

Argo Workflows

Wilmer Moina-Rivera

2024-05-15

Table of Contents

1. Introducción	1
1.1. Introducción a Kubernetes.....	1
1.2. ¿Qué es Argo Workflows y por qué es importante?	1
1.3. Casos de Uso Comunes de Argo Workflows en la Industria	2
2. Instalación y Configuración	3
2.1. Requisitos Previos.....	3
2.2. Instalación de Argo Workflows	3
2.3. Configuración Inicial y Verificación.....	4
3. Primeros Pasos con Argo Workflows.....	5
3.1. ¿Qué es un workflow?	5
3.2. Crear un workflow:	5
3.3. Usando la Interfaz de Usuario	5
3.4. Ver el workflow	6
3.5. Uso de la CLI	7
4. Tipos de Templates	10
4.1. Template de Contenedor.....	10
4.2. Etiquetas en un Template.....	11
4.3. Work Templates	12
4.4. Template DAG	12
4.5. Bucles (Loops)	14
4.6. Templates de Orquestación	16
4.7. Manejador de Salida	16
5. Entradas y Salidas	19
5.1. Parámetros	19
5.2. Parámetros de Entrada	19
5.3. Parámetros de Salida	20
5.4. Artefactos	22
6. Argo Events	25
6.1. Prerequisitos	25
6.2. Conecta tu Navegador al Servidor MinIO	25
6.3. Instalar y Configurar Argo Events	26
6.4. Agregar un Sensor y Disparador de Minio	28
6.5. Crear un Flujo de Trabajo	29
7. Conclusiones	32

Chapter 1. Introducción

1.1. Introducción a Kubernetes

Kubernetes es un sistema de orquestación de contenedores de código abierto que automatiza la implementación, el escalado y la gestión de aplicaciones en contenedores. Fue desarrollado originalmente por Google y ahora es mantenido por la Cloud Native Computing Foundation. Kubernetes permite a los desarrolladores y administradores de sistemas desplegar aplicaciones de manera consistente y escalable. Además, optimiza la confiabilidad y eficiencia, minimizando tanto el tiempo como los recursos necesarios para las operaciones diarias.

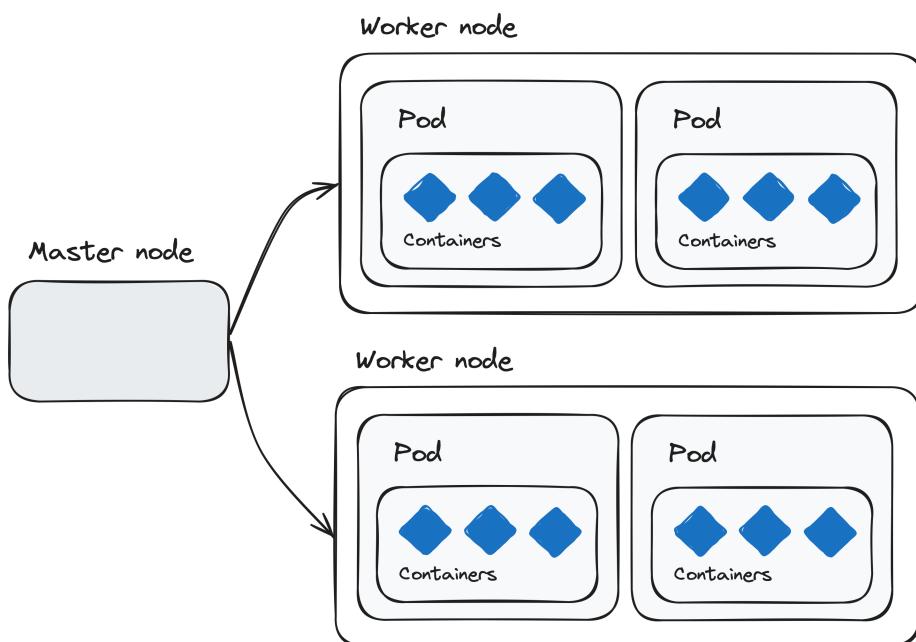


Figure 1. Clúster de Kubernetes (o k8s).

Lectura recomendada



- [¿Qué es Kubernetes?](#)
- [Según Google](#)
- [Según Azure](#)
- [Según AWS](#)

1.2. ¿Qué es Argo Workflows y por qué es importante?

Argo Workflows es un motor de flujo de trabajo de código abierto y nativo de contenedores, implementado como un Custom Resource Definition (CRD) en Kubernetes. Permite orquestar trabajos paralelos de manera eficiente, ideal para ejecutar tareas computacionalmente intensivas en menos tiempo. Su diseño optimizado para contenedores reduce la sobrecarga operativa, lo que lo hace crucial para entornos de producción flexibles y escalables.

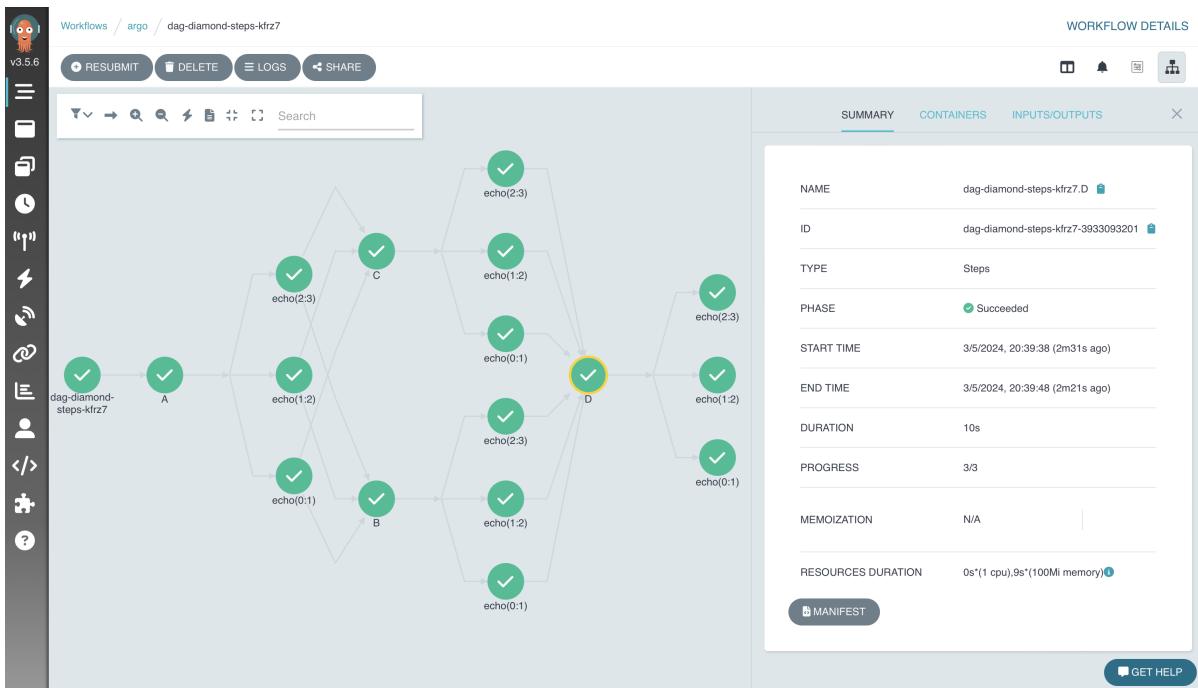


Figure 2. Interfaz de usuario (UI) de Argo Workflows.

Lectura recomendada



- [¿Qué es Argo Workflows?](#)
- [Documentación oficial](#)
- [Prácticas recomendadas](#)

1.3. Casos de Uso Comunes de Argo Workflows en la Industria

Argo Workflows se utiliza ampliamente para automatizar pipelines de machine learning, procesamiento de datos y lotes, y automatización de infraestructura. También es esencial en la implementación de CI/CD y en otros escenarios que requieren orquestación robusta de tareas. Su capacidad para manejar tareas altamente paralelas sin la complejidad de sistemas basados en VM lo hace popular en diversas aplicaciones industriales.



Lectura recomendada

- [Argo Workflows & Events 2023 User Survey Results](#)

Chapter 2. Instalación y Configuración

2.1. Requisitos Previos

Para seguir este seminario, se requiere tener conocimientos básicos de Kubernetes y contenedores. Se recomienda tener acceso a un clúster de Kubernetes local (por ejemplo, Minikube) o remoto para realizar las prácticas. Además, es necesario tener instalado el cliente kubectl y minikube en su máquina local.

- Para instalar minikube, siga las instrucciones en la página oficial de minikube: [minikube](#).
- Para instalar kubectl, siga las instrucciones en la página oficial de Kubernetes: [kubectl](#).
- La versión de kubernetes utilizada en este seminario es la v1.30.0.



2.1.1. Iniciar un Clúster de Kubernetes Local

Arranque un clúster de Kubernetes local utilizando Minikube con el siguiente comando:

```
minikube start
```

Salida esperada:

```
wilmermoina-rivera@wmoinar-pc /tmp > [21:14:55]
minikube start
😊 minikube v1.33.0 en Darwin 14.4.1 (arm64)
↳ Controlador docker seleccionado automáticamente. Otras opciones: qemu2, ssh
✖ Using Docker Desktop driver with root privileges
👍 Starting "minikube" primary control-plane node in "minikube" cluster
🌟 Pulling base image v0.0.43 ...
🕒 Descargando Kubernetes v1.30.0 ...
  > preloaded-images-k8s-v18-v1...: 319.81 MiB / 319.81 MiB 100.00% 34.93 M
  > gcr.io/k8s-minikube/kicbase...: 434.52 MiB / 434.52 MiB 100.00% 25.46 M
🔥 Creating docker container (CPUs=2, Memory=4600MB) ...
🕒 Preparando Kubernetes v1.30.0 en Docker 26.0.1...
  ▪ Generando certificados y llaves
  ▪ Iniciando plano de control
  ▪ Configurando reglas RBAC...
🕒 Configurando CNI bridge CNI ...
🕒 Verifying Kubernetes components...
  ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟 Complementos habilitados: storage-provisioner, default-storageclass
🎉 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

wilmermoina-rivera@wmoinar-pc /tmp > [21:15:42]
kubectl version
Client Version: v1.29.1
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
Server Version: v1.30.0
```

Figure 3. Minikube v1.33.0

2.2. Instalación de Argo Workflows

Una vez que Kubernetes esté listo, podemos instalar Argo Workflows en nuestro clúster. Argo se instala normalmente en un espacio de nombres llamado **argo**. Así que vamos a crearlo:

```
kubectl create ns argo
```

A continuación, dirígete a la página de [lanzamientos](#) y busca la versión que deseas usar (se recomienda la última versión).

De la sección [Controller and Server](#) copia y ejecuta los comandos de kubectl.

A continuación se muestra un ejemplo de los comandos de instalación; asegúrate de actualizar el comando para instalar el número de versión correcto:

```
kubectl apply -n argo -f https://github.com/argoproj/argo-workflows/releases/download/v3.5.6/install.yaml
```

2.3. Configuración Inicial y Verificación

¿Qué se ha instalado?

Tomará cerca de un minuto para que todos los despliegues estén disponibles. Vamos a ver qué se instaló mientras esperamos.

El [Workflow Controller](#) es responsable de ejecutar los [workflows](#):

```
kubectl -n argo get deploy workflow-controller
```

Y el Servidor de Argo proporciona una interfaz de usuario y API:

```
kubectl -n argo get deploy argo-server
```

Antes de continuar, vamos a esperar (alrededor de 1 a 2 minutos) a que nuestros despliegues estén disponibles:

```
kubectl -n argo wait deploy --all --for condition=Available --timeout 2m
```

Para establecer en espacio de nombres predeterminado para Argo Workflows, puedes ejecutar el siguiente comando:



```
kubectl config set-context --current --namespace=argo
```

Cómo configurar y verificar la instalación de Argo Workflows para asegurar su correcto funcionamiento.

Chapter 3. Primeros Pasos con Argo Workflows

3.1. ¿Qué es un workflow?

Un workflow se define como un recurso de Kubernetes. Cada workflow consta de una o más plantillas ([templates](#)), una de las cuales se define como el punto de entrada. Cada template puede ser de varios tipos; en este ejemplo, tenemos un template que es un contenedor.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: hello
spec:
  serviceAccountName: argo ①
  entrypoint: main ②
  templates:
    - name: main
      container: ③
        image: wmoinar/wholesay ④
        command: ["cowsay"]
```

- ① Esta es la cuenta de servicio con la que se ejecutará el workflow
- ② El primer template que se ejecutará en el workflow
- ③ Esta es una template de contenedor
- ④ La imagen que se utilizará para ejecutar el contenedor

Dado que un workflow es un [recurso](#) más de Kubernetes, puedes utilizar [kubectl](#) con ellos.

3.2. Crear un workflow:

```
kubectl -n argo apply -f examples/1_hello_world.yaml
```

Podemos verificar que el workflow se ha creado y completado con el siguiente comando:

```
kubectl -n argo get pod -l workflows.argoproj.io/workflow=hello
```

3.3. Usando la Interfaz de Usuario

[argo-server](#) maneja la autenticación de los clientes en la interfaz de usuario (UI), exigiendo que estos proporcionen su token de Kubernetes para autenticarse. Puedes encontrar más información en la documentación oficial de Argo: [Saber más](#).

Para facilitar el acceso durante este laboratorio, configuraremos el modo de autenticación a `server`, lo que nos permitirá omitir el proceso de inicio de sesión en la UI temporalmente. Además, desactivaremos la comunicación segura por HTTPS.

```
kubectl patch deployment argo-server \
--namespace argo \
--type='json' \
-p='[{"op": "replace", "path": "/spec/template/spec/containers/0/args", "value": [
"server",
"--auth-mode=server",
"--secure=false"
]},
{"op": "replace", "path":
"/spec/template/spec/containers/0/readinessProbe/httpGet/scheme", "value": "HTTP"
}]'
```

Redesplegamos el servidor de Argo para aplicar los cambios:

```
kubectl -n argo rollout status --watch --timeout=600s deployment/argo-server
```

Luego puedes visualizar la interfaz de usuario ejecutando un reenvío de puerto:

```
kubectl -n argo port-forward --address 0.0.0.0 svc/argo-server 2746:2746 > /dev/null &
```

Ahora puedes hacer clic aquí [Argo workflow](#) para acceder a la UI.

3.4. Ver el workflow

Abre la pestaña "Argo Server" y deberías ver la interfaz de usuario:

The screenshot shows the Argo Workflows interface. On the left is a sidebar with various icons for navigation. The main area displays a workflow named 'hello'. A large green circle with a white checkmark is prominently displayed, indicating success. To its right, the workflow details are listed:

NAME	hello
ID	hello
POD NAME	hello
HOST NODE NAME	minikube
TYPE	Pod
PHASE	Succeeded
START TIME	2/5/2024, 21:03:05 (34m50s ago)
END TIME	2/5/2024, 21:03:10 (34m45s ago)
DURATION	5s
PROGRESS	1/1
MEMOIZATION	N/A
RESOURCES DURATION	0s*(1 cpu),4s*(100Mi memory)

At the bottom of the details panel are buttons for 'MANIFEST', 'LOGS', and 'EVENTS', along with a 'GET HELP' button.

Figure 4. Workflow hello-work

3.5. Uso de la CLI

Para ejecutar workflows, la forma más fácil es utilizando la CLI de Argo, puedes descargarla de la siguiente manera (o en repositorio de [Argo](#)):

```
curl -sLO https://github.com/argoproj/argo-workflows/releases/download/v3.5.6/argo-linux-amd64.gz
gunzip argo-linux-amd64.gz
chmod +x argo-linux-amd64
mv ./argo-linux-amd64 /usr/local/bin/argo
```

Para verificar que se instaló correctamente:

```
argo version
```

Deberías ver algo como esto:

```
argo: v3.5.6
BuildDate: 2024-04-19T21:32:35Z
GitCommit: 555030053825dd61689a086cb3c2da329419325a
GitTreeState: clean
GitTag: v3.5.6
GoVersion: go1.21.9
Compiler: gc
Platform: darwin/amd64
```

¡Vamos a ejecutar un workflow!

```
argo submit -n argo --serviceaccount argo --watch examples/2_hello_world_cli.yaml
```

Deberías ver que el workflow se completa exitosamente después de aproximadamente 1 minuto:

```
Name: hello-world-q2dbl
Namespace: argo
ServiceAccount: argo
Status: Succeeded
Conditions:
  PodRunning: False
  Completed: True
Created: Fri May 03 00:00:28 +0200 (11 minutes ago)
Started: Fri May 03 00:00:28 +0200 (11 minutes ago)
Finished: Fri May 03 00:00:38 +0200 (11 minutes ago)
Duration: 10 seconds
Progress: 1/1
ResourcesDuration: 4s*(100Mi memory),0s*(1 cpu)
```

STEP	TEMPLATE	PODNAME	DURATION	MESSAGE
✓	hello-world-q2dbl	whalesay	hello-world-q2dbl	4s

Puedes listar los workflows fácilmente:

```
argo list -n argo
```

NAME	STATUS	AGE	DURATION	PRIORITY	MESSAGE
hello-world-q2dbl	Succeeded	33m	10s	0	
hello-work	Succeeded	2h	10s	0	
hello	Succeeded	3h	10s	0	

Obtén detalles sobre un workflow específico. `@latest` es un alias para el workflow más reciente:

```
argo get -n argo @latest
```

Y puedes ver los logs de ese workflow:

```
argo logs -n argo @latest
```

```
hello-world-q2dbl: -----
hello-world-q2dbl: < hello world >
hello-world-q2dbl: -----
hello-world-q2dbl: \## . ==
hello-world-q2dbl: \## ## ## ==
```

```
hello-world-q2dbl:      ## ## ## ##      ===
hello-world-q2dbl:      /'-----'\_/_/ ===
hello-world-q2dbl: ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
hello-world-q2dbl: \_----- o      _/ 
hello-world-q2dbl: \_ \_      _/ 
hello-world-q2dbl: \_ \_-----/
hello-world-q2dbl: time="2024-05-02T22:00:31.948Z" level=info msg="sub-process exited"
argo=true error=<nil>"
```

Finalmente, puedes obtener ayuda:

```
argo --help
```

Chapter 4. Tipos de Templates

Existen varios tipos de templates, divididas en dos categorías diferentes: **work** y **orchestration**.

La primera categoría define el trabajo a realizar. Esto incluye:

- Container
- Container Set
- Data
- Resource
- Script

La segunda categoría orquesta el trabajo:

- DAG (Directed Acyclic Graph)
- Steps
- Suspend



Lectura recomendada

- [Workflow Templates](#)

4.1. Template de Contenedor

Un template de contenedor es el tipo más común de template. Veamos un ejemplo completo:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: container-
spec:
  entrypoint: main
  templates:
  - name: main
    container:
      image: wmoinar/whalesay
      command: [cowsay]
      args: ["hello world"]
```

Ahora, ejecutemos el workflow:

```
argo submit -n argo --serviceaccount argo --watch examples/3_container_workflow.yaml
```

4.2. Etiquetas en un Template

Las etiquetas en un template (también conocidas como variables de template) son una forma de sustituir datos en tu workflow en tiempo de ejecución. Las etiquetas de template están delimitadas por `{{}}` y serán reemplazadas en tiempo de ejecución.

Las etiquetas disponibles dependen del tipo de template, y hay varias globales que puedes utilizar, como `{{workflow.name}}`, que se reemplaza por el nombre del workflow:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: template-tag-
spec:
  entrypoint: main
  templates:
    - name: main
      container:
        image: wmoinar/whalesay
        command: [cowsay]
        args: ["hello {{workflow.name}}"]
```

Envía este workflow:

```
argo submit -n argo --serviceaccount argo --watch examples/4_template_tag.yaml
```

Puedes ver la salida ejecutando:

```
argo logs @latest
```

Deberías ver algo como esto:

```
template-tag-jnfzc: -----
template-tag-jnfzc: < hello template-tag-jnfzc >
template-tag-jnfzc: -----
template-tag-jnfzc: \## . .
template-tag-jnfzc: \## ## ## == ==
template-tag-jnfzc: ## ## ## ## === ==
template-tag-jnfzc: /*****\ / === ==
template-tag-jnfzc: ~~~ {~~ ~~~ ~~~ ~~~ ~~~ / ===- ~~~
template-tag-jnfzc: \----- o -----/
template-tag-jnfzc: \----- / /
template-tag-jnfzc: \----- / /
template-tag-jnfzc: time="2024-05-03T11:04:21.923Z" level=info msg="sub-process
exited" argo=true error=<nil>"
```

4.3. Work Templates

¿Qué otros tipos de work templates existen?

Un **template de conjunto de contenedores** permite ejecutar múltiples contenedores en un solo pod. Esto es útil cuando quieres que los contenedores comparten un espacio de trabajo común, o cuando deseas consolidar el tiempo de activación del pod en un paso de tu workflow.

Un **template de datos** te permite obtener datos de un almacenamiento (por ejemplo, S3). Esto es útil cuando cada elemento de datos representa un trabajo que necesita ser realizado.

Un **template de recurso** te permite crear un recurso de Kubernetes y esperar a que cumpla con una condición (por ejemplo, exitoso). Esto es útil si deseas interoperar con otro sistema de Kubernetes, como AWS Spark EMR.

Un **template de script** te permite ejecutar un script en un contenedor. Esto es muy similar a un template de contenedor, pero con un script añadido.

Cada tipo de template que realiza un trabajo, lo hace ejecutando un pod. Puedes usar `kubectl` para ver estos pods:

```
kubectl get pods -l workflows.argoproj.io/workflow
```

Puedes identificar los pods de workflow por la etiqueta `workflows.argoproj.io/workflow`.

Deberías ver algo como esto:

NAME	READY	STATUS	RESTARTS	AGE
container-9br6p	0/2	Completed	0	115m
hello	0/2	Completed	0	17h
hello-work	0/2	Completed	0	16h
template-tag-jnfzc	0/2	Completed	0	79m

4.4. Template DAG

Un template DAG es un tipo común de template de orquestación. Veamos un ejemplo completo:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: dag-
spec:
  entrypoint: main
  templates:
    - name: main ①
      dag:
        tasks:
```

```

- name: a
  template: whalesay
- name: b
  template: whalesay
  dependencies:
    - a
- name: whalesay ②
  container:
    image: wmoinar/whalesay
    command: [cowsay]
    args: ["hello world"]

```

En este ejemplo, tenemos dos templates:

- ① El template "main" es nuestro nuevo DAG.
- ② El template "whalesay" es el mismo que en el ejemplo de contenedor.

El DAG tiene dos tareas: "a" y "b". Ambas ejecutan el template "whalesay", pero como "b" depende de "a", no comenzará hasta que "a" haya completado exitosamente.

Ejecutemos el workflow:

```
argo submit -n argo --serviceaccount argo --watch examples/5_dag_workflow.yaml
```

Deberías ver algo como esto:

STEP	TEMPLATE	PODNAME	DURATION	MESSAGE
✓ dag-whm2j	main			
└─✓ a	whalesay	dag-whm2j-whalesay-660405645	4s	
└─✓ b	whalesay	dag-whm2j-whalesay-610072788	5s	

¿Notaste cómo "b" no comenzó hasta que "a" completó?

Abre la pestaña del Servidor Argo ([Ir](#)) y navega al workflow, deberías ver dos contenedores.

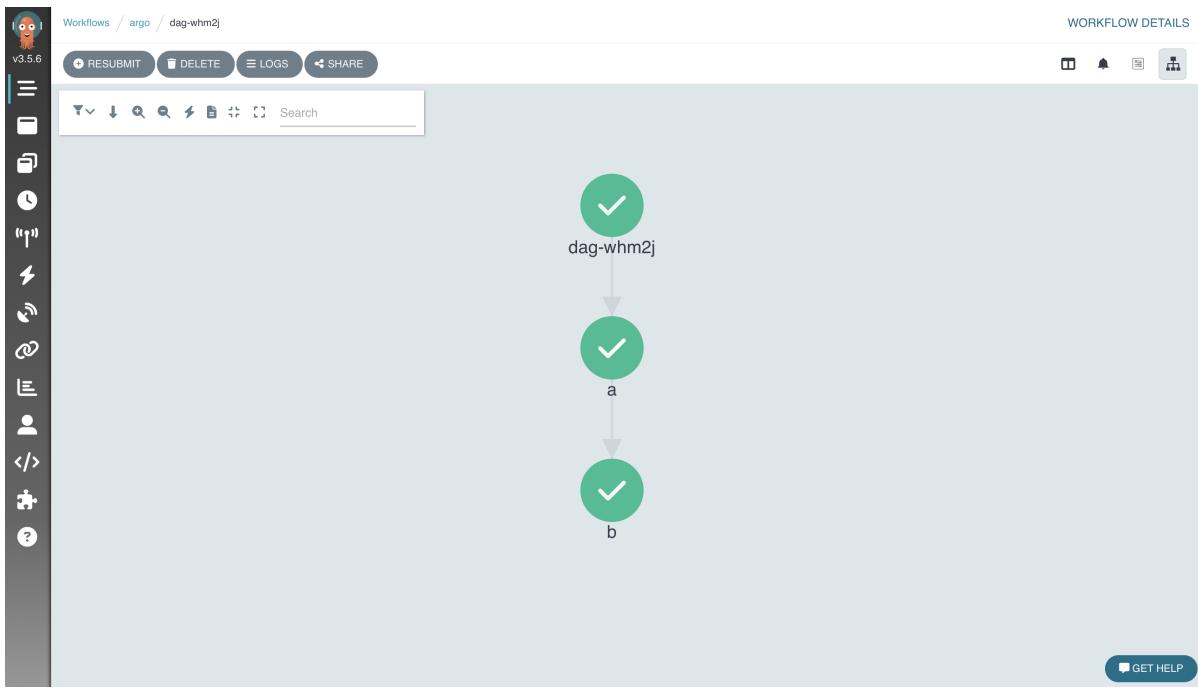


Figure 5. DAG Workflow

4.5. Bucles (Loops)

La capacidad de ejecutar trabajos de procesamiento paralelo en gran escala es una de las características clave de Argo Workflows. Veamos cómo utilizar los bucles para esto.

4.5.1. withItems

Un DAG te permite iterar sobre una serie de elementos utilizando `withItems`:

```

dag:
  tasks:
    - name: print-message
      template: whalesay
      arguments:
        parameters:
          - name: message
            value: "{{item}}"
      withItems:
        - "hello world"
        - "ciao mondo"
  
```

En este ejemplo, se ejecutará una vez por cada uno de los elementos listados. Aquí podemos ver una etiqueta de template. `{{item}}` será reemplazada por "hello world" y "ciao mondo". Los DAGs se ejecutan en paralelo, así que ambas tareas se iniciarán al mismo tiempo.

Ejecuta el workflow:

```
argo submit --watch examples/6_with_items_workflow.yaml
```

Deberías ver algo como esto:

STEP	TEMPLATE	PODNAME
DURATION	MESSAGE	
✓ with-items-49mjs	main	
└─✓ print-message(0:hello world)	whalesay	with-items-49mjs-whalesay-1503367336
4s		
└─✓ print-message(1:ciao mondo)	whalesay	with-items-49mjs-whalesay-1791799948
5s		

Observa cómo los dos elementos se ejecutaron al mismo tiempo.

4.5.2. withSequence

También puedes iterar sobre una secuencia de números utilizando `withSequence`:

```
dag:  
  tasks:  
    - name: print-message  
      template: whalesay  
      arguments:  
        parameters:  
          - name: message  
            value: "{{item}}"  
      withSequence:  
        count: 4
```

Como de costumbre, ejecútalo:

```
argo submit --watch examples/7_with_sequence_workflow.yaml
```

Deberías ver algo como esto:

STEP	TEMPLATE	PODNAME	DURATION
MESSAGE			
✓ with-sequence-tnm67	main		
└─✓ print-message(0:0)	whalesay	with-sequence-tnm67-whalesay-1551215270	6s
└─✓ print-message(1:1)	whalesay	with-sequence-tnm67-whalesay-2190561570	5s
└─✓ print-message(2:2)	whalesay	with-sequence-tnm67-whalesay-1568889174	8s
└─✓ print-message(3:3)	whalesay	with-sequence-tnm67-whalesay-1540049194	7s

Observa cómo se ejecutaron 5 pods al mismo tiempo y que sus nombres incluyen el valor del ítem, indexado desde cero.

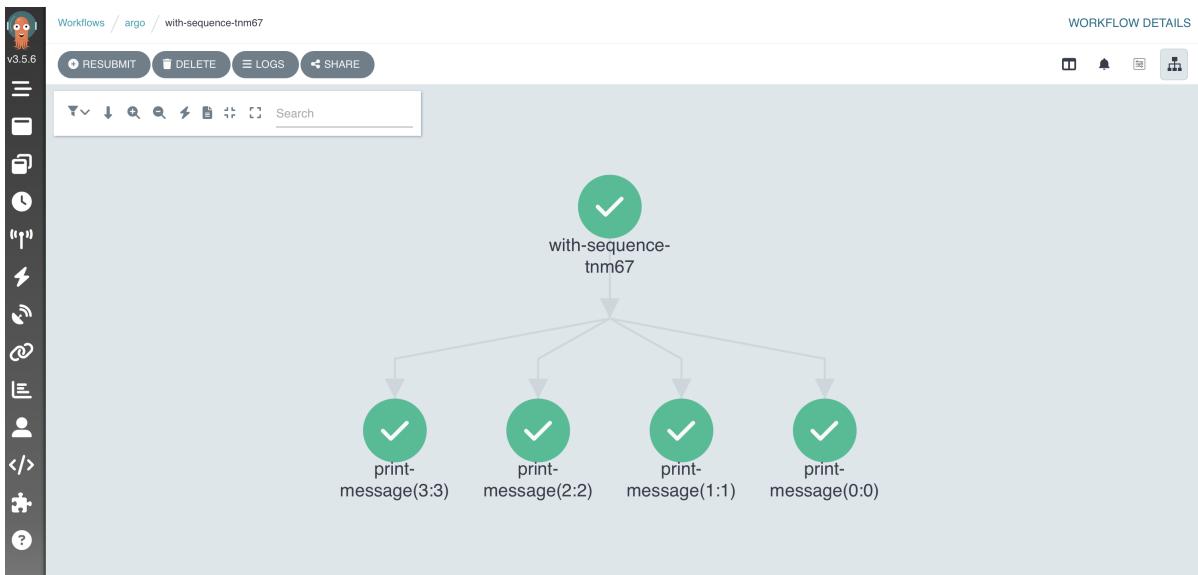


Figure 6. WitSequence Workflow

4.6. Templates de Orquestación

Hemos aprendido que un **template DAG** es un tipo de template de orquestación. ¿Qué otros tipos de templates de orquestación existen?

Los **Steps template** te permite ejecutar una serie de pasos en secuencia.

Los **suspend template** te permite suspender automáticamente un workflow, por ejemplo, mientras se espera una aprobación manual o mientras un sistema externo realiza algún trabajo.

4.7. Manejador de Salida

Si necesitas realizar una tarea después de que algo haya terminado, puedes utilizar un manejador de salida. Los manejadores de salida se especifican usando **onExit**:

```

dag:
  tasks:
    - name: a
      template: whalesay
      onExit: tidy-up
  
```

Simplemente indica el nombre del template que debe ejecutarse al finalizar.

Veamos un ejemplo completo:

```
cat examples/8_exit_handler_workflow.yaml
```

Ejecútalo:

```
argo submit --watch examples/8_exit_handler_workflow.yaml
```

Deberías ver:

STEP MESSAGE	TEMPLATE	PODNAME	DURATION
✓ exit-handler-7mq52 main			
└─✓ a	whalesay	exit-handler-7mq52-whalesay-970555148	4s
└─✓ a.onExit	tidy-up	exit-handler-7mq52-tidy-up-3394673695	5s

Se puede utilizar varios manejadores de salida en un solo template, y un manejador global de salida para todo el workflow. por ejemplo:

```
entrypoint: main
onExit: global-cleanup ①
templates:
- name: main
  dag:
    tasks:
      - name: step-one
        template: primary-action
        onExit: cleanup-after-step-one ②

      - name: step-two
        template: secondary-action
        dependencies: [step-one]
        onExit: cleanup-after-step-two ③

    - name: primary-action
      container:
        image: wmoinar/whalesay
        command: [cowsay]
        args: ["Doing primary action"]

    - name: secondary-action
      container:
        image: wmoinar/whalesay
        command: [cowsay]
        args: ["Doing secondary action"]

    - name: cleanup-after-step-one
      container:
        image: wmoinar/whalesay
        command: [cowsay]
        args: ["Cleanup after step one"]

    - name: cleanup-after-step-two
      container:
```

```

image: wmoinar/whalesay
command: [cowsay]
args: ["Cleanup after step two"]

- name: global-cleanup
  container:
    image: wmoinar/whalesay
    command: [cowsay]
    args: ["Performing global cleanup"]

```

- ① Este manejador se ejecuta cuando todo el workflow finaliza o falla.
- ② Manejador de salida específico para esta tarea.
- ③ Otro manejador de salida específico para esta segunda tarea.

Ejecútalo:

```
argo submit --watch examples/9_exit_multi_handler_workflow.yaml
```

Deberías ver:

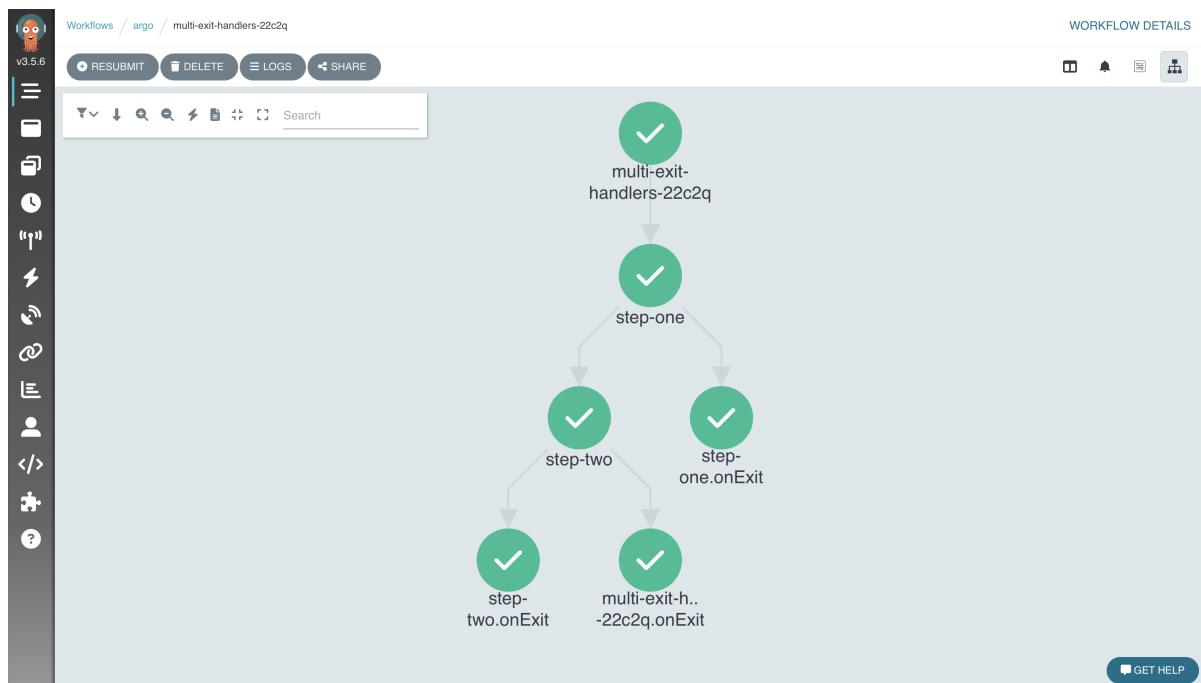


Figure 7. Multiple Exit Handler Workflow

Chapter 5. Entradas y Salidas

5.1. Parámetros

Los parámetros son un tipo de entrada o salida que consisten en valores de texto simples. A diferencia de los artefactos, que pueden ser archivos o conjuntos de datos complejos, los parámetros son ideales para transmitir información sencilla y directa a lo largo de un workflow.

5.2. Parámetros de Entrada

Veamos un ejemplo:

```
- name: main
  inputs:
    parameters:
      - name: message
  container:
    image: wmoinar/whalesay
    command: [cowsay]
    args: ["{{inputs.parameters.message}}"]
```

Este template declara que tiene un parámetro de entrada llamado "message".

Ve el workflow completo:

```
cat examples/10_input_parameters_workflow.yaml
```

Observa cómo el propio workflow tiene argumentos.

Ejecútalo:

```
argo submit --watch examples/10_input_parameters_workflow.yaml
```

Deberías ver:

STEP	TEMPLATE	PODNAME	DURATION	MESSAGE
✓ input-parameters-pxrmj	main	input-parameters-pxrmj	7s	

Si un workflow tiene parámetros, puedes cambiar los parámetros usando **-p** con la CLI:

```
argo submit --watch examples/10_input_parameters_workflow.yaml -p message='¡Bienvenido  
a Argo!'
```

Deberías ver:

STEP	TEMPLATE	PODNAME	DURATION	MESSAGE
✓ input-parameters-qg452	main	input-parameters-qg452	11s	

Revisa la salida en los logs:

```
argo logs @latest
```

Deberías ver:

```
: -----  
: < ¡Bienvenido a Argo! >  
: -----  
: \      ## .  
:   \    ## ## ==  
:     ## ## ## ## ===  
:     /*****\__/_/ ===  
: ~~~ {~~ ~~~ ~~~ ~~~ ~~~ / ===- ~~~  
:     \____ o __/  
:       \ \ _/  
:       \_\_/_/  
: time="2024-05-04T21:43:14.087Z" level=info msg="sub-process exited" argo=true  
error=<nil>
```

5.3. Parámetros de Salida

Los parámetros de salida pueden provenir de varios lugares, pero típicamente el más versátil es de un archivo. En este ejemplo, el contenedor crea un archivo con un mensaje en él:

```
- name: whalesay  
  container:  
    image: wmoinar/whalesay  
    command: [sh, -c] //  
    args: ["echo -n hello world > /tmp/hello_world.txt"] ①  
  outputs:  
    parameters:  
      - name: hello-param  
        valueFrom:  
          path: /tmp/hello_world.txt ②
```

① Crea un archivo con un mensaje.

② Referencia al archivo creado.

En los templates de tipo DAG y de pasos (steps), es posible utilizar las salidas de una tarea como

entradas para otra tarea, mediante el uso de etiquetas de template. Esto facilita la interconexión y dependencia directa entre tareas dentro del workflow.

```
dag:  
  tasks:  
    - name: generate-parameter ①  
      template: whalesay  
    - name: consume-parameter ②  
      template: print-message  
      dependencies:  
        - generate-parameter  
      arguments:  
        parameters:  
          - name: message  
            value: "{{tasks.generate-parameter.outputs.parameters.hello-param}}" ③
```

- ① La tarea que genera el parámetro.
- ② La tarea que consume el parámetro.
- ③ El valor del parámetro generado.

Ve el workflow completo:

```
cat examples/11_output_parameters_workflow.yaml
```

Ejecútalo:

```
argo submit --watch examples/11_output_parameters_workflow.yaml
```

Deberías ver:

STEP	TEMPLATE	PODNAME
DURATION	MESSAGE	
✓ 5s	parameters-t64zg	main
	└─✓ generate-parameter	whalesay
	└─✓ consume-parameter	print-message
		parameters-t64zg-whalesay-2318848988
		parameters-t64zg-print-message-796340527
4s		

Lectura recomendada



- [Parámetros](#)
- [Parámetros de Entrada](#)

5.4. Artefactos

Un artefacto es simplemente un archivo que se comprime y almacena en un almacenamiento de objetos (como [S3](#), [MinIO](#), [GCS](#), etc.).

En Argo, hay dos tipos de artefactos:

- Un **artefacto de entrada** es un archivo descargado desde el almacenamiento (p. ej., S3) y montado como un volumen dentro del contenedor.
- Un **artefacto de salida** es un archivo creado en el contenedor que luego se sube al almacenamiento.

Los artefactos generalmente se cargan en un bucket dentro de algún tipo de almacenamiento como AWS S3 o GCP GCS. Llamamos a este almacenamiento un repositorio de artefactos. En este laboratorio usaremos MinIO.

5.4.1. Artefactos de Salida

Cada tarea dentro de un workflow puede producir artefactos de salida. Para especificar un artefacto de salida, debes incluir **outputs** en el manifiesto. Cada artefacto de salida declara:

```
- name: save-message
  container:
    image: wmoina/whalesay
    command: [sh, -c]
    args: ["cowsay hello world > /tmp/hello_world.txt"]
  outputs:
    artifacts:
      - name: hello-art ①
        path: /tmp/hello_world.txt ②
```

① La ruta dentro del contenedor donde se puede encontrar.

② Un nombre para poder referenciarlo.

Cuando el contenedor se completa, el archivo se copia fuera de él, se comprime y se almacena.

Los archivos también pueden ser directorios, por lo que cuando se encuentra un directorio, todos los archivos se comprimen en un archivo y se almacenan.

5.4.2. Artefactos de Entrada

Para declarar un artefacto de entrada, debes incluir **inputs** en el manifiesto. Cada artefacto de entrada debe declarar:

```
- name: print-message
  inputs:
    artifacts:
      - name: message ①
```

```
path: /tmp/message ②
container:
  image: wmoinar/whalesay
  command: [sh, -c]
  args: ["cat /tmp/message"]
```

① Su nombre.

② La ruta donde debe ser creado.

Si el artefacto era un directorio comprimido, se descomprimirá y desempaquetará en la ruta especificada.

5.4.3. Entradas y Salidas (artefactos)

- Para ejecutar este ejemplo necesitaremos instalar MinIO. Sigue las instrucciones en la sección de [Argo-Events.Prerrequisitos](#).
- Asegúrate de que MinIO esté instalado y funcionando antes de continuar.
- Crear un bucket llamado `argo-artifacts` en MinIO.

No puedes usar entradas y salidas de forma aislada; necesitas combinarlas utilizando un template de pasos o DAG, como en el siguiente ejemplo:

```
- name: main
dag:
  tasks:
    - name: generate-artifact
      template: save-message
    - name: consume-artifact
      template: print-message
      dependencies:
        - generate-artifact
      arguments:
        artifacts:
          - name: message
            from: "{{tasks.generate-artifact.outputs.artifacts.hello-art}}"
```

En el ejemplo anterior, `arguments` se usa para declarar el valor de la entrada del artefacto. Esto utiliza una etiqueta de template. En este ejemplo, `{{tasks.generate-artifact.outputs.artifacts.hello-art}}` se convierte en la ruta del artefacto en el repositorio.

La tarea `consume-artifact` debe ejecutarse después de `generate-artifact`, por lo que usamos `dependencies` para declarar esa relación.

Veamos el workflow DAG completo:

```
cat examples/12_artifacts_workflow.yaml
```

Ejecutemos un ejemplo:

```
argo submit --watch examples/12_artifacts_workflow.yaml
```

Deberías ver:

STEP	TEMPLATE	PODNAME	DURATION	MESSAGE
✓ artifacts-qvcpn	main			
└─✓ generate-artifact	save-message	artifacts-qvcpn-3260493969	7s	
└─✓ consume-artifact	print-message	artifacts-qvcpn-2991781604	8s	

Lectura recomendada

- [Artifacts overview](#)
- [Configuring an artifact repository](#)
- [Key-only artifacts](#)
- [Referencing artifact repositories](#)



Chapter 6. Argo Events

6.1. Prerequisites

Para ejecutar los ejemplos de Argo Events, primero necesitas tener Minio instalado. Puedes seguir la guía oficial de Minio en el siguiente enlace: [Guía de inicio rápido de Minio](#), o puedes usar el siguiente comando para instalar Minio en tu clúster de Minikube:

```
kubectl apply -f examples/minio.yaml
```

Para verificar que Minio se ha instalado correctamente, ejecuta el siguiente comando:

```
kubectl -n argo wait deploy minio --for condition=Available --timeout 2m
```

Dar acceso a Minio para almacenar artefactos de Argo Events:

```
kubectl apply -f examples/minio_artifacts.yaml
```

Después de aplicar las configuraciones, reinicia el controlador de workflows:

```
kubectl rollout restart deployment workflow-controller -n argo
```

6.2. Conecta tu Navegador al Servidor MinIO

Para acceder a la interfaz de Minio, crea un reenvío de puerto:

```
kubectl -n argo port-forward --address 0.0.0.0 svc/minio 9001:9001 > /dev/null & #
```

Accede a la Consola de MinIO abriendo un navegador en la máquina local y navegando a <http://127.0.0.1:9001>. Inicia sesión en la Consola con las credenciales `argoproj | UVseminario24`. Estas son las credenciales predeterminadas del usuario raíz.

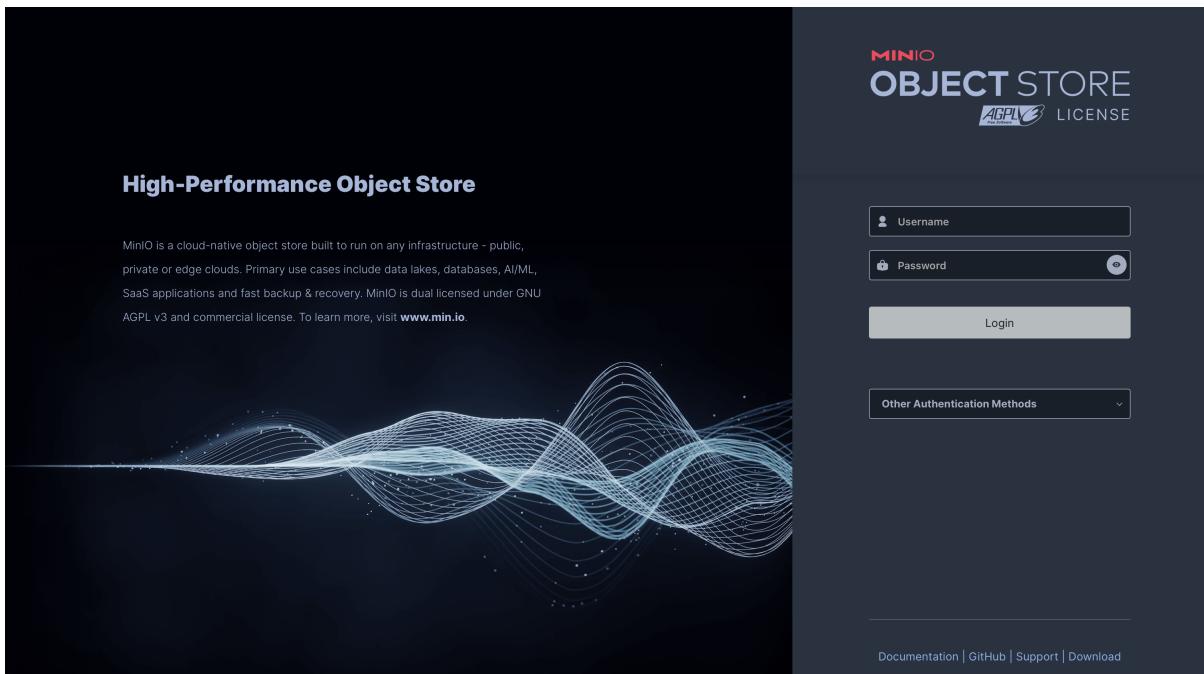


Figure 8. Minio Dashboard

6.3. Instalar y Configurar Argo Events

Argo Events normalmente se instala en un espacio de nombres llamado argo-events, así que vamos a crearlo:

```
kubectl create ns argo-events
```

A continuación, dirígete a la página de versiones y encuentra la versión que deseas usar (se prefiere la última versión completa).

A continuación, un ejemplo del comando de instalación:

```
kubectl apply -n argo-events -f https://github.com/argoproj/argo-events/releases/download/v1.9.1/install.yaml
```

Argo Events también requiere la creación de un eventbus. Este es un punto central al que se envían todos los eventos. El eventbus se crea en el mismo espacio de nombres que el controlador de Argo Events.

Hay un ejemplo disponible en el repositorio de GitHub de Argo Events. Ejecuta el siguiente comando para crear el eventbus:

```
kubectl apply -n argo-events -f https://raw.githubusercontent.com/argoproj/argo-events/stable/examples/eventbus/native.yaml
```

Queremos que Argo Events active un flujo de trabajo cuando se agregue un archivo a Minio. Para lograr esto, agregaremos un eventsource de Minio que escuchará los eventos de Minio.

El eventsource requerirá credenciales de Minio. Proporcionaremos estas credenciales en forma de un secreto. Visualiza el secreto con:

```
cat examples/minio_secret.yaml
```

Implementa el secreto con:

```
kubectl apply -n argo-events -f examples/minio_secret.yaml
```

Antes de implementar el eventsource, necesitamos crear un bucket en Minio. Puedes hacerlo desde la interfaz, utiliza el nombre **argoproj**.

Una vez que hayas creado el bucket, visualiza el eventsource con:

```
apiVersion: argoproj.io/v1alpha1
kind: EventSource
metadata:
  name: minio
  namespace: argo-events
spec:
  minio:
    example:
      bucket:
        name: argoproj ①
        endpoint: minio.argo:9000 ②
        events: ③
          - s3:ObjectCreated:Put
          - s3:ObjectRemoved:DeleteMarkerCreated

    insecure: true ④
    accessKey: ⑤
      key: accesskey
      name: minio-creds
    secretKey: ⑥
      key: secretkey
      name: minio-creds
```

① Nombre del bucket

② Punto de conexión del servicio S3

③ Lista de eventos a los que suscribirse. Visita <https://docs.minio.io/docs/minio-bucket-notification-guide.html>

④ Tipo de conexión

⑤ Clave dentro del secreto de K8s que almacena la clave de acceso

⑥ Clave dentro del secreto de K8s que almacena la clave secreta

Implementa el eventsource con:

```
kubectl apply -n argo-events -f examples/minio_eventsources.yaml
```

6.3.1. ¿Qué se instaló?

Puede tardar aproximadamente 1 minuto en que todas las implementaciones estén disponibles. Veamos qué se instaló mientras esperamos.

El Gestor de Controladores de Eventos:

```
kubectl -n argo-events get deploy controller-manager
```

Un estado de eventbus statefulset:

```
kubectl -n argo-events get statefulsets eventbus-default-stan
```

Un pod de eventsource de Minio:

```
kubectl -n argo-events get pod -l eventsource-name=minio
```

6.4. Agregar un Sensor y Disparador de Minio

Necesitamos instalar un Sensor que sea llamado por el EventSource. El Sensor es responsable de activar la creación de un flujo de trabajo cuando se recibe un evento. Veamos la configuración del Sensor:

```
cat examples/minio_sensor.yaml
```

Despliega con el siguiente comando:

```
kubectl apply -n argo-events -f examples/minio_sensor.yaml
```

Esto crea finalmente un pod en nuestro espacio de nombres `argo-events` que es responsable de activar flujos de trabajo cuando se reciben eventos.

"This service account does not have permission to create workflows in the argo namespace. We therefore need to give it permission to do so."

Necesitamos otorgar permisos a la cuenta de servicio `default` para crear workflows. Para hacer esto, necesitamos crear un `ClusterRole` y un `ClusterRoleBinding`:

```
kubectl apply -f examples/minio_sa.yaml
```

Podemos ver que está configurado para observar el bucket de Minio llamado `argoproj`. Si creamos o eliminamos un archivo en este bucket, se activará un evento.

6.5. Crear un Flujo de Trabajo

Cargamos un archivo en el bucket `argoproj` en Minio. Puedes hacerlo desde la interfaz de Minio:

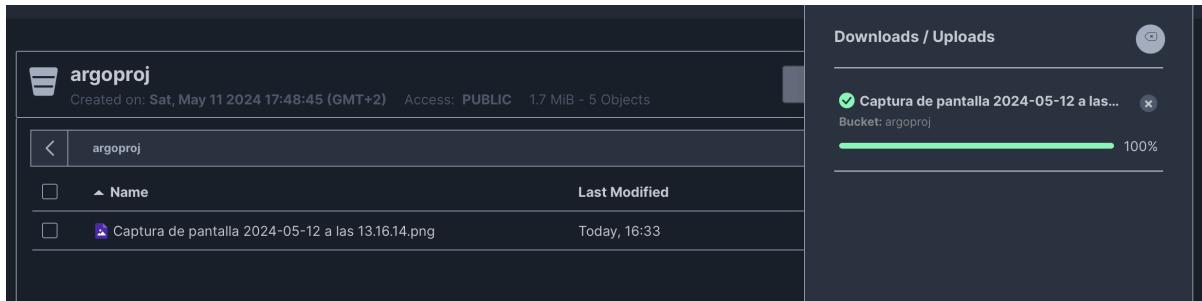


Figure 9. Carga de Archivo

Esto debería activar un evento en el EventSource de Minio. El Sensor debería detectar este evento y activar un workflow.

Veamos los registros del pod del sensor:

```
kubectl -n argo-events logs -l sensor-name=minio
```

```
{
  "level": "info",
  "ts": 1715524387.6334236,
  "logger": "argo-events.sensor",
  "caller": "sensors/listener.go:433",
  "msg": "Successfully processed trigger 'minio-workflow-trigger'",
  "sensorName": "minio",
  "triggerName": "minio-workflow-trigger",
  "triggerType": "Kubernetes",
  "triggeredBy": [
    "example-dep"
  ],
  "triggeredByEvents": [
    "31336131313836312d313362342d343766662d613663332d633336303932623961643035"
  ]
}
```

Si todo ha ido bien hasta ahora, deberías ver un mensaje similar al anterior. Esto indica que el Sensor ha detectado un evento y ha activado un flujo de trabajo.

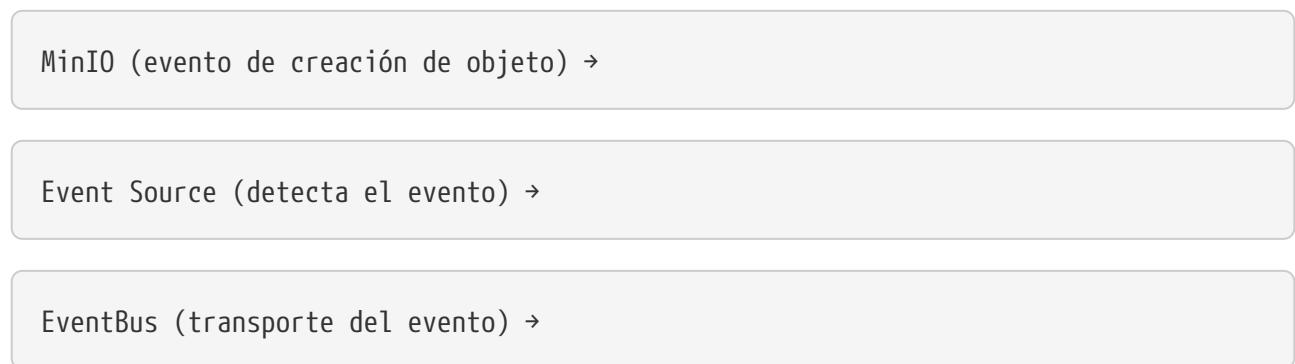
Desde la interfaz de Argo, puedes ver el flujo de trabajo activado:



Figure 10. Argo UI

6.5.1. Proceso Detallado

1. Evento en MinIO:
 - Un usuario sube un archivo a `my-bucket` en MinIO.
 - MinIO genera un evento `s3:ObjectCreated:*`.
 2. Detección del Evento por el Event Source:
 - El Event Source `minio-event-source` monitorea `argoproj` y detecta el evento de creación de objeto.
 - El Event Source envía el evento al EventBus.
 3. Transporte del Evento por el EventBus:
 - El EventBus recibe el evento del Event Source y lo envía a los Sensors suscritos.
 4. Activación del Sensor:
 - El Sensor `minio-sensor` está suscrito a los eventos del Event Source `minio-event-source`.
 - El Sensor recibe el evento y activa su trigger configurado.
 5. Ejecución del Workflow en Argo Workflows:
 - El trigger del Sensor somete un nuevo Workflow de Argo (`triggered-workflow`).
 - El Workflow se ejecuta según la configuración definida, en este caso, imprime un mensaje usando la imagen `wmoinar/whalesay`.
 6. Diagrama del Flujo de Trabajo:



Sensor (recibe el evento y activa el trigger) →

Argo Workflows (ejecuta el workflow).

Chapter 7. Conclusiones

En este laboratorio, hemos explorado los conceptos fundamentales y la configuración básica de Argo Workflows, una poderosa herramienta de orquestación de flujos de trabajo para Kubernetes. A continuación, se resumen los puntos clave tratados:

1. Comprensión de Argo Workflows:

- Argo Workflows es una herramienta nativa de Kubernetes que facilita la creación y gestión de flujos de trabajo complejos.
- Permite la definición de tareas que se ejecutan en contenedores y su organización en estructuras secuenciales o paralelas.

2. Instalación y Configuración:

- Instalamos Argo Workflows en un clúster de Kubernetes, asegurando que todos los componentes necesarios estuvieran en funcionamiento.
- Configuramos y verificamos la instalación para garantizar que el entorno esté listo para ejecutar flujos de trabajo.

3. Creación y Ejecución de Workflows:

- Aprendimos a definir workflows en YAML, especificando tareas y sus relaciones mediante templates.
- Ejecutamos workflows utilizando tanto la CLI de Argo como la interfaz de usuario web, observando cómo se desarrollan y completan las tareas.

4. Tipos de Templates:

- Exploramos diversos tipos de templates, incluyendo contenedores, scripts, y DAGs, que nos permiten modelar tareas de manera flexible y escalable.
- Entendimos cómo utilizar los templates de orquestación como DAG y Steps para estructurar flujos de trabajo complejos.

5. Argo Events:

- Configuramos Argo Events para reaccionar a eventos externos, integrando MinIO como fuente de eventos.
- Implementamos Event Sources, EventBus y Sensors para detectar eventos en MinIO y desencadenar la ejecución de workflows en Argo.

6. Automatización y Escalabilidad:

- Demostramos cómo Argo Workflows y Argo Events juntos pueden automatizar procesos en respuesta a eventos, mejorando la eficiencia y escalabilidad de las operaciones.

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFlvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
```