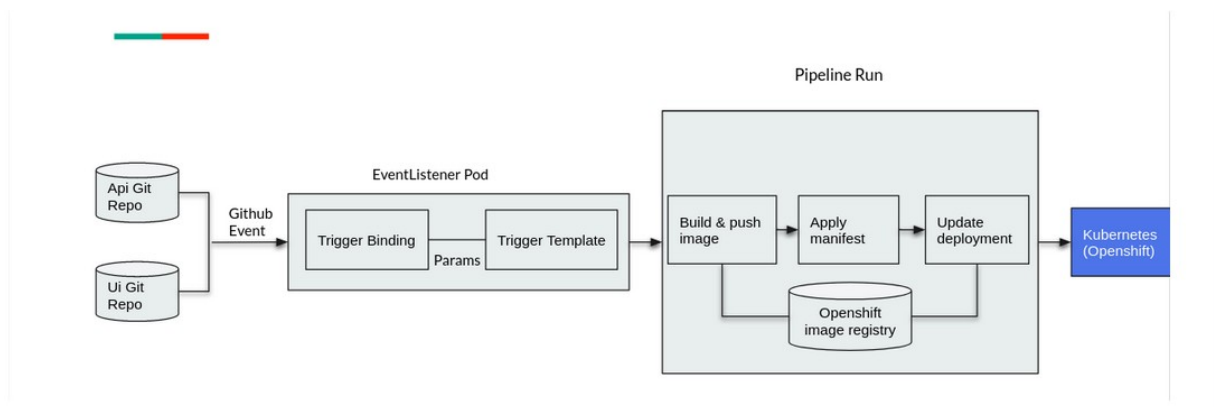
```
$ tkn task list
```

NAME	DESCRIPTION	AGE
apply-manifests		3 hours ago
git-clone	These Tasks are Git...	1 week ago
s2i	Source-to-Image (S2...	1 week ago
update-deployment		3 hours ago

Como podemos ver ahora tenemos creada las dos tareas nuevas. La tarea s2i y git-clone son predeterminadas de Tekton

Pipeline

Adjunto imagen de un esquema que representa perfectamente lo que hará el pipeline que vamos a crear



Tendremos dos repositorios para la app (Backend y Frontend) Cuando haya un push en uno de los dos repositorios, github se comunicará con la api de Openshift a través de un EventListener, el EventListener avisará a los disparadores para que estos ejecuten nuestro pipeline. El pipeline clonará de nuevo el repositorio actualizado, construirá la imagen y desplegará la aplicación actualizada.

```
apiVersion: tekton.dev/v1beta1
```

```
kind: Pipeline
```

```
metadata:
```

```
  name: build-and-deploy
```

```
spec:
```

```
  workspaces:
```

```
    - name: shared-workspace
```

```
  params:
```

```
    - name: deployment-name
```

```
      type: string
```

```
      description: name of the deployment to be patched
```

```
- name: git-url
  type: string
  description: url of the git repo for the code of deployment
- name: git-revision
  type: string
  description: revision to be used from repo of the code for
deployment
  default: master
- name: IMAGE
  type: string
  description: image to be build from the code
tasks:
- name: fetch-repository
  taskRef:
    name: git-clone
    kind: ClusterTask
workspaces:
- name: output
  workspace: shared-workspace
params:
- name: url
  value: $(params.git-url)
- name: subdirectory
  value: ""
- name: deleteExisting
  value: "true"
- name: revision
  value: $(params.git-revision)
- name: build-image
```

```
taskRef:
  name: buildah
  kind: ClusterTask
params:
- name: IMAGE
  value: $(params.IMAGE)
workspaces:
- name: source
  workspace: shared-workspace
runAfter:
- fetch-repository
- name: apply-manifests
  taskRef:
    name: apply-manifests
  workspaces:
  - name: source
    workspace: shared-workspace
  runAfter:
  - build-image
- name: update-deployment
  taskRef:
    name: update-deployment
  params:
  - name: deployment
    value: $(params.deployment-name)
  - name: IMAGE
    value: $(params.IMAGE)
  runAfter:
  - apply-manifests
```

```
oc create -f
https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/01_pipeline/04_pipeline.yaml
```

En el interior del pipeline no hay nada indicado, sino que todo está construido con variables de entornos (Parámetros). Por tanto a la hora de ejecutar el pipeline por primera vez debemos de indicarle el contenido de cada variable de entorno.

Ahora ejecutaremos el pipeline 2 veces, una para el backend y otra para el frontend

BACKEND

```
tkn pipeline start build-and-deploy \
-w
name=shared-workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/01_pipeline/03_persistent_volume_claim.yaml \
-p deployment-name=pipelines-vote-api \
-p git-url=https://github.com/apalgar24/pipelines-vote-api.git \
-p
IMAGE=image-registry.openshift-image-registry.svc:5000/apalgar24-dev/pipelines-vote-api \
--use-param-defaults
```

FRONTEND

```
tkn pipeline start build-and-deploy \
-w
name=shared-workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/01_pipeline/03_persistent_volume_claim.yaml \
-p deployment-name=pipelines-vote-api \
-p
IMAGE=image-registry.openshift-image-registry.svc:5000/apalgar24-dev/pipelines-vote-api \
--use-param-defaults
```

```
ercontent.com/apalgar24/pipelines-triggers/master/01_pipeline/
03_persistent_volume_claim.yaml \
-p deployment-name=pipelines-vote-ui \
-p git-url=https://github.com/apalgar24/pipelines-vote-ui.git \
-p
IMAGE=image-registry.openshift-image-registry.svc:5000/apalgar24-
dev/pipelines-vote-ui \
--use-param-defaults
```

Indicamos

- Volumen persistente va a utilizar, en mi caso la definición del volumen la tengo en un repositorio git,
- Nombre del deployment, esta variable nos servirá a la hora de ejecutar la tarea de despliegues
- Url del repositorio git que contiene el código fuente y el Dockerfile
- Registro donde guardaremos la imagen construida por el pipeline

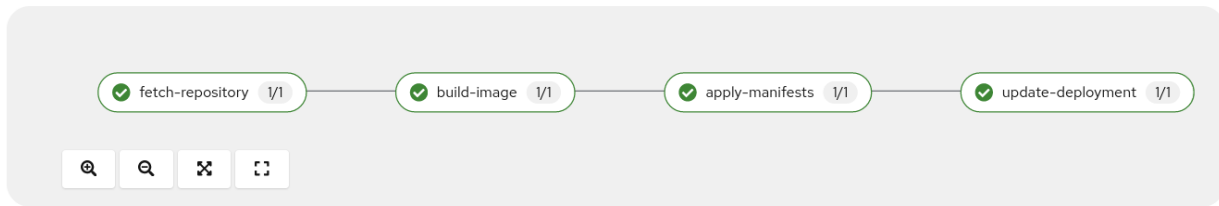
He de recalcar que en mi caso guardo las imágenes en local, también podremos guardarlas en una registradora online como docherhub.

Si hacemos un listado los runs de nuestro pipeline podemos ver que se ha ejecutado correctamente.

```
bash-4.4 ~ $ tkn pipelinerun ls
```

NAME	STARTED	DURATION	STATUS
build-and-deploy-run-fysdf	1 hour ago	1m42s	Succeeded
build-and-deploy-run-gsvfs	2 hours ago	2m4s	Succeeded

PipelineRun details



PipelineRuns > PipelineRun details

PLR build-deploy-pipelines-vote-ui-56xc2 Succeeded Actions

[Details](#) [YAML](#) [TaskRuns](#) [Parameters](#) [Logs](#) [Events](#)

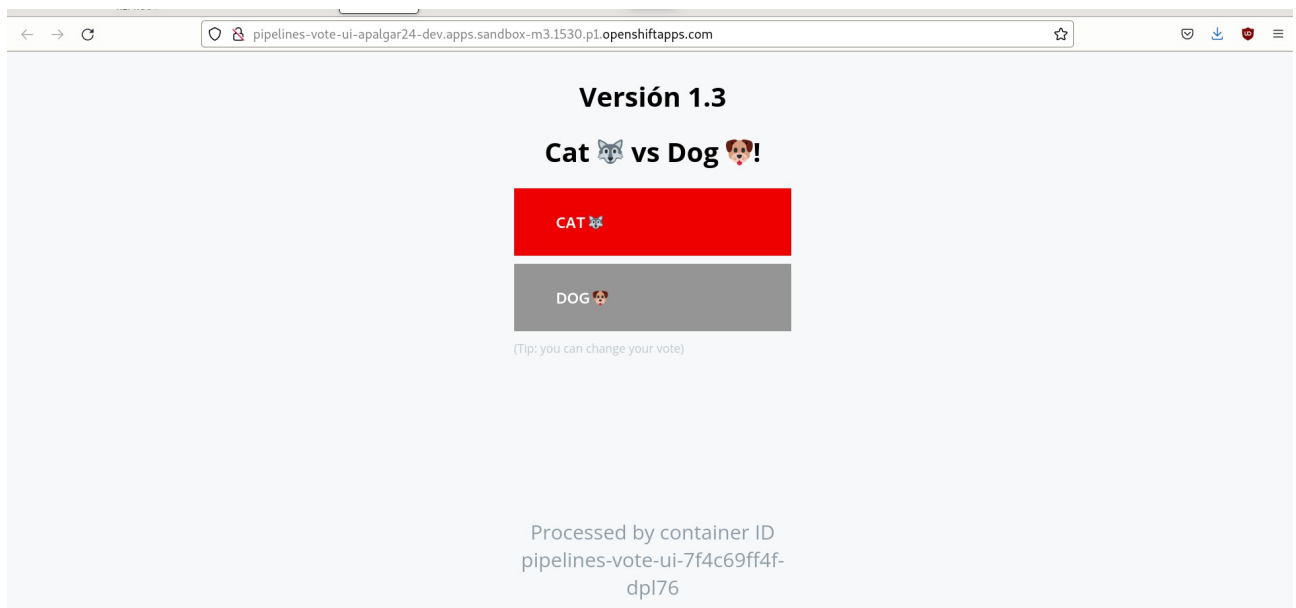
[Download](#) | [Download all task logs](#) | [Expand](#)

- fetch-repository
- build-image**
- apply-manifests
- update-deployment

```
build-image
STEP 6/6: CMD ["python", "./app.py"]
COMMIT image-registry.openshift-image-registry.svc:5000/apalgar24-dev/pipelines-vote-ui
--> 6d14fc8f5e3
Successfully tagged image-registry.openshift-image-registry.svc:5000/apalgar24-dev/pipelines-vote-ui:latest
6d14fc8f5e35fef048bb43764723aded546879d17154871c769398f5ba3aa02b
Getting image source signatures
Copying blob sha256:2515df60886245a12001711789f0e7434f951cf8ffe4bc16ae6508f6f45d19b7
Copying blob sha256:784f7f93393c6b22225807b7ad38da19dc227402886a17bff8975cd47c42d632
Copying blob sha256:4575332388311bec93cf76da8cee9d1781b7d0f4bedddbc96417e6a682b294a5
Copying blob sha256:33204bfe17ee0408ce75330d99e2f997942362bba4f2d3c250e9669b988c8db8
Copying blob sha256:02cdeed2df3409a79d00372a58c5d765f0901c4c8983b698b3929b00c646dc8d
Copying config sha256:6d14fc8f5e35fef048bb43764723aded546879d17154871c769398f5ba3aa02b
Writing manifest to image destination
Storing signatures
sha256:e2de0810a1f681e230e07b43714bb26ed30912bef60ef8492bcf3951ba7c46f7image-registry.openshift-image-registry.svc:5
```

Desde la interfaz gráfica podemos ver el estado de las tareas (en mi caso todas exitosas) y también podemos ver el log de cada una de las tareas del pipeline.

En la demo de este proyecto mostraré mejor como desplazarnos por el interfaz gráfico, en esta memoria me centraré en los pasos necesarios para hacer el despliegue.



Como podemos ver la aplicación ha sido desplegado con éxito

Triggers

Como hemos comentado antes, los triggers son disparadores que nos ayudarán a ejecutar el pipeline en cuanto la api de Openshift detecte algún cambio en nuestro repositorio git

Trigger template

```
apiVersion: triggers.tekton.dev/v1beta1
kind: TriggerTemplate
metadata:
  name: vote-app
spec:
  params:
    - name: git-repo-url
      description: The git repository url
    - name: git-revision
      description: The git revision
      default: master
    - name: git-repo-name
```

```

description: The name of the deployment to be created / patched

resourcetemplates:
- apiVersion: tekton.dev/v1beta1
  kind: PipelineRun
  metadata:
    generateName: build-deploy-$(tt.params.git-repo-name)-
  spec:
    serviceAccountName: pipeline
    pipelineRef:
      name: build-and-deploy
    params:
      - name: deployment-name
        value: $(tt.params.git-repo-name)
      - name: git-url
        value: $(tt.params.git-repo-url)
      - name: git-revision
        value: $(tt.params.git-revision)
      - name: IMAGE
        value: image-registry.openshift-image-registry.svc:5000/apalgar24-dev/$(tt.params.git-repo-name)
    workspaces:
      - name: shared-workspace
        volumeClaimTemplate:
          spec:
            accessModes:
              - ReadWriteOnce
            resources:
              requests:
                storage: 500Mi

```

```

oc create -f
https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/03_triggers/02_template.yaml

```

Como su nombre indica es una plantilla que usaremos para definir los parámetros y el registro donde se guardarán nuestras imágenes.

Trigger binding

```

apiVersion: triggers.tekton.dev/v1beta1
kind: TriggerBinding

```

```
metadata:
  name: vote-app
spec:
  params:
  - name: git-repo-url
    value: $(body.repository.url)
  - name: git-repo-name
    value: $(body.repository.name)
  - name: git-revision
    value: $(body.head_commit.id)
```

```
oc create -f
https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/03_triggers/01_binding.yaml
```

Este rellena los parámetros del trigger template recogiendo la información a partir del evento que ocurra (push a nuestro rep git) .

Trigger interceptor

```
apiVersion: triggers.tekton.dev/v1beta1
kind: Trigger
metadata:
  name: vote-trigger
spec:
  serviceAccountName: pipeline
  interceptors:
  - ref:
      name: "github"
      params:
      - name: "secretRef"
        value:
          secretName: github-secret
```

```
        secretKey: secretToken
      - name: "eventTypes"
        value: ["push"]
    bindings:
      - ref: vote-app
    template:
      ref: vote-app
---
apiVersion: v1
kind: Secret
metadata:
  name: github-secret
type: Opaque
stringData:
  secretToken: "1234567"
```

oc create -f

https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/03_triggers/03_trigger.yaml

Este trigger lo usa el EventListener como referencia ya que indica que estamos utilizando github, el secreto que se va a utilizar para añadir el webhook ...

EventListener

```
apiVersion: triggers.tekton.dev/v1beta1
kind: EventListener
metadata:
  name: vote-app
spec:
  serviceAccountName: pipeline
  triggers:
    - triggerRef: vote-trigger
```

```
oc create -f
https://raw.githubusercontent.com/apalgar24/pipelines-triggers/master/03\_triggers/04\_event\_listener.yaml
```

Este componente configura un servicio y escucha eventos. También conecta un TriggerTemplate a un TriggerBinding.

Importante exponer el trigger ya que la url proporcionada del route, será el webhook que añadamos en github

```
oc expose svc el-vote-app
```

Git Webhook

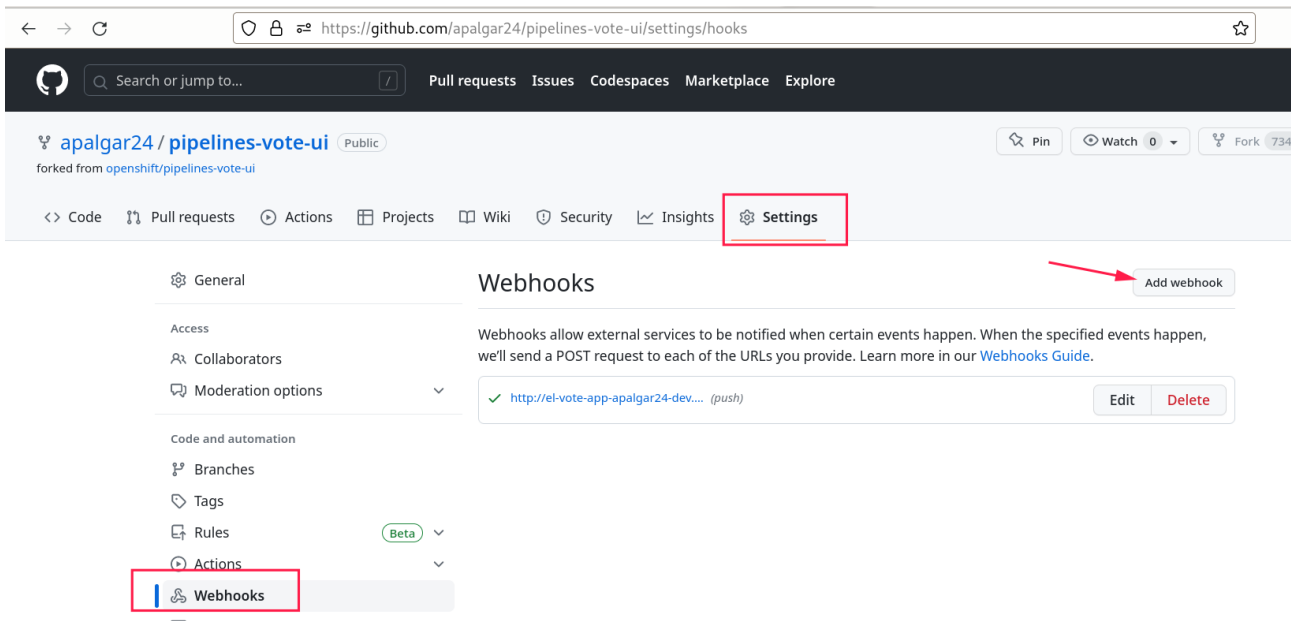
Para sacar la url de nuestro webhook debemos de teclear lo siguiente en nuestra línea de comandos

```
echo "$(oc get route el-vote-app --template='http://{{.spec.host}}')"
```

También podemos hacerlo desde la interfaz gráfica mirando el route de nuestro trigger interceptor

Una vez tengamos la URL, debemos de dirigirnos a nuestros dos repositorios con el código fuente de nuestra app (backend y frontend)

Settings → Webhook → add webhook



Payload URL

<http://el-vote-app-apalgar24-dev.apps.sandbox-m3.1530.p1.openshiftapps.com>

Content type

application/json

Secret

1234567

Webhooks / Manage webhook

Settings Recent Deliveries

We'll send a `POST` request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, `x-www-form-urlencoded`, etc). More information can be found in [our developer documentation](#).

Payload URL *

`http://el-vote-app-apalgar24-dev.apps.sandbox-m3.1530.p1.op`

Content type

application/json

Secret

If you've lost or forgotten this secret, you can change it, but be aware that any integrations using this secret will need to be updated. — [Change Secret](#)

Which events would you like to trigger this webhook?

Just the `push` event.

Send me **everything**.

Comprobación despliegue continuo con webhook

Como hemos visto en la captura anterior, nuestra app estaba en la versión 1.3, cambiaremos algo en el `index.html` del frontend, haremos un `commit` y posteriormente un `push` y veremos como comienza la construcción del nuevo frontend y el despliegue de la nueva version.

```
adrian@debadri:~/git/pipelines-vote-ui$ git commit -am "versión 2.3"
[master d767e74] versión 2.3
  Committer: adrian <adrian@debian-BULLSEYE-live-builder-AMD64>
  Tu nombre y correo fueron configurados automáticamente basados
  en tu usuario y nombre de host. Por favor verifica que son correctos.
  Tu puedes suprimir este mensaje configurándolos de forma explícita. Ejecuta el
  siguiente comando y sigue las instrucciones de tu editor
  para modificar tu archivo de configuración:

  git config --global --edit

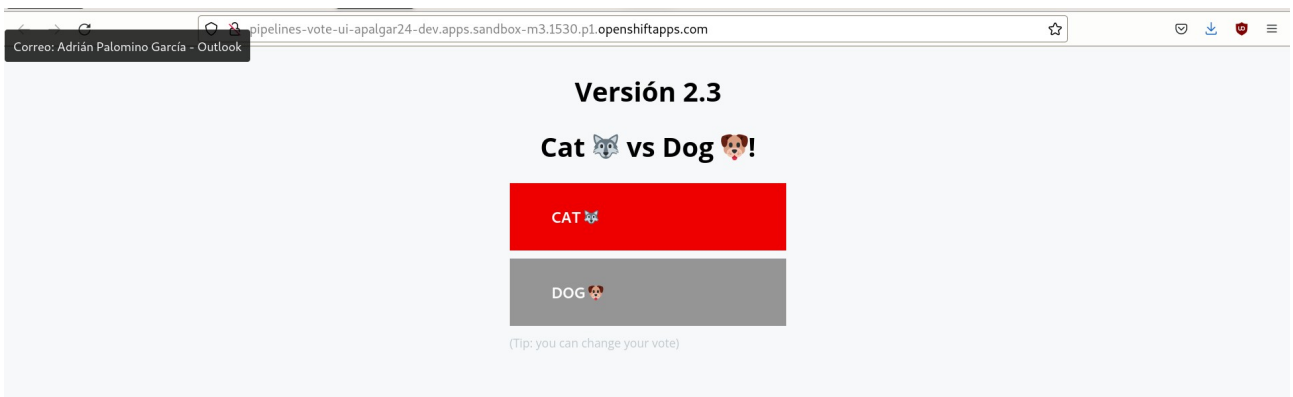
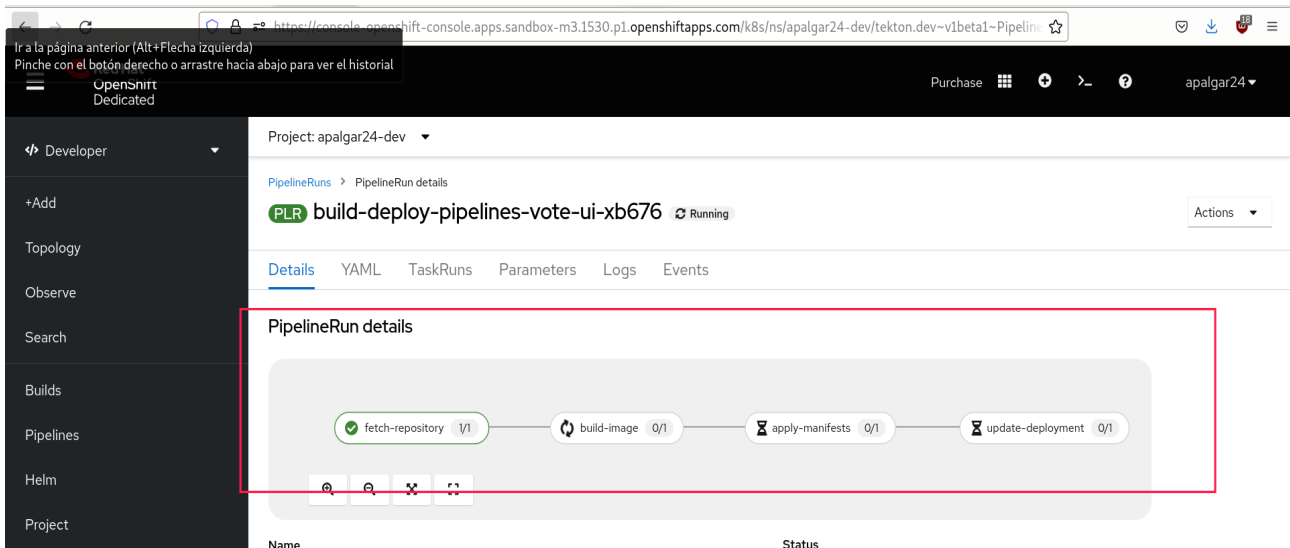
Tras hacer esto, puedes arreglar la identidad usada para este commit con:

  git commit --amend --reset-author

1 file changed, 1 insertion(+), 1 deletion(-)
adrian@debadri:~/git/pipelines-vote-ui$ git push
Enumerando objetos: 11, listo.
Contando objetos: 100% (11/11), listo.
Compresión delta usando hasta 8 hilos
Comprimiendo objetos: 100% (8/8), listo.
Escribiendo objetos: 100% (8/8), 752 bytes | 752.00 KiB/s, listo.
Total 8 (delta 4), reusado 0 (delta 0), pack-reusado 0
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To github.com:apalgar24/pipelines-vote-ui.git
   bldfdfd..d767e74  master -> master
adrian@debadri:~/git/pipelines-vote-ui$ █
```

Una vez hacemos el push, automáticamente se ejecuta el pipeline y empieza a construir y desplegar la nueva versión de la aplicación

Una vez finalice todo, veremos el cambio en nuestra web



Ya tenemos la versión 2.3 disponible en nuestra aplicación!

Tolerancia a fallos, Escalabilidad y Balanceo de carga

Como ya sabemos, estas características nos la ofrece Openshift de manera automática, simplemente teniendo varios pods en nuestro despliegue ya es más que suficiente para tener tolerancia a fallos y un balanceador de carga.

Aquí vamos a hacer unas pequeñas pruebas para verificar el funcionamiento de estas características que ofrece Openshift

Empezaremos con la escalabilidad.

Como he dicho en varias ocasiones la manera de hacerlo con el entorno gráfico lo demostraré en vivo en la presentación del proyecto, aquí me centraré en ir directo al grano.

```
oc scale deployment pipelines-vote-ui -replicas=2
```

De esta manera nuestra aplicación estará respaldada por dos pods.

```
$ oc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
el-vote-app-865d5897c7-cz42w	1/1	Running	0	64m
pipelines-vote-api-694c864ff4-w5p77	1/1	Running	0	64m
pipelines-vote-ui-b79b97d94-qbgmr	1/1	Running	0	20m
pipelines-vote-ui-b79b97d94-z46qs	1/1	Running	0	19m
workspace2fcf1d329f444c3-5568fcb659-45n7v	2/2	Running	0	68m

Como podemos ver, en el despliegue pipelines-vote-ui (frontend) tengo dos pods en ejecución

Si nosotros intentamos borrar por ejemplo el pod pipelines-vote-ui-b79b97d94-z46qs, se generará automáticamente otro pod sustituyendo el que hemos borrado, a esto se le llama Tolerancia a fallos.

oc delete pod pipelines-vote-ui-b79b97d94-z46qs

Comprobamos si se ha generado otro diferente

```
bash-4.4 ~ $ oc delete pod pipelines-vote-ui-b79b97d94-z46qs
```

```
pod "pipelines-vote-ui-b79b97d94-z46qs" deleted
```

```
bash-4.4 ~ $ oc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
el-vote-app-865d5897c7-cz42w	1/1	Running	0	67m
pipelines-vote-api-694c864ff4-w5p77	1/1	Running	0	67m
pipelines-vote-ui-b79b97d94-f8264	1/1	Running	0	3s
pipelines-vote-ui-b79b97d94-qbgmr	1/1	Running	0	23m
workspace2fcaf1d329f444c3-5568fcb659-45n7v	2/2	Running	0	72m

Como podemos ver el pod se ha borrado correctamente y se ha generado un nuevo pod llamado pipelines-vote-ui-b79b97d94-f8264

Por último demostraremos el balanceo de carga haciendo unos cuantos curl a nuestra aplicación

```
OpenShift command line terminal
Terminal 1 x +
</html>bash-4.4 ~ $ curl http://pipelines-vote-ui-apalgar24-dev.apps.sandbox-m3.1530.p1.openshiftapps.com
<!DOCTYPE html>
<html>
bash-4.4 ~ $ curl http://pipelines-vote-ui-apalgar24-dev.apps.sandbox-m3.1530.p1.openshiftapps.com
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Cat 🐱 vs Dog 🐶 </title>
<base href="/index.html">
<meta name = "viewport" content = "width=device-width, initial-scale = 1.0">
<meta name="keywords" content="docker-compose, docker, stack">
<meta name="author" content="Tutum dev team">
<link rel="stylesheet" href="/static/stylesheets/style.css" />
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-awesome/4.4.0/css/font-awesome.min.css">
</head>
<body>
<div id="content-container">
<div id="content-container-center">
<h1>Cat 🐱 vs Dog 🐶 </h1>
<form id="choice" name='form' method="POST" action="/">
<button id="a" type="submit" name="vote" class="a" value="a">Cat 🐱 </button>
<button id="b" type="submit" name="vote" class="b" value="b">Dog 🐶 </button>
</form>
<div id="tip">
(Tip: you can change your vote)
</div>
<div id="result">
</div>
<div id="hostname">
Processed by container ID pipelines-vote-ui-b79b97d94-f8264
</div>
</div>
</body>
</html>
```

```
OpenShift command line terminal
Terminal 1 x +
</body>
</html>bash-4.4 ~ $ curl http://pipelines-vote-ui-apalgar24-dev.apps.sandbox-m3.1530.p1.openshiftapps.com
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Cat 🐱 vs Dog 🐶</title>
<base href="/index.html">
<meta name = "viewport" content = "width=device-width, initial-scale = 1.0">
<meta name="keywords" content="docker-compose, docker, stack">
<meta name="author" content="Tutum dev team">
<link rel='stylesheet' href="/static/stylesheets/style.css" />
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-awesome/4.4.0/css/font-awesome.min.css">
</head>
<body>
<div id="content-container">
<div id="content-container-center">
<h1>Cat 🐱 vs Dog 🐶</h1>
<form id="choice" name='form' method="POST" action="/">
<button id="a" type="submit" name="vote" class="a" value="a">Cat 🐱</button>
<button id="b" type="submit" name="vote" class="b" value="b">Dog 🐶</button>
</form>
<div id="tip">
(Tip: you can change your vote)
</div>
<div id="result">
</div>
<div id="hostname">
Processed by container ID pipelines-vote-ui-b79b97d94-qbgmr
</div>
</div>
</div>
</div>
```

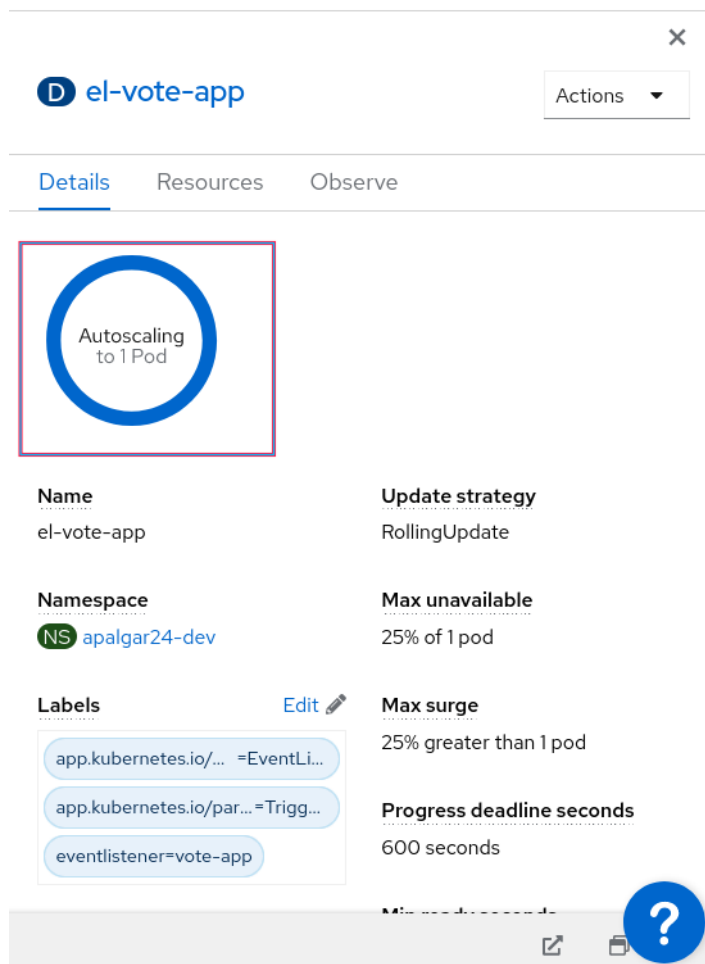
Como podemos ver en las capturas, hay veces que accede a un pod y otras veces accede a otro pod

Autoescalado

El autoescalado puede llegar a ser muy útil si estudiamos la carga que tiene nuestro despligue, es decir si veo que mi aplicación suele estar por debajo del 50% de uso de CPU pero de vez en cuando supera los 75%, podemos incrementar el número de pods cuando sobrepase ese porcentaje y disminuir el número de pod cuando vuelva a la normalidad.

En este caso haré un autoescalado de mi trigger que dispara el pipeline. Para ello ejecutamos los siguiente

```
oc autoscale deployment el-vote-app --min=1 --max=3 --cpu-percent=15
```



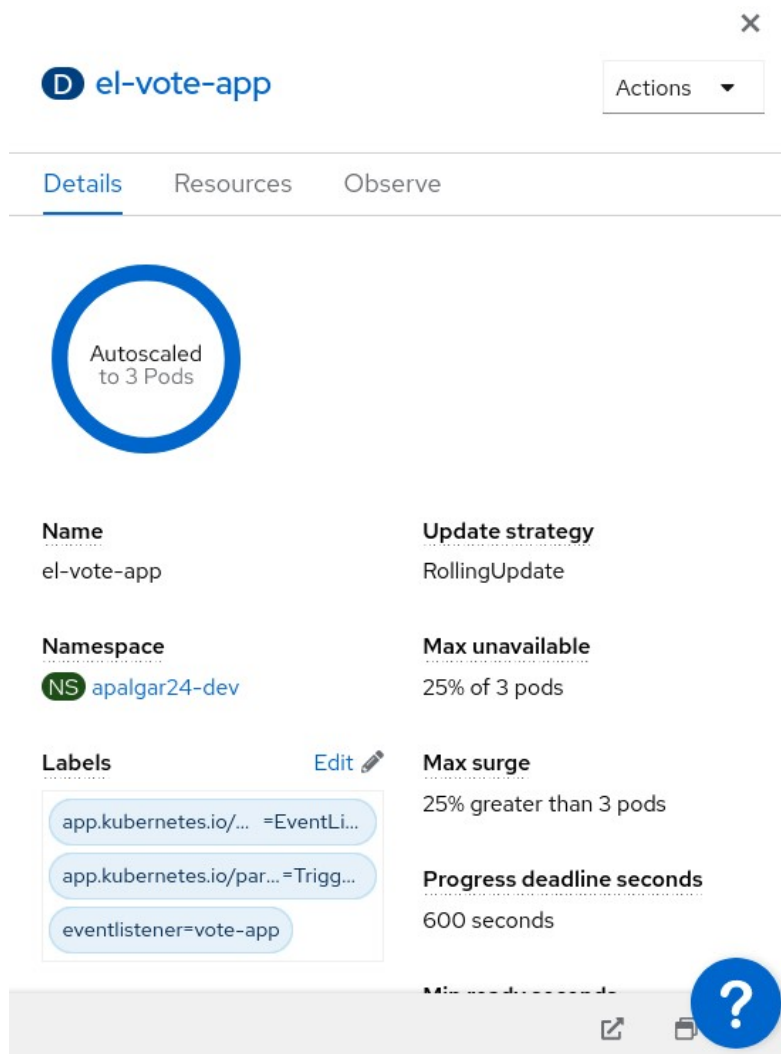
el-vote-app Actions

Details Resources Observe

Autoscaling to 1 Pod

Name	el-vote-app	Update strategy	RollingUpdate
Namespace	apalgar24-dev	Max unavailable	25% of 1 pod
Labels	<code>app.kubernetes.io/...=EventLi...</code> <code>app.kubernetes.io/par...=Trigg...</code> <code>eventlistener=vote-app</code>	Max surge	25% greater than 1 pod
		Progress deadline seconds	600 seconds

Como podemos ver en la interfaz gráfica, el deployment ha sido autoescalado, ahora cuando el despliegue sobrepase el 15% de la CPU, se autoescalará a 3 pods



He realizado un push para forzar al trigger a realizar la acción de llamada a la api de Openshift y de esta manera he conseguido que se autoescale a 3 pods

Rollback

Una de las características más llamativas y útiles, es el rollback, ya que gracias a este recurso que nos ofrece OpenShift, podemos volver a una versión anterior de nuestra aplicación en caso de que la última versión desplegada dé problemas en producción.

Para poder hacer un rollback, primero debemos de mapear el historial de versiones de nuestro despliegue, por tanto debemos de ejecutar lo siguiente en nuestra terminal.

```
oc rollout history deployment pipelines-vote-ui
```

```
$ oc rollout history deployment pipelines-vote-ui
deployment.apps/pipelines-vote-ui
REVISION  CHANGE-CAUSE
2          <none>
7          <none>
8          <none>
13         <none>
14         <none>
15         <none>
16         <none>
17         <none>
19         <none>
20         <none>
21         <none>
```

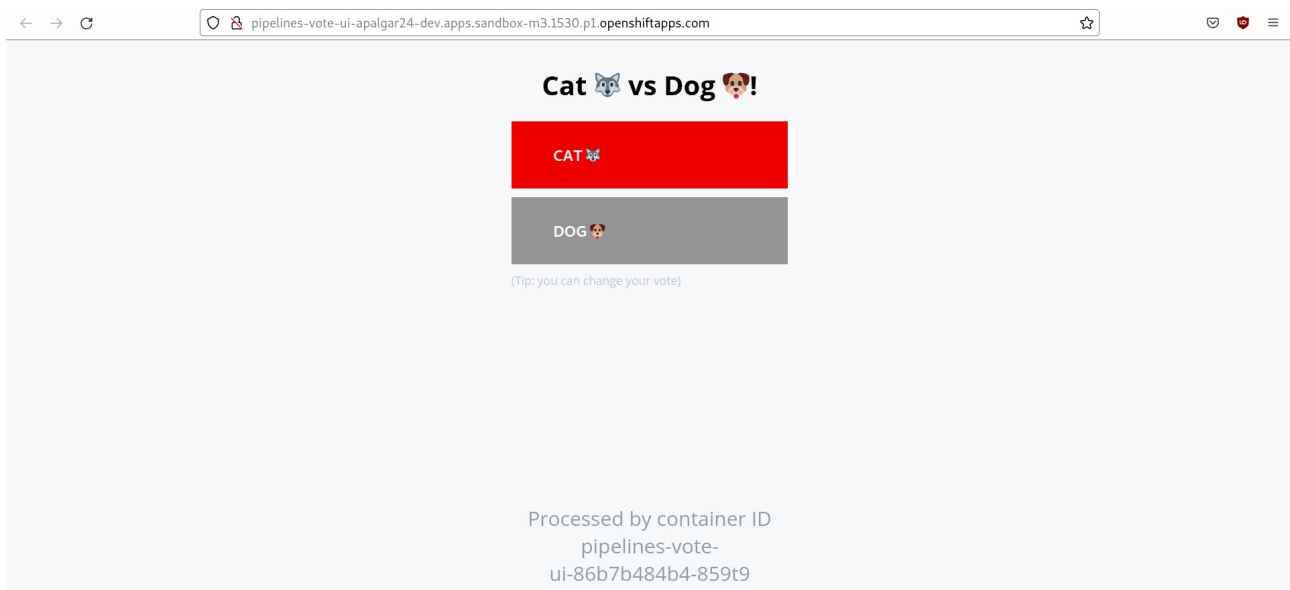
Como podemos ver tenemos un montón de versiones a las que volver atrás.

Para hacer el rollback a la versión seleccionada debemos de ejecutar lo siguiente

```
oc rollout undo deployment/pipelines-vote-ui --to-revision=19
```

```
oc rollout undo deployment/pipelines-vote-ui --to-revision=19
deployment.apps/pipelines-vote-ui rolled back
```

El rollback ha sido ejecutado con éxito, ahora accederemos a la web:

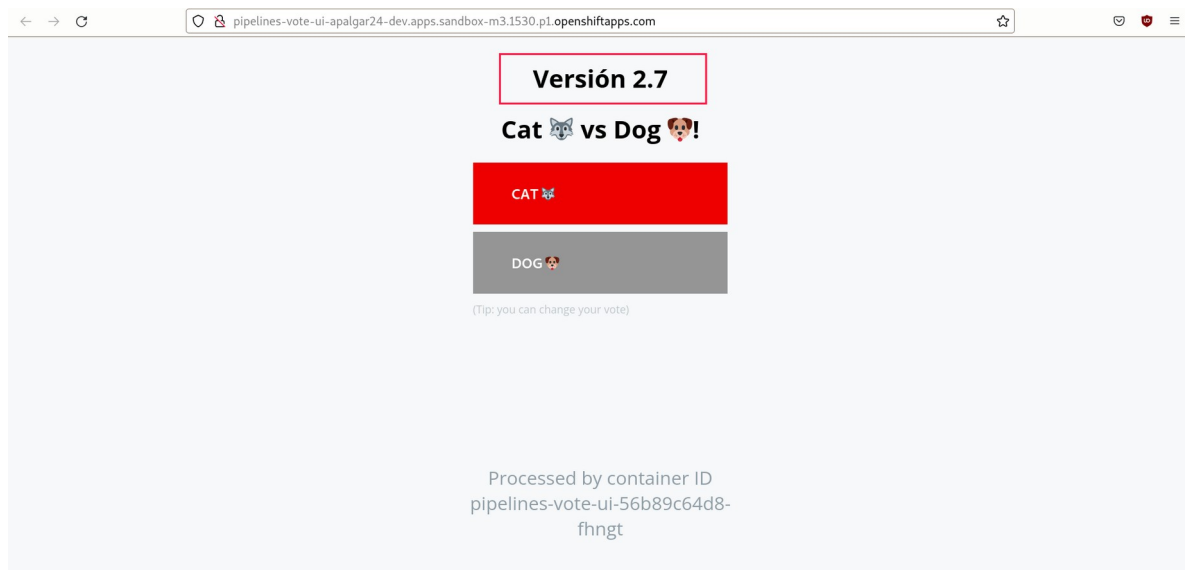


Efectivamente ha retrocedido a una versión anterior, ahora volveremos a la actual de nuevo, para ello miramos en nuestro historial que versión es la más reciente y ejecutamos el rollback

```
$ oc rollout undo deployment/pipelines-vote-ui --to-revision=22
```

```
deployment.apps/pipelines-vote-ui rolled back
```

Comprobamos la web



Como podemos ver, ha vuelto a la versión mas reciente.

SonarQube

SonarQube es una plataforma de análisis estático de código (Static Code Analysis) diseñada para evaluar la calidad y seguridad del código fuente. Se utiliza como una herramienta de análisis automatizado que ayuda a los equipos de desarrollo a identificar y corregir problemas en el código, así como a mantener altos estándares de calidad.

Para desplegar SonarQube, debemos de descargarnos desde la terminal, el deployment

```
curl -o sonarqube-h2-db-template.yml  
https://raw.githubusercontent.com/edwin/sonarqube-on-openshift-4/master/sonarqube-h2-db-template.yml
```

Una vez descargado, aplicamos el despliegue en nuestro Openshift

```
oc create -f sonarqube-h2-db-template.yml
```

Por último ejecutamos el despliegue con el siguiente comando

```
oc new-app --template sonarqube-h2-db
```

```
oc new-app --template sonarqube-h2-db
```

```
--> Deploying template "apalgar24-dev/sonarqube-h2-db" to project apalgar24-dev
```

```
SonarQube (H2 DB)
```

```
-----
```

```
Sonarqube service, with H2 DB.
```

```
NOTE: Data will not be gone despite restarts, but dont use this for production usage.
```

```
A Sonarqube service has been created in your project. You can access using admin/admin.
```

```
* With parameters:
```

```
* SonarQube Service Name=sonar
```

```
* Memory Limits=2Gi
```

```
--> Creating resources ...
```

```
route.route.openshift.io "sonar" created
```

```
deploymentconfig.apps.openshift.io "sonar" created
```

```
service "sonar" created
```

```
persistentvolumeclaim "sonar-data-pv" created
```

```
persistentvolumeclaim "sonar-logs-pv" created
```

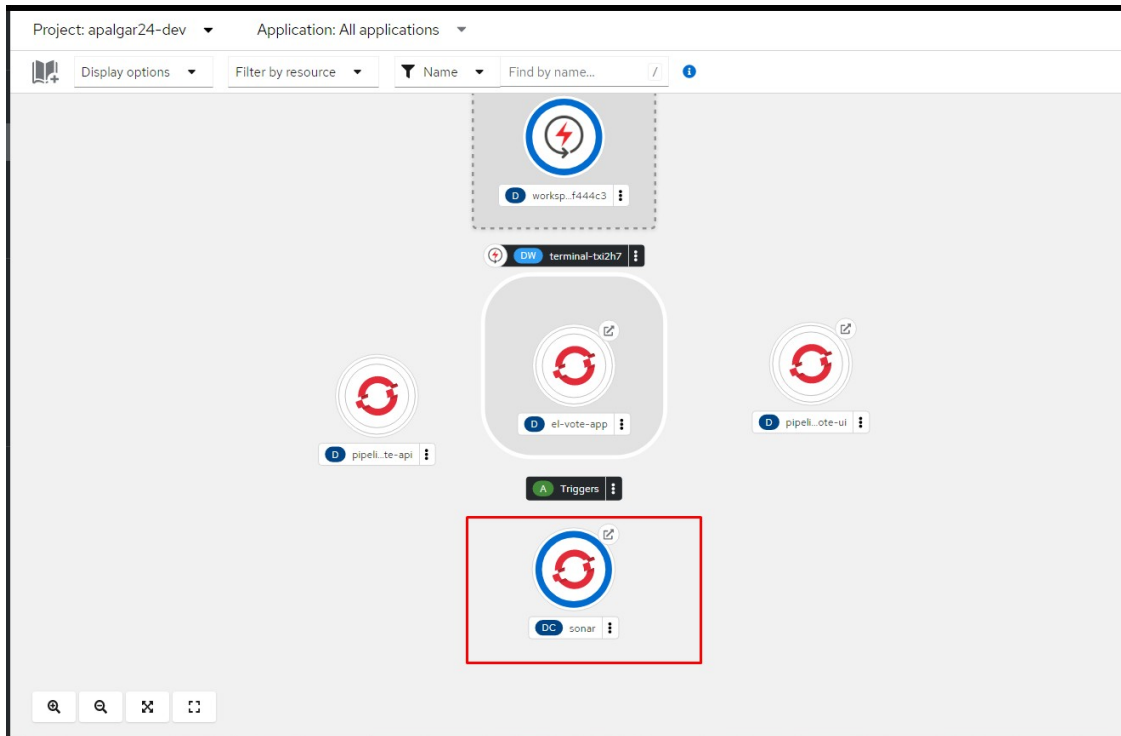
```
persistentvolumeclaim "sonar-extensions-pv" created
```

```
--> Success
```

```
Access your application via route 'sonar-apalgar24-dev.apps.sandbox-  
m3.1530.p1.openshiftapps.com'
```

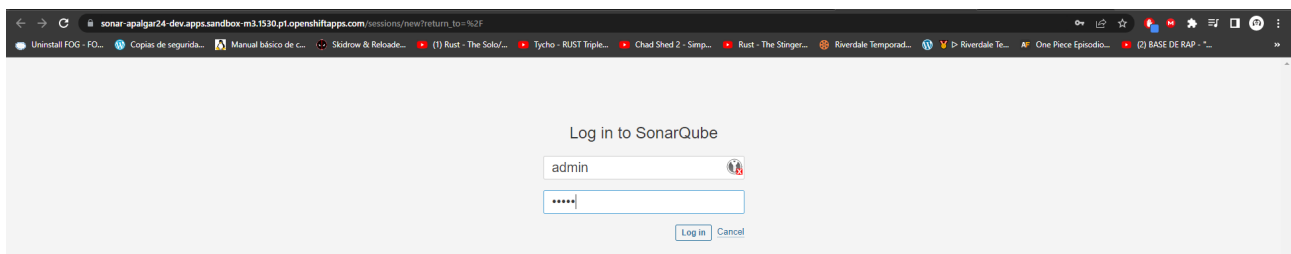
```
Run 'oc status' to view your app.
```

Como podemos ver se nos ha creado todo los recursos necesarios para levantar la aplicación, ahora solo queda esperar a que se despliegue todo

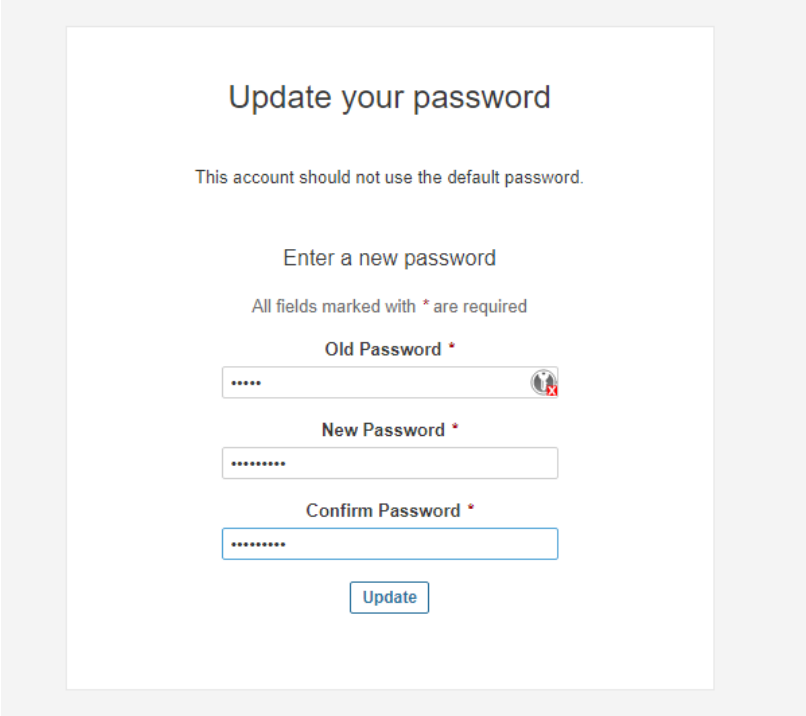


Ahí está nuestra aplicación el topología de nuestro proyecto.

Ahora debemos de loguearnos , por defecto el usuario es “admin” y la contraseña “admin”



Nos obliga como es obvio a cambiar la contraseña de administrador



Update your password

This account should not use the default password.

Enter a new password

All fields marked with * are required

Old Password *

.....

New Password *

.....

Confirm Password *

.....

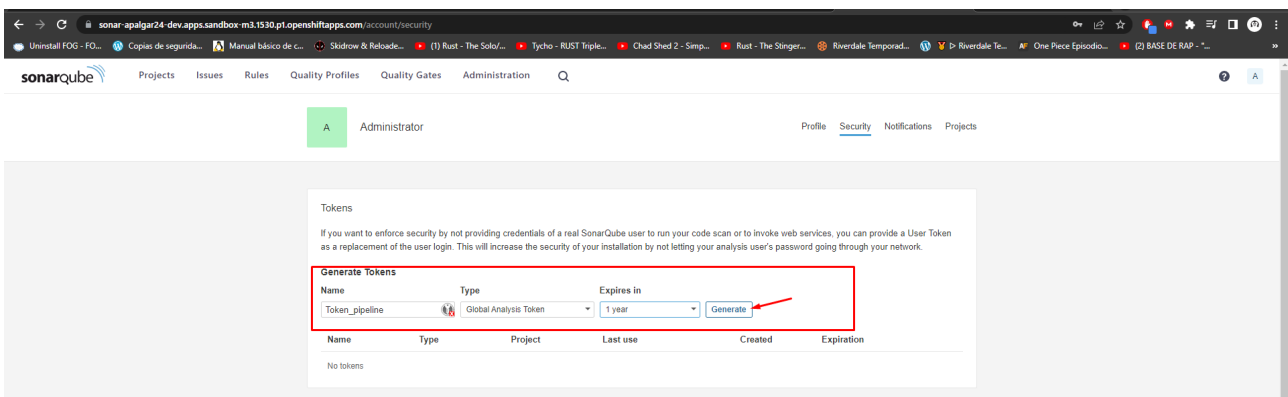
Update

Lo siguiente que haremos será sacar un token de acceso para luego implementarlo en nuestro pipeline para que pueda acceder y realizar el testeo de la app

Para ello :

Inicia sesión en SonarQube con una cuenta que tenga privilegios de administrador.

1. En la parte superior derecha de la interfaz de SonarQube, haz clic en tu nombre de usuario y selecciona "My Account" (Mi cuenta) en el menú desplegable.
2. En la página de tu cuenta, selecciona la pestaña "Security" (Seguridad) en la parte superior.
3. En la sección "Tokens", haz clic en el botón "Generate Tokens" (Generar tokens).
4. En la ventana emergente, ingresa un nombre descriptivo para el token en el campo "Token Name" (Nombre del token).
5. Si deseas asignar permisos específicos al token, puedes seleccionar los roles correspondientes en la lista desplegable "Token Permissions" (Permisos del token). Esto determinará qué acciones puede realizar el token en SonarQube.
6. Haz clic en el botón "Generate" (Generar) para crear el token.



Tokens

If you want to enforce security by not providing credentials of a real SonarQube user to run your code scan or to invoke web services, you can provide a User Token as a replacement of the user login. This will increase the security of your installation by not letting your analysis user's password going through your network.

Generate Tokens

Name: Type: Expires in:

! New token "Token_pipeline" has been created. Make sure you copy it now, you won't be able to see it again!

`sqa_4d00eed37cdf6d5b59594171ada96bad10162b8c`

Name	Type	Project	Last use	Created	Expiration	
Token_pipeline	Global		Never	May 24, 2023	May 23, 2024	<input type="button" value="Revoke"/>

Ahí tendríamos generado nuestro token

Ahora debemos de crear un secreto con el token y la url de nuestro SonarQube

kind: Secret

apiVersion: v1

metadata:

name: sonarqube-access

data:

SONARQUBE_TOKEN: sqa_4d00eed37cdf6d5b59594171ada96bad10162b8c

SONARQUBE_URL: https://sonar-apalgar24-dev.apps.sandbox-m3.1530.p1.openshiftapps.com/

type: Opaque

Una vez creado el secreto, debemos de instalarnos la tarea sonarqube-scanner del catálogo de Tekton

```
tkn hub search sonarqube-scanner
```

```
tkn hub install task sonarqube-scanner
```

Y aquí es donde lamentablemente, debo de decir que la prueba gratuita de Openshift no me deja instalar tareas del catálogo de Tekton, solo me deja jugar con las tareas ya instaladas por defecto, por tanto no puedo hacer una demostración, pero puedo explicar los pasos posteriores ya que son muy sencillos, de hecho es lo mas sencillo.

Solo quedaría agregar la tarea sonarqube-scanner a nuestro pipeline que creamos anteriormente.

Osea que el pipeline quedaría tal que así

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  creationTimestamp: '2023-05-17T06:43:43Z'
  generation: 1
  managedFields:
    - apiVersion: tekton.dev/v1beta1
      fieldType: FieldsV1
      fieldsV1:
        'f:spec':
          .: {}
          'f:params': {}
          'f:tasks': {}
          'f:workspaces': {}
      manager: kubectl-create
      operation: Update
      time: '2023-05-17T06:43:43Z'
  name: build-and-deploy
  namespace: apalgar24-dev
  resourceVersion: '738725163'
  uid: 2252ddfc-1d9d-417f-87b5-afb8b78311cf
spec:
  params:
```

```

- description: name of the deployment to be patched
  name: deployment-name
  type: string
- description: url of the git repo for the code of deployment
  name: git-url
  type: string
- default: master
  description: revision to be used from repo of the code for d
deployment
  name: git-revision
  type: string
- description: image to be build from the code
  name: IMAGE
  type: string
tasks:
- name: fetch-repository
  params:
  - name: url
    value: $(params.git-url)
  - name: subdirectory
    value: ''
  - name: deleteExisting
    value: 'true'
  - name: revision
    value: $(params.git-revision)
  taskRef:
    kind: ClusterTask
    name: git-clone
  workspaces:
  - name: output
    workspace: shared-workspace
- name: code-analysis
  params:
  - name: SONAR_HOST_URL
    value: >-
      https://sonar-apalgar24-dev.apps.sandbox-
m3.1530.p1.openshiftapps.com
  runAfter:
  - fetch-repository
  taskRef:
    kind: Task
    name: sonarqube-scanner

```

```
workspaces:
  - name: source-dir
    workspace: shared-workspace
  - name: sonar-settings
    workspace: sonar-settings

- name: build-image
  params:
    - name: IMAGE
      value: $(params.IMAGE)
  runAfter:
    - code-analysis
  taskRef:
    kind: ClusterTask
    name: buildah
  workspaces:
    - name: source
      workspace: shared-workspace
- name: code-analysis
  params:
    - name: SONAR_HOST_URL
      value: >-
        https://sonar-apalgar24-dev.apps.sandbox-
m3.1530.p1.openshiftapps.com
  runAfter:
    - fetch-repository
  taskRef:
    kind: Task
    name: sonarqube-scanner
  workspaces:
    - name: source-dir
      workspace: shared-workspace
    - name: sonar-settings
      workspace: sonar-settings

- name: apply-manifests
  runAfter:
    - build-image
  taskRef:
    kind: Task
    name: apply-manifests
  workspaces:
```



```
- name: source
  workspace: shared-workspace
- name: update-deployment
  params:
    - name: deployment
      value: $(params.deployment-name)
    - name: IMAGE
      value: $(params.IMAGE)
  runAfter:
    - apply-manifests
  taskRef:
    kind: Task
    name: update-deployment
workspaces:
  - name: shared-workspace
    - name: sonar-settings
```

Los cambios respecto al pipeline original es que hemos añadido la tarea code-analysis (sonarqube-scanner) entre medio de las tarea de clonado y de construcción

Es decir que el orden sería al siguiente (Clonas el repositorio → Testeas la app → Co,pilas la imagen → Despliegas la aplicación)

Conclusiones y propuestas

En este proyecto podemos hacer infinidad de propuestas para mejorar, ya que la intención en este proyecto era hacer una vista general de Openshift, aprender a como usarlo desde la terminal y hacer una pequeña demostración de un despliegue continuo muy simple.

ArgoCD

Una de las propuesta mas interesantes puede ser la implementación de ArgoCd en nuestro Openshift, esta herramienta sincroniza la configuración de nuestra aplicación definida en los yamls alojados en nuestro repositorio git, con el estado de actual de nuestra aplicación.

Es decir si nosotros queremos modificar el deployment de nuestra aplicación por ejemplo, solo debemos de cambiar el ficherp que define nuestro deployment en nuestro repositorio git, hacemos un push y automáticamente ArgoCd detectará que el estado actual de la aplicación no esta sincronizado con el repositorio git, y efectuará los cambios. De esta manera siempre garantizamos que nuestra aplicación esta corriendo sobre la configuración correcta.

SonarQube

A pesar de haber investigado esta herramienta, ha sido una pena no poder implementarla en este proyecto por las limitaciones de nuestro Openshift gratuito, nada más que añadir sobre esta herramienta, es perfecta para hacer testeos a nuestra app.

Terraform

En caso de tener una versión de pago de Openshift, podríamos implementar Terraform, una herramienta que crea una estructura de máquinas virtuales sobre la que instaurr nuestro gestor de contenedores, en este caso Openshift.

Conclusión

En conclusión, este proyecto me ha encantado ya que he podido probar una herramienta de pago muy potente como es Openshift, además he descubierto una herramienta de Integración continua muy amena (Tekton) ya que prácticamente esta todo lo que necesitas en su catálogo de tareas y se hace muy sencillo montar un pipeline.

Si que es verdad que al ser un entorno de pruebas, he tenido muchas limitaciones como volúmenes persistentes, instalaciones de tareas del catálogo de Tekton, tener varias aplicaciones corriendo ... A pesar de ello he intentado exprimir al máximo lo que me ofrecía este entorno de pruebas.

Dificultades e inconvenientes

En mi caso la mayoría de dificultades han sido por limitaciones del entorno de prueba de Openshift y por la poca documentación gratuita sobre Openshift, ahora entro en mas detalles:

Limitaciones

- Limitación número de apps corriendo: No puedo crear mas de 4 apps simultáneamente, yo por ejemplo tengo tres que no pueden faltar: Frontend, Backend y Trigger, a la hora de desplegar un jenkins con un Sonarqube a la vez, no podía debido a límite de apps
- Limite de Volúmenes Persistentes: He tenido que ir borrando volúmenes persistentes ha medida que se van haciendo los pipelineruns ya que solo puedo tener 5 volúmenes persistentes simultáneos
- Prohibido instalar tareas tekton: No tengo acceso al catálogo de tareas de Tekton y solo puedo trabajar con las tareas por defecto.

Poca documentación

Obviamente si buscas información sobre Openshift o Tekton, encontrarás mucha, pero siempre es muy básica y poco detallada, a la hora de buscar cosas en concreto o pipelines avanzados, casi no

hay nada en internet, la mayoría de gente te hace una breve introducción para luego intentar venderte su curso sobre el tema. A diferencia de kubernetes, la mayoría de documentación gratis es sobre kubernetes

Bibliografía

<https://edwin.baculsoft.com/2022/01/deploying-sonarqube-to-openshift-4-by-using-openshift-template/>

<https://hub.tekton.dev/tekton/task/sonarqube-scanner>

https://access.redhat.com/documentation/es-es/openshift_container_platform/4.4/html/pipelines/creating-applications-with-cicd-pipelines

<https://openwebinars.net/academia/portada/openshift/>

<https://www.redhat.com/architect/cicd-pipeline-openshift-tekton>

<https://openai.com/blog/chatgpt>

<https://docs.openshift.com/container-platform/4.9/cicd/pipelines/securing-webhooks-with-event-listeners.html>

<https://tekton.dev/docs/triggers/eventlisteners/>

<https://medium.com/@nikhilthomas1/cloud-native-cicd-on-openshift-with-openshift-pipelines-tektoncd-pipelines-part-3-github-1db6dd8e8ca7>