UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

Ciclo completo de CI/CD con Dagger y Kubernetes

Autor:
Daniel Vieites Torres

Tutores:
Juan Carlos Pichel Campos
Francisco Maseda Muiño

Grado en Ingeniería Informática

2025

Trabajo de Fin de Grado presentado en la Escuela Técnica Superior de Ingeniería de la Universidad de Santiago de Compostela para la obtención do Grado en Ingeniería Informática

Resumen

Ante la creciente complejidad de los flujos DevOps, resulta conveniente identificar herramientas de *Continuous Integration/Continuous Delivery* (CI/CD) que unifiquen los *pipelines* y reduzcan los costes de mantenimiento. Este Trabajo de Fin de Grado investiga si Dagger, un motor programable basado en contenedores, puede simplificar y acelerar la integración y entrega continuas frente a los métodos convencionales predominantes.

Se construyó un *monorepo* con una aplicación de prueba (*frontend*: Vue + Typescript; *backend*: Typescript; MongoDB) y se diseñaron e implementaron dos módulos de Dagger para los ciclos de CI y CD, empleando el SDK que proporciona Dagger para el lenguaje de programación Go. Además, se desplegó la aplicación en tres *clusters* de KinD, gestionados por ArgoCD bajo filosofía GitOps, con la infraestructura definida mediante Helm y Kubernetes.

La evaluación cuantitativa incluyó una prueba en la que se ejecutó la función `endtoend` del módulo de CI. Esta mostró una reducción del tiempo de ejecución del 40 % cuando los parámetros de entrada cambiaban entre ejecuciones, y del 94 % cuando los parámetros permanecían sin cambios. Esto es gracias al sistema de gestión de caché nativo de Dagger. El análisis cualitativo destacó ventajas en cuanto al mantenimiento, la portabilidad (ejecución sobre *runtime* OCI), la reproducibilidad y la escalabilidad, frente a una curva de aprendizaje moderada.

En conclusión, Dagger se posiciona como una alternativa sólida para estandarizar y modernizar los ciclos de vida del *software*, facilitando la integración continua y reduciendo la dependencia de *scripts* heterogéneos. Entre otras mejoras futuras, se sugieren: validar el funcionamiento de los módulos en *clusters* gestionados en la nube, gestionar de mejor manera los secretos y añadir herramientas de monitorización y de análisis de imágenes de Docker.

Índice general

Índice de figuras

Índice de tablas

Índice de Listings

Capítulo 1

Introducción

En este Trabajo de Fin de Grado se pretende demostrar y evaluar la viabilidad, eficiencia y flexibilidad de Dagger[1] como motor programable de ciclos de CI/CD (*Continuous Integration/Continuous Delivery*)[3, 6], con el fin de estandarizar y modernizar los ciclos de vida del desarrollo de software. Esto es importante debido a la actual complejidad que involucra mantener *pipelines* de este tipo en entornos de desarrollo con un gran volumen de tecnologías.

1.1. Objetivos

Objetivos principales

Se usará Dagger con el fin de implementar la lógica necesaria para llevar a cabo los ciclos de CI y CD de una aplicación de prueba. Se evaluarán las ventajas que tiene su uso frente a métodos convencionales, entre las que destacarán su portabilidad, al correr sobre un *runtime* de OCI (*Open Container Initiative*[26]), como Docker[16]; y su capacidad programática, ya que se pueden crear *pipelines* implementando funciones en el lenguaje conocido para el desarrollador. En vez de coordinar *scripts* creados a mano en diferentes entornos, el programador es capaz de componer acciones reusables, utilizando un lenguaje de programación y una API (*Application Programming Interface*) a su disposición.

Se van a proporcionar ejemplos de módulos creados con Dagger, los cuales estarán especialmente diseñados para cumplir los ciclos tanto de CI como de CD de la aplicación de prueba. De esta manera se podrá comprobar que este mismo proceso se puede llevar a cabo para cualquier aplicación.

Lo que sigue son las preguntas a las que este Trabajo de Fin de Grado busca responder:

- ¿Qué grado de complejidad tiene desarrollar módulos de Dagger para la gestión de un ciclo de CI/CD de una aplicación?
- ¿Vale realmente la pena aprender a utilizar esta herramienta?

- ¿Es capaz de aumentar la velocidad de desarrollo de una aplicación?
- ¿Es fácilmente integrable en cualquier tipo de aplicación?
- ¿Qué puntos débiles corrige Dagger frente al uso de otros métodos convencionales?
- ¿Qué desafíos, limitaciones o desventajas se encuentran al trabajar con Dagger?

Objetivos secundarios

Para lograr los objetivos principales es necesario llevar a cabo varios pasos:

- Creación de un *monorepo*[7] en GitHub.
- Diseño y creación de una aplicación de prueba. Esta consistirá en una página web de gestión de un zoo, la cual realizará peticiones a una API REST que estará conectada a una base de datos.
- Implementación de un *pipeline* CI/CD con Dagger.
- Entorno orquestado por Kubernetes[8], configurado a través de una Chart de Helm[9].
- Análisis comparativo de las ventajas que ofrece Dagger frente a métodos convencionales.

1.2. Estructura de la memoria

- Capítulo 1: Introducción.

  Este capítulo, en el cual se describen la finalidad del proyecto, las tecnologías a utilizar, de manera breve; y la estructura, a grandes rasgos, del trabajo en sí.

- Capítulo 2: Estado del arte y fundamentos teóricos.

  En el segundo capítulo se detallan los conceptos más importantes de CI/CD. Además, se estudia la evolución de las herramientas DevOps[10], incluyendo Dagger como una de las últimas y más innovadoras herramientas en este sector.

- Capítulo 3: Diseño y arquitectura del sistema.

  Aquí se describe la organización de repositorio, así como las tecnologías utilizadas para implementar cada una de las piezas de *software* del trabajo. Por lo tanto, se habla de la aplicación de prueba y de los módulos de Dagger. También se explica cómo se ha organizado la infraestructura de despliegue.

- Capítulo 4: Implementación del *pipeline* con Dagger.

  Aquí se detallan los pasos que se han dado para crear los *pipelines* con Dagger, utilizando el SDK para definirlos como código. Este es el núcleo del proyecto.

- Capítulo 5: Pruebas y resultados.

  En este capítulo se presentan las pruebas que se han llevado a cabo. Se habla de las dificultades que se han tenido, así como de las ventajas que ofrece Dagger frente a otras tecnologías, aportando comparaciones cuantitativas y cualitativas.

- Capítulo 6: Conclusiones y líneas futuras.

  Finalmente, se resumen los hechos que se han obtenido, se valora el resultado final del uso de Dagger y se indica si ha cumplido con las expectativas. Además, se añaden puntos de mejora o extensiones del proyecto.

Capítulo 2

Estado del arte y fundamentos teóricos

Antes de empezar a escribir código, se deben entender los conceptos fundamentales que permitirán llevar a cabo este Trabajo de Fin de Grado.

Dagger busca mejorar el desarrollo de un ciclo completo de CI/CD de una aplicación. Por lo tanto, es fundamental definir los conceptos de *Continuous Integration* y el *Continuous Delivery*. Una vez se comprenda a qué se refieren esos términos, se podrán entender los métodos y tecnologías convencionales que permiten implementar dichos procesos. Será entonces cuando se pueda introducir Dagger, un herramienta innovadora para implementar *pipelines*.

2.1. CI/CD

CI/CD son las siglas de *Continuous Integration*/*Continuous Delivery* o, en casos más específicos, este último también se puede conocer como *Continuous Deployment*. Se trata de un conjunto de pasos automatizados, utilizados en el desarrollo de *software* para llevar el código desde su implementación inicial hasta el despliegue de la aplicación. Estos pasos incluyen:

- Integración de cambios en el código.
- Compilación de la aplicación con los cambios realizados.
- Realización de pruebas.
- Creación y publicación de imágenes de Docker y paquetes NPM.
- Despliegue de la aplicación.

Continuous Integration

Se basa en la integración de código de manera constante, día a día, en un repositorio compartido por programadores. Cada uno de los programadores realiza

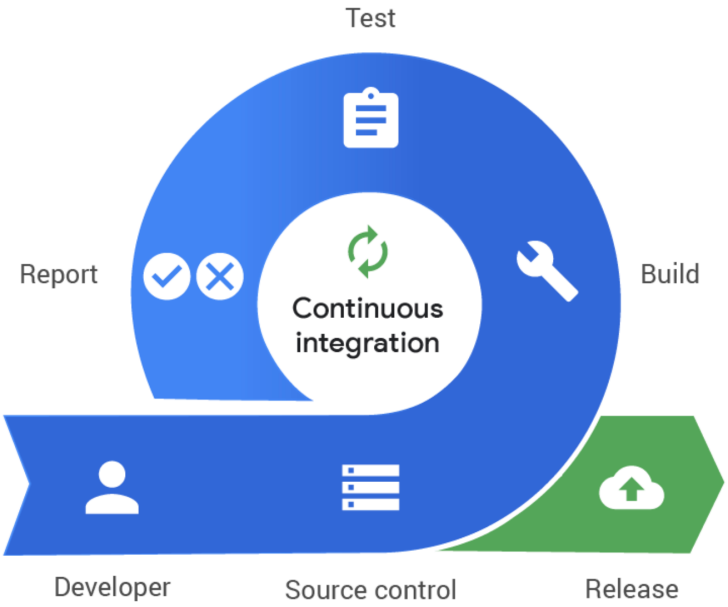

Figura 2.1: Proceso de integración continua[4].

cambios en el código y lo integra en el repositorio. Una vez se realizan cambios, estos deben pasar una serie de pruebas antes de poder ser incorporados de forma definitiva en el código fuente de la aplicación (Figura 2.1).

Desde hace años se utilizan sistemas de control de versiones para gestionar el código de cualquier proyecto. Este tipo de herramientas permite a un equipo controlar el estado del código en cada momento, siendo capaces de conocer el historial de los cambios realizados, saber quién ha hecho cada cambio y tener la capacidad de revertir alguna modificación en el caso de ser necesario. La herramienta de control de versiones más utilizada hoy en día, y la que se utiliza en este proyecto, es Git[12].

La integración de código en un repositorio no se trata simplemente de modificar una porción de un archivo y subirlo. El código debe ser probado antes de integrarlo completamente en el núcleo de la aplicación. Durante el proceso de integración continua, cada vez que se modifica algo de código, se debe:

- Construir la aplicación.
- Pasar pruebas de funcionalidad.
- Pasar el *linting* del propio código, es decir, verificar que el código cumpla ciertos estándares de estilo.
- Reportar cualquier error en el caso de que exista.

Todo lo anterior se debe realizar de manera automatizada, con el fin de integrar el código modificado en el repositorio lo más rápido posible, evitando en la medida de lo posible la intervención humana.

Figura 2.2: Proceso de despliegue continuo[5].

Continuous Delivery

Tras haber construido la aplicación durante el proceso de integración continua, toca desplegar la aplicación. El despliegue automático de nuevas versiones de una aplicación que han pasado el ciclo de CI se conoce como "despliegue continuo" (Figura 2.2).

Esto tiene como requisito que la aplicación que se está construyendo tenga el despliegue como uno de los pasos en su ciclo de vida, lo cual no tiene por qué ser así. En este Trabajo de Fin de Grado sí que ocurre, ya que la aplicación de prueba que se construye es una página web, junto con una API y una base de datos.

Es necesario que exista relación entre los desarrolladores y los encargados de desplegar la aplicación. Sin embargo, hoy en día encontramos en muchas empresas un conjunto de metodologías conocidas como DevOps[10], lo cual implica que ciertos integrantes de un equipo deben tener conocimiento tanto del desarrollo de la aplicación como del despliegue de la misma.

Esta transición a la cultura DevOps permite a los equipos desplegar sus aplicaciones más fácilmente. Además, incluye la necesidad de que el despliegue sea una parte muy importante en el proceso de desarrollo.

Al igual que en la integración continua, en este ciclo también es necesario automatizar el proceso despliegue de una aplicación. Esto permitirá disminuir la posibilidad de error humano.

Con el despliegue continuo podemos tener *feedback* más rápido por parte del usuario, lo que permitirá mejorar y corregir errores más rápidamente. Además, se despliegan con más frecuencia cambios realizados en la aplicación, por lo que los errores en producción son menos probables y, en el caso de que los haya, más fáciles de corregir. Esto también es gracias a llevar un historial de los cambios mediante una herramienta de control de versiones como Git.

GitOps & ArgoCD

Basándose en la filosofía DevOps, mencionada antes, que abarca tanto el ciclo de CI como el de CD, existen un conjunto de prácticas en las que se utiliza Git como fuente de verdad para la gestión de la infraestructura y las aplicaciones. Esto es conocido como GitOps[22]. Utilizar Git como la única fuente de información permite gestionar de manera consistente la infraestructura de las aplicaciones. Realizar cambios en un repositorio produce la ejecución de *workflows* o

secuencias de acciones de CI/CD, los cuales reflejan en el entorno de despliegue correspondiente los cambios que se han llevado a cabo.

La herramienta que realiza estas prácticas de GitOps, y que se utiliza en este Trabajo de Fin de Grado, es ArgoCD[23]. ArgoCD es una herramienta que, como su nombre indica, facilita el despliegue continuo. Supervisa repositorios de Git para aplicar los cambios que se realizan en ellos automáticamente en el *cluster* (ver Sección A.1).

2.2. Ecosistema de herramientas

Un *pipeline* moderno se compone de diferentes tipos de herramientas, cada una con sus características y finalidades. Se pueden agrupar en los ciclos que se han indicado anteriormente, CI y CD. El grupo de herramientas de CI facilitan la construcción y empaquetado de la aplicación que se va a construir, mientras que las de CD permiten desplegar la aplicación empaquetada previamente.

Herramientas de construcción y empaquetado

Como cabe esperar, los pasos mencionados en la Sección 2.1, que forman parte de la integración continua, van a depender del tipo de aplicación que se esté construyendo, y de las tecnologías que se estén utilizando. Además, esta secuencia de acciones pueden incluir unos pocos comandos en trabajos o proyectos sencillos, o necesitar varios *scripts* complejos en el caso de aplicaciones más avanzadas. Por lo tanto, es necesario tener una herramienta de construcción que permita realizar los pasos mencionados anteriormente, sin la necesidad de memorizar cada uno de los comandos o *scripts* que hay que ejecutar.

Make

Para ello existe `make`[13], una aplicación de línea de comandos que permite definir bloques de comandos o reglas, aportando a cada bloque un nombre u objetivo que se pretende obtener ejecutando dicha regla. Se suele crear un archivo llamado `Makefile` para definir todas las reglas que se precisen.

Un ejemplo muy típico de compilación de un programa escrito en `C` sería el que se puede observar en el Listing 1.

Just

En este Trabajo de Fin de Grado se utiliza una herramienta de construcción más moderna y polivalente llamada `just`[14]. Este software tiene la misma finalidad que `make`, ejecutar comandos específicos de un proyecto. Pero este incluye muchas más funcionalidades (ver Sección A.1).

```
# Compiler
CC = gcc

# Compiler options
CFLAGS = -Wall -g

# Final executable name
TARGET = my_program

# The object files (.o) needed by the program
# Make infers automatically that .o depends on the corresponding
↪  .c
OBJS = main.o hello.o

# --- Rules ---

# The first rule is the one executed by default with *make*
# It declares that to create the TARGET, it needs the OBJS
$(TARGET): $(OBJS)
  $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# *.PHONY* tells Make that *clean* is not a file
.PHONY: clean
clean:
  rm -f $(TARGET) $(OBJS)
```

Listing 1: Makefile para compilación de un programa en C.

En la Listing 2 se puede ver que se hace uso de comandos y sintaxis propios de sistemas UNIX (por ejemplo, Bash), por lo que se trata de un requisito de software contar con un entorno UNIX o compatible (Linux, macOS, WSL, etc.) para su correcta ejecución.

Docker

Docker es una herramienta que permite empaquetar aplicaciones, creando imágenes con las dependencias necesarias para que la aplicación se lance sin problemas. Las imágenes generadas se pueden ejecutar, creando contenedores, que son entornos completamente aislados del contexto de la máquina en la que han levantado. Estos contenedores son muy ligeros en cuanto a espacio y uso de recursos, ya que almacenan únicamente lo necesario para correr el *software* que queremos desplegar (ver Sección A.1).

```
# --- ALIASES ---
# Defines shortcuts (aliases) for longer commands.
alias dv := down_vol

# --- DEFAULT RECIPE ---
# This is the recipe that runs if you just type 'just' in the
# terminal.
# By default, it invokes the 'just -l' recipe, which lists all
# available recipes.
# The '_' prefix indicates that it is a helper recipe, not
# intended to be called directly by the user.
_default:
  just -l

# --- INTERNAL (PRIVATE) RECIPES ---
_build_zoo_base:
  #!/usr/bin/env bash
  if [[ "$(docker images -f reference=zoo-base | wc -l | xargs)"
  ↪  != "2" ]]
  then
    docker build --target base -t zoo-base .
  fi

# Accepts two parameters: 'entrypoint' and 'command'.
_run entrypoint command:
  # '@' at the beginning of a command line prevents 'just' from
  # printing the command before executing it.
  @just _build_zoo_base
  docker run --rm -w /app -v $PWD:/app --env-file .env
  ↪  --entrypoint={{entrypoint}} zoo-base {{command}}

# --- PUBLIC RECIPES ---
init:
  @just _run "yarn" "install"

down_vol:
  docker compose down -v
```

Listing 2: Extracto de justfile utilizado en el proyecto. También referenciado en la Sección A.1.

La mayor ventaja que proporciona el empaquetado de aplicaciones con Docker es la portabilidad. Aunque hay que tener en cuenta la arquitectura de la máquina, las imágenes se pueden lanzar en cualquier entorno con Docker instalado, lo cual evita el conocido problema de: "En mi máquina funciona".

Plataformas de despliegue

Docker Compose

Con Docker se es capaz de gestionar varios servicios desplegados en distintos contenedores. Pero existe una herramienta que apareció poco después y que facilita esta tarea, llamada "Docker Compose"[17]. Esta permite desplegar entornos con múltiples contenedores para desarrollar localmente (ver Sección A.1).

Docker Compose no es una plataforma de producción, se utiliza principalmente con el fin de desarrollar de manera local, y es muy útil en el caso de querer hacer pruebas rápidas de una aplicación sencilla. Se puede tomar como un precursor conceptual a la orquestación más compleja que realiza Kubernetes.

Kubernetes, Helm & KinD

El proyecto Kubernetes nació un año después que Docker. Es una herramienta de *software* que permite orquestar contenedores. Gestiona el ciclo de vida de las aplicaciones en contenedores que viven en un *cluster*. Entre sus características principales destacan:

- Escalado automático.

  Aumenta o disminuye automáticamente el número de contenedores en ejecución. Esto va a depender de la cantidad de réplicas de una misma aplicación que se hayan indicado en su configuración. Kubernetes siempre va a intentar mantener el estado del *cluster* cumpliendo los parámetros que se indicaron en las plantillas de configuración de cada uno de los servicios.

- Reparación.

  Si un contenedor falla, se reinicia o se reemplaza por otra instancia del mismo servicio, garantizando la continuidad de este.

- Descubrimiento de servicios y balanceo de carga.

  Se exponen los contenedores entre ellos y/o a Internet. Además, permite distribuir el tráfico de red, evitando así sobrecargas.

Más información en la Sección A.1.

Complementando a Kubernetes tenemos Helm (Figura 2.3), que se trata de un gestor de paquetes para Kubernetes. Su propósito es ayudar a instalar recursos y administrar el ciclo de vida de las aplicaciones de Kubernetes. Además, las Charts de Helm son formatos de archivos YAML que permiten definir los objetos

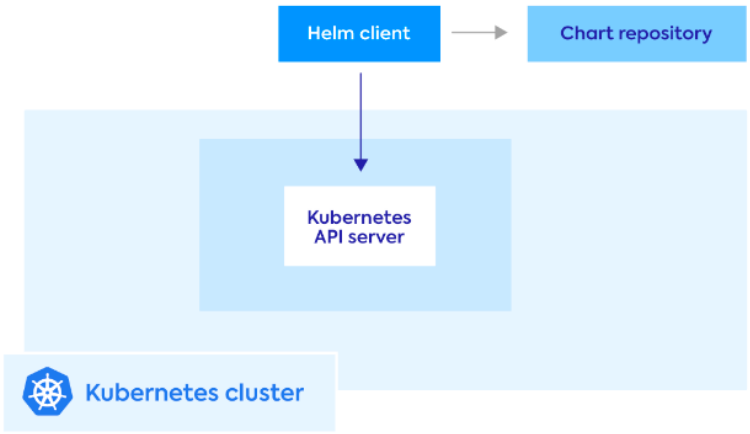

Figura 2.3: Arquitectura de Helm[24].

de Kubernetes de una manera dinámica. Esto facilita la definición de aplicaciones mucho más complejas .

Para desplegar una aplicación de Kubernetes es necesario tener un *cluster* preparado para soportar este tipo de recursos. Un *cluster* de Kubernetes es un conjunto de máquinas o nodos que trabajan juntos con el fin de ejecutar y gestionar aplicaciones que corren dentro de contenedores. Para crear los *clusters* en los que se despliega la aplicación se utiliza KinD[20].

KinD (*Kubernetes in Docker*) permite crear *clusters* de Kubernetes de manera local. Se utiliza esta herramienta debido a su sencillo uso. Simplemente utilizando el comando que se muestra en el Listing 3 se puede construir un *cluster* en el que desplegar cualquier aplicación de Kubernetes.

```
1 kind create cluster
```

Listing 3: Comando para crear un *cluster* con KinD.

Para gestionar los propios recursos de Kubernetes que existen dentro del *cluster* se utiliza la herramienta `kubectl`[21]. Se trata de un CLI oficial de Kubernetes que permite comunicarse con el controlador principal del *cluster*, utilizando la API de Kubernetes. Un ejemplo de uso se puede observar en el Listing 4.

```
1 kubectl apply -f "argo/argo_dev.yaml" --context kind-dev
```

Listing 4: Aplicar la configuración de ArgoCD con `kubectl`.

El uso de estas dos herramientas se ve más en detalle en la Sección 3.2.

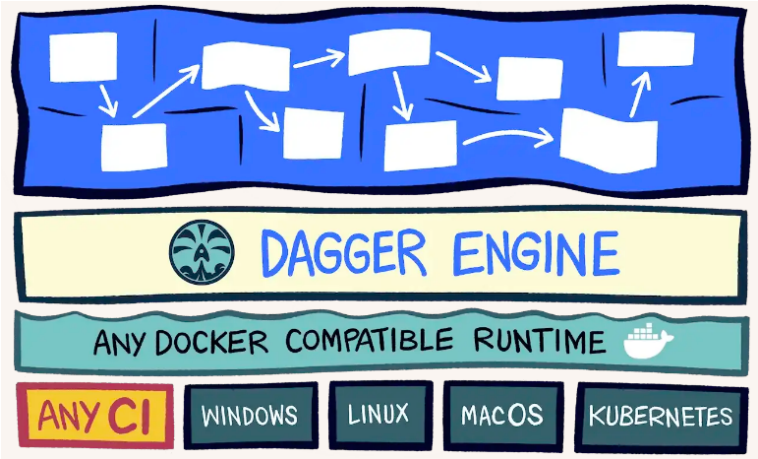

Figura 2.4: *Pipelines* con Dagger sobre un *runtime* compatible con Docker[25].

2.3. Dagger

Dagger es el pilar fundamental de este Trabajo de Fin de Grado. Se trata de un kit de desarrollo de *software* que permite a los desarrolladores crear *pipelines* CI/CD y ejecutarlos en cualquier sitio.

La principal idea de los creadores de Dagger es poder crear *pipelines* portables, que no sea necesario implementarlos de nuevo cada cierto tiempo debido a cambios en el entorno de desarrollo o de pruebas. Esta portabilidad se consigue permitiendo a los desarrolladores utilizar cualquier *runtime* de OCI para ejecutar las funciones que forman parte del SDK[28] de Dagger, así como las que definen los propios desarrolladores. Además, desde el principio se ha evitado el uso de archivos YAML, que está siendo el lenguaje de configuración más utilizado por parte de la mayoría de aplicaciones.

CUE

Dagger comenzó utilizando un lenguaje de configuración muy potente llamado CUE[27]. Este se podría ver como una extensión de JSON, pero con más funcionalidades. Todo lo escrito en JSON se puede traducir a CUE, pero no al revés. En el Listing 5 se puede ver un ejemplo de código de Dagger en el que se utiliza CUE para lanzar un "plan". En él se descarga la imagen de Docker de Alpine y se almacena en un registro levantado en la máquina local.

Esta es la primera vez que se pueden ejecutar *pipelines* definidas de manera programática, ejecutadas sobre un *runtime* de OCI, y de manera relativamente funcional. Además, se puede ejecutar toda la secuencia de acciones de manera local, permitiendo realizar pruebas y buscar errores sin necesidad de hacer uso de otras herramientas de CI como GitHub Actions[29].

Además, Dagger hace un uso exhaustivo de la caché. Todas las acciones son

```
package main

import (
    "dagger.io/dagger"
    "universe.dagger.io/docker"
)

dagger.#Plan & {
    actions: {
        pull: docker.#Pull & {
            source: "alpine"
        }
        push: docker.#Push & {
            image: pull.output
            dest:  "localhost:5042/alpine"
        }
    }
}
```

Listing 5: Código de Dagger con CUE.

cacheadas automáticamente. Esto es una funcionalidad muy útil, ya que va a hacer que un *pipeline* se ejecute en algunos casos un 90 % más rápido, como ha ocurrido en pruebas realizadas en este Trabajo de Fin de Grado, que se comentan más adelante. Si se pone como ejemplo la ejecución de tests sobre una aplicación, la primera vez que se lanza el *pipeline*, este tiene que ejecutarse completamente:

1. Instalar las dependencias.
2. Compilar la aplicación.
3. Correr los tests.

Dependiendo del tipo de aplicación y de la conexión a Internet, esto puede tardar desde unos segundos hasta varios minutos. Pero con Dagger solo ocurre una vez. La primera vez. Gracias a la caché, todas las acciones repetitivas, como la instalación de dependencias o la compilación de la aplicación, se almacenan en la caché, ahorrando así mucho tiempo a la hora de realizar pruebas de cualquier tipo. Mientras tanto, con otras herramientas se tendría que esperar siempre la misma cantidad de tiempo para cada una de las veces que se quiere lanzar el *pipeline*. Además, Dagger puede ejecutar estas acciones de manera local, mientras que utilizando métodos convencionales habría que depender de aplicaciones que pueden fallar o no estar disponibles en cierto momento.

CI/CD como código

Dagger ha querido ir más allá pensando que los desarrolladores deberían ser capaces de crear sus *pipelines* de la misma manera que crean sus aplicaciones: escribiendo código. Es así como nace el SDK de Go[30] para Dagger, el cual se utiliza en este Trabajo de Fin de Grado. Go es un lenguaje de programación con muchos casos de uso. Desde la creación de servicios de red y en la nube, hasta aplicaciones CLI y desarrollo web. Es conocido también por su simplicidad en cuanto a sintaxis y por tener una librería estándar muy completa, aportando muchas de las herramientas necesarias para realizar proyectos comunes. Además, facilita la implementación de programación concurrente, gracias a las *goroutines*, similares a los hilos de ejecución de un sistema operativo, pero mucho más ligeros. En el Listing 6 se puede observar un ejemplo de código utilizando el SDK de Go.

```
1  package main
2
3  import (
4          "context"
5          "dagger/dagger/internal/dagger"
6          "fmt"
7          "strconv"
8  )
9
10 type Backend struct {
11         Name string
12         Base *dagger.Container
13         Secrets SecMap
14 }
15
16 func (m *Backend) Build(ctx context.Context) *dagger.Container {
17         build := m.Base.
18                 WithWorkdir("/app").
19                 WithExec([]string{"lerna", "run", "--scope",
                    ↪ "@vieites-tfg/zoo-backend", "build"}).
20                 WithExec([]string{"ncc", "build",
                    ↪ "./packages/backend/dist/index.js", "-o",
                    ↪ "./dist"})
21
22         return build
23 }
24
25 func (m *Backend) Ctr(ctx context.Context) *dagger.Container {
26         build := m.Build(ctx)
```

```
27
28          compiled := build.File("/app/dist/index.js")
29          pkgJson :=
            ↪  build.File("/app/packages/backend/package.json")
30
31          dag := dagger.Connect()
32          back := dag.
33                  Container().From("node:20-alpine").
34                  WithExposedPort(3000).
35                  WithEnvVariable("NODE_ENV", "production").
36                  WithEnvVariable("YARN_CACHE_FOLDER",
                    ↪  "/.yarn/cache").
37                  WithMountedCache("/.yarn/cache",
                    ↪  dag.CacheVolume("yarn-cache")).
38                  WithWorkdir("/app").
39                  WithFile("/app/package.json", pkgJson).
40                  WithFile("/app/index.js", compiled).
41                  WithExec([]string{"yarn", "install",
                    ↪  "--production"}).
42                  WithEntrypoint([]string{"node", "index.js"})
43
44          return back
45  }
```

Listing 6: Ejemplo del SDK de Go de Dagger.

Todo lo anterior hace de Go una elección excelente para empezar la lista de lenguajes de programación sobre los que el equipo de Dagger implementaría su propio SDK, que hasta el día de hoy incluyen: Typescript, Java, PHP, Node y Python.

GraphQL

El equipo de Dagger es capaz de desarrollar SDKs específicos para cada lenguaje tan rápidamente gracias al uso de GraphQL[31], un lenguaje para manipulación y consulta de datos.

Se puede ver su funcionamiento en la Figura 2.5. El SDK de cada uno de los lenguajes funcionan como traductores del código que escribes en dicho lenguaje a sentencias que entiende el motor de Dagger. Esto es a través de la API de GraphQL de Dagger. El motor de Dagger es el que se encarga de ejecutar las instrucciones en un entorno controlado. De esta manera, no son los propios SDKs los que corren los programas dependiendo del lenguaje, sino que funcionan como clientes de la API para traducir la secuencia de acciones y ejecutarse en un mismo entorno.

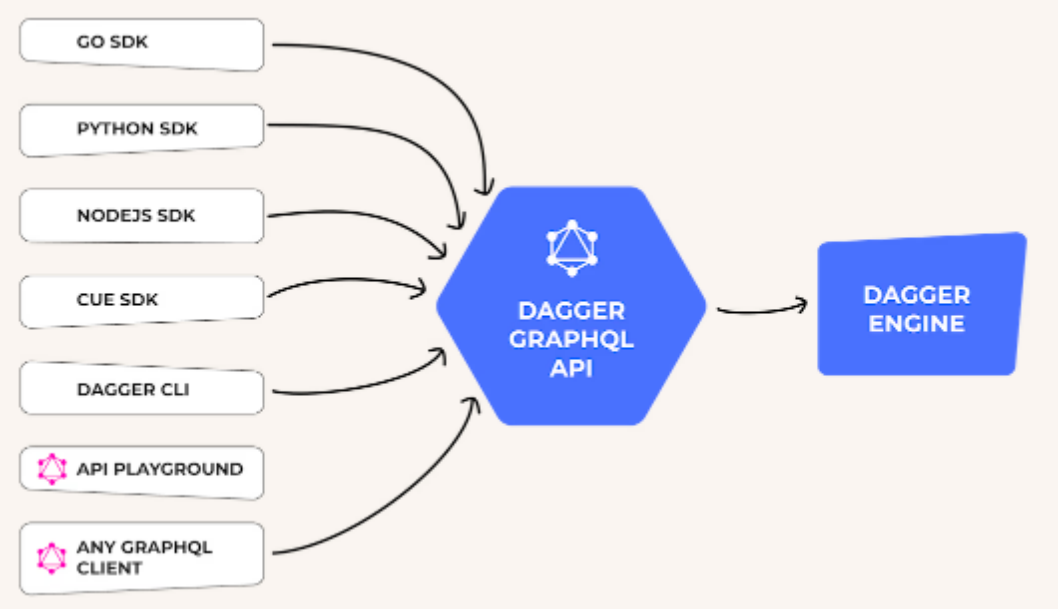

Figura 2.5: Uso de la API de GraphQL para Dagger[32].

Dagger *functions*

Tras varias mejoras y nuevas versiones, el equipo de Dagger implementa las "funciones de Dagger". Estas funciones son el componente principal de Dagger hoy en día. Cada una de las operaciones principales de Dagger se pueden llamar a través de una función, utilizando una API. Además, estas se pueden encadenar, generando *pipelines* dinámicas en una sola llamada. De esta manera, se puede decir que gracias a Dagger se pueden programar ciclos CI/CD en uno de los lenguajes disponibles, bien conocidos en el sector.

Daggerverse

La existencia de las funciones de Dagger permite la creación de módulos, un conjunto de funciones que toman una entrada y producen una salida en concreto. Los módulos creados por la comunidad se pueden encontrar en el Daggerverse[33]. Este es el lugar en el que se comparten módulos de Dagger, los cuales se pueden reutilizar para añadir funcionalidades a otro módulo, sin necesidad de volver a crear una lógica de programación ya implementada por otra persona o equipo de desarrolladores.

CLI

Otra funcionalidad que tiene Dagger es su CLI[34] (*Command-Line Interface*). A través de ella puedes llamar a funciones de módulos de Dagger, tanto de de tu

sistema de archivos local como directamente de un repositorio de Git en el que se encuentre el código fuente de un módulo.

En el Listing 7 se muestra un ejemplo de ejecución de una de las funciones definidas en un módulo correspondiente a este Trabajo de Fin de Grado. Toda la documentación al respecto se puede encontrar en el Apéndice B.

```
dagger call --sec-env=file//../../.env backend service up --ports
↪  3000:3000
```

Listing 7: Comando para lanzar el *backend* del proyecto.

Resumen

Dagger es una herramienta revolucionaria que redefine la manera de crear *pipelines* de CI/CD, proporcionando la capacidad de implementar estos *pipelines* de manera programática y evitar que los desarrolladores tengan que lidiar con archivos de configuración estáticos como YAML, o con la creación de diferentes *scripts*. Ofrece SDKs en una variedad de lenguajes de programación, tales como Go, Python o Node, que actúan como clientes del motor central de Dagger.

Una de sus ventajas más significativas es su portabilidad, la cual se consigue al ejecutar todas las operaciones sobre un *runtime* de OCI, como Docker. De esta manera se garantiza que cualquier *pipeline* definido con Dagger funcione de igual manera sin importar la máquina en la que se ejecuta.

Otro de sus pilares es la gestión que hace de la caché. Dagger cachea cada una de las acciones ejecutadas. Gracias a esto, tras la primera ejecución de un *pipeline*, las siguientes ejecuciones son significativamente más rápidas. Así se reducen los tiempos de espera, permitiendo a los desarrolladores trabajar más rápido.

Finalmente, la evolución hacia las "Dagger *functions*" y la creación de módulos permite un gran nivel de reutilización. Estos módulos, que pueden ser compartidos a través del Daggerverse, junto con su CLI para invocarlos, proponen una manera muy poderosa de crear *pipelines* para construir, probar y desplegar aplicaciones.

Capítulo 3

Diseño y arquitectura del sistema

3.1. Estructura general

El código del trabajo y todo lo que abarca se encuentra almacenando en GitHub. Se han creado los repositorios necesarios en una misma organización de GitHub.

Entre los repositorios creados se pueden encontrar:

- `zoo`.

  Este es el repositorio principal. Se trata de un *monorepo*[7], en el que se encuentra implementado todo el código necesario dentro del presente trabajo de investigación. En el repositorio se encuentran:

  - La aplicación de prueba sobre la que se apoya el proyecto, y que da nombre al repositorio, debido a que se trata de una aplicación de gestión de un zoo.
  - Los módulos de Dagger para realizar los ciclos de CI y CD.
  - Otros archivos, como *scripts* y archivos de configuración.

- `helm-repository`.

  Este repositorio alberga las Charts de Helm que definen la estructura necesaria para desplegar la aplicación de prueba.

- `state`.

  Se trata del repositorio que, siguiento el enfoque GitOps, almacena los valores que poblarán los recursos de Kubernetes según el entorno en el que se despliegue la aplicación. Además, en este repositorio también existe una rama de despliegue, de la cual ArgoCD lee los manifiestos de los recursos que debe desplegar para cada uno de los entornos.

En la Figura 3.1 se muestra un diagrama de la disposición de los repositorios y la relación entre ellos.

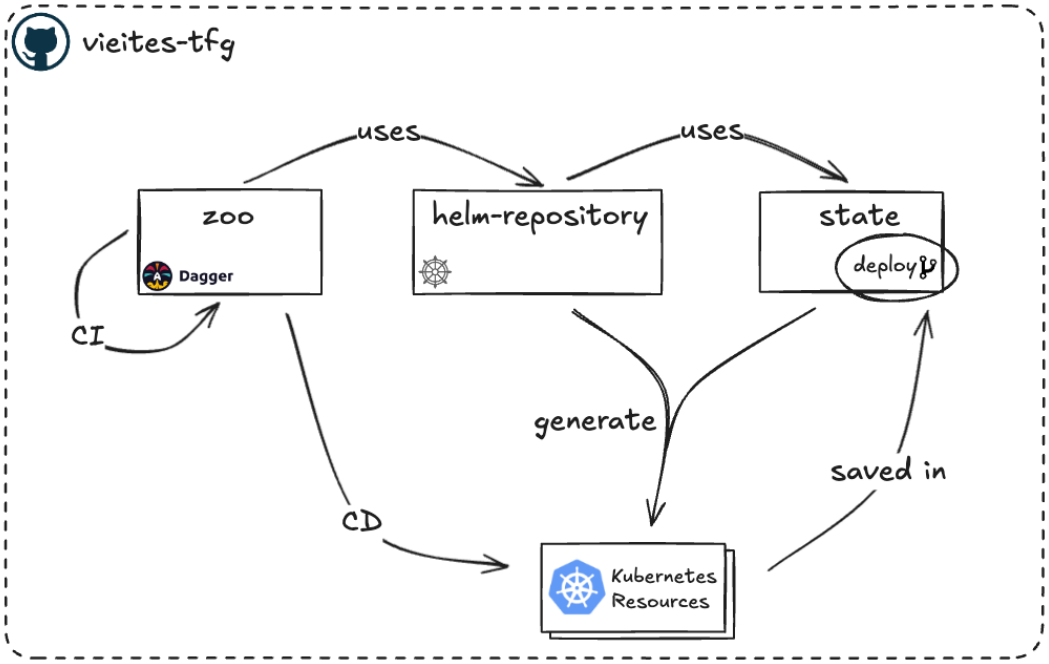

Figura 3.1: Diagrama de la organización de GitHub. Imagen creada con excalidraw.com.

3.2. zoo

Como se ha comentado anteriormente, el repositorio `zoo` está estructurado como un *monorepo*. Un *monorepo* es un repositorio con diferentes proyectos, los cuales se encuentran interrelacionados de una manera bien definida. A lo largo de esta sección se justifica la elección de este tipo de estructura, a la vez que se explica cómo están implementados las diferentes piezas de software.

Aplicación de prueba

La primera razón para escoger un *monorepo* como estructura del repositorio es el hecho de querer crear una aplicación relativamente pequeña, una página web que consta de un *frontend* y un *backend* (se hará referencia a estos como "paquetes" a partir de ahora). Por lo tanto, se hace más sencillo gestionar estos dos paquetes si se ubican en un único repositorio.

Otra ventaja de utilizar un *monorepo* tiene que ver con el *software* utilizado para crear los paquetes de la aplicación de prueba. Ambos se implementan utilizando Node.js, en lenguaje Typescript[35]. Los paquetes tienen dependencias propias, y se puede dar el caso de que ambos utilicen una o varias dependencias iguales. Usar un *monorepo* permite tener esas dependencias en un mismo lugar, evitando su duplicado. Con esto se consigue reducir el tiempo de construcción de

los paquetes.

Sin embargo, es necesaria una herramienta que permita manejar los paquetes de manera independiente. Alguno de los motivos para tener esta preferencia pueden ser: que haya dos equipos de desarrolladores, uno para cada paquete; o que se quiera publicar versiones, hacer tests, u otro tipo de tarea sobre cada paquete por separado. La herramienta que se utiliza en este Trabajo de Fin de Grado se llama Lerna[36]. Este *software* está específicamente diseñado para gestionar *monorepos* de proyectos de Node.js. Entre las ventajas que proporciona se encuentran:

- Gestión de tareas locales.
- Cacheo local de salidas de comandos, con posibilidad de que dicha caché sea compartida entre entornos, por ejemplo, con agentes de CI.
- Detección de paquetes afectados por cambios en el código.
- Análisis de la estructura del proyecto.

Por los beneficios anteriormente comentados, y más, es por lo que se ha elegido esta herramienta para gestionar el *monorepo*.

En cuanto a las tecnologías que se utilizan en la aplicación, ya se ha mencionado Typescript como lenguaje principal. Este lenguaje permite tener un sistema tipado, lo cual puede ser útil para detectar muchos errores comunes mediante el análisis estático en tiempo de construcción. Esto reduce las posibilidades de errores en tiempo de ejecución.

El *backend* está completamente desarrollado utilizando dicho lenguaje. Su funcionalidad es proporcionar una API REST que el *frontend* pueda utilizar para realizar cambios en la base de datos. Se usa MongoDB[37] como base de datos debido a que es fácil de gestionar y porque solo se almacena información sobre animales, sin ningún tipo de relación entre ellos, en una única tabla o documento de la base de datos.

El *frontend* se ha implementado utilizando Vue.js[38], un *framework* que permite construir interfaces web mediante componentes reactivos. Se ha escogido este frente a otras opciones debido pequeña curva de aprendizaje inicial gracias a su API intuitiva. Además, el propio *framework* está construido utilizando Typescript, por lo que tiene compatibilidad de primera clase con este lenguaje.

En la Figura 3.2 se puede ver un diagrama que muestra cómo es la comunicación entre los paquetes de la aplicación, y con la base de datos, junto con las tecnologías que se utiliza en cada uno de ellos.

Módulos de Dagger

Se integran también en el repositorio los módulos de Dagger de CI y de CD. Estos módulos se incluyen en el *monorepo* para facilitar la referencia a los paquetes que constituyen la aplicación de prueba. Además, resulta lógico que se ubiquen en el mismo lugar una aplicación y las herramientas que permiten su

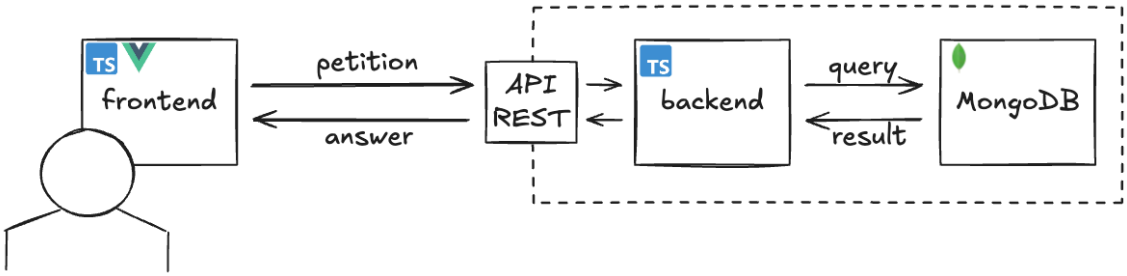

Figura 3.2: Comunicación entre paquetes de la aplicación. Imagen creada con excalidraw.com.

evolución, como son cualquier tipo de *software* que realice las funciones de CI y de CD.

Ambos módulos se realizan utilizando el SDK del lenguaje Go que proporciona Dagger. Se ha escogido este lenguaje debido al conocimiento previo que ya se tenía de este. Además, es el lenguaje en el que está implementado el propio Dagger.

Uno de los módulos se encarga del ciclo de CI, es decir, de realizar los tests de la aplicación, del *linting* o análisis del código en sí, y de la publicación de imágenes de Docker y paquetes NPM. Está organizado de manera que se pueden gestionar cada uno de los paquetes de la aplicación de manera independiente. Esto también es posible gracias al uso de Lerna, que ya se ha comentado anteriormente.

El segundo de los módulos, el de CD, realiza la tarea de publicación de los recursos de Kubernetes, los cuales son posteriormente obtenidos por ArgoCD para su despliegue completo. Esto se consigue haciendo uso de los repositorios `helm-repository` y `state`, en los cuales se almacenan las Charts de Helm y los valores que pueblan dichas Charts, respectivamente.

Se detalla más profundamente la implementación de los módulos de Dagger en el Capítulo 4.

Creación y configuración de los *clusters*

La fase final del ciclo de una aplicación es el despliegue. En este Trabajo de Fin de Grado se levantan tres *clusters* de KinD de manera local. Estos son los lugares en los que se despliega la aplicación. Generalmente se tienen diferentes *clusters* con el fin de probar la aplicación en entornos distintos antes de desplegarla en el principal, que sería el de producción. El hecho de crearlos todos localmente hace que sean más sencillas las pruebas relacionadas con el despliegue. En equipos de desarrollo reales, los entornos de producción se encuentran en la nube. Sin embargo, sí que se pueden llegar a tener entornos locales para realizar pruebas de la aplicación.

Los *clusters* se crean con el *script* que se muestra en el Listing 8 (se han puesto comentarios en vez de código en algunas partes para reducir su tamaño,

a modo de pseudocódigo). Este *script* está escrito para funcionar en sistemas UNIX, en Bash, por lo que es un requisito utilizar un sistema operativo como MacOS o una distribución de Linux para probar el *script*. No funcionará en un sistema operativo Windows. A modo de resumen del *script*:

```
# Variables globales
# ---

for ENV in "${ENVS[@]}"; do
  case "${ENV}" in
    dev)
      BANNER_TEXT="We are in DEV";;
    # ... otros entornos
  esac

  CONTEXT="kind-${ENV}"

  kind create cluster --config "${CLUSTER_DIR}/kind_${ENV}.yaml"

  kubectl apply -f "${INGRESS_MANIFEST}" --context "${CONTEXT}"
  # Se espera a que se construya

  kubectl create namespace argocd || true

  cat "${SOPS_DIR}/age.agekey" |
    kubectl create secret generic sops-age -n argocd \
    --context ${CONTEXT} --from-file=keys.txt=/dev/stdin

  # Se descarga el repositorio de la Chart
  helm install argocd argo/argo-cd -n argocd \
    -f argo/values.yaml \
    # ... otros *flags*

  # Se espera a que se instale la Chart de Argo

  kubectl apply -f "${ARGO_DIR}/argo_${ENV}.yaml" \
    --context "${CONTEXT}"
  # Se espera a que se apliquen los cambios

  # Se aplica el banner a la interfaz de Argo

  # Se obtiene la clave inicial del usuario "admin" y se muestra
  # por pantalla
```

```
  PASSWORDS+="${current_pass}"
done

printf "${PASSWORDS}"
```

Listing 8: *Script* de creación de los *clusters*. También referenciado en la Sección A.2.

- Se crean tres *clusters* con su configuración específica. Se puede ver el archivo de configuración del *cluster* de `dev` en el Listing 9.
- Se instala en cada *cluster* un controlador de Ingress, lo cual permitirá acceder a la aplicación a través de una URL customizada desde el exterior.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: dev
networking:
  apiServerPort: 6443
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready="
  extraPortMappings:
  - containerPort: 80
    hostPort: 8080
    protocol: TCP
  - containerPort: 443
    hostPort: 8443
    protocol: TCP
```

Listing 9: Configuración del *cluster* de `dev`.

- Se incluye como Secret de Kubernetes el valor de la clave de cifrado que permite desencriptar los secretos. El proceso de creación de cifrado y descifrado se explica en la Sección 3.2.
- Se instala la Chart de Helm de ArgoCD. A esta Chart se le pasan una serie de valores, de los cuales se habla en la Sección 3.2. Más información sobre cómo acceder a la interfaz de ArgoCD en el Apendice B.

- Se aplica la configuración específica de ArgoCD para el entorno que se está creando. Se puede ver la configuración para ArgoCD en el entorno de `dev` en el Listing 10. En dicha configuración se indica el repositorio, la ruta desde la raíz donde se encuentran los recursos y la rama de donde ArgoCD debe obtener los archivos que definen los recursos que se van a desplegar.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: app-dev
  namespace: argocd
spec:
  project: default
  source:
    repoURL: 'https://github.com/vieites-tfg/state.git'
    path: dev
    targetRevision: deploy
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: dev
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

Listing 10: Configuración de ArgoCD en `dev`.

- Se añade un banner en la parte superior de la interfaz para distinguir cada uno de los *clusters*.
- Se muestra la contraseña del usuario `admin`, para poder hacer *log in* a través de la interfaz de ArgoCD.

En la Figura 3.3 se puede observar cómo se sincroniza ArgoCD, en cada uno de los *clusters*, con el repositorio de estado. ArgoCD reacciona cada vez que se realizan cambios en el directorio que le corresponde, dentro de la rama de despliegue del repositorio `state`. En ese momento, obtiene de nuevo todos los archivos de definición de los recursos y se sincroniza con el estado deseado.

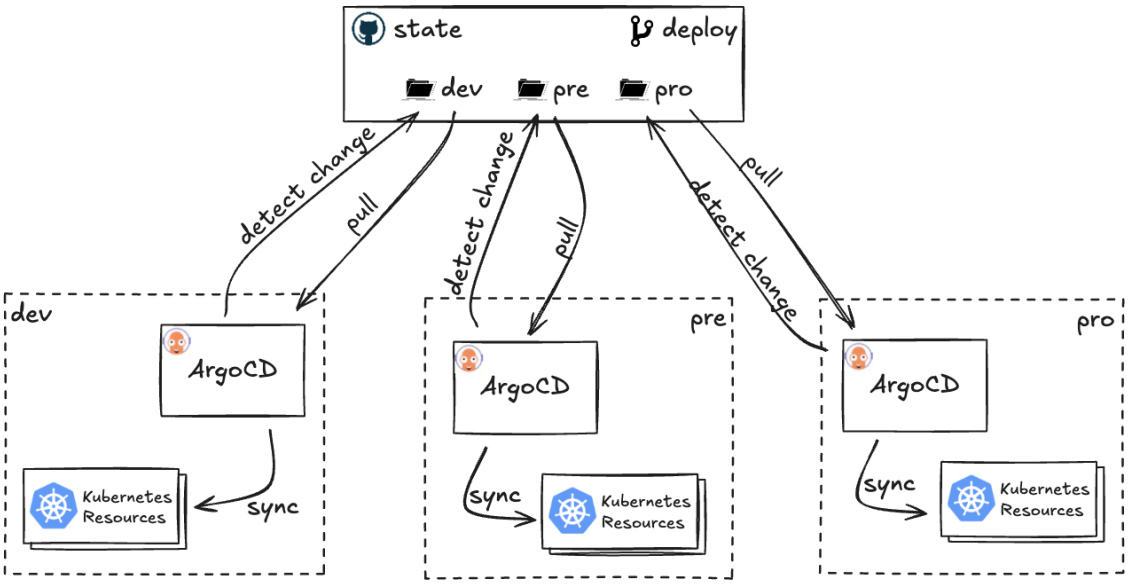

Figura 3.3: Clusters y comunicación con el repositorio de estado. Imagen creada con excalidraw.com.

Gestión de secretos

La aplicación de prueba consta de una base de datos, la cual tiene usuario y contraseña. Este es un ejemplo de datos que es necesario almacenar como un secreto o Secret de Kubernetes. Debido a que se está utilizando un método *pull*[11] de despliegue de la aplicación; es decir, la herramienta que realiza el despliegue, ArgoCD, "tira" (hace *pull*) del repositorio que le indicamos como objetivo; es necesario almacenar en el repositorio los secretos encriptados previamente. Esto es una buena práctica para evitar que se filtren sin querer los datos al exterior, aunque el repositorio sea privado.

Existe un apartado completamente dedicado a como se realiza la gestión de secretos en la Sección A.2.

Función de cada *cluster* y promoción de entornos

A continuación se explica para qué se utiliza cada *cluster* y cuál es el proceso de despliegue de la aplicación en cada uno de los entornos.

- `dev`.

  Se trata del *cluster* de desarrollo. En este se despliega la aplicación en el momento en el que se añade una nueva funcionalidad, ya sea en el *frontend* o en el *backend*. Esto implica, en términos de GitHub:

  1. Crear una *Pull Request* (PR) en la que se implementa la nueva funcionalidad. Esta debe ser lo más reducida posible, cumpliendo con la

filosofía de CI. Esto implica que la intención del equipo de desarrollo debe ser desplegar nuevas funcionalidades o correcciones de errores en producción en el menor tiempo posible. Se puede conseguir esto planeando PRs cortas en cuanto a tiempo de desarrollo, evitando que el equipo tenga demasiado trabajo en progreso y asignando los recursos necesarios para que cada PR se lleve a cabo lo más rápidamente[39].

2. Implementar la funcionalidad o realizar la corrección pertinente. A medida que se implementa, se puede, y es una buena práctica, ejecutar localmente el ciclo de CI para asegurarnos de que se pasan las pruebas y el *linting* del código. Esta es una de las ventajas principales de utilizar Dagger. El desarrollador puede comprobar de manera local si el código actualizado es capaz de pasar el *pipeline* de CI, lo cual evita tener errores inesperados a la hora de integrar el código en la rama principal.
3. Revisar que la tarea que correspondía hacer en dicha PR se ha realizado correctamente.
4. Integrar la funcionalidad o corrección en la rama principal del repositorio.

Tras haber terminado todos los pasos anteriores, se ejecuta un *workflow* de GitHub que realiza todo el ciclo de CI y CD, independientemente del entorno en el que se vaya a desplegar. El *workflow* es el que se ve en el Listing 26. Para más información sobre los pasos que son necesarios para la promoción de entornos ver la Sección A.3.

Lo que se consigue con el *workflow* mencionado es ejecutar los *pipelines* de CI y de CD para la versión actual de la aplicación, lo cual permite desplegarla completamente en el entorno que le corresponde. El entorno en el que se despliega depende del evento que produce la ejecución del *workflow*, que puede ser: una integración de código en la rama principal, la creación de una *prerelease* o la creación de una *release*. Los eventos anteriores hacen que se despliegue en `dev`, `pre` y `pro`, respectivamente.

- `pre`

  Este es el entorno de pre-producción. Este tipo de entornos están diseñados para simular el entorno de producción real, y funciona como prueba final previa a la publicación de una aplicación. En el caso de este Trabajo de Fin de Grado, como ya se ha comentado, todos los entornos son idénticos, pero en equipos y entornos reales, cada uno de ellos tiene características distintas.

- `pro`

  Finalmente, el entorno de producción. Aquí es donde se despliega la aplicación de manera abierta a los usuarios. Como se lleva insistiendo a lo largo de este capítulo, lo normal es que estos entornos se encuentren en la nube.

Para mayor facilidad de pruebas y debido a que no es la finalidad de este Trabajo de Fin de Grado, se ha decidido crear todos los *clusters* de forma local.

En la Figura 3.4 se muestra cómo el *workflow* escucha los diferentes eventos que hacen que se ejecute el ciclo de CI/CD. Dependiendo del evento que ocurre, se va a utilizar una `tag` diferente y se va a desplegar en el entorno (`env`) correspondiente.

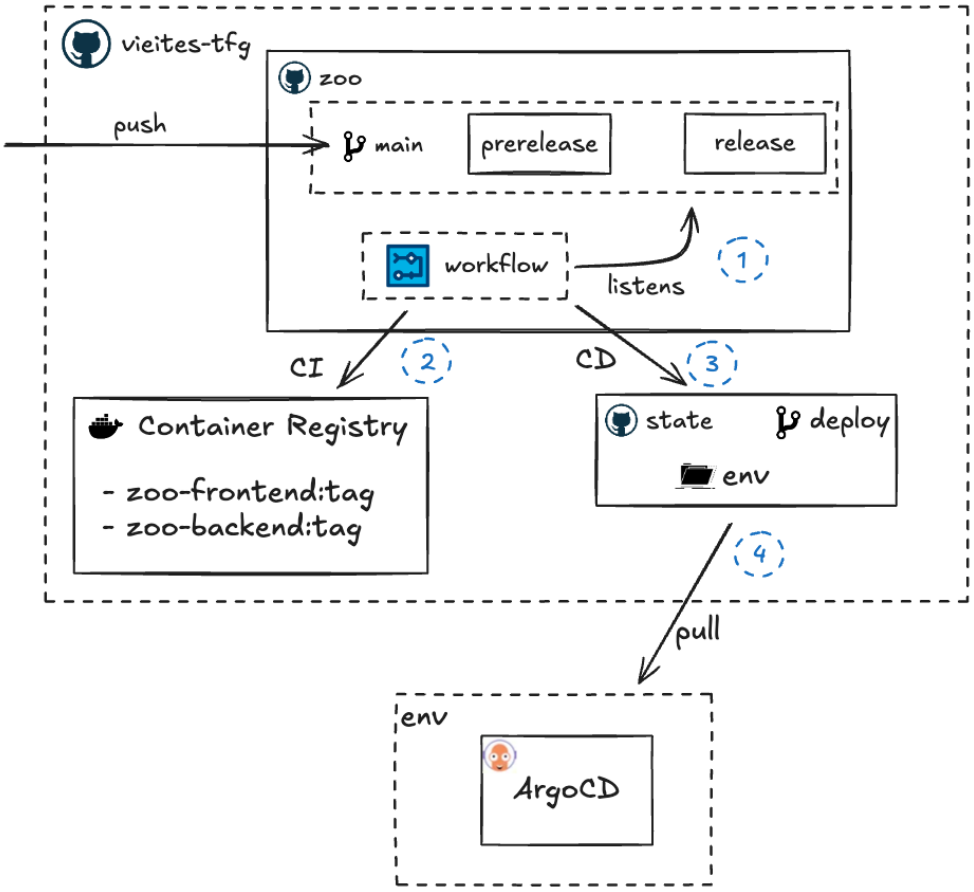

Figura 3.4: Diagrama del proceso de promoción de entornos. Imagen creada con excalidraw.com. También referenciado en la Sección A.3.

3.3. helm-repository

Es una práctica habitual tener un repositorio con todas las Charts de Helm que utiliza el equipo de desarrollo. Esto permite tener un lugar centralizado para que los desarrolladores compartan y los usuarios encuentren aplicaciones para desplegar en Kubernetes.

Este repositorio tiene como finalidad lo que se acaba de comentar, funcionar como el lugar en el que se almacena la Chart de Helm de la aplicación de prueba.

La Figura 3.5 muestra que se ha creado una Chart *umbrella* llamada `zoo`. Una Chart de este tipo funciona como una agrupación de más *subcharts* que están interrelacionadas. Normalmente se utiliza este método para gestionar despliegues de aplicaciones complejas en Kubernetes. La definición de la Chart `zoo` se puede ver en el Listing 11. En esta se indican las dependencias (*subcharts*) que agrupa la Chart. Entre estas se encuentran las Charts propias de la aplicación, `zoo-frontend` y `zoo-backend`. Además, se puede ver que se utiliza una Chart ya definida en un repositorio conocido, perteneciente a Bitnami[45]. De este repositorio se obtiene la Chart de MongoDB. De esta manera, se utiliza una Chart ya implementada y desarrollada por expertos, proporcionando muchas opciones de configuración y ofreciendo mucha más seguridad de lo que se tendría en el caso de crear una Chart de MongoDB propia para la aplicación.

```yaml
apiVersion: v2
name: zoo
description: Umbrella chart to deploy frontend, backend and mongo
version: 0.0.7
appVersion: "0.0.0"
dependencies:
  - name: zoo-frontend
    version: 0.0.0
    repository: file://charts/zoo-frontend
  - name: zoo-backend
    version: 0.0.0
    repository: file://charts/zoo-backend
  - name: mongodb
    repository: https://charts.bitnami.com/bitnami
    version: 15.0.0
    condition: mongo.internal.enabled
```

Listing 11: Definición de la Chart *umbrella* de la aplicación.

También se puede observar en la Figura 3.5 los recursos que se crean para cada una de las *subcharts* de la aplicación. Para más detalle en cuanto al funcionamiento de cada uno de los recursos, ver la Sección A.1.

Se detalla el proceso de desarrollo de la Chart de la aplicación en la Sección A.4.

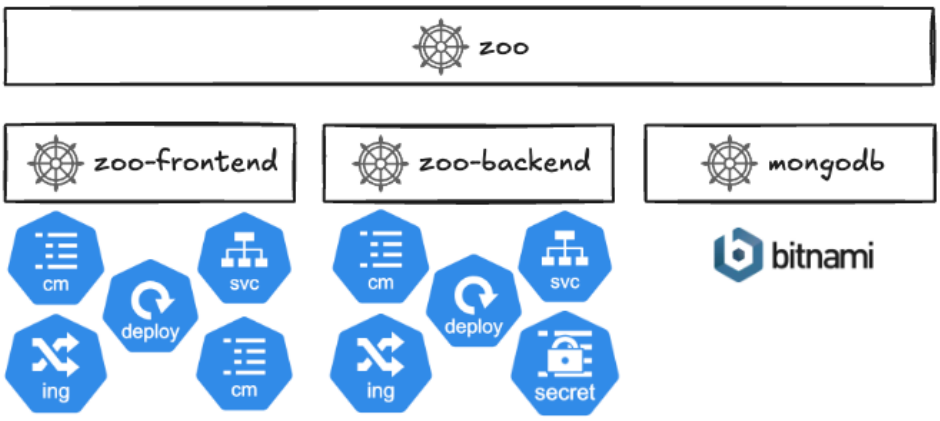

Figura 3.5: Diagrama de organización de las Charts de la aplicación. Imagen creada con excalidraw.com.

3.4. state

El repositorio de estado funciona como única fuente de verdad. Aquí se definen los valores que pueblan la Chart de la aplicación. Se utiliza una herramienta llamada `helmfile`[46]. Esta permite integrar Charts de Helm y valores, aunque estos se encuentren en lugares diferentes. En la Figura 3.6 se muestra la estructura del repositorio de estado, incluyendo la rama principal y la rama de despliegue.

Se puede ver el archivo de configuración de `helmfile` en el Listing 12. Los archivos del apartado de `environments` indican el *namespace* y la versión de la Chart que se va a utilizar. Los valores de las líneas 19-22 son aquellos que no dependen del entorno en el que se quiere desplegar, mientras que los de las líneas 23-26 sí que dependen del entorno, y se almacenan en un directorio dedicado a cada uno de ellos. En estos últimos archivos de valores dependientes del entorno se indica, por ejemplo, la *tag* a utilizar de las imágenes de Docker.

```
1   repositories:
2   - name: helm-repository
3     url: https://raw.githubusercontent.com/vieites-tfg/helm-repos
      ↪ itory/gh-pages/
4
5 environments:
6   dev:
7     values:
8       - ./dev.yaml
9   # ... otros entornos
10
11 ---
```

```
12
13 releases:
14   - name: zoo-{{ .Environment.Name }}
15     namespace: {{ .Environment.Name }}
16     chart: helm-repository/zoo
17     version: {{ .Values.version }}
18     values:
19       - zoo-frontend.yaml
20       - zoo-backend.yaml
21       - mongodb.yaml
22       - global.yaml
23       - {{ .Environment.Name }}/zoo-frontend.yaml
24       - {{ .Environment.Name }}/zoo-backend.yaml
25       - {{ .Environment.Name }}/mongodb.yaml
26       - {{ .Environment.Name }}/global.yaml
27       - global:
28           ghcrSecret:
29             enabled: true
30             password: {{ requiredEnv "CR_PAT" | quote }}
31       - zoo-backend:
32           mongo:
33             root:
34               user: {{ requiredEnv "MONGO_ROOT" | quote }}
35               password: {{ requiredEnv "MONGO_ROOT_PASS" | quote
                  ↪ }}
36       - mongodb:
37           auth:
38             rootUser: {{ requiredEnv "MONGO_ROOT" | quote }}
39             rootPassword: {{ requiredEnv "MONGO_ROOT_PASS" |
                ↪ quote }}
```

Listing 12: Archivo de configuración de `helmfile`.

Un ejemplo de archivo de valores se puede observar en el Listing 13. Como se ha comentado, el valor de la *tag* se ve indicado en la línea 3, con el formato que se menciona en la Sección A.3.

En la Sección 4.2 se explica cómo se especifica el entorno y, por lo tanto, los valores que se van a utilizar para poblar la Chart de la aplicación.

Este repositorio también es el lugar en el que se almacenan los archivos que definen los recursos que se van a desplegar en cada entorno. En la Figura A.2 se puede ver la estructura que tiene uno de los directorios (`dev`) de la rama de despliegue del repositorio *state*. La estructura de dicho directorio en la rama `deploy` es la misma para los directorios correspondientes a los demás entornos.

```yaml
zoo-backend:
  image:
    tag: "69ab8a1e"
  mongo:
    service:
      name: "zoo-dev-mongodb"
  ingress:
    hostTemplate: "api-zoo-dev.example.com"
```

Listing 13: Archivo de valores de `zoo-backend` en `dev`

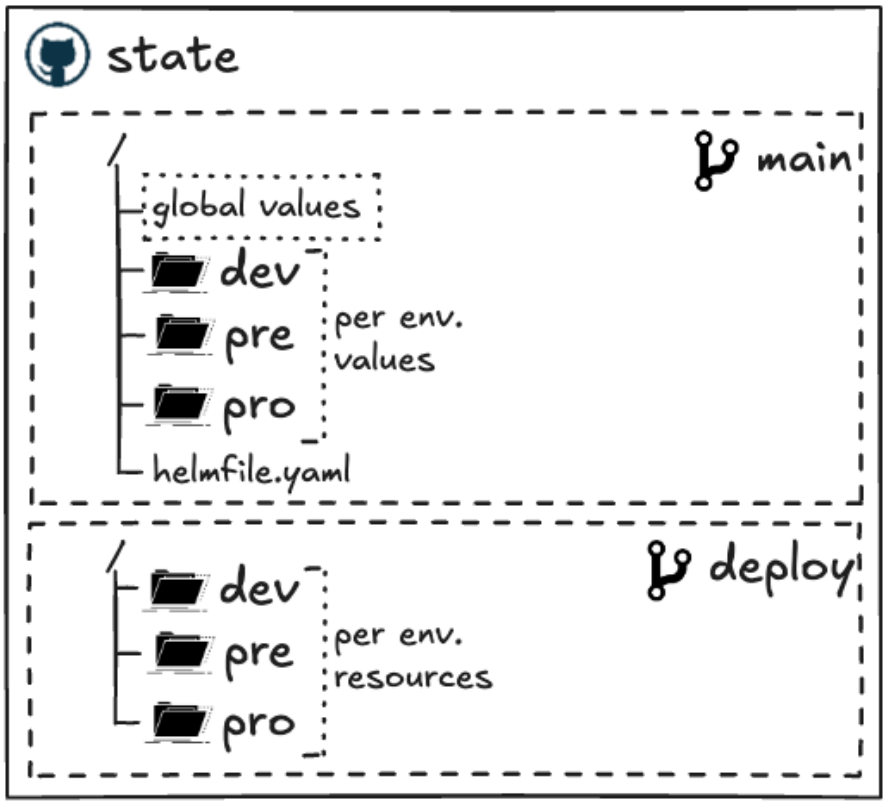

Figura 3.6: Diagrama de la estructura del repositorio de estado. Imagen creada con excalidraw.com.

Capítulo 4

Implementación del *pipeline* con Dagger

Este capítulo describe la implementación del *pipeline* con Dagger. Se parte de una versión inicial sin la herramienta, y se llega a un ciclo completo, plenamente integrado con esta herramienta. El objetivo es demostrar que Dagger aporta mayor flexibilidad a los desarrolladores para construir *pipelines* de CI y CD, sin depender de tener un entorno de desarrollo ni de una configuración específica para ejecutarlos.

4.1. CI

Tras realizar una primera aproximación de la aplicación de prueba, el siguiente paso consiste en realizar el ciclo de CI. En este ciclo se realizan los tests de la aplicación, el *linting*, es decir, verificar que el código cumpla ciertos estándares de estilo; y la publicación de imágenes de Docker y de paquetes NPM. Al principio se comenzó a implementar el ciclo completo sin el uso de Dagger. Se utilizó un `justfile` con los comandos necesarios para realizar tanto los tests como el *linting* de los paquetes de la aplicación, y se crearon dos *scripts*: uno para la publicación de imágenes y otro para la publicación de paquetes NPM. Además, se crea un Dockerfile para poder realizar las pruebas *end-to-end* del *frontend* con Cypress[47].

La desventaja con respecto al Dagger es evidente: *scripts* y comandos que "funcionan en mi máquina". Siguiendo esta práctica, sería necesario que todo el equipo de desarrollo utilizara el mismo entorno de desarrollo, y aún así, no se garantizaría el funcionamiento del ciclo de CI, debido a posibles configuraciones que cada desarrollador haya hecho a su sistema que pueda diferir de las configuraciones de los demás. Con este método, los desarrolladores no tienen margen de maniobra en cuanto al sistema que deben utilizar.

Ahora se explica el diseño e implementación con Dagger del mismo ciclo de

CI.

Una vez el ciclo de CI sin Dagger se ha realizado al completo, se pasó a traducir los scripts y comandos a un módulo de CI con Dagger, utilizando el SDK de Go.

En la Figura 4.1 se puede comprobar la estructura de este módulo. Se ha optado por implementar un objeto principal `Ci`, el cual tiene la capacidad de hacer uso de cualquiera de los otros dos objetos customizados, uno específicamente diseñado para gestionar el *frontend* y otro para el *backend*. Este diseño concuerda con la propia estructura y filosofía del *monorepo*: elementos separados que realizan tareas diferentes, pero que se interrelacionan desde un punto central (Lerna para el *monorepo*, y el objeto `Ci` para el módulo).

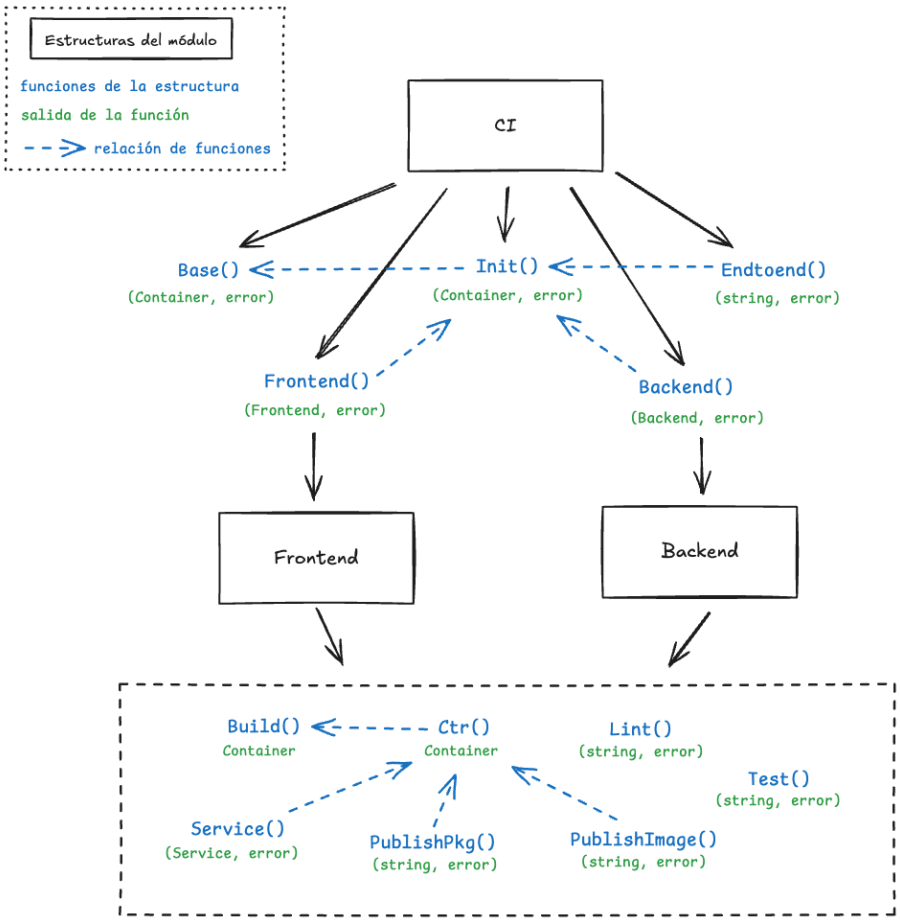

Figura 4.1: Diagrama del diseño del módulo de CI con Dagger. Imagen creada con excalidraw.com

Por lo tanto, la idea del diseño del módulo es la misma que la del *monorepo*, tener un elemento principal desde el que hacer uso de cualquiera de los otros tipos customizados[48], que se corresponden con los paquetes de la aplicación.

En Dagger se hace uso del "encadenamiento". Esto permite llamar a las fun-

ciones de los objetos que devuelven las propias funciones. Así, se puede llamar a la función del objeto principal del módulo que devuelve el objeto `Backend`, y se tendrá acceso a las funciones que están definidas para este paquete.

Para entender cómo funciona, se explicará un ejemplo sencillo con pseudocódigo del propio módulo. En el Listing 14 se puede ver un ejemplo de dos funciones del módulo de CI (se han escrito comentarios de lo que se hace en el código para reducir su tamaño). La primera función corresponde con el objeto principal del módulo. Este construye la imagen base y posteriormente crea un objeto `Backend`, el cual devuelve como referencia al final de la función. El hecho de que este objeto sea devuelto por la función, permite acceder a las funciones que este tiene definidas.

Una función definida para `Backend` es `Test`, que aparece justo después. En esta se utiliza la imagen base que se ha pasado como parámetro en la construcción del objeto. Se ejecuta dentro del contenedor que define dicha imagen el comando que permite correr los tests del paquete *backend* de la aplicación. Como se puede comprobar, se utiliza Lerna para ejecutar el *script* `test`, definido en el `package.json` del paquete correspondiente, indicado mediante el parámetro *scope*.

```
// Ci (main.go)

type Ci struct {
        // +required
        SecEnv  *dagger.Secret
        secrets secrets
}

func (m *Ci) Backend(
ctx context.Context,
// +defaultPath="/"
src *dagger.Directory,
) (*Backend, error) {
  base, _ := m.Init(ctx, src)

  // Se obtienen las claves y los valores de los secretos para
  // el *backend*.

  return &Backend{
    Name:    "backend",

    // Imagen base, con todo lo necesario para ejecutar los
    // comandos que se precise.
    Base:    base,
```

```
25
26      // Claves y valores de los secretos obtenidos antes.
27      Secrets: SecMap{Keys: keys, Values: values},
28    }, nil
29  }
30
31  // Backend (backend.go)
32
33  func (m *Backend) Test(ctx context.Context) (string, error) {
34    return m.Base.
35      WithExec([]string{"lerna", "run", "test", "--scope",
        ↪  "@vieites-tfg/zoo-backend"}).
36      Stdout(ctx)
37  }
38
```

Listing 14: Funciones del módulo de Dagger de CI.

La manera en la que se ejecuta a través de un comando la cadena de funciones anterior es como aparece en el Listing 15. El comando se realiza en el directorio de trabajo del módulo de CI. Se hace una llamada a Dagger, se pasan los valores requeridos por el módulo, los cuales se indican en la definición del objeto principal; y finalmente se empieza a llamar a las funciones que tiene cada uno de los objetos. A las funciones se las llama en formato "kebab-case", aunque en el propio código estén en "camelCase". Por lo tanto, primero se realiza la llamada a la función `backend` y, posteriormente, sabiendo que esta función devuelve un tipo `Backend`, se llama a la función `test` que este tiene definida.

```
1  dagger call --sec-env=file://../../.env backend test
```

Listing 15: Encadenamiento de funciones del módulo de CI.

De esta manera se puede lanzar la función que se prefiera de cualquiera de los paquetes. También, esta estructura y diseño del módulo facilita la escalabilidad de la aplicación, permitiendo añadir fácilmente más paquetes en el caso de ser necesario. Lo único que habría que hacer sería crear un objeto nuevo con funciones para las tareas que se quieran realizar en este, y añadir una función en el objeto principal del módulo que devuelva el nuevo objeto creado para el paquete.

Junto con la ejecución de los tests y del *linter*, se añade también al módulo la posibilidad de levantar por separado cada uno de los paquetes. Esto quiere decir que se permite acceder de manera local a los servicios de los paquetes, la página web por un lado y la API por el otro.

Antes de implementar esa lógica en el módulo de CI, se desarrolla lo necesario para conseguir levantar la aplicación entera de la manera más sencilla posible. En este caso, se utiliza un `Dockerfile` con varios pasos, en los cuales se construye y se prepara la imagen para la ejecución de cada uno de los paquetes. Esto se consigue con construcciones *multi-stage*, un ejemplo de esto se encuentra en el Listing 18.

Con el `Dockerfile` anterior y con la construcción de un `docker-compose.yaml`, se consigue levantar la aplicación de manera local. Tras conseguir esto, se traduce de nuevo la lógica al módulo de Dagger. Se crea en cada uno de los objetos del módulo una función `service`, que permiten levantar los paquetes por separado y acceder a ellos de manera local. Gracias a estas funciones es posible realizar los tests *end-to-end* de la aplicación, donde una única función es capaz de realizar los siguientes pasos:

- Tests y *linting* de cada uno de los paquetes.
- Levantar los servicios de los paquetes, permitiendo su comunicación de manera interna.
- Correr los tests de la aplicación con Cypress.

Todo lo anterior integrado en el módulo y ejecutado con una única llamada a una función del objeto principal de la aplicación.

Aquí es donde realmente se ve la potencia de Dagger. Todo se programa en un lenguaje conocido para el desarrollador, con sintaxis que no depende del sistema en el que se está desarrollando y sin la aparición de problemas relacionados con el entorno de ejecución, ya que funciona independientemente del sistema. Esto es gracias a ejecutar todo sobre un *runtime* de OCI, como Docker. Para realizar todo lo anterior sin Dagger, habría que: lanzar los comandos de Lerna de manera local, levantar los servicios con Docker Compose y correr los tests de Cypress con un `Dockerfile` específico para la causa. Y aún haciendo todo esto, no se tendría la flexibilidad de realizar pruebas y obtener *logs* que se tiene con Dagger. Además, Dagger permite acceder al contenedor en cualquier momento de la ejecución del *pipeline*.

4.2. CD

El ciclo de CD involucra: la construcción de los recursos de Kubernetes, el encriptado de los secretos y la publicación de todos los manifiestos necesarios en el repositorio de estado.

Para comenzar con este ciclo, se crearon un par de *scripts*, uno que permitía construir las imágenes de Docker y su publicación y otro que facilitaba la publicación de los paquetes NPM. Esto tiene las desventajas ya comentadas, son *scripts* que se ejecutan de manera local y que, por lo tanto, dependen del entorno en el que se ejecutan. En el caso de cambiar de entorno de desarrollo, sería necesario

crear nuevos *scripts* o acomodarlos al nuevo entorno. Esto solo trae problemas innecesarios a los desarrolladores, que se tienen que centrar en solucinar estos inconvenientes, externos a la finalidad del desarrollo de la aplicación en sí.

Tras crear los *scripts* anteriores, se pasó a la construcción de las Charts de Helm de la aplicación. Se diseñó la estructura de las Charts y los componentes que iban a tener cada uno de los servicios de la aplicación. Entonces se definieron todos los recursos con Helm y se realizaron pruebas hasta que finalmente todo funcionaba como se esperaba.

La construcción de las Charts era necesario realizarlo tras ser capaz de publicar las imágenes de Docker al Container Registry. Esto es porque Kubernetes utiliza imágenes construidas que obtiene de cualquier registro que se le indique, no se le puede pasar un *Dockerfile* para que construya la imagen.

Todo el proceso de creación de las Charts se ha realizado en local, utilizando comandos de Helm. Una vez comprobada la funcionalidad de esta, se procedió a la separación de responsabilidades: un repositorio para las Charts por un lado (`helm-repository`), y otro repositorio con los valores que pueblan las Charts por otro lado (`state`). Esto facilita el desarrollo y mantenimiento de las Charts. Además, se tiene una única fuente de verdad para obtener los valores.

Con los distintos repositorios creados, es necesario utilizar la herramienta `helmfile` para integrar Charts y valores. El archivo de configuración de esta herramienta se incluye en el repositorio de estado, manteniendo así la norma de tener toda la gestión de valores en un mismo lugar.

Así, ya es posible desplegar la aplicación utilizando Kubernetes. Lo siguiente es hacer uso de Dagger para permitir realizar el despliegue. Se comienza utilizando un método *push* de despliegue. En el propio módulo de Dagger que se creó inicialmente, se utiliza `helmfile` para desplegar la aplicación en un *cluster* de KinD. No existe ArgoCD, ni varios entornos. Por ahora es sencillo y directo, se crea un *cluster* con un módulo de KinD para Dagger, y se ejecuta el comando de `helmfile` necesario para desplegar la aplicación.

A continuación se decide emplear el método *pull*, ya comentado, donde existe una herramienta que es la que se encarga de obtener los recursos que se van a desplegar. Esa herramienta es ArgoCD. El módulo de CD de Dagger pasa de desplegar la aplicación a ser el encargado de publicar en la rama `deploy` del repositorio `state` los recursos que se van a desplegar en cada entorno. Esto implica

1. Realizar el *template* de los recursos con `helmfile`, es decir, mostrar el renderizado de estos pero sin desplegarlos.
2. Obtener los secretos y encriptarlos.
3. Crear los archivos que dice a ArgoCD cómo desencriptar los secretos,
4. Publicar todo en la rama de despliegue del repositorio de estado.

Tras tener en funcionamiento el módulo, se procede a crear el *workflow* de

GitHub, cuyo funcionamiento se explica en la Sección A.3. En resumen, se encarga de ejecutar los módulos de CI y CD, ejecutando los tests *end-to-end* previamente a la publicación de las nuevas imágenes de Docker y actualizando el repositorio de estado con la definición de los recursos en el entorno que proceda, utilizando las nuevas imágenes creadas.

De esta manera, se integra todo lo que se ha creado:

- Los módulos de Dagger de CI y de CD.
- Las Charts de Helm y valores.
- Los *clusters* de KinD con los entornos levantados, con ArgoCD instalado en cada uno de ellos.
- Un *workflow* de GitHub que enlaza todo lo anterior.

Capítulo 5

Pruebas

En este capítulo se realizan pruebas con el fin de medir, cuantitativa y cualitativamente, las ventajas y/o desventajas que tiene el uso de Dagger para la gestión de un ciclo completo de CI/CD. Se proponen dos pruebas diferentes:

- Ejecución de los tests *end-to-end* con Dagger.
- Construcción y ejecución de tests de los paquetes de la aplicación. Tanto con Dagger como sin él.

En cada uno de los apartados dedicados a las diferentes pruebas se indica lo que implica realizarlos.

5.1. Entorno de pruebas

Las pruebas se realizan en un ordenador portátil con las siguientes características de *software* y *hardware*:

- *PC*: MacBook Air, 13-inch, 2024
- *Chip*: Apple M3
- *RAM*: 16 GB
- *Sistema Operativo*: MacOs Sequoia 15.5
- *Red*: Por cable, aproximadamente 1 GB simétrico.

Las versiones de los binarios utilizados se pueden encontrar en el Apéndice B.

5.2. Prueba 1

Lo que se pretende comprobar con esta prueba es ver cómo afecta el almacenamiento de caché que lleva a cabo Dagger tras varias ejecuciones de una función. Se realizarán medidas de tiempo bajo ciertas condiciones, con el fin de obtener diferentes valores y observar cuánto tiempo permite ahorrar Dagger gracias a su

gestión de la caché. Se ejecuta varias veces la función `endtoend` implementada en el módulo de CI. Dicha función realiza las siguientes acciones:

- Ejecuta los tests y el *linting* del paquete del *backend* de la aplicación.
- Realiza el *linting* del *frontend*.
- Levanta los servicios de ambos paquetes.
- Ejecuta los tests del *frontend* con Cypress.

En definitiva, con la función que se va a utilizar en esta prueba, se realiza un testeo integral de toda la aplicación.

Antes realizar esta prueba se borra la caché que pudieran tener almacenada tanto de Dagger como de Docker. Para esto se utilizan los comandos del Listing 16.

```
dagger core engine local-cache prune

docker system prune -af
```

Listing 16: Borrado de caché de Dagger y Docker.

Para probar la función, simplemente es necesario ejecutar el comando que se muestra en el Listing 17.

```
dagger call --sec-env=file://../../.env endtoend
```

Listing 17: Testeo integral de la aplicación con el módulo de CI.

Se lanza el comando anterior cuatro veces, realizando un cambio en el código de la aplicación entre cada una de las ejecuciones. Esto tiene como finalidad evitar que se haga un uso total de la caché, debido a que Dagger comprueba los parámetros de entrada que se pasan a cada función (el código fuente, en este caso) para decidir si va a utilizar la caché o no. Si los parámetros de entrada cambian, no hace uso de la caché de dicha función, y la ejecuta. En el caso de que no cambien los parámetros de entrada, se hace uso de la salida almacenada en la caché para esa función.

En la Figura 5.1 se pueden observar los tiempos que se obtienen tras realizar las ejecuciones del comando anterior. Los resultados en azul muestran los tiempos obtenidos haciendo cambios en el código de la aplicación entre cada una de las ejecuciones. Se observa que, tras la primera vez, se reduce el tiempo de ejecución un 40 %. A partir de ahí, el tiempo se mantiene constante, y se continuaría con este tiempo en consecuentes ejecuciones, como muestra la línea azul punteada. En rojo se muestran los tiempos obtenidos posteriormente, esta vez sin realizar

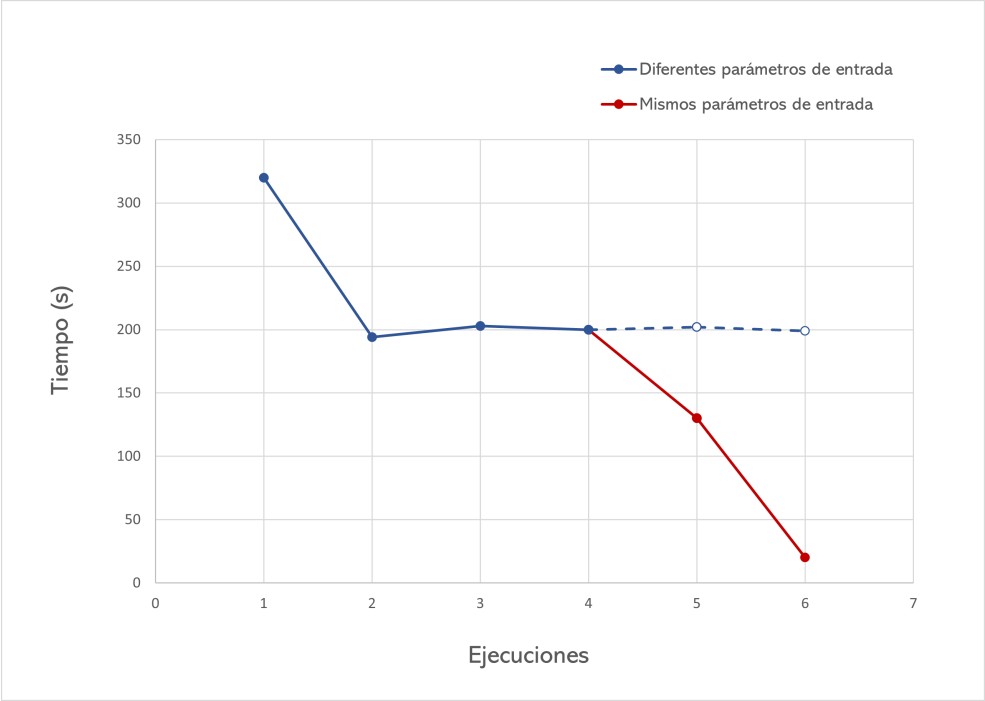

Figura 5.1: Tiempos de ejecución, con y sin cambios en el código de la aplicación entre ejecuciones.

cambios en el código y, por lo tanto, manteniendo los parámetros de entrada de la función iguales a como estaban en la última ejecución. Esto tiene como resultado una caída drástica del tiempo de ejecución, debido a lo comentado previamente. Dagger comprueba que la entrada de la función para la ejecución actual es la misma que la anterior, la cual tiene almacenada en caché. Por lo tanto, no ejecuta la función, y toma como salida la que ya había obtenido anteriormente. Así, se obtiene una reducción con respecto al tiempo inicial de un 94 %.

Por lo tanto, se puede observar la buena gestión que Dagger hace de la caché. Es capaz de reducir un tercio el tiempo de ejecución de la misma función, con cambios en el código entro ejecuciones. En el caso de realizar pruebas sin hacer cambios en los parámetros de entrada, el tiempo de ejecución se reduce un 94 %. Este último valor de 20 segundos no es realista para el caso de uso de estos módulos, donde la entrada de las funciones (el código fuente) debería cambiar constantemente. Sin embargo, puede ser muy útil conocer esta capacidad de Dagger para aplicarlo en otras situaciones.

5.3. Prueba 2

La finalidad de esta prueba es comparar cómo se gestionan acciones básicas en el desarrollo de *software*, como son la construcción de la aplicación y la ejecución

de tests, tanto utilizando el módulo de Dagger como sin usarlo. Por lo tanto, se busca un resultado cualitativo, en el que lo importante reside en la facilidad del desarrollador para implementar y mantener lo necesario para llevar a cabo dichas acciones.

El módulo de Dagger de CI es el encargado de realizar las acciones comentadas. Se proporciona una función de construcción y de ejecución de tests para cada uno de los paquetes. El ciclo de construcción y testeo de la aplicación está creado íntegramente utilizando un único lenguaje de programación, lo cual ofrece simplicidad tanto a la hora de la implementación como del mantenimiento. Se puede utilizar lógica de programación para adaptar el *pipeline* a distintos escenarios. El desarrollador también puede reutilizar código de manera muy sencilla, mediante la creación de funciones, a las cuales se las puede llamar desde cualquier sitio. El código está organizado y se previene la duplicación de lógica de programación.

La ejecución del módulo se puede hacer de manera local gracias a que Dagger lanza sus funciones sobre un *runtime* de OCI, como Docker. Dagger también facilita la reproducibilidad gracias a que construye grafos para definir dependencias entre funciones, donde cada nodo del grafo define su entrada y su salida. Además, lo anterior también permite gestionar de mejor manera la caché, haciendo que las ejecuciones sean más rápidas, como ya se ha visto en la primera prueba de este capítulo.

En el caso de que no se use Dagger, el desarrollador tendría que optar por métodos convencionales. Comenzaría por crear un `Dockerfile` que permitiera realizar la construcción de los paquetes de la aplicación. Para realizar los tests del *frontend*, haría falta otro `Dockerfile` específico para utilizar Cypress. Estos tests implicarían también implementar un archivo de Docker Compose, el cual permite levantar todos los servicios al mismo tiempo con posibilidad de comunicación entre ellos, necesario para realizar los tests *end-to-end* del *frontend*.

Además, extender la lógica de programación implicaría crear *scripts*, los cuales habría que acomodar a un nuevo entorno de desarrollo en el caso de que este cambie en algún momento. Esto añade más carga al desarrollador, que tiene que mantener dichos *scripts* a lo largo de los diferentes entornos en los que se quieran ejecutar. Crear diferentes *scripts* hace la reutilización de código más complicada, y puede llevar a la duplicación de lógica de programación. Finalmente, también se encuentra el problema de la reproducibilidad, siendo esto más complicado de conseguir debido a las pequeñas diferencias que pueda haber en la configuración del entorno de desarrollo de cada programador.

Sin embargo, la realidad es que las tecnologías y herramientas que se utilizan de manera convencional son más conocidas que Dagger. Por lo tanto, inicialmente puede ser lógico encaminarse por el uso de estas tecnologías, ya que permite iniciar la implementación de este tipo de ciclos más rápidamente. Además, aunque la API de Dagger es intuitiva si ya conoces las otras herramientas convencionales, su amplitud introduce una pequeña curva de aprendizaje. Pero, como se ha comentado, el mantenimiento a la larga es más sencillo con Dagger, debido a que

todo se encuentra en un mismo lugar e implementado con un mismo lenguaje de programación, como Go, Python o Typescript (entre otros). Y, lo más importante, Dagger tiene una gran capacidad de portabilidad gracias a que ejecuta todas las funciones en entornos aislados, como los contenedores de Docker. Esto es una ventaja frente a los *scripts*, que es necesario mantener en diferentes entornos en el caso de que no se utilice Dagger.

En la Tabla 5.1 se proporciona una tabla a modo de resumen, con las diferencias que se encuentran entre las dos opciones de implementación que se han comentado.

Aspecto	*Con Dagger*	*Con métodos convencionales*
Configuración inicial	Requiere un mayor esfuerzo para aprender su API	Más rápido de implementar al utilizar herramientas conocidas
Mantenimiento	Centralizado en un solo lenguaje; facilita la reutilización y evita duplicación	Requiere mantener múltiples archivos, lo cual es propenso a inconsistencias
Reproducibilidad	Alta, gracias a la ejecución en entornos aislados	Menor, debido a las posibles diferencias entre entornos locales de desarrollo
Flexibilidad	Alta, permitiendo aplicar lógica de programación a todo el *pipeline*	Limitada por la necesidad de creación y adaptación de diferentes *scripts*
Caché	Optimizada de forma automática	Manual, más dificil de mantener y optimizar
Portabilidad	Elevada, todo se ejecuta en contenedores	Menor, requiere adaptar *scripts* o configuraciones según el entorno
Escalabilidad	Adecuado para cualquier proyecto, destacando en aquellos más grandes	Difícil de escalar conforme crece la lógica
Curva de aprendizaje	Moderada, API extensa pero intuitiva si se tiene experiencia previa con contenedores	Baja, herramientas ampliamente conocidas
Madurez del ecosistema	En crecimiento activo	Herramientas maduras con amplia documentación y soporte

Tabla 5.1: Diferencias entre Dagger y métodos convencionales.

Capítulo 6

Conclusiones y posibles ampliaciones

A lo largo del Trabajo de Fin de Grado se explica todo el proceso de diseño e implementación de: la aplicación de prueba, las Charts de Helm que permiten el despliegue de la aplicación en Kubernetes y los módulos de Dagger para los ciclos de CI y CD. Esto último es la razón por la que se ha hecho este Trabajo de Fin de Grado.

Crear un módulo de Dagger implica tener conocimientos de programación, pero también de creación de entornos aislados, como son los contenedores de Docker. Además, requiere estudiar la API que proporciona, la cual es más intuitiva en el caso de tener los conocimientos que se acaban de mencionar. Por lo tanto, es una herramienta que tiene una curva de aprendizaje moderada, sobre todo inicialmente.

Una vez se comienza el desarrollo de los módulos, el hecho de que se utilice un lenguaje de programación como Go facilita mucho la gestión de la lógica de programación, permitiendo crear funciones que se pueden llamar desde cualquier otro lugar dentro del módulo. Tras tener más conocimiento de la herramienta, la dificultad de desarrollar módulos de CI y CD radica más en el diseño de los propios módulos que en la programación en sí.

La capacidad de Dagger para crear módulos de CI/CD de manera programática facilita en gran medida el mantenimiento de estos a largo plazo, ya que no se utilizan diferentes herramientas o tecnologías que puedan depender del entorno de desarrollo. A lo largo de la implementación de los módulos, el programador se da cuenta de las ventajas que tiene poder centralizar toda la lógica de programación. Cabe mencionar la gran portabilidad que proporciona Dagger gracias a que se ejecuta sobre un *runtime* de OCI, lo cual permite ejecutar los módulos creados con esta herramienta de manera local en una gran variedad de entornos de desarrollo. Además, se encuentran de manera pública módulos en el Daggerverse, los cuales están implementados por otros desarrolladores. Estos se pueden integrar muy fácilmente en otros módulos en desarrollo.

Por otro lado, la gestión que realiza Dagger de la caché permite al desarrollador reducir en gran medida los tiempos de ejecución de funciones que se lanzan de manera repetitiva a lo largo del proceso de desarrollo de la aplicación. Un ejemplo de esto se ha visto en las pruebas realizadas en el Capítulo 5.

Todo lo mencionado anteriormente hace de Dagger una excelente opción para crear módulos de este tipo para cualquier aplicación. Aprender a utilizar esta herramienta puede facilitar en gran medida la implementación, el mantenimiento, la portabilidad, la flexibilidad, la escalabilidad y la reproducibilidad de *pipelines* de CI/CD para una aplicación.

6.1. Vías de mejora

A continuación, se mencionan elementos que se pueden mejorar en un futuro con respecto al trabajo realizado:

- Posible mejora del diseño de los módulos de CI/CD.
- Añadir más funcionalidades a la aplicación de prueba.
- Despliegue de la aplicación en proveedores en la nube, en vez de levantar todos los *clusters* de KinD de manera local.
- Gestión de secretos más avanzada.
- Añadir una herramienta de monitorización y observabilidad de métricas de la aplicación, tales como DataDog[49] o Prometheus[50].
- Escalado horizontal[51], con el fin de distribuir la carga entre múltiples instancias, mejorando la escalabilidad, la tolerancia a fallos y la disponibilidad de la aplicación.
- Análisis de imágenes de Docker y `Dockerfiles`, para mayor seguridad, empleando herramientas como Trivy[52].
- Realización de encuestas o entrevistas a más desarrolladores que prueben la herramienta, con el fin de obtener más *feedback* y validar con mayor firmeza las conclusiones a las que se han llegado en este Trabajo de Fin de Grado.

Apéndice A

Manuales técnicos

Todo el código fuente se encuentra en la siguiente organización de GitHub: vieites-tfg. En este se puede encontrar tres repositorios y un registro de contenedores de GitHub, donde se publican las imágenes y los paquetes NPM de la aplicación.

A.1. Descripción de tecnologías

Just

Herramienta con la misma funcionalidad que Make, pero con más funcionalidades, entre las cuales destacan:

- Poder pasar parámetros a las "recetas" (las "reglas" en `make`).
- Crear *aliases* para las recetas.
- Cargar archivos `.env`.
- Poder definir recetas como *scripts* en el lenguaje que se prefiera, simplemente añadiendo el *shebang*[15] correspondiente.
- Ser capaz de ser invocado desde cualquier subdirectorio.

Como se puede comprobar en el Listing 2, el archivo de configuración de `just`, en este caso nombrado habitualmente `justfile`, tiene una estructura similar a la de un `Makefile`. La diferencia principal es que los nombres de las recetas no hacen referencia a un archivo objetivo que se supone que se debe crear al ejecutar el bloque de comandos, sino que se trata simplemente del nombre de la receta.

Docker

Docker permite empaquetar aplicaciones, creando imágenes con las dependencias necesarias para que la aplicación se lance sin problemas. Las imágenes

generadas se pueden ejecutar, creando contenedores, que son entornos completamente aislados del contexto de la máquina en la que han levantado.

El proceso más habitual a la hora de construir una imagen de Docker es definir un `Dockerfile` como el del Listing 18. En este se indica, paso a paso, todo el proceso de instalación de dependencias y compilación del código fuente, necesario para lanzar la aplicación. En el `Dockerfile` mencionado, se puede observar que, además, se hace uso de *multi-stage builds*, distintos estados de la construcción. Esto permite construir imágenes de Docker más pequeñas y optimizadas separando el proceso de construcción en distintas fases.

```
# Base

FROM node:20 AS base

WORKDIR /app

COPY package.json lerna.json yarn.lock* ./

COPY packages packages/

RUN yarn install

RUN yarn global add lerna@8.2.1

RUN yarn global add @vercel/ncc

# Frontend build stage

FROM base AS frontend-build

WORKDIR /app

RUN lerna run --scope @vieites-tfg/zoo-frontend build

# Frontend

FROM nginx:alpine AS frontend

WORKDIR /usr/share/nginx/html

COPY --from=frontend-build /app/packages/frontend/dist .

EXPOSE 80
```

```
34
35 CMD ["nginx", "-g", "daemon off;"]
```

Listing 18: Extracto de `Dockerfile` utilizado en el proyecto.

El comando del Listing 19, muestra en la línea 1 cómo se construye la imagen que se define en el `Dockerfile` del directorio de trabajo actual (`.`), con el nombre `my-image`. Con el siguiente comando se ejecuta la imagen. Las *flags* indican:

```
1 docker build -t my-image .
2
3 docker run --rm -d -p 8080:80 my-image
```

Listing 19: Construir y levantar una imagen de Docker.

- `--rm`

  Se eliminará el contenedor creado al finalizar su ejecución.

- `-d`

  El contenedor correrá en *background*.

- `-p 8080:80`

  Se mapea el puerto 8080 de la máquina local al puerto 80 del contenedor.

Docker Compose

Con Docker se es capaz de gestionar varios servicios desplegados en distintos contenedores. Pero existe una herramienta que apareció poco después y que facilita esta tarea, llamada "Docker Compose"[17]. Esta permite simular entornos con múltiples contenedores para desarrollar localmente.

En el archivo que se muestra en el Listing 20, se puede observar cómo se configura el despliegue de tres servicios diferentes. Cada uno de los servicios se construye a partir de una imagen de Docker. Las imágenes correspondientes al *frontend* y al *backend* de la aplicación (`zoo-frontend` y `zoo-backend`, respectivamente) se generan y almacenan en un registro de GitHub al finalizar el ciclo de CI. Una vez publicadas, se pueden descargar indicando en el campo `image` el registro en el que están almacenadas junto con su nombre, como se puede ver en las líneas 3 y 15.

```
1 services:
2   zoo-frontend:
3     image: ghcr.io/vieites-tfg/zoo-frontend
4     container_name: zoo-frontend
```

```
    hostname: zoo-frontend
    ports:
      - "8080:80"
    depends_on:
      - zoo-backend
    environment:
      NODE_ENV: production
      YARN_CACHE_FOLDER: .cache

  zoo-backend:
    image: ghcr.io/vieites-tfg/zoo-backend
    container_name: zoo-backend
    hostname: zoo-backend
    ports:
      - "3000:3000"
    depends_on:
      - mongodb
    environment:
      NODE_ENV: production
      YARN_CACHE_FOLDER: .cache
      MONGODB_URI:
        ↪ "mongodb://${MONGO_ROOT}:${MONGO_ROOT_PASS}@mongodb:${M
        ↪ ONGO_PORT:-27017}/${MONGO_DATABASE}?authSource=admin"

  mongodb:
    image: mongo:7.0
    container_name: zoo-mongo
    hostname: mongodb
    environment:
      - MONGO_INITDB_DATABASE=${MONGO_DATABASE}
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_ROOT}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_ROOT_PASS}
    ports:
      - "${MONGO_PORT_HOST:-27017}:${MONGO_PORT:-27017}"
    volumes:
      - ./mongo-init/:/docker-entrypoint-initdb.d/
      - mongo_data:/data/db

volumes:
  mongo_data:
```

Listing 20: `docker-compose.yaml` usado en el proyecto.

Utilizando el comando del Listing 21, y teniendo en cuenta el Listing 20:

- Se levantan los tres servicios.
- Se les pasarán las variables de entorno indicadas.
- Se podrá acceder a ellos a través de los puertos establecidos en, siendo el primer número el puerto local y el segundo el puerto del contenedor (`<local>:<contenedor>`).
- Se compartirán los volúmenes mencionados.

```
1 docker compose up
```

Listing 21: Despliegue con Docker Compose.

Los valores de las variables de entorno, los indicados como `${variable}`, se obtienen de un archivo `.env`, el cual debe estar presente en el mismo directo que el archivo de configuración. En otro caso, es posible indicar la ruta al archivo mediante el campo `env_file`, dentro de cada uno de los servicios configurados.

Kubernetes

Se trata de una herramienta de *software* que permite orquestar contenedores y el ciclo de vida de las aplicaciones en contenedores que viven en un *cluster*.

En la Figura A.1, se puede observar la estructura de un *cluster* de Kubernetes, el cual se compone de dos tipos principales de servidores ("nodos"):

- El *Control Plane*.

  Toma todas las decisiones. Se encarga de que todo el sistema funcione como debe.

- Los nodos de trabajo.

  Donde realmente se ejecutan las aplicaciones. Reciben órdenes del *Control Plane*. Puede y suele haber más de un nodo de trabajo en un *cluster*.

Kubernetes permite definir diferentes elementos en archivos `YAML`. Estos archivos describen el estado que deseamos que tenga el sistema en todo momento. Kubernetes se encarga de procesar estos archivos e intentar hacer que el estado real del sistema sea igual al estado deseado.

Entre los elementos que se pueden construir se encuentran los siguientes:

- Pod.

  Es la unidad más pequeña que se puede crear. Puede tener uno o más contenedores, pero lo normal es que tenga solo uno. Su función es encapsular y ejecutar la aplicación que le corresponda, que se indica mediante una imagen de Docker.

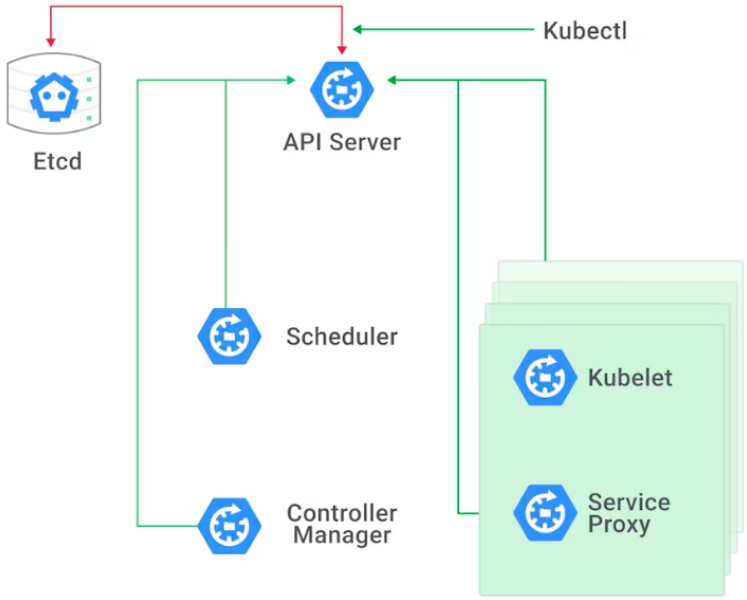

Figura A.1: Arquitectura de Kubernetes[18].

- Deployment.

  Se trata de un controlador de Pods. Normalmente se utiliza este tipo de elementos en vez de crear Pods directamente. Esto es porque le puedes indicar la cantidad de Pods (réplicas) que deseas que haya en todo momento en el sistema, y el Deployment lo hace por ti.

- Service.

  Debido a que los Pods son efímeros, es decir, se crean y se destruyen constantemente, cambiando así su dirección IP; es necesario tener un elemento que funcione como punto fijo de acceso a un Pod. Para eso sirve un Service. Estos proporcionan un nombre y una IP únicos y fijos para los Pods.

- Ingress.

  Es un elemento más avanzado que un Service. Permite gestionar el acceso desde Internet, dirigiendo las peticiones hacia los servicios correctos dentro del *cluster*.

- ConfigMap.

  Este elemento está diseñado para almacenar valores no sensibles. Se definen en formato clave-valor. Permite separar la configuración de una aplicación de su código.

- Secret.

  Es muy similar a un ConfigMap, pero está diseñado para almacenar y gestionar información sensible. Su función es guardar datos que no se deberían

mostrar a simple vista en la configuración de una aplicación, como contraseñas o *tokens* de autenticación.

ArgoCD

Herramienta diseñada específicamente para Kubernetes. Automatiza el despliegue de las aplicaciones, supervisando repositorios de Git para aplicar los cambios que se realizan en ellos automáticamente en el *cluster*. Su uso permite no tener que realizar actualizaciones manuales en entornos de producción, y hace que el entorno siempre esté sincronizado con el código definido en el repositorio. Esto se consigue gracias al proceso de reconciliación que realiza ArgoCD cada vez que detecta cambios en el repositorio que se le ha indicado.

A.2. Gestión de secretos

Para encriptar los secretos se utilizan dos herramientas:

- `age`[40].

  `age` es una herramienta creada con Go que permite encriptar y desencriptar archivos. Una vez instalada, es necesario crear unas claves privada y pública. Estas se proporcionan en el manual de usuario (ver Apéndice B) para poder probar la aplicación.

- SOPS[41].

  SOPS (*Secrets OPerationS*) permite editar archivos encriptados, pero con la capacidad de realizar la encriptación de campos específicos de tipos de archivos como YAML o JSON. Esta herramienta tiene compatibilidad con `age`, por lo que es un recurso excelente para encriptar recursos de Kubernetes, ya que estos se definen mediante archivos en formato YAML.

  El archivo de configuración de SOPS es como el que se muestra en el Listing 22. En él se indican: el formato de los nombres de los archivos que tiene que encriptar (línea 2), los campos que *no* tiene que encriptar (línea 3) y la herramienta de encriptado junto con la clave pública que le sirve para realizar la encriptación (línea 4).

```
1 creation_rules:
2   - path_regex: ".*\.ya?ml$"
3     unencrypted_regex: "^(apiVersion|metadata|kind|type)$"
4     age: age15peyc7 #...
```

Listing 22: Archivo de configuración de SOPS.

Con las claves pública y privada, y la configuración de SOPS, se es capaz de encriptar los Secrets de Kubernetes durante el ciclo de CD, en el módulo de Dagger. Los archivos encriptados se suben a la rama de despliegue del repositorio de estado, junto con los demás recursos que definen la aplicación.

Para desencriptar los secretos, ArgoCD necesita información: la clave privada creada con `age` y las herramientas necesarias para gestionar archivos encriptados con SOPS. La clave se le proporciona durante uno de los pasos de la creación de los *clusters*, que se muestra en la Sección 3.2. Sin embargo, es necesario decirle a ArgoCD cómo utilizarla.

Para ello es necesario el uso de otras dos herramientas:

- `kustomize`[42].

  `kustomize` permite modificar valores de definiciones de recursos de Kubernetes sin necesidad de realizar cambios directamente en el archivo original,

o bien crear recursos completamente nuevos a partir de otros. Las customizaciones se definen igual que cualquier otro recurso de Kubernetes, como se muestra en el Listing 23. En dicho archivo, el cual se debe llamar `kustomization.yaml`, se indican: `resources`, que son los archivos que se van a incluir tal y como estén definidos; y `generators`, archivos que muestran cómo construir recursos a partir de otros. El generador que se utiliza en este Trabajo de Fin de Grado se muestra en el Listing 24. Estos archivos los lee ArgoCD, los interpreta, y así sabe cómo comportarse y las herramientas que tiene que utilizar cuando encuentra los archivos que definen los recursos, como `secrets.yaml` y `non-secrets.yaml`.

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - non-secrets.yaml
generators:
  - secret_generator.yaml
```

Listing 23: Archivo `kustomization.yaml`.

- `ksops`[43].

  `ksops` (kustomize-SOPS) es un *plugin* de `kustomize` para gestionar recursos encriptados con SOPS. Se utiliza, sobre todo, para desencriptar Secrets o ConfigMaps de Kubernetes encriptados con SOPS. En el generador del Listing 24 se ve cómo se le indica a ArgoCD que debe utilizar el *plugin* `ksops` para desencriptar el archivo con el nombre `secrets.yaml`.

```yaml
apiVersion: viaduct.ai/v1
kind: ksops
metadata:
  name: secret-generator
  annotations:
    config.kubernetes.io/function: |
      exec:
        path: ksops
files:
  - secrets.yaml
```

Listing 24: Generador de los secretos.

Lo único que falta es instalar estas herramientas dentro de ArgoCD, `kustomize` y `ksops`, e indicarle dónde se encuentra la clave privada que usará para desencriptar los secretos.

Esto se consigue con los valores que muestran en la línea 26 del Listing 8. El archivo `values.yaml` tiene el contenido que se muestra en el Listing 25. Con estos valores se consigue:

- Indicar las *flags* que tiene que utilizar ArgoCD a la hora de ejecutar comandos con `kustomize` (líneas 1-4).
- Crear variables de entorno (líneas 6-11) que guardan información sobre (de arriba a abajo, respectivamente) el directorio raíz de archivos de configuración del sistema y la ruta en la que se puede encontrar la clave privada de `age`.
- Construir dos volúmenes (líneas 13-18), uno para almacenar los binarios de `kustomize` y `ksops`, y otro para la clave privada que se ha introducido en el *cluster* en las líneas 20-22 del Listing 8.
- Instalar los binarios de `kustomize` y `ksops` en el volumen `custom-tools` creado previamente (líneas 20-21).
- Permitir a ArgoCD acceder a los binarios y a la clave, montando los volúmenes que contienen estos elementos dentro del servidor principal de ArgoCD (líneas 33-41).

```
1  configs:
2    cm:
3      kustomize.buildOptions: "--enable-alpha-plugins --enable-exec"
4      ui.bannerpermanent: "true"
5
6  repoServer:
7    env:
8      - name: XDG_CONFIG_HOME
9        value: /.config
10     - name: SOPS_AGE_KEY_FILE
11       value: /.config/sops/age/keys.txt
12
13   volumes:
14     - name: custom-tools
15       emptyDir: {}
16     - name: sops-age
17       secret:
18         secretName: sops-age
19
20   initContainers:
21     - name: install-ksops
22       image: viaductoss/ksops:v4
23       command: ["/bin/sh", "-c"]
```

```
24        args:
25          - echo "Installing KSOPS and Kustomize...";
26            mv ksops /custom-tools/;
27            mv kustomize /custom-tools/;
28            echo "Done.";
29        volumeMounts:
30          - mountPath: /custom-tools
31            name: custom-tools
32
33    volumeMounts:
34      - mountPath: /usr/local/bin/kustomize
35        name: custom-tools
36        subPath: kustomize
37      - mountPath: /usr/local/bin/ksops
38        name: custom-tools
39        subPath: ksops
40      - name: sops-age
41        mountPath: /.config/sops/age
```

Listing 25: Valores que pueblan la Chart de ArgoCD.

Los archivos: `kustomization.yaml`, `secret_generator.yaml`, `secrets.yaml` y `non-secrets.yaml`; todos ellos son los ficheros que se disponen en la rama de despliegue `deploy` del repositorio `state`. Por lo tanto, son los archivos que ArgoCD obtiene y utiliza para desplegar toda la aplicación en los distintos entornos.

En la Figura A.2 se puede ver el ciclo completo de encriptado y desencriptado de secretos

A.3. Promoción de entornos

Referenciado en las Secciones 3.2 y 3.4.

Estos son los pasos que se realizan en el *workflow* que realiza el despliegue al entorno correspondiente:

1. Se clonan los repositorios necesarios (líneas 12-15).
2. Se instala Dagger (líneas 17-18).
3. Se determina el entorno y la *tag* que se le pondrá a la imagen de Docker de los paquetes de la aplicación (*backend* y *frontend*) (líneas 20-29).
    - En `dev` la *tag* se trata de los ocho primeros caracteres del último *commit* que se ha realizado. De esta manera se sabe a ciencia cierta el código que conforma la aplicación en dicha imagen, y facilita la detección de errores. El entorno es `dev` siempre que el evento que haya

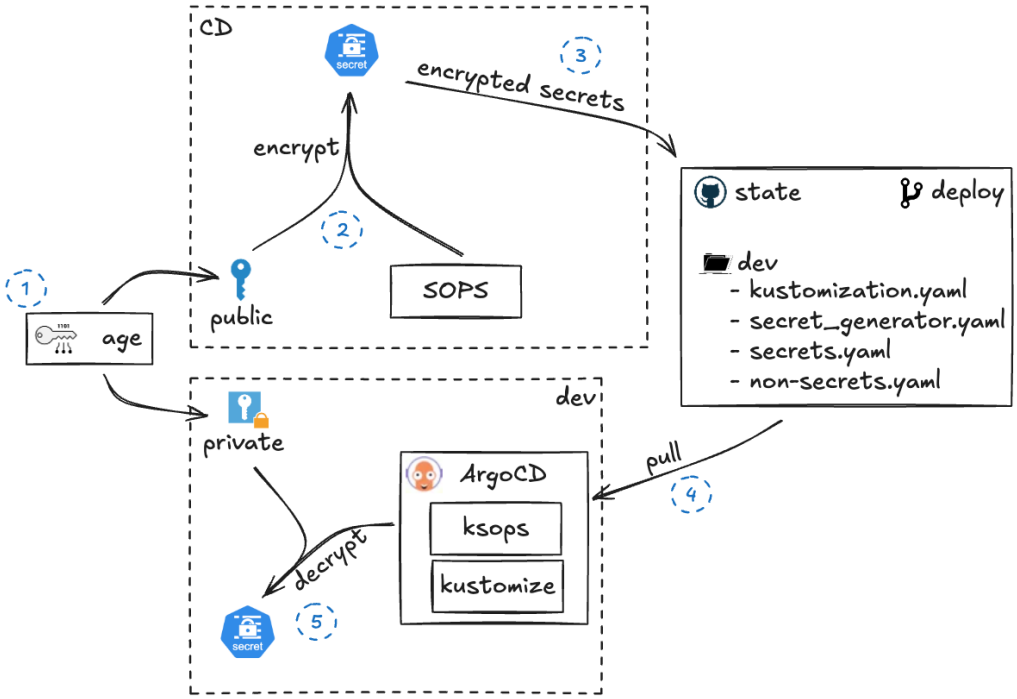

Figura A.2: Encriptado y desencriptado de secretos. Imagen creada con excalidraw.com. También referenciado en las Secciones 3.4 y A.2.

disparado el *workflow* sea un *push* de la PR a la rama principal, en este caso `main`.

- En `pre` se publica la imagen y los recursos siempre que el evento que lanza el *workflow* sea la creación de una *prerelease*. La *tag* que se utiliza es la que se le pone al nombre de la *prerelease*. Esta debería tener formato SemVer[44] con una coletilla "`snapshot`" (e.j. `1.2.3-snapshot`). Se utiliza esta coletilla con el fin de dar a entender que dicha imagen es una copia o "captura" de lo que sería la versión final de la imagen, la que se publicaría en el entorno de producción.
- En `pro` se despliega la aplicación cuando el evento que lanza el *workflow* no es una *prerelease*, sino una *release*. Al igual que en `pre`, la *tag* sigue el formato SemVer, pero en este caso sin la coletilla que se usaba antes (e.j. `1.2.3`).

4. Se crean los archivos necesarios con las variables almacenadas en GitHub (líneas 31-39). Se proporcionan estos archivos y datos en los manuales de usuario del Capítulo B para su testeo en local.
5. Se ejecuta el ciclo de CI para ambos paquetes de la aplicación y se publica cada una de las imágenes, indicando la *tag* que se ha determinado previamente (líneas 51-57). Además, es necesario actualizar los valores de las *tags*

en el repositorio de estado, el cual tiene un campo específico para indicar este dato (líneas 45-49). Más información en la Sección 3.4.

6. Se ejecuta el ciclo de CD, aportando los parámetros necesarios, entre los que se encuentran los archivos que se han construido previamente (líneas 59-69).

Finalmente, la instancia de ArgoCD instalada en el entorno se sincroniza con el repositorio, obtiene los recursos de Kubernetes que se han almacenado en este y despliega la aplicación.

```
1  on:
2    push:
3      branches: [ main ]
4    release:
5      types: [ published ]
6    workflow_dispatch:
7
8  jobs:
9    cicd:
10     runs-on: ubuntu-24.04
11     steps:
12       - % Clona el repositorio "zoo" en la ruta "/zoo"
13
14       - % Clona el repositorio "state" en la ruta "/state"
15         % utilizando el token STATE_REPO
16
17       - name: Install Dagger
18         uses: dagger/dagger-for-github@8.0.0
19
20       - name: Determine environment
21         id: env_tag
22         run: |
23           % Determina el entorno en el que se despliega,
24           % teniendo en cuenta el *trigger* que ha lanzado
25           % el workflow:
26           %   - *push*    -> main
27           %   - *release* -> *release* o *pre-release*
28
29           echo "environment=${envi}" >> "$GITHUB_OUTPUT"
30           echo "tag=${tag}" >> "$GITHUB_OUTPUT"
31
32       - name: Recreate needed files
33         run: |
```

```
          % Recrea el archivo .env para tenerlo disponible
          % en "zoo"
          echo "CR_PAT=${{ secrets.CR_PAT }}" >> ./.env
          % .... se incluyen todas las variables

          % Almacena la clave privada de "age"
          % Crea el archivo de configuracion de SOPS

      - name: Run Dagger CI module
        run: |
          tag=${{ steps.env_tag.outputs.tag }}

          update_state () {
            % Actualiza el valor en "state" de la *tag* de
            % la imagen para que se despliegue la que se
            % acaba de publicar
          }

          dagger call --sec-env=file://.env backend \
              publish-image --tag "${tag}"
          update_state "zoo-backend" "${tag}"

          dagger call --sec-env=file://.env frontend \
              publish-image --tag "${tag}"
          update_state "zoo-frontend" "${tag}"

      - name: Run Dagger CD module
        run: |
          dagger call -m "./dagger/cd" \
            --socket=/var/run/docker.sock \
            --kind-svc=tcp://localhost:3000 \
            --config-file=file://cluster/kind_local.yaml \
            deploy \
            --sec-env=file://.env \
            --env=${{ steps.env_tag.outputs.environment }} \
            --age-key=file://sops/age.agekey \
            --sops-config=file://sops/.sops.yaml
```

Listing 26: *Workflow* de CI/CD.

En la Figura 3.4 se muestra el ciclo de eventos que se realizan previa y posteriormente a la ejecución del *workflow*.

A.4. Desarrollo de la Chart de la aplicación

Referenciado en la Sección 3.3.

El proceso de desarrollo de la Chart comienza con la propia creación de esta. Tras realizar pruebas de construcción de la aplicación, se construye un archivo comprimido con el comando que se muestra en el Listing 27. De esta manera se obtiene un archivo `.tgz` con toda la definición de la Chart de la aplicación. Posteriormente, se almacena el archivo comprimido en un directorio `temp`.

```
1 helm package zoo
```

Listing 27: Generación de un archivo comprimido de la Chart.

Por último, se publica la Chart a través de GitHub Pages. Esto se consigue gracias a un *workflow* que existe en el repositorio, el cual se ve en el Listing 28 (se han comentado las acciones que se realizan en cada paso para reducir su tamaño). El *workflow* se lanza únicamente cuando existe un directorio `temp` con archivos en su interior (líneas 10-11). Este realiza los comandos necesarios para actualizar el archivo `index.yaml`, el cual define las diferentes versiones de la Chart y la URL pública donde se pueden descargar. Además, actualiza las ramas principal y `gh-pages`, siendo esta última en la que se puede encontrar el archivo índice ya mencionado.

```
1  name: Release Helm Charts
2
3  concurrency: release-helm
4
5  on:
6    workflow_dispatch:
7    push:
8      branches:
9        - main
10     paths:
11       - 'temp/**'
12
13 permissions:
14   contents: write
15
16 jobs:
17   release:
18     runs-on: ubuntu-latest
19     steps:
20       - % Clona la rama principal del repositorio en "src/"
```

```
21
22      - % Clona la rama "gh-pages" del repositorio en "dest/"
23
24      - name: Install Helm
25        uses: azure/setup-helm@v4.3.0
26
27      - name: Update New Files and push to main branch
28        shell: bash
29        working-directory: src
30        run: |
31          % Se genera o actualiza el "index.yaml" con la URL
32          % donde se van a alojar las diferentes versione de la
33          % Chart
34
35          % Se guarda el .tgz del directorio "temp/" con la Chart
36          % en la rama de "gh-pages", el "index.yaml" en la raiz
37          % de la misma rama y se suben los cambios la rama
38          % principal
39        env:
40          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
41
42      - name: Push New Files
43        shell: bash
44        working-directory: dest
45        run: |
46          % Se guardan los cambios en la rama "gh-pages",
47          % realizando finalmente la publicacion de la version
48          % de la Chart
49        env:
50          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Listing 28: *Workflow* de publicación de la Chart en GitHub Pages.

Apéndice B

Manuales de usuario

Este Trabajo de Fin de Grado se compone de tres repositorios bien diferenciados y con su propia funcionalidad. Todos ellos se pueden encontrar en esta organización, vieites-tfg.

- `zoo`: Es el repositorio principal del trabajo. En él se encuentra todo el código fuente, desde la aplicación de prueba hasta la implementación de los módulos de Dagger. Es el único necesario para probar las diferentes funcionalidades.
- `helm-repository`: Alberga las Chart de Helm que describen el despliegue de la aplicación de prueba en Kubernetes.
- `state`: En él se almacenan los valores de las Charts de Helm correspondientes a cada uno de los entornos en los que se puede desplegar la aplicación. Funciona como única fuente de verdad. También es el lugar de donde ArgoCD obtiene los recursos de Kubernetes, con el fin de ser desplegados.

B.1. zoo

Estructura general del repositorio

```
argo
|- argo_dev.yaml
|- argo_pre.yaml
|- argo_pro.yaml
|- values.yaml
cluster
|- kind_dev.yaml
|- kind_local.yaml
|- kind_pre.yaml
|- kind_pro.yaml
```

```
dagger
|- cd
|- ci
docker-compose.yml
Dockerfile
example.env
mongo-init
|- init-zoo.js
packages
|- backend
|- frontend
scripts
|- create_envs.sh
|- image.sh
|- push_package.sh
sops
|- .sops.yaml
|- age.agekey
```

Listing 29: Estructura del repositorio `zoo`.

Este repositorio funciona como un *monorepo*, lo cual indica que todo el código fuente se encuentra en este lugar.

Aplicación de prueba

Consta de un *frontend* y un *backend*, ambos creados con Typescript, y utilizando Vue para el *frontend*. Se gestiona utilizando una herramienta de gestión de *monorepos* llamada Lerna. En la raíz del repositorio se pueden encontrar archivos de configuración de Lerna y Node.js. El código correspondiente al *frontend* y al *backend* se incluyen en el directorio `packages`. El *backend* se conecta a una base de datos de animales de un zoo, y proporciona una API REST que el *frontend* consume, con el fin de poder realizar acciones CRUD sobre la base de datos.

Dagger

En el directorio `dagger` se encuentran las implementaciones de los módulos correspondientes a los flujos de CI y CD. Estos se pueden ejecutar localmente teniendo Dagger y Docker instalados. Se utilizan en el *workflow* de GitHub encargado de realizar todo el flujo de testeo, publicación y despliegue.

Otros

También se pueden encontrar varios ejecutables en el directorio de `scripts`, de los cuales `create_envs.sh` es el más interesante. Este permite levantar los

clusters en local para probar el despliegue de la aplicación en los tres entornos posibles: `dev`, `pre` y `pro`.

En todos los *clusters* se instala ArgoCD, aplicación creada específicamente para Kubernetes y que utiliza el método *pull*, siguiendo la filisofía GitOps, leyendo los recursos a desplegar del repositorio de estado (`state`) mencionado anteriormente. Este tiene una rama `deploy`, en la que se suben los archivos necesarios para que ArgoCD lea y despliegue la aplicación.

Se configuran los *clusters* y las propias instancias de ArgoCD de manera diferente para cada uno de los *clusters*. Estas configuraciones se pueden encontrar en los directorios `cluster` y `argo`.

Prueba mínima

Aquí se describen los requisitos de *software* y pasos a seguir para probar de la manera más simple la aplicación. Con esta prueba se comprobará que se pueden obtener los recursos de Kubernetes y levantar la aplicación en los diferentes entornos, pudiendo visualizar todos estos recursos en ArgoCD.

Los elementos que van a influir en esta prueba serán: los *clusters* junto con sus configuraciones y las de ArgoCD, y el repositorio de estado, en el cual ya debería haber recursos preparados para desplegar.

Requisitos de software

A continuación, en la Tabla B.1 se indica el software junto con las versiones utilizadas para el desarrollo del proyecto.

Es necesario tener en cuenta que *no* se ha probado en un sistema operativo Windows, por lo que no se asegura su funcionamiento en este. Sí se ha probado en MacOS y distribuciones Linux.

Software	*Version*
Git	2.48.1
Just	1.39.0
Docker	27.5.1
Kubectl	1.33
Kind	0.27.0
Helm	3.17.3

Tabla B.1: Software y versiones utilizadas durante el desarrollo.

Cómo probar

1. Clonar este repositorio.

   Se clona el repositorio y se accede al directorio.

```
git clone https://github.com/vieites-tfg/zoo ~/zoo
cd ~/zoo
```

Listing 30: Clonado y acceso al repositorio.

2. Clave privada.

```
mkdir -p ./sops
echo "AGE-SECRET-KEY-1CTS4S4QNNZ9N9YXXM288LSE9VKPJ220E57ZHC4558WM
↪  Z8LG2QWKQFFER8C" > ./sops/age.agekey
```

Listing 31: Almacenamiento de la clave privada de encriptado.

3. Ejecutar el *script* de creación de los entornos.

```
./scripts/create_envs.sh
```

Listing 32: *Script* de creación de los entornos.

El *script* anterior:

- Crea tres *clusters* (`dev`, `pre` y `pro`), con tres contextos diferentes (`kind-{{cluster}}`), con su propia configuración.
- Introduce la clave privada, creada previamente, en cada uno de los *clusters*, para permitir a ArgoCD desencriptar los secretos.
- Instala ArgoCD en cada uno de los *clusters*, con sus respectivas configuraciones, obteniendo cada uno los recursos de despliegue del entorno que le toca.

4. Acceso a los *clusters*.

   A medida que se van creando los *clusters*, las contraseñas del usuario `admin` de ArgoCD se van mostrando. También se muestran todas al finalizar la ejecución del *script*.

   Para poder acceder a cada uno de los *clusters*, lo primero que hay que hacer es mapear un puerto local libre al puerto 443 del servidor de ArgoCD. Esto se consigue con el comando que se muestra en el Listing 33:

```
kubectl port-forward svc/argocd-server -n argocd --context
↪  kind-{{cluster}} 8086:443
```

Listing 33: Disponer un puerto en local para acceder a ArgoCD.

En el anterior comando, hay que cambiar {{`cluster`}} por aquel al que se quiera acceder. Se podrá acceder a ArgoCD a través del navegador en `localhost:8086`

Se pide usuario y contraseña para entrar, que son `admin` y la contraseña de dicho *cluster*, mostrada en la salida del *script* que se ha ejecutado antes.

5. Acceder a la aplicación. (opcional)

   Lo primero que hay que hacer es configurar los *hosts* del ordenador para que se resuelvan las rutas como `localhost`. Para ello se puede ejecutar el comando del Listing 34

```
1 just check_hosts dev pre pro
```

Listing 34: Configuración del *host* para acceder a las URLs de la aplicación.

El comando anterior modifica el archivo `/etc/hosts`, incluyendo las líneas necesarias para poder resolver las rutas de acceso a la aplicación. Es necesario tener permisos de usuario o poner la contraseña de este en el caso de que se pida.

Hay que tener en cuenta que cada entorno tiene su propio puerto, que son:

- `dev`: 8080
- `pre`: 8081
- `pro`: 8082

Los anteriores puertos se podrían modificar, pero sería necesario actualizar tanto las configuraciones de los *clusters* en `zoo/cluster` como los valores de los puertos del Ingress en el repositorio de estado `state` para cada uno de los entornos. Esto implicaría tener que hacer un nuevo despliegue con el fin de actualizar los valores en los propios recursos de Kubernetes. No sería necesario publicar nuevas imágenes.

Ahora se puede acceder a la aplicación de gestión del zoo a través de las siguientes rutas en el navegador:

- *frontend*: `zoo-{{entorno}}.example.com:{{puerto_entorno}}`
- *backend*: `api-zoo-{{entorno}}.example.com:{{puerto_entorno}}`

Conclusión

Se comprueba que funciona el despliegue de la aplicación para cualquiera de los entornos. Esta configuración permite al desarrollador tener a su disposición cada una de las versiones y comprobar que el despliegue se realiza correctamente. El uso de ArgoCD y la capacidad de obtener los recursos de una única fuente

Software	*Version*
Dagger	latest
act	0.2.79

de verdad hace de este flujo de despliegue algo esencial en cualquier equipo de desarrollo.

Es necesario mencionar que, en entornos de producción reales, los *clusters* se encontrarían en la nube. Sin embargo, se podría mantener el *cluster* de desarrollo `dev`, con el fin de realizar implementaciones y probar su funcionamiento sin depender de herramientas remotas.

A continuación se indica cómo probar los módulos de Dagger.

Prueba de Dagger

En este caso se va mostrar cómo probar tanto los módulos de Dagger como el *workflow* completo de CI/CD, desde la simulación de implementación de una nueva característica en la aplicación, pasando por su despliegue en todos los entornos.

Requisitos de software

Incluyendo los de la prueba anterior.

Para la simulación de una nueva *feature* se utilizar *act*, herramienta que permite ejecutar *workflows* de GitHub de manera local. Con ella se simulará el disparo de los eventos que hacen que el *workflow* se ejecute, y se comportará de la misma manera que haría en remoto.

Cómo probar

1. Creación de archivos `.env` y `.secrets.yaml`

   Es necesario modificar los archivos `example.env` y `example.secrets.yaml` con los datos pertinentes. Los archivos quedarían como se muestra en los Listings 35 y 36.

   Se puede copiar el contenido anterior y pegarlo en su respectivo archivo. Posteriormente, hay que cambiar el nombre de estos, eliminando la parte de `example`.

2. Prueba local del módulo de CI.

   El módulo de CI de Dagger se divide en funciones para *frontend* y para *backend*, por separado. Las funciones son las mismas para ambos, pero internamente se comportan diferente.

```
# Se ha dividido en dos STATE_REPO, pero es un solo string
MONGO_DATABASE=zoo
MONGO_ROOT=carer
MONGO_ROOT_PASS=carerpass
CR_PAT=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
STATE_REPO=github_pat_11AOOYJPI0zETh9qTHsxYE_MbGvGQbWlZ52j5AhCwqV
↪ 4ofE0zkGlwVrXdOM4KFUt3e52GWN3ZD7t0QEG8q
```

Listing 35: Archivo de secretos `.env`.

```
# Se han dividido en dos STATE_REPO, age y SOPS_PRIVATE_KEY,
# pero son un solo string
MONGO_DATABASE: zoo
MONGO_ROOT: carer
MONGO_ROOT_PASS: carerpass
CR_PAT: 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
STATE_REPO: github_pat_11AOOYJPI0zETh9qTHsxYE_MbGvGQbWlZ52j5AhCwq
↪ V4ofE0zkGlwVrXdOM4KFUt3e52GWN3ZD7t0QEG8q
SOPS_CONFIG_FILE: |
  creation_rules:
      - path_regex: ".*\\.ya?ml$"
        unencrypted_regex: "^(apiVersion|metadata|kind|type)$"
        age: age15peyc7pedj8gjqwnarat6s3u87wy4j5xtf7t96vuj74m3l9x
        ↪ q5ys0r4sag
SOPS_PRIVATE_KEY: AGE-SECRET-KEY-1CTS4S4QNNZ9N9YXXM288LSE9VKPJ220
↪ E57ZHC4558WMZ8LG2QWKQFFER8C
```

Listing 36: Archivo de secretos `.secrets.yaml`, para pruebas con `act`.

```
mv example.env .env
mv example.secrets.yaml .secrets.yaml
```

Listing 37: Cambio de nombre de archivos ocultos.

En la Figura B.1 se muestra un diagrama de la implementación de este módulo.

En el diagrama se ve:

- `CI`: La estructura principal, con cinco funciones, dos de las cuales permiten acceder a las estructuras de *frontend* y *backend*, por separado.
- `Frontend` y `Backend`: Estructuras dedicadas, con implementación diferente para cada una de las funciones que proporcionan, que en este

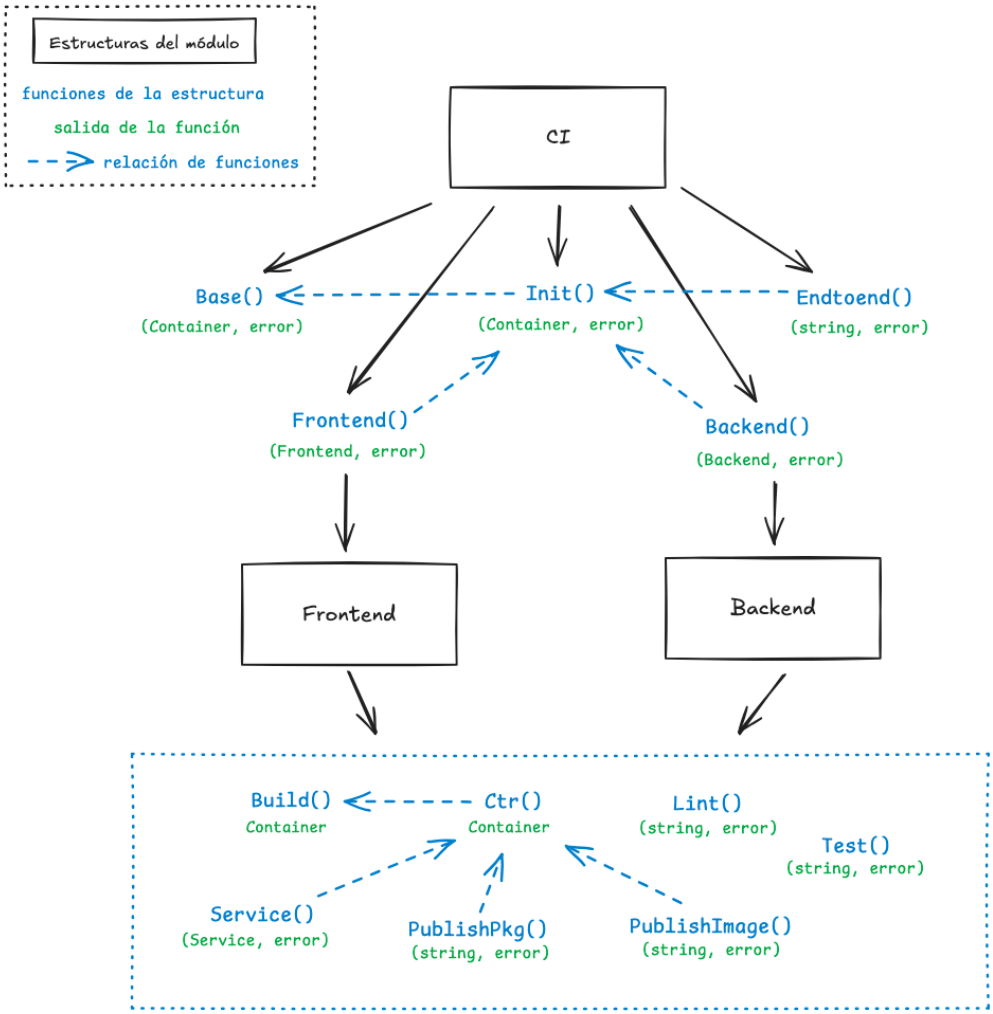

Figura B.1: Estructura del módulo de CI.

caso son las mismas para ambos.

Para poder utilizar el comando `dagger`, es necesario estar en un directorio de trabajo en el que exista un módulo de Dagger, o bien proporcionarlo con la opción `-m`. Para facilitar la explicación, se ejecutarán los comandos desde el directorio correspondiente al módulo de CI.

```
cd dagger/ci
```

Listing 38: Acceder al módulo de Dagger de CI.

Se pueden obtener las funciones de `CI`, junto con los campos de la estructura, con el comando del Listing 39:

```
dagger functions
```

Listing 39: Comprobación de las funciones disponibles con Dagger.

Para conocer las funciones y campos de las demás estructuras se ejecuta el comando del Listing 40:

```
dagger call backend --help
dagger call frontend --help
```

Listing 40: Obtener las funciones de cada uno de los objetos customizados del módulo de Dagger de CI.

`CI` tiene un parámetro requerido, que se trata del archivo `.env` que se ha creado anteriormente. En el Listing 41 se muestra cómo se ejecutarían los tests *end-to-end* de la aplicación, teniendo en cuenta que hay que encontrarse en el directorio del módulo y que el archivo `.env` se ha creado correctamente en la raíz del repositorio:

```
dagger call --sec-env=file://../../.env endtoend
```

Listing 41: Ejecutar la función de prueba íntegra de los paquetes de la aplicación con Dagger.

Otro ejemplo sería, levantar el *frontend* y el *backend* y hacer que se comuniquen de manera local.

Simplemente, ejecutando los comandos anteriores en terminales diferentes, los servicios serán capaces de comunicarse. Estos estarán disponibles en `localhost:{{puerto}}`. Para acceder a la API se añade la ruta `/animals`.

```
dagger call --sec-env=file://../../.env backend service up
 ↪  --ports 3010:3000
dagger call --sec-env=file://../../.env frontend service up
 ↪  --ports 8090:80
```

Listing 42: Levantamiento de los servicios de los paquetes de la aplicación con Dagger.

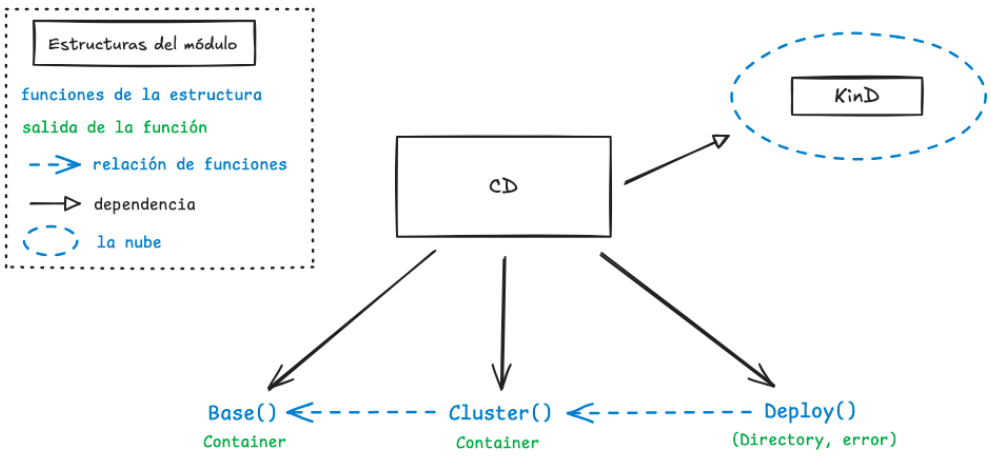

Figura B.2: Estructura del módulo de CD.

En los comandos anteriores, `up` es una función del tipo `Service`, propio de Dagger, que se devuelve en la función `service`, como se muestra en el diagrama anterior. `--ports` es un argumento de la función `up`.

```
dagger call --sec-env=file://../../.env [backend|frontend] lint
dagger call --sec-env=file://../../.env [backend|frontend] test
dagger call --sec-env=file://../../.env [backend|frontend]
 ↪  publish-image --tag "{{tag}}"
dagger call --sec-env=file://../../.env [backend|frontend]
 ↪  publish-pkg
```

Listing 43: Otras posibles funciones a ejecutar del módulo de CI con Dagger.

3. Prueba local del módulo de CD.

   El módulo de CD tiene la estructura que se muestra en la Figura B.2.

   La función principal de este módulo es `deploy`, que se encarga de:

   - Construir todos los recursos de Kubernetes, haciendo uso de los repositorios `helm-repository` y `state`.

- Encriptar los secretos.
- Subir los cambios al repositorio de estado.

Para conseguir esto, se hace uso de este módulo, que permite crear un *cluster* de KinD.

No se recomienda la ejecución en local de este módulo, pero se explica igual cómo se haría.

Para realizar un despliegue habría que ingresar al directorio del módulo y ejecutar el comando que se muestra en el Listing 44:

```
cd ~/zoo/dagger/cd

just deploy dev
# just deploy pre
# just deploy pro

# lo anterior ejecuta:
dagger call \
    --socket=/var/run/docker.sock \
    --kind-svc=tcp://localhost:3000 \
    --config-file=file://../../cluster/kind_local.yaml \
    launch \
    --sec-env=file://../../.env \
    --env={{env}} # <- cambiar {{env}} por "dev", "pre" o
    ↪  "pro" \
    --age-key=file://../../sops/age.agekey \
    --sops-config=file://../../sops/.sops.yaml
```

Listing 44: Despliegue con el módulo de Dagger de CD.

4. Prueba con `act`.

   `act` es una herramienta que permite ejecutar *workflows* de GitHub en local, pudiendo indicar el *trigger* que dispara el *workflow*.

   De esta manera, se puede probar cómo sería el flujo de ejecución en el caso de que se introdujera en la rama principal una nueva característica de la aplicación.

   El *workflow* encargado de realizar las operaciones de CI y CD es el que se encuentra en `.github/workflows/cicd.yaml`.

   La definición de los objetos del *trigger* `release` se encuentran en el directorio `.github/workflows/events`.

   Una vez se tenga `act` instalado, se puede iniciar la *promoción de entornos*.

Antes de nada, para comprobar que se ha realizado el cambio, se realizarán una serie de pasos para asegurarse de que las *tags* de la imagen de `dev` es nueva. El comando del Listing 45 creará una rama nueva con el nombre `environments-test`.

```
git switch -c environments-test
```

Listing 45: Creación de una nueva rama con Git.

Ahora se puede cambiar el título de la página web. Este se encuentra en el archivo `packages/frontend/src/components/MainTitle.vue`, de "Zoo" a algo como "Mi zoo".

Se debe crear un *commit* para que la imagen use como *tag* los ocho primeros caracteres de este.

```
git add .
git commit -m 'cambio de nombre'
```

Listing 46: Creación del *commit* del cambio realizado en el código.

Así se utilizará el *commit* que se acaba de crear para generar el nombre de la *tag* de las imágenes.

dev

Ahora se puede subir todo al entorno de "dev", simulando un `push` (*trigger* por defecto de `act`) a la rama principal, como si el cambio anterior se hubiera realizado en una PR y si hubieran aprobado los cambios. Esto se puede hacer, desde la raíz del repositorio, con el comando del Listing 47.

```
# se utiliza el archivo .secrets.yaml creado anteriormente
act --secret-file .secrets.yaml
```

Listing 47: Simulación de *push* a la rama principal con `act`.

Una vez terminada la ejecución, si se tiene levantado el *cluster* de `dev`, se debería poder sincronizar ArgoCD. Para comprobar que se ha actualizado la imagen correctamente, se puede visualizar la *tag* que está en uso en el Deployment del *frontend* `zoo-dev-frontend`, y buscar el campo `spec.template.spec.containers.image`, en la pestaña de "*Live manifest*". Este campo debería contener la nueva imagen generada.

También se puede entrar en la propia web, en `zoo-dev.example.com:8080`, para ver el nuevo título.

pre

Para pasar los cambios que se han realizado al entorno de `pre`, sería necesario crear una `pre-release`. Se va a simular su creación utilizando la configuración de dicho evento, que se encuentra en `.github/events/prerelease.json`.

Lo mejor para comprobar que se actualiza correctamente, es cambiar el nombre de la `pre-release`, en el campo `name` del archivo del evento. Por ejemplo, `0.0.123-snapshot`.

Con el comando del Listing 48 se realizará la simulación.

```
act release --eventpath .github/events/prerelease.json
 ↪  --secret-file .secrets.yaml
```

Listing 48: Simulación de creación de una *prerelease* con `act`.

El archivo de comprobación del recurso es, en este caso, `zoo-pre-frontend`, y el mismo campo que antes.

La URL sería `zoo-dev.example.com:8081`.

pro

Ahora, es necesario realizar algo parecido al paso anterior. Se modifica el archivo `.github/events/release.json`, y se pone el mismo `name` que antes, pero sin la coletilla `snapshot`, es decir, `0.0.123`.

Se simula el cambio al entorno de `pro` con el comando del Listing 49.

```
act release --eventpath .github/events/release.json --secret-file
 ↪  .secrets.yaml
```

Listing 49: Simulación de creación de una *release* con `act`.

Conclusión

Se puede comprobar que el ciclo completo funciona correctamente, y la flexibilidad que da poder ejecutar los módulos de Dagger en local. Esto facilita el desarrollo de características nuevas para la aplicación sin miedo a que se produzca un fallo durante la ejecución del *workflow* de GitHub en la nube. Así, el resultado que va a dar la ejecución de dicho *workflow* ya se sabe de antemano, porque se ha probado de manera local, en un entorno controlado gracias a Docker. Por lo tanto, se sabe que este se va a comportar de la misma manera en cualquier otro sistema.

Prueba la API

La API está disponible en `https://api-{{env}}-zoo.example.com/animals`. Tiene definidos los *endpoints* de la Tabla B.2.

Acción	*endpoint*	*Funcionalidad*
`GET`	`/animals`	Obtener todos los animales
`GET`	`/animals/{id}`	Obtener un animal mediante su ID
`POST`	`/animals`	Añadir un nuevo animal
`PUT`	`/animals/{id}`	Actualizar el animal con cierto ID

Tabla B.2: Endpoints de la API.

En los siguientes Listings se muestran peticiones de ejemplo que se pueden utilizando, por ejemplo, Postman.

```
// GET http://localhost:3000/animals
```

Listing 50: API REST, obtener la lista de animales.

```
// GET http://localhost:3000/animals/{{ID_FROM_LAST_LIST}k
```

Listing 51: API REST, obtener animal.

```
// POST http://localhost:3000/animals/
{
    name: "Marcus",
    species: "Tiger",
    birthday: "2010-05-16",
    genre: "male",
    diet: "Carnivore",
    condition: "Healthy"
}
```

Listing 52: API REST, crear un animal.

B.1.1. helm-repository

En este reposito se albergan las Charts Helm correspondientes a la aplicación que se implementa en el repositorio principal `zoo`.

```
// PUT http://localhost:3000/animals/{{ID_FROM_LAST_POST}}
{
    condition: "Injured",
    notes: "Recovering from minor foot injury."
}
```

Listing 53: API REST, actualizar un animal.

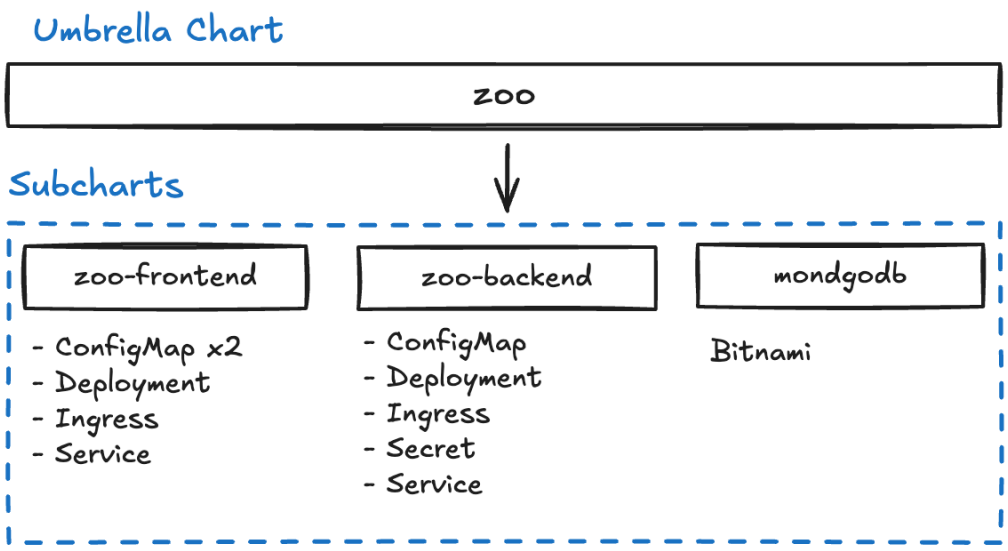

Figura B.3: Estructura del módulo de CD.

En la Figura B.3 se muestra un diagrama de la disposición de las Charts.

Como se puede comprobar, existe una *Chart Umbrella* `zoo`, la cual se encarga de indicar las subcharts a crear. Entre estas hay dos creadas específicamente para la aplicación, `zoo-backend` y `zoo-frontend`, perfectamente autodescritas. Por último está la Chart de MongoDB, obtenida del repositorio de Bitnami.

Estructura del repositorio

```
helm
|- zoo-0.0.0.tgz
|- zoo-0.0.1.tgz
|- zoo-0.0.2.tgz
|- zoo-0.0.3.tgz
zoo
|- Chart.yaml
|- charts
|   |- zoo-backend
```

```
|   |   |- Chart.yaml
|   |   |- templates
|   |       |- configMap.yaml
|   |       |- deployment.yaml
|   |       |- ingress.yaml
|   |       |- secret.yaml
|   |       |- service.yaml
|   |
|   |- zoo-frontend
|       |- Chart.yaml
|       |- templates
|           |- configmap-js.yaml
|           |- configmap.yaml
|           |- deployment.yaml
|           |- ingress.yaml
|           |- service.yaml
|
|- templates
    |- _helpers.tpl
    |- ghcr-secret.yaml
```

Listing 54: Estructura del repositorio `helm-repository`.

B.1.2. state

En este repositorio se almacenan los valores necesarios para las Charts de Helm correspondientes a la aplicación, que se encuentran en helm-repository.

Estructura del repositorio

Se ven los siguientes archivos:

- `global`: En estos se indican valores globales a toda la aplicación.
- `zoo-backend`, `zoo-frontend` y `mongodb`: Archivos de valores de cada una de las Subcharts.
- `helmfile.yaml.gotmpl`: Archivo en el que se indican los repositorios a utilizar y los valores a incluir en las Charts. Se utiliza con la herramienta helmfile para generar la plantilla de todos los recursos que se van a construir.

Existen directorios para cada uno de los posibles entornos, con valores específicos de dicho entorno para cada una de las Subcharts o valores globales.

```
dev
|- global.yaml
|- mongodb.yaml
|- zoo-backend.yaml
|- zoo-frontend.yaml
pre
|- global.yaml
|- mongodb.yaml
|- zoo-backend.yaml
|- zoo-frontend.yaml
pro
|- global.yaml
|- mongodb.yaml
|- zoo-backend.yaml
|- zoo-frontend.yaml
dev.yaml
global.yaml
helmfile.yaml.gotmpl
mongodb.yaml
pre.yaml
pro.yaml
zoo-backend.yaml
zoo-frontend.yaml
```

Listing 55: Estructura del repositorio `state`.

Rama de despliegue

La rama de despliegue `deploy` es la rama en la que se publican los recursos que ArgoCD va a leer para construir la aplicación.

Estructura

```
dev
|- kustomization.yaml
|- non-secrets.yaml
|- secret_generator.yaml
|- secrets.yaml
pre
|- kustomization.yaml
|- non-secrets.yaml
|- secret_generator.yaml
```

```
10 |- secrets.yaml
11 pro
12 |- kustomization.yaml
13 |- non-secrets.yaml
14 |- secret_generator.yaml
15 |- secrets.yaml
```

Listing 56: Estructura de la rama `deploy` en `state`.

Se puede comprobar que hay un directorio con los recursos correspondientes de cada uno de los entornos posibles.

En ellos se encuentran los seguientes archivos:

- `non-secrets`: Los recursos de Kubernetes que no son secretos.
- `secrets`: Los recursos de Kubernetes que son secretos, encriptados utilizando SOPS y age.
- `kustomization`: Archivo que lee la herramienta kustomize e indica los recursos y el generador a utilizar para crear los secretos desencriptados.
- `secret_generator`: Indica la necesidad del uso de la herramienta ksops con el fin de desencriptar los secretos.

Bibliografía

[1] Dagger.io. “Dagger Documentation — Dagger.” Dagger.io, 2022, *https://docs.dagger.io*

[2] Dagger.io. “Dagger — Blog.” Dagger.io, 2025, *https://dagger.io/blog/*. Accedido el 15 de junio de 2025

[3] Fowler, Martin. “Continuous Integration.” Martinfowler.com, 18 Jan. 2024, *https://martinfowler.com/articles/continuousIntegration.html*.

[4] PagerDuty, Inc. “What Is Continuous Integration?” PagerDuty, 20 Nov. 2020, *https://www.pagerduty.com/resources/devops/learn/what-is-continuous-integration*.

[5] Amazon Web Services, Inc. “¿Qué Es La Entrega Continua? – Amazon Web Services.” Amazon Web Services, Inc., 2024, *https://aws.amazon.com/es/devops/continuous-delivery*.

[6] Fowler, Martin. “Bliki: ContinuousDelivery.” Martinfowler.com, 2013, *https://martinfowler.com/bliki/ContinuousDelivery.html*.

[7] Nx. “Monorepo Explained.” Monorepo.tools, 2025, *https://monorepo.tools*

[8] Los autores de Kubernetes. “Orquestación de Contenedores Para Producción.” Kubernetes, 2025, *https://kubernetes.io/es*.

[9] Helm. “Helm.” Helm.sh, 2019, *https://helm.sh*.

[10] Wikipedia Contributors. “DevOps.” Wikipedia, Wikimedia, 1 Dec. 2019, *https://en.wikipedia.org/wiki/DevOps*.

[11] Nasser, Mohammed. “Push vs. Pull-Based Deployments.” DEV Community, 25 Nov. 2024, *https://dev.to/mohamednasser018/push-vs-pull-based-deployments-4m78*. Accedido el 14 de junio de 2025.

[12] Git. “Git.” Git-Scm.com, 2024, *https://git-scm.com*.

[13] Wikipedia Contributors. “Make (Software).” Wikipedia, 10 July 2021, en *https://en.wikipedia.org/wiki/Make_(software)*.

[14] casey. "GitHub - Casey/Just: Just a Command Runner." GitHub, 2025, *https://github.com/casey/just*. Accedido el 14 de junio de 2025.

[15] Wikipedia Contributors. "Shebang (Unix)." Wikipedia, 13 Aug. 2021, *https://en.wikipedia.org/wiki/Shebang_(Unix)*.

[16] Wikipedia Contributors. "Docker (Software)." Wikipedia, 16 Nov. 2019, *https://en.wikipedia.org/wiki/Docker_(software)*.

[17] "Overview of Docker Compose." Docker Documentation, 10 Feb. 2020, *https://docs.docker.com/compose*.

[18] Kong. "What Is Kubernetes? Examples and Use Cases." Kong Inc., 2024, *https://konghq.com/blog/learning-center/what-is-kubernetes*.

[19] The Kubernetes Authors. "Cluster Architecture." Kubernetes, 2025, *https://kubernetes.io/docs/concepts/architecture*.

[20] The Kubernetes Authors. "Kind." Kind.sigs.k8s.io, 2025, *https://kind.sigs.k8s.io*.

[21] The Kubernetes Authors. "Command Line Tool (Kubectl)." Kubernetes, 2025, *https://kubernetes.io/docs/reference/kubectl*.

[22] GitLab. "¿Qué Es GitOps?" Gitlab.com, GitLab, 9 Feb. 2022, *https://about.gitlab.com/es/topics/gitops*. Accedido el 20 de junio de 2025.

[23] Argo Porject Authors. "Argo CD." Github.io, 2025, *https://argoproj.github.io/cd*. Accedido el 20 de junio de 2025.

[24] Flexera. "Kubernetes Helm: K8s Application Deployment Made Simple." Spot.io, 12 Sept. 2024, *https://spot.io/resources/kubernetes-architecture/kubernetes-helm-k8s-application-deployment-made-simple*. Accedido el 20 de junio de 2025.

[25] Dagger. "Introducing Dagger: A New Way to Create CI/CD Pipelines — Dagger." Dagger.io, 2022, *https://dagger.io/blog/public-launch-announcement*. Accedido el 21 de junio de 2025.

[26] Wikipedia Contributors. "Open Container Initiative." Wikipedia, Wikimedia Foundation, 12 Nov. 2024, *https://en.wikipedia.org/wiki/Open_Container_Initiative*.

[27] CUE. "The CUE Language Specification." CUE, 16 Apr. 2025, *https://cuelang.org/docs/reference/spec*. Accedido el 21 de junio de 2025.

[28] Wikipedia Contributors. "Conjunto de Herramientas de Desarrollo de Software." Wikipedia, 15 Aug. 2006, *https://es.wikipedia.org/wiki/Kit_de_desarrollo_de_software*. Accedido el 21 de junio de 2025.

[29] GitHub. "Features · GitHub Actions." GitHub, 2025, *https://github.com/features/actions*.

[30] "The Go Programming Language." Go.dev, 2025, *https://go.dev*. Accedido el 21 junio de 2025.

[31] The GraphQL Foundation. "GraphQL: A Query Language for APIs." Graphql.org, 2012, *https://graphql.org*.

[32] Dagger. "Introducing the Dagger GraphQL API — Dagger." Dagger.io, 2022, *https://dagger.io/blog/graphql*. Accedido el 21 de junio de 2025.

[33] Dagger. "Daggerverse." Daggerverse.dev, 2025, *https://daggerverse.dev*. Accedido el 21 de junio de 2025.

[34] Dagger. "ContentKeeper Content Filtering." Dagger.io, 2025, *https://docs.dagger.io/reference/cli*. Accedido el 21 de junio de 2025

[35] Microsoft. "TypeScript - JavaScript That Scales." Typescriptlang.org, 2025, *https://www.typescriptlang.org*.

[36] Nx. "Lerna · a Tool for Managing JavaScript Projects with Multiple Packages." Lerna.js.org, 2025, *https://lerna.js.org*.

[37] MongoDB. "MongoDB." MongoDB, 2024, *https://www.mongodb.com*.

[38] You, Evan. "Vue.js." Vuejs.org, 2014, *https://vuejs.org*.

[39] LinearB, Inc. "Cycle Time Breakdown: Tactics for Reducing PR Review Time — LinearB Blog." Linearb.io, 4 Aug. 2021, *https://linearb.io/blog/reducing-pr-review-time*. Accedido el 10 de julio de 2025.

[40] Valsorda, Filippo. "FiloSottile/Age." GitHub, 21 Apr. 2022, *https://github.com/FiloSottile/age*.

[41] getsops. "Getsops/Sops." GitHub, 9 Jan. 2024, *https://github.com/getsops/sops*.

[42] Kubernetes SIGs. "Kustomize - Kubernetes Native Configuration Management." Kustomize.io, 2025, *https://kustomize.io*.

[43] viaduct-ai. "GitHub - Viaduct-Ai/Kustomize-Sops: KSOPS - a Flexible Kustomize Plugin for SOPS Encrypted Resources." GitHub, 28 Jan. 2025, *https://github.com/viaduct-ai/kustomize-sops*. Accedido el 11 de julio de 2025.

[44] Preston-Werner, Tom. "Versionado Semántico 2.0.0." Semantic Versioning, 2024, *https://semver.org/lang/es*.

[45] Broadcom. "Applications - Bitnami.com." Bitnami.com, 2024, *https://bitnami.com/stacks?stack=helm*. Accedido el 12 de julio de 2025.

[46] Helmfile Authors. "Helmfile." Readthedocs.io, 2025, *https://helmfile.readthedocs.io/en/latest*. Accedido el 12 de julio de 2025.

[47] Cypress. "JavaScript End to End Testing Framework." JavaScript End to End Testing Framework — Cypress.io, 2025, *https://www.cypress.io*.

[48] Dagger. "Custom Types — Dagger." Dagger.io, 2022, *https://docs.dagger.io/api/custom-types*. Accedido el 13 de julio de 2025.

[49] Datadog. "Modern Monitoring & Analytics — Datadog." Modern Monitoring & Analytics, 2019, *https://www.datadoghq.com*.

[50] Prometheus. "Prometheus - Monitoring System & Time Series Database." Prometheus.io, 2025, *https://prometheus.io*.

[51] León, Manuel. "Escalabilidad Horizontal Y Vertical: Modelos de Escalado — Arsys." Blog de Arsys.es, 3 July 2019, *https://www.arsys.es/blog/escalado-horizontal-vs-vertical*.

[52] aquasecurity. "Trivy." Trivy.dev, 2024, *https://trivy.dev/latest*.