Gestión y desarrollo de servicios distribuidos en un monorepo

Curso académico 2020/2021
Trabajo Fin de Máster

Autor: Javier García González
Tutor: Francisco Gortázar Bellas

Resumen 3

Objetivos 4

Introducción 5

Descripción de la aplicación 6
- Servicios 8
  - Server 8
  - Users 9
- Librerías 9
  - Eslint-config 10
  - Prettier-config 10

Infraestructura 11
- Gestión de proyectos monorepo con Lerna 11
- Gestión de releases 12
  - Publicación de una nueva versión de una librería 12
- Publicación de una nueva imagen Docker 12
- CLI (Command Line Interface) 13
  - bootstrap 13
  - build 13
  - lint 13
  - lint-fix 14
  - publish 14
  - test 15
  - create-env 15
  - up 15

Desarrollo en local 16
- El papel de Lerna 17

Integración continua (CI) 18
- Workflow para integrar las librerías 18
  - Ejemplo de commit que no contiene cambios que afectan a ninguna librería 18
- Workflows para integrar los servicios 19
  - Ejemplo de la integración de un servicio durante la ejecución del workflow del servicio Server 20

Entrega continua (CDE) **22**
Workflow para entregar (publicar) librerías 22
Ejemplo de commit que contiene cambios que afectan a alguna librería 22
Workflows para entregar (publicar) servicios 25
Ejemplo de un commit que contiene cambios en el servicio Server 26
Despliegue continuo (CD) **29**
Ejemplo de un commit que contiene cambios en el servicio Server 29
Conclusiones **31**
Bibliografía **33**

Resumen

En este trabajo se describe la gestión de un proyecto de ejemplo en el que varios servicios y librerías, escritos en Typescript y ejecutados sobre NodeJS, son desplegados en servicios en la nube de forma automatizada. La estructura del monorepo ayuda a resolver algunos retos que se presentan a la hora de desarrollar una aplicación distribuída.

Para la gestión de paquetes (servicios y librerías) dentro del monorepo se ha utilizado Lerna. Una herramienta que ayuda a mantener el versionado entre los distintos paquetes (módulos Javascript registrados en NPM - Node Package Manager) actualizado durante el desarrollo y gestiona la publicación de los paquetes y sus dependencias dentro del proyecto.

Para incrementar la calidad del código, este es probado con tests unitarios que se ejecutan sobre el framework Jest.

Los servicios se distribuyen de forma automatizada a un proveedor de servicios en la nube, en este caso: Okteto. Para ello, los servicios se despliegan en contenedores Docker, manejados ("orquestrados") en un nivel más alto por Kubernetes. En local, se hace uso de Docker Compose para levantar todos los servicios necesarios para el desarrollo.

La forma en que se ha conseguido automatizar el proceso desde que un cambio se produce hasta que se entrega y despliega ha sido Github Actions, ya que el repositorio reside en la plataforma Github. El código se ha ido añadiendo de forma progresiva e incremental siguiendo el modelo de desarrollo TBD (Trunk Based Development).

Por último, cabe destacar que las operaciones de construcción y pruebas de los servicios y librerías son iniciados mediante el uso de un fichero "Makefile" que contiene todos los comandos necesarios para construir, probar, levantar el entorno local o realizar las labores necesarias al comenzar con el desarrollo del proyecto desde cero, entre otras.

Objetivos

Este trabajo se enfoca en la gestión de un monorepo que incluye varios servicios distribuídos y librerías que también se desarrollan dentro de este proyecto con una alta dependencia entre ellos.

El proyecto cubre los siguientes objetivos:

- **Investigar experiencia de desarrollo**: gestión de las dependencias internas entre los diferentes servicios y paquetes de la aplicación. Para ello, se ha utilizado una herramienta llamada Lerna, que tiene entre sus distintas aplicaciones la capacidad de enlazar los paquetes que dependen unos de otros entre sí para mejorar la experiencia de desarrollo.
- **Integración continua**: determinar una estrategia para asegurar que los cambios que se producen en la rama de producción no corrompan el estado del proyecto. Determinar qué estrategia se adapta mejor al proyecto monorepo donde se comparten el lenguaje y las tecnologías de mantenimiento de calidad de software y pruebas.
- **Entrega continua**: generar versiones de cada paquete (servicios y librerías) de forma independiente cuando se produce un cambio en alguno de ellos.
- **Despliegue continuo**: cada cambio en la rama de producción debe verse reflejado en un entorno de producción accesible por el usuario final. El despliegue debe llevarse a cabo sobre los servicios a los que les afectan los cambios que han sido añadidos al código fuente del proyecto.

Introducción

El término "monorepo" ha ganado mucha popularidad en los últimos años en la comunidad del desarrollo de software, especialmente en proyectos con servicios distribuidos. Un monorepo es un patrón arquitectónico que especifica que todo el código de un proyecto resida en un único repositorio, independientemente del número de servicios que contenga. En un monorepo pueden convivir servicios que usan diferentes estrategias de desarrollo, tecnologías e incluso lenguajes de programación.

Cada servicio se desarrolla, se mantiene, se integra y se despliega de forma independiente. Esto permite la automatización de muchos de estos procesos, permitiendo que los cambios específicos de un servicio sólo afecten a este, por lo que no es necesario volver a realizar acciones sobre el conjunto de todos los servicios que forman el proyecto. Este es uno de los retos más importantes a la hora de trabajar con un único repositorio de código. En este proyecto se han utilizado varias tecnologías para conseguir este objetivo, desde herramientas para gestionar módulos escritos en Javascript y su publicación en un registro, así como herramientas para gestionar el proceso de integración, entrega y despliegue continuo.

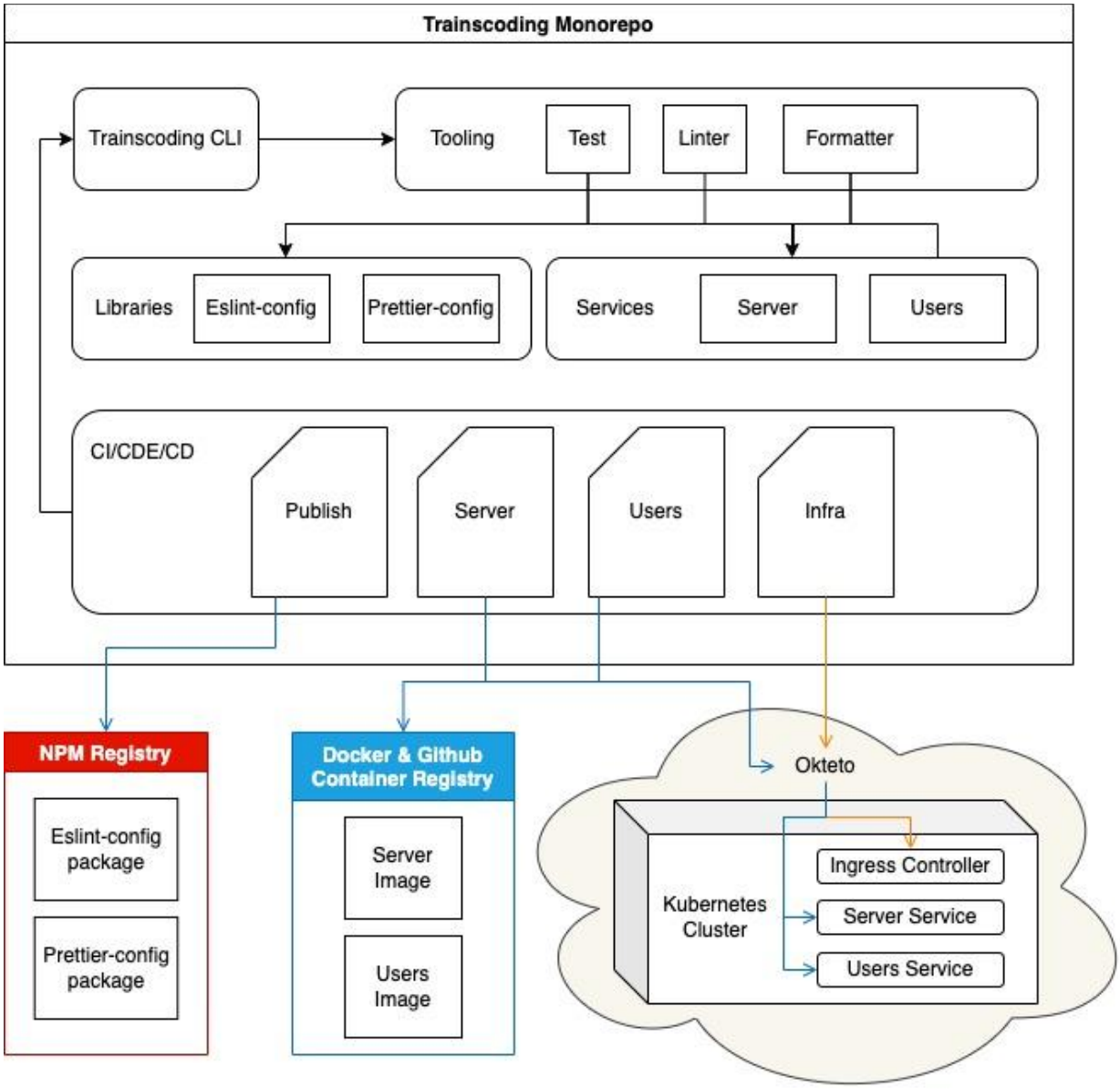
Además, a medida que un sistema va aumentando en complejidad y tamaño, incrementa la dificultad de mantener una visión de conjunto. A menudo se vuelve más complejo tener una idea clara de qué servicios componen un sistema en concreto, lo que dificulta la labor de desarrollo y mantenimiento. Mantener todo el código en un único repositorio permite que se puedan seguir las mismas estrategias para cada servicio a la hora de realizar controles de calidad (pruebas) o ejecutar análisis (comprobaciones de estilo). Es más sencillo tener una estratégia única desarrollada en un monorepo que reproducirla en proyectos donde los servicios se alojan en diferentes repositorios.

Otro de los retos es la interacción entre los servicios del proyecto y el versionado de los mismos. Cuando uno de los servicios se actualiza, el resto de servicios que dependen de él tienen que actualizarse también para que puedan empezar a utilizar la nueva versión del código que ha sido publicada.

Existen diferentes ventajas y desventajas a la hora de decidirse por este tipo de arquitectura en un proyecto. A lo largo de este documento se detalla cada una de estas.

Descripción de la aplicación

Con el objetivo de explicar la gestión de un proyecto monorepo se ha desarrollado una aplicación de ejemplo llamada "Trainscoding". Esta aplicación consiste en varios servicios y librerías desarrolladas en Typescript[1] que se despliegan de forma distribuída en un proveedor de servicios en la nube, en este caso, Okteto[2].



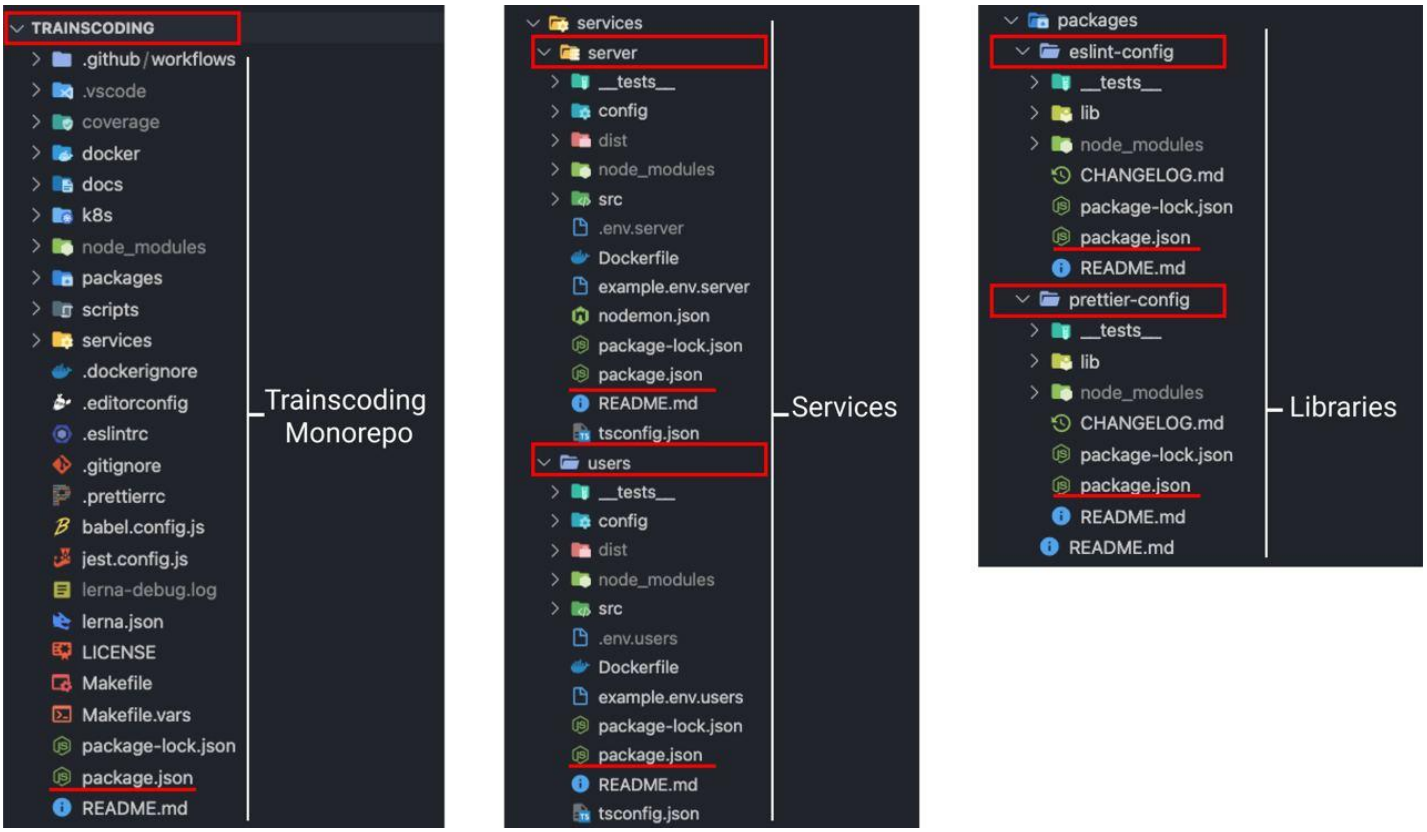
En la imagen anterior se exponen a los diferentes integrantes de la arquitectura en el contexto del monorepo. Dentro del proyecto se encuentra la interfaz de línea de

[1] https://www.typescriptlang.org/
[2] https://okteto.com/

comandos que tiene acceso a las operaciones disponibles para todos los paquetes independientes. La raíz del proyecto contiene las herramientas necesarias para probar, comprobar la calidad del código e incluso publicar los artefactos en sus diferentes versiones.

En la imagen se aprecian los diferentes agentes del proyecto están contenidos en un único repositorio pero todos están aislados entre sí. Todo el código base está en un único repositorio donde se encuentra tanto la lógica de negocio como la infraestructura.

Como se ha mencionado en la introducción, los servicios no han sido implementados en su totalidad, por lo que la aplicación no es completamente funcional. Trainscoding quiere emular una aplicación en la que distintos usuarios pueden codificar vídeos de un formato a otro con facilidad. Debido al enfoque centrado en la gestión de un proyecto monorepo no se ha completado la funcionalidad pretendida en esta aplicación de ejemplo. Para este proyecto se ha querido hacer énfasis en la infraestructura y la arquitectura de los servicios desde el punto de vista de la gestión en un monorepo, por lo que la complejidad de estos servicios no es alta, pero sirve para poner en práctica un ejemplo real.



En la imagen anterior se expone una captura de pantalla del editor de texto desde el cual se ha desarrollado la aplicación. En la primera figura que aparece en el

lado izquierdo de la imagen se aprecia una vista a primer nivel de profundidad donde se ven los distintos elementos que componen el proyecto. Cabe destacar que el proyecto principal tiene un archivo "package.json" en la raíz del directorio. Esto significa que Trainscoding también es un paquete que podría ser publicado en registro de módulos Javascript. En este caso, no existe un caso de uso por el que el paquete que contiene el monorepo deba ser publicado. Además, este archivo "package.json" permite definir las dependencias intrínsecas a todo el proyecto en su conjunto, por ejemplo, herramientas como Lerna (para la gestión del monorepo) o Jest (el framework utilizado para ejecutar los tests) y también contiene los comandos que componen la aplicación de línea de comandos (CLI) del propio proyecto. Desde esta CLI se lanzan todas las operaciones que permiten tanto desarrollar el proyecto como integrarlo, entregarlo y desplegarlo.

En las dos figuras a la derecha del árbol de ficheros del proyecto se encuentran desplegados los directorios que albergan los servicios y las librerías. Destacan los nombres de los mismos y los consiguientes archivos "package.json" que dotan a los servicios y librerías de independencia, fundamental en un escenario monorepo como este.

Servicios

Los servicios en Trainscoding son aplicaciones ejecutadas sobre NodeJS[3] autocontenidas y desplegadas sobre su propio espacio dentro de la aplicación general. Estos servicios son los que contienen la lógica de negocio.

Los servicios siguen la arquitectura hexagonal que pretende desacoplar los puntos de comunicación al exterior con el núcleo de la aplicación, asegurando el aislamiento entre las tecnologías que implementan la funcionalidad y el propio dominio del servicio.

Server

Se trata de un servidor web que actúa como punto de entrada de todas las peticiones que llegan al servidor de trainscoding desde el cliente.
Desde este servidor se exponen el resto de endpoints que conforman la aplicación. Como el resto de los servicios que implementan un servidor web lo hace mediante el uso de framework ExpressJS[4].

[3] https://nodejs..org/
[4] https://expressjs.org/

Users

Este servicio se encarga de la gestión de usuarios. Como se ha mencionado anteriormente, no se han implementado todas las funcionalidades ya que se trata de un servicio de ejemplo. Todas las peticiones a la API de Trainscoding que tienen el prefijo "users" son redireccionadas a este servicio.

Librerías

En un proyecto modularizado en servicios, cada servicio es independiente y autocontenido, lo que dificulta la tarea de compartir y reutilizar código entre los mismos. En Javascript existe el concepto de módulo que puede ser exportado para después ser importado en cualquier proyecto. NPM[5] es una herramienta de línea de comando que permite instalar y mantener módulos descargados desde un registro propio, facilitando la reutilización mencionada anteriormente.

En Trainscoding, las librerías son módulos publicados en el registro de NPM para compartir código entre los diferentes servicios. Además, estas librerías son públicas, por lo que cualquier proyecto puede utilizarlas por medio de una instalación con NPM.

Para describir con detalle el funcionamiento de las librerías, es preciso introducir qué tecnologías usan a su vez estas librerías, puesto que su funcionamiento está fuertemente ligado a estas:

- Eslint[6]: un "linter" cuya función es analizar estáticamente (en tiempo de construcción) el código en búsqueda de errores de sintaxis y de estilo. Esta librería contiene un conjunto de reglas por defecto, pero es altamente configurable a través de complementos (plugins). En Trainscoding, Eslint es usado para alertar a los desarrolladores de la existencia de errores y avisos sobre la sintaxis del código.
- Prettier[7]: es una librería que contiene un conjunto de reglas específico que permite formatear el código de forma opinionada. Su propósito es alinear a la comunidad de desarrollo en Javascript para adoptar un estilo de programación común.

Ambas librerías se utilizan en dos fases: desarrollo e integración. Durante el tiempo de desarrollo, es responsabilidad del desarrollador mantener el código limpio de errores y malas prácticas, por lo que puede hacer uso de estas herramientas en cualquier momento, teniendo más sentido en el instante anterior de enviar cambios al repositorio remoto. Durante la fase de integración, la herramienta de integración continua hace uso

[5] https://www.npmjs.com/
[6] https://eslint.org/
[7] https://prettier.io/

de estas librerías externas por medio de la CLI de Trainscoding y de las librerías propias que se detallan a continuación en la sección de infraestructura.

Eslint-config

Contiene las reglas de formato que se aplican al código que contienen los servicios. Para ello, se usa un conjunto de reglas que Eslint entiende y aplica sobre los análisis. En el ecosistema de Javascript no hay un consenso general acerca de qué reglas son las óptimas para todos los proyectos, por lo que existen multitud de convenciones que se pueden usar. En este caso, el módulo Eslint-config sigue las reglas sugeridas por la propia comunidad de Eslint y hace uso del conjunto de reglas llamado "eslint:recommended:". La ventaja de usar una librería personalizada para aplicar en los servicios de Trainscoding es que permite la modificación de este conjunto de reglas recomendadas en cualquier momento. Además, este paquete contiene algunas configuraciones necesarias para que el análisis del código se realice correctamente en función del entorno que se está utilizando en la aplicación, o los tests que se están ejecutando.

Además, la propia estructura del módulo de NPM permite ejecutar tests unitarios específicos para este conjunto de reglas, de modo que se protege el código de cara a futuras ediciones que puedan ocurrir, a parte del añadido de calidad que supone contar con tests en el módulo.

Prettier-config

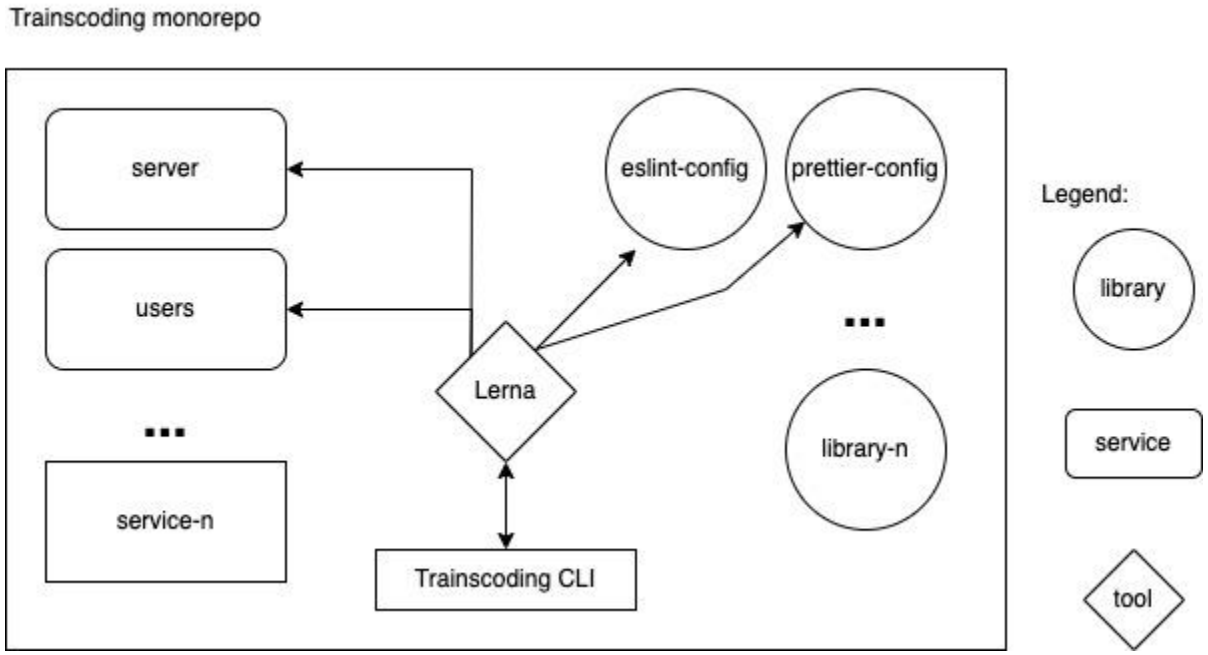
Prettier-config tiene el mismo propósito que la librería anterior, Eslint-config. Haciendo uso de los módulos de se puede utilizar una configuración común a todos los servicios de Trainscoding. A diferencia de Eslint-config, en Prettier-config se han añadido algunas reglas adicionales para completar el conjunto de reglas pre-definido inicialmente por la librería Prettier.

Infraestructura

Como se ha indicado en el resumen, la gestión de la infraestructura en este proyecto monorepo se ha realizado usando Lerna[8]. Lerna es una herramienta para manejar proyectos Javascript "multi-paquete". Trainscoding está dividida en librerías y servicios atendiendo al propósito de los mismos, pero en definitiva, estos son paquetes (o módulos) que permiten la importación y exportación de código.

Gestión de proyectos monorepo con Lerna

Lerna resulta muy conveniente a la hora de realizar tareas repetitivas sobre un número indeterminado de paquetes, pero también permite mantener sincronizada y actualizada las dependencias entre los paquetes de un mismo proyecto.



En la imagen anterior se describe, de forma general, la manera en la que los distintos agentes de Trainscoding interactúan entre sí. A través de Lerna, los distintos comandos disponibles en CLI (Command Line Interface) de Trainscoding permiten ejecutar acciones específicas por paquete o hacerlo de manera global.

[8] https://github.com/lerna/lerna

Gestión de releases

En este proyecto se realizan dos tipos de releases. Por un lado, las librerías se publican en el registro de NPM para facilitar que el código pueda ser usado por los servicios. Y a su vez, los servicios son publicados como imágenes Docker[9] en los registros de Github[10] y Dockerhub[11]. Esas imágenes son usadas para su posterior despliegue en el "cluster" de Kubernetes[12] de Okteto.

Publicación de una nueva versión de una librería

Trainscoding ha sido dada de alta como organización en NPM. Esto permite agrupar todos los paquetes (librerías) del proyecto dentro de un mismo contexto.
Una vez que el paquete ha sido publicado, este puede ser instalado como dependencia en otros paquetes siguiendo los pasos habituales en el ecosistema de Javascript. En la sección de donde se detalla la funcionalidad de la interfaz de línea de comandos, en el apartado publish, se detalla el proceso seguido en la herramienta de CI.

Publicación de una nueva imagen Docker

Cada servicio dispone de un archivo "Dockerfile" que contiene los pasos para construir la imagen. Cuando alguno de los servicios es editado, la herramienta de CI Github Actions[13] reacciona a ese cambio, ejecutando el "pipeline" o "workflow" correspondiente a los servicios que han cambiado. Idealmente, cada commit debe contener un cambio atómico y específico (siguiendo el modelo de desarrollo TBD - Trunk Based Development[14]), por lo que generalmente se espera que sólo un servicio haya sido modificado por cada commit[15].

Si todos los pasos proceden de manera satisfactoria en el "pipeline", una imagen con una nueva etiqueta (tag) por cada servicio actualizado es publicada en los registros mencionados anteriormente. En la sección Entrega Continua se describe este proceso en detalle.

[9] https://www.docker.com/
[10] https://ghcr.io/
[11] https://hub.docker.com/
[12] https://kubernetes.io/
[13] https://docs.github.com/es/actions/
[14] https://trunkbaseddevelopment.com/
[15] https://en.wikipedia.org/wiki/Commit_(version_control)/

CLI (Command Line Interface)

Trainscoding dispone de un conjunto de comandos que son utilizados tanto en tiempo de desarrollo como en tiempo de construcción del proyecto, generalmente en dentro de las herramientas de integración continua (Github Actions).

Estos comandos han sido descritos en un fichero llamado "Makefile". Make[16] es una tecnología desarrollada para sistemas operativos de tipo Unix/Linux y, en un principio, fue concebida para gestionar las dependencias de un proyecto en 1976. Make es capaz de determinar qué operaciones han de realizarse para compilar/crear un fichero dado. Por su versatilidad, también es posible utilizar Make como un lanzador de comandos, por lo que ha ganado mucha popularidad en los últimos años con el auge de los sistemas automatizados.

A continuación se detallan los comandos disponibles en Trainscoding. Muchos de ellos disparan acciones conocidas como "scripts" en el ecosistema de NPM, y todas ellas están contenidas en el módulo general de trainscoding (en el fichero package.json).

bootstrap

Ejecuta el comando "bootstrap" de Lerna. Consiste en sincronizar las interdependencias de los distintos paquetes de trainscoding. Por ejemplo, durante el tiempo de desarrollo, se crean enlaces simbólicos entre las dependencias internas del proyecto para que sea posible acceder al código de una librería que aún no se ha publicado en tiempo real. Además, si alguno de los paquetes que dependen de otros están apuntando a versiones anteriores, este proceso se encarga de actualizar el fichero "package.json" (fichero que determina, entre otras cosas, qué dependencias y su versiones tiene un paquete) para que use la versión más reciente, sin necesidad de ninguna acción manual. Esto se vuelve especialmente práctico en proyectos con multitud de paquetes que dependen entre sí.

build

Genera los artefactos de los servicios que se utilizarán en las imágenes de Docker. Se puede ejecutar para construir todos los servicios a la vez o cada uno de ellos por separado. Este comando permite construir un servicio en particular, por medio del uso de variables de entorno, o de construir todos los servicios a la vez.

lint

Ejecuta el linter (Eslint) haciendo uso de la librería propia "eslint-config". Este proceso es rápido, por lo que se ha decidido que la única posibilidad de ejecutarlo sea

[16] https://www.gnu.org/software/make/

sobre todo el código por razones de simplicidad. En caso de que hubiera un error en el estilo del código, la operación enviará un código de error a la salida de la consola, por lo que se detendrá la ejecución del "pipeline" en el sistema de integración continua (CI).

lint-fix

Ejecuta el formateador (Prettier) haciendo uso de la librería propia "prettier-config", de forma idéntica a como lo hace el comando anterior. En esta ocasión también se ha optado por ejecutar esta acción sobre el total del código a analizar.

En caso de que el código presente alguna malformación de estilo, y siempre que la herramienta Prettier sea capaz de lidiar con ello, se llevará a cabo una sobreescritura del código para garantizar así que este esté formateado de la manera deseada. Este comando es utilizado en la CI de los servicios porque resulta conveniente a la hora de resolver pequeños errores sin la necesidad de ninguna intervención humana, por lo que es idóneo en sistemas automáticos.

Si existen errores demasiado complejos para Prettier, entonces la operación resultará en error, por lo que el progreso del pipeline se detendrá, obligando así a los desarrolladores a tomar acción para arreglar el problema.

publish

Este comando llama al script "publish:ci". Por medio de Lerna, los paquetes que han sido modificados desde la última publicación de los mismos serán propuestos para su re-publicación. Lerna ofrece diferentes opciones a la hora de publicar un paquete en un registro a elección del usuario. En caso de Trainscoding, el registro utilizado es NPM Registry.

Para evitar que la interfaz de Lerna realice preguntas al usuario, la herramienta de CI en caso de Trainscoding, se utiliza una opción permite extraer información de los mensajes de los commits del proyecto para realizar un suposición del cambio que se quiere realizar. Lerna utiliza la sintaxis de Conventional Commits[17] para ello. En caso de no encontrar el tipo de lanzamiento (release) deseado, la versión de publicación será parche (patch).

En Trainscoding, las únicas publicaciones de paquetes realizadas sobre el registro de NPM son las librerías (eslint-config y prettier-config). Aunque los servicios también disponen de la arquitectura necesaria para ser publicados en un registro de paquetes, la intención de los mismos no es la de ser reutilizados, al contrario que las librerías. Por ello, la configuración de Lerna excluye a los servicios de su lista de candidatos a ser publicados.

[17] https://www.conventionalcommits.org/en/v1.0.0/

test

Se encarga de la ejecución de los tests. Los servicios y librerías de Trainscoding cuentan con tests unitarios, y estos se ejecutan dentro de un contenedor de Docker para incrementar la fiabilidad de los mismos cuando distintas personas trabajan en el desarrollo del proyecto. Gracias al framework de testing Jest[18], existe la posibilidad de ejecutar todos los tests a la vez o ejecutarlos para un paquete en concreto.

create-env

Ejecuta un sencillo script de Bash[19] que se encarga de generar los archivos archivos que contienen las variables de entorno de los servicios, necesarios al comienzo del desarrollo del proyecto. Esto es útil cuando un nuevo desarrollador clona el repositorio en su máquina local.

up

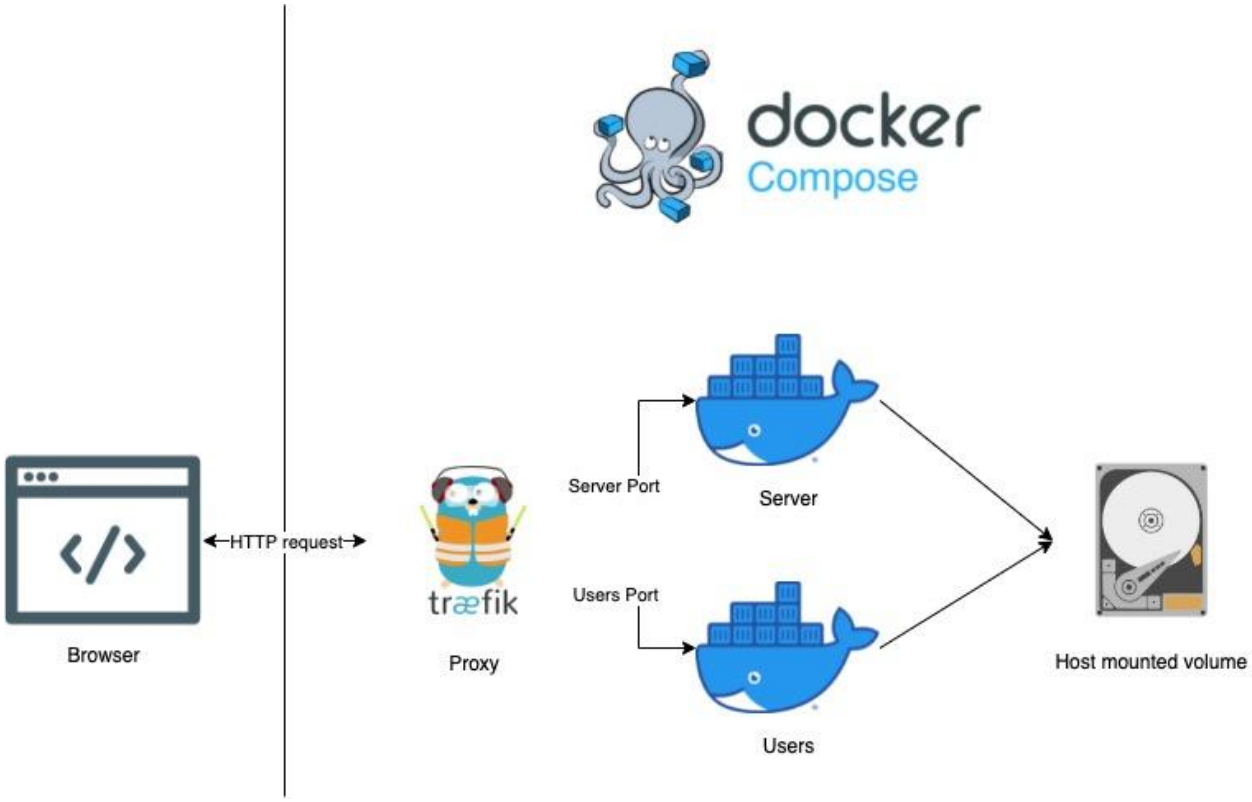
Levanta el entorno de desarrollo local, que será detallado en profundidad en la siguiente sección.

[18] https://jestjs.io/
[19] https://www.gnu.org/software/bash/

Desarrollo en local

El entorno de desarrollo pretende emular la situación en la que Trainscoding se despliega en el entorno de producción. Para conseguir que todos los servicios se estén ejecutando a la vez se hace uso de Docker Compose[20] y Traefik[21], fundamentalmente. Al contrario que en el entorno de producción, no se ha usado la misma infraestructura basada en Kubernetes porque se ha querido reducir la cantidad de recursos necesarios para ejecutar el proyecto entero, así como minimizar la complejidad a la hora de realizar cambios de forma continua.

Docker Compose es una herramienta que permite ejecutar varios contenedores, definidos como servicios, a los que se les puede inyectar una configuración para modificar su funcionamiento. Usando una imagen de NodeJS como base, común a todos los servicios, y montando el código del repositorio usando volúmenes, es posible modificar el código que se ejecuta dentro de los contenedores ya que en realidad el código reside en la máquina anfitriona y no dentro de la imagen del contenedor.



En la imagen anterior se describe el lugar que ocupa Traefik en arquitectura de desarrollo en local. Traefik, en concreto Traefik Proxy, es un enrutador que cumple el

[20] https://docs.docker.com/compose/
[21] https://doc.traefik.io/traefik/

mismo cometido que el Ingress en Kubernetes. Para evitar el uso de puertos explícitos para acceder a los diferentes servicios de Trainscoding, y así, emular la situación real que se da en el entorno de producción donde el usuario nunca ve los puertos internos en los que los servicios son desplegados, es necesario configurar Traefik Proxy para que actúe como un intermediario entre las peticiones HTTP que llegan al puerto 80 de Trainscoding. La configuración de los servicios incluyen dos etiquetas, una para habilitar Treafik y que este tenga el cuenta al servicio como un candidato a redireccionar peticiones, y otra que especifica el prefijo de la ruta de la URL que se va a utilizar para decidir qué peticiones son dirigidas a un servicio o a otro.

El papel de Lerna

En un entorno local, tener acceso a los cambios en caliente es esencial para conseguir un ritmo de trabajo productivo. El proceso que siguen las librerías que se publican en el registro de NPM no se adapta de forma adecuada a un proyecto monorepo. En Trainscoding, las librerías son dependencias de los servicios. Cuando cambia el código en estas, se necesita generar una nueva versión para que los servicios puedan utilizar el código más reciente. Para evitar este proceso, y sobre todo, para poder comprobar de forma inmediata si los cambios realizados en las librerías tienen el efecto deseado en los servicios que las consumen, se utiliza Lerna para conectar los servicios a la versión más reciente de las librerías antes de que se publique una nueva versión. Lerna realiza enlaces simbólicos en la carpeta donde se almacenan los módulos (dependencias) de cada servicio, para que los cambios que aún no han sido publicados puedan ser consumidos sin necesidad de publicar una nueva release. Además, Lerna es capaz de mantener actualizados los ficheros (package.json) donde se especifica la versión de las dependencias. De este modo, si un paquete “A” tiene como dependencia a un paquete “B”, y “B” publica una nueva release, el archivo package.json de “A” será modificado por Lerna para que apunte a la nueva versión de “B”, y propondrá a “A” como candidato para publicar una nueva versión ya que su código ha cambiado.

Integración continua (CI)

El proceso de integración continua consiste en comprobar y garantizar que el estado del proyecto sigue siendo el deseado tras añadir código al repositorio. Para ello, dependiendo del objetivo a integrar, se ejecutan los tests disponibles y se comprueba el estilo del código.

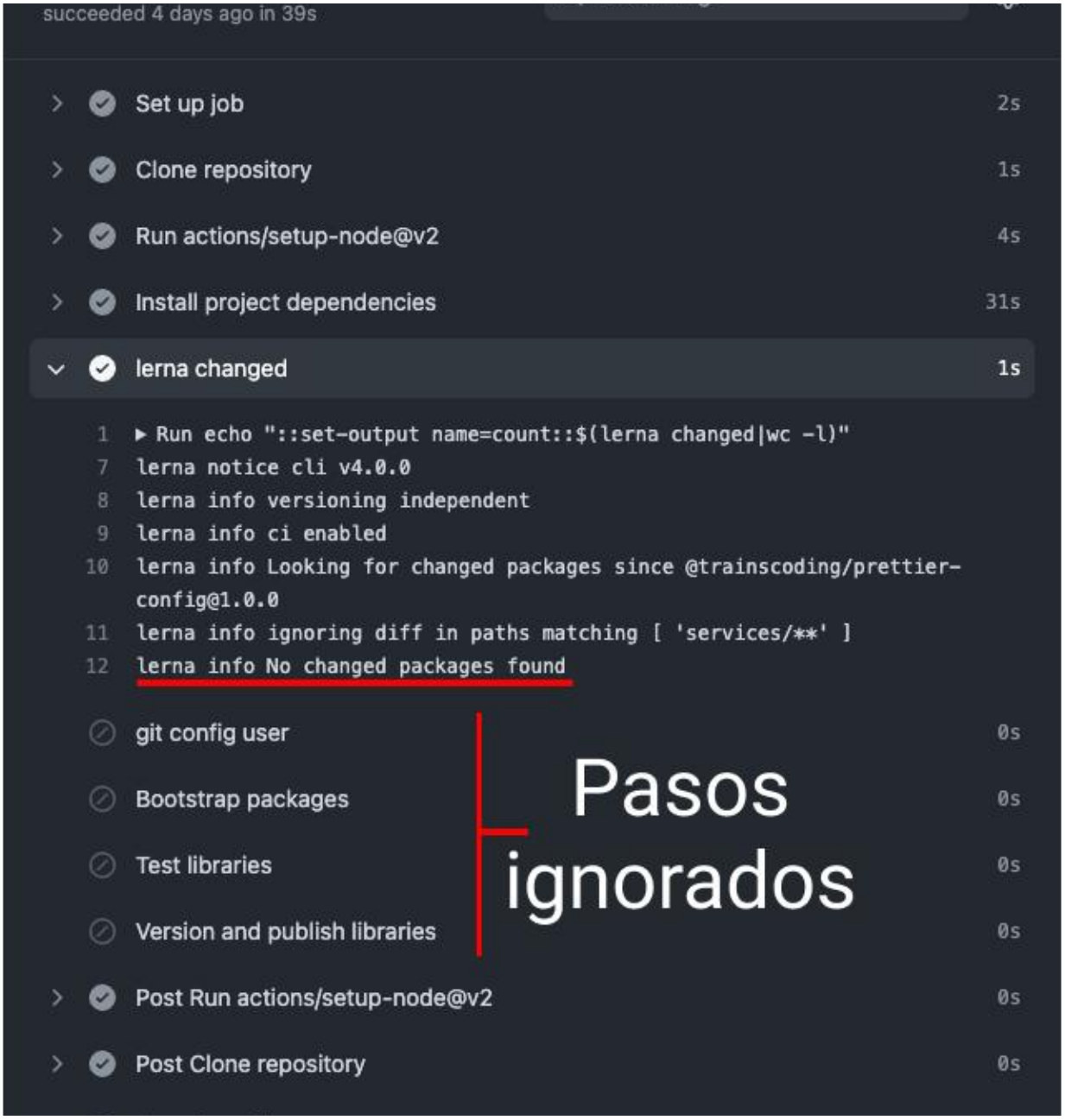
Como se ha mencionado anteriormente en este documento, la herramienta utilizada para llevar a cabo la integración continua es Github Actions. En Github Actions se pueden definir "workflows", un itinerario donde cada paso tiene un cometido específico. Para esta sección de integración continua se va a profundizar en los pasos de estos workflows que tienen el cometido de integrar, es decir, asegurar que los cambios producidos en el repositorio no rompen el estado actual del proyecto.

Workflow para integrar las librerías

En caso de que Lerna detecte que el código de alguna de las librerías ha cambiado desde la última versión publicada en el registro de NPM, se ejecutan los tests sobre las librerías únicamente en un contenedor Docker con la misma versión de NodeJS que se utiliza en los entornos de producción. Por motivos de simplicidad, sólo se comprueba si alguna de las librerías ha sufrido algún cambio. Para un enfoque más óptimo convendría comprobar cuál de las librerías han sido las que han cambiado y ejecutar sólo los tests para estas. El workflow se ejecuta por cada commit que se empuja a la rama main del proyecto en Github, lo cual es costoso, por eso se comprueba cuanto antes si es necesario ejecutar todo el proceso o no.

Ejemplo de commit que no contiene cambios que afectan a ninguna librería

En la siguiente imagen se expone una ejecución del workflow en el que Lerna no detecta ningún cambio y, por tanto, los pasos involucrados en la publicación de las librerías son ignorados, lo que supone un ahorro en recursos y en tiempo.



El workflow está accesible desde este enlace.

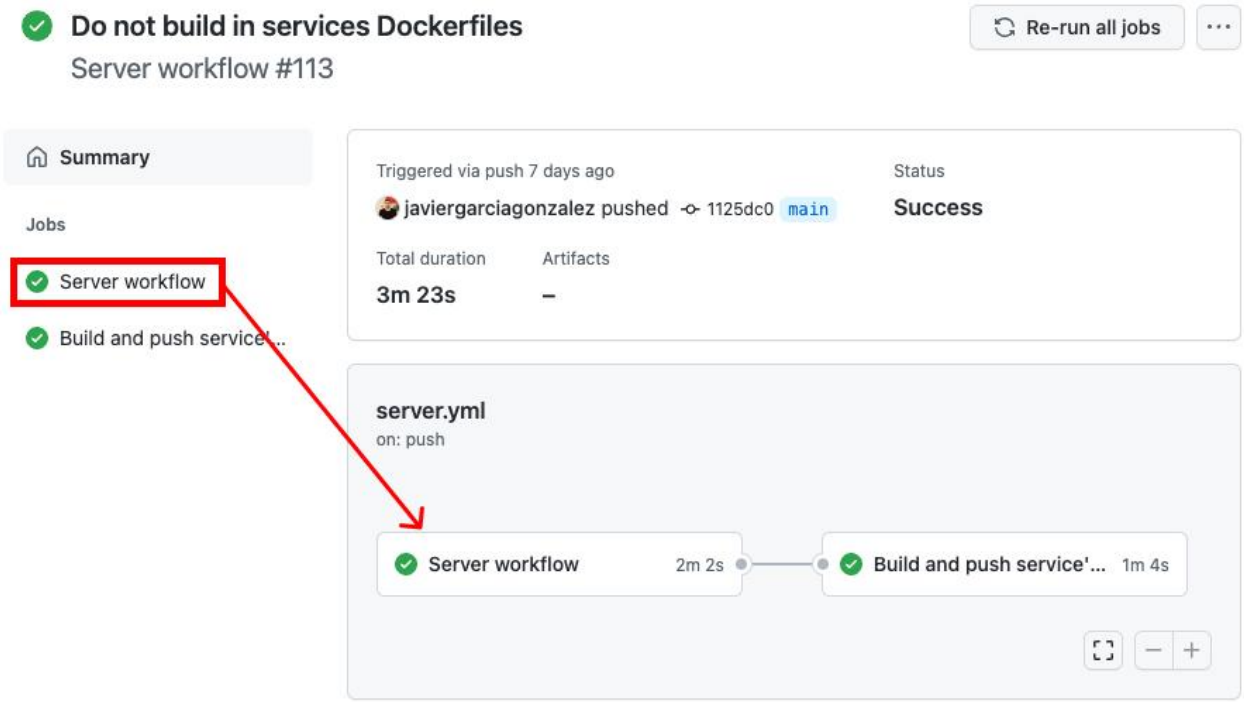
Workflows para integrar los servicios

De modo similar a las librerías, los servicios también ejecutan los tests dentro de un contenedor Docker. Sin embargo, para los servicios también se ejecuta el linter y el formateador de código (que usan precisamente las librerías de Trainscoding), antes de proceder a los siguientes pasos (entrega y despliegue). En el caso de los servicios, debido a la complejidad de estos y a la cantidad de pasos involucrados en el workflow,

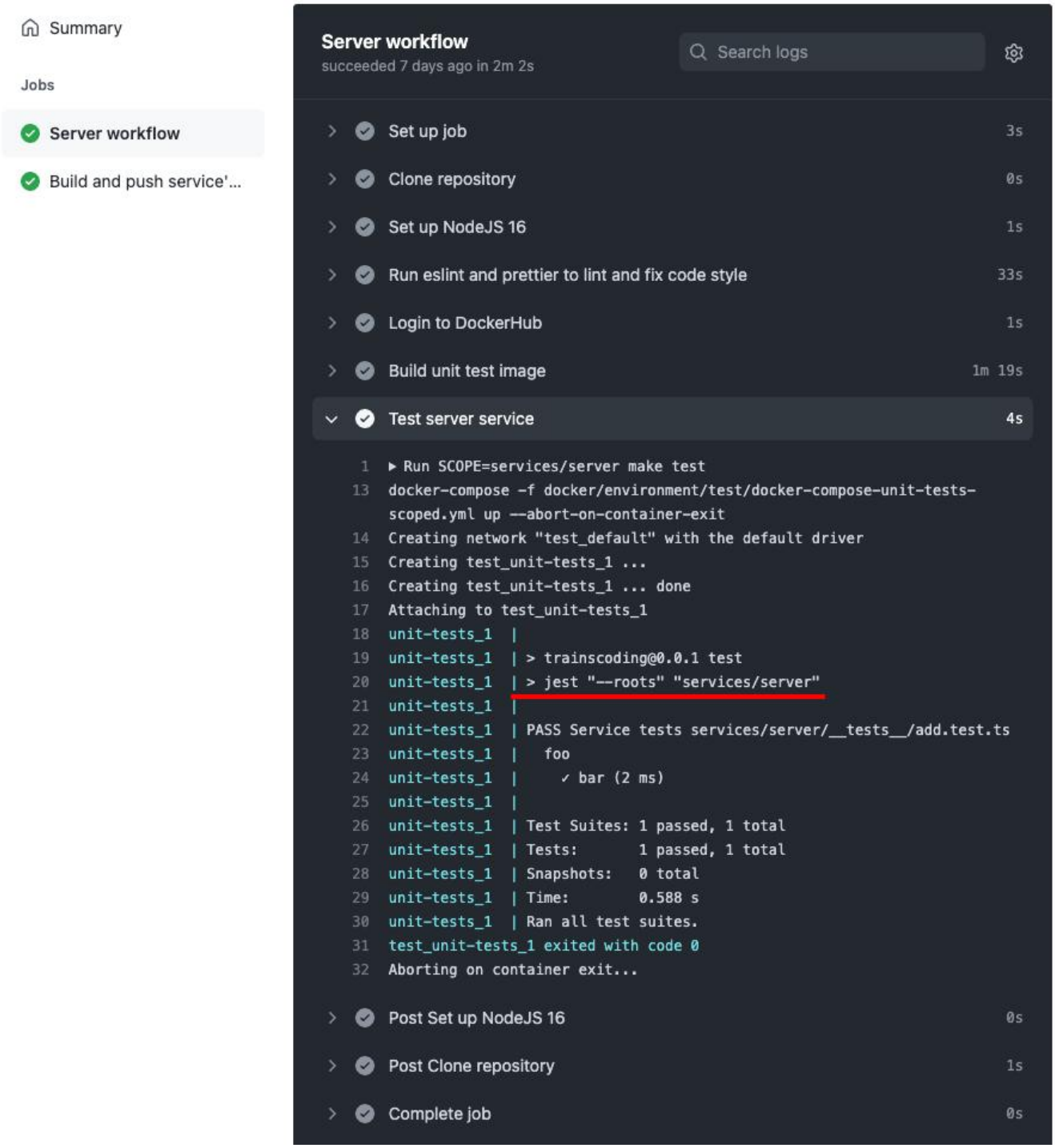
se ha optado por mantener un workflow distinto por servicio. De modo que estos workflows se ejecutan cuando algún fichero que componen el servicio o el template de Kubernetes que se utiliza para desplegar la aplicación cambian. El siguiente paso para mejorar la configuración de estos workflows para los servicios sería reutilizar la lógica de estos para no tener que mantener un archivo distinto por cada servicio en la aplicación ya que es una solución que no se adapta al crecimiento del proyecto.
Los respectivos workflows de los servicios están disponibles en estos enlaces server y users.

Ejemplo de la integración de un servicio durante la ejecución del workflow del servicio Server

Los workflows de los servicios están divididos en dos secciones. La primera, integra el servicio, donde se comprueba que el código está preparado para una segunda fase donde se entrega y despliega.



A continuación se muestra el contenido de la primera parte del workflow expuesto en la imagen anterior, donde se aprecia que los tests que se están ejecutando son únicamente sobre los que contiene el servicio "Server", en este caso, los tests de prueba que han sido añadidos con meros fines demostrativos.



Esta ejecución del workflow puede ser consultada en este enlace.

Entrega continua (CDE)

En la entrega continua se toma como información de entrada el resultado de la aplicación de la integración continua para, una vez comprobado que el software está preparado, construir el empaquetado (artefacto) que se usará en el entorno de producción y su posterior publicación en los registros necesarios para que puedan ser usados en la siguiente fase automatizada de despliegue.

De nuevo, el proceso de entrega continua es diferente en relación a las librerías y los servicios.

Workflow para entregar (publicar) librerías

Una vez se han ejecutado correctamente los tests de las librerías que han sufrido cambios desde la última versión se procede a su publicación en el registro de NPM. También por medio de la herramienta Lerna, como en paso anterior, se publican los paquetes siguiendo las reglas de "versionado" explicadas anteriormente en la sección publish.

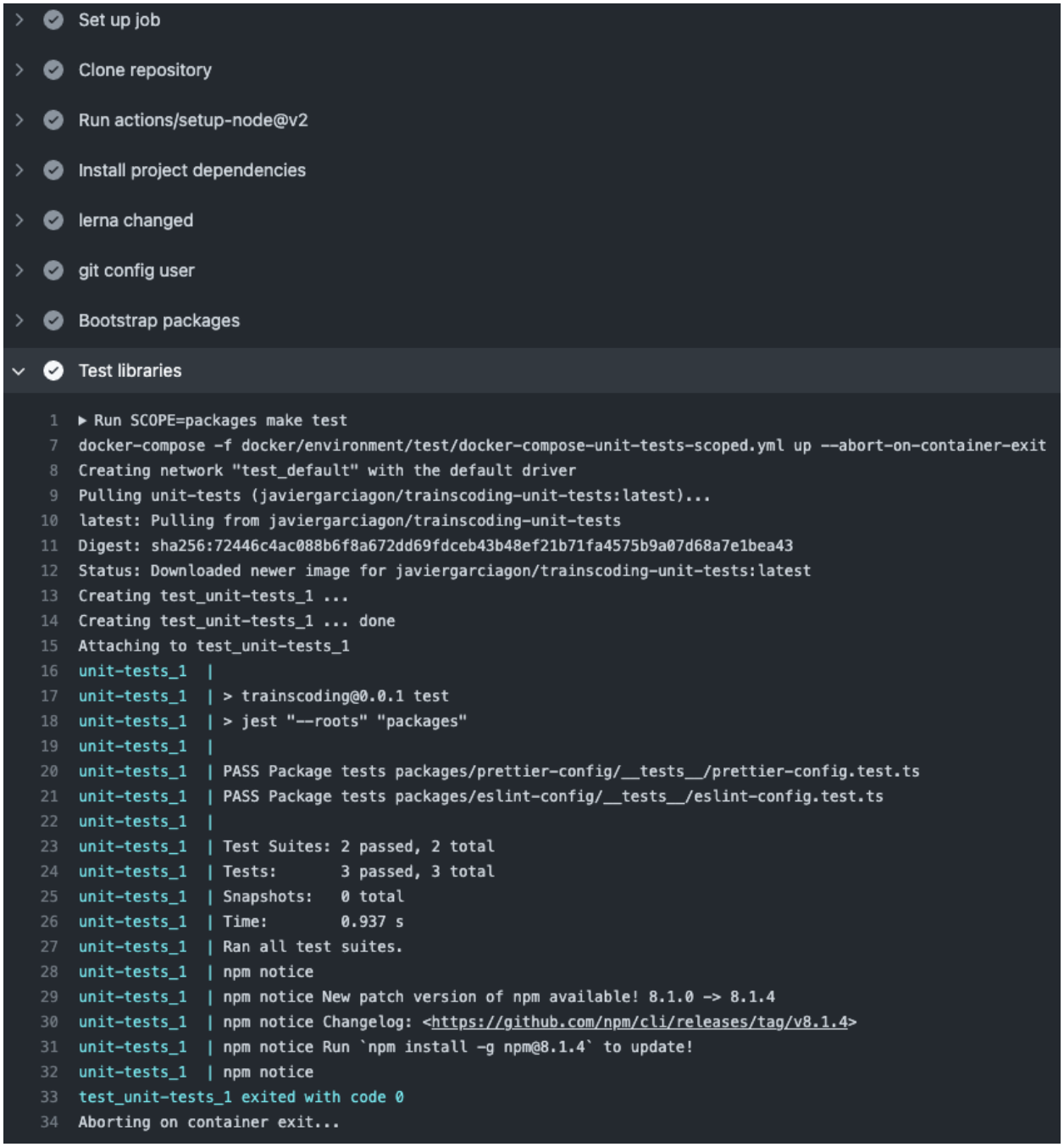
Las ejecuciones de estos workflows pueden ser visitados en este enlace.

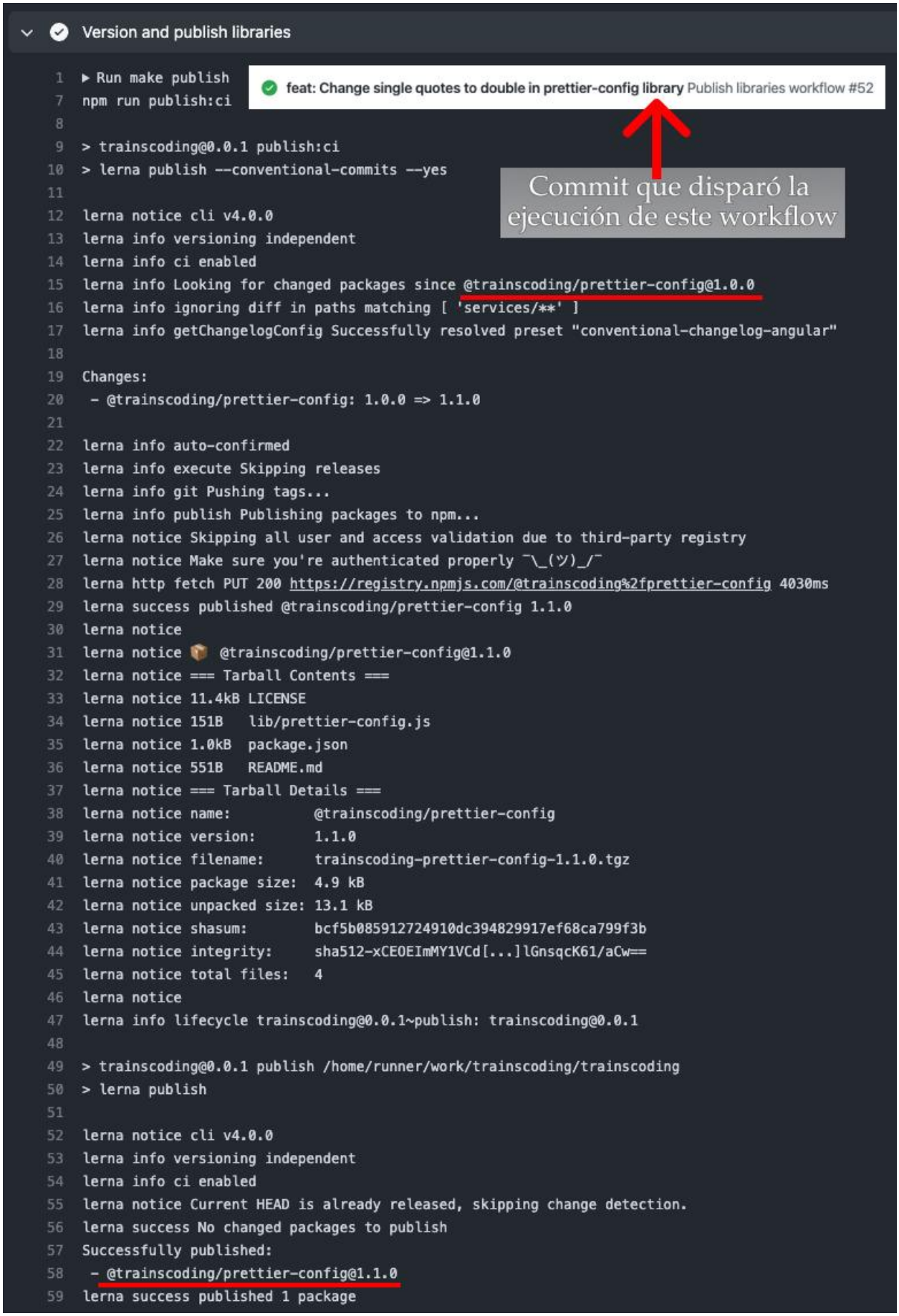
Ejemplo de commit que contiene cambios que afectan a alguna librería

Para este caso, se presenta un cambio en el que la librería Prettier-config ha sufrido la modificación de una de sus reglas. A partir de este cambio, se ha optado por reemplazar todas las comillas simples del código por comillas dobles. Este cambio contiene una nueva feature, por lo que siguiendo las convenciones de Semver[22], se debe publicar una nueva versión "minor": implementa nueva funcionalidad pero no rompe la estructura en la que se consume esta librería. Gracias a que el mensaje del commit ha sido escrito siguiendo el modelo que propone conventional-commits y a que Lerna es capaz de entenderlo, se ha publicado de forma automática una nueva versión para la librería Prettier-config, que ha pasado de la versión "1.0.0" a la versión "1.1.0".

En este caso, todos los pasos que componen el workflow han sido ejecutados, por lo que la librería ha sido testeada antes de proceder a su publicación.

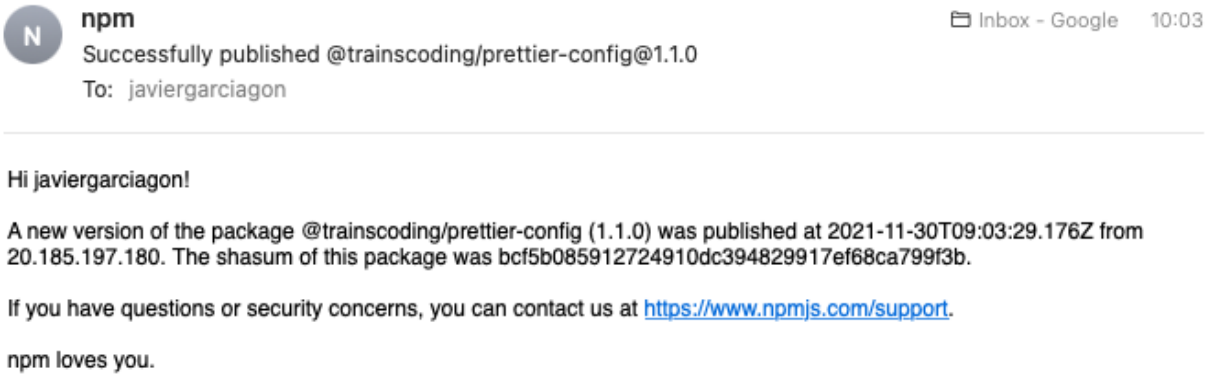
[22] https://semver.org/lang/es


Set up job
Clone repository
Run actions/setup-node@v2
Install project dependencies
lerna changed
git config user
Bootstrap packages
Test libraries
▶ Run SCOPE=packages make test
docker-compose -f docker/environment/test/docker-compose-unit-tests-scoped.yml up --abort-on-container-exit
Creating network "test_default" with the default driver
Pulling unit-tests (javiergarciagon/trainscoding-unit-tests:latest)...
latest: Pulling from javiergarciagon/trainscoding-unit-tests
Digest: sha256:72446c4ac088b6f8a672dd69fdceb43b48ef21b71fa4575b9a07d68a7e1bea43
Status: Downloaded newer image for javiergarciagon/trainscoding-unit-tests:latest
Creating test_unit-tests_1 ...
Creating test_unit-tests_1 ... done
Attaching to test_unit-tests_1
unit-tests_1 |
unit-tests_1 | > trainscoding@0.0.1 test
unit-tests_1 | > jest "--roots" "packages"
unit-tests_1 |
unit-tests_1 | PASS Package tests packages/prettier-config/__tests__/prettier-config.test.ts
unit-tests_1 | PASS Package tests packages/eslint-config/__tests__/eslint-config.test.ts
unit-tests_1 |
unit-tests_1 | Test Suites: 2 passed, 2 total
unit-tests_1 | Tests: 3 passed, 3 total
unit-tests_1 | Snapshots: 0 total
unit-tests_1 | Time: 0.937 s
unit-tests_1 | Ran all test suites.
unit-tests_1 | npm notice
unit-tests_1 | npm notice New patch version of npm available! 8.1.0 -> 8.1.4
unit-tests_1 | npm notice Changelog: <https://github.com/npm/cli/releases/tag/v8.1.4>
unit-tests_1 | npm notice Run `npm install -g npm@8.1.4` to update!
unit-tests_1 | npm notice
test_unit-tests_1 exited with code 0
Aborting on container exit...




```
▶ Run make publish
npm run publish:ci

> trainscoding@0.0.1 publish:ci
> lerna publish --conventional-commits --yes

lerna notice cli v4.0.0
lerna info versioning independent
lerna info ci enabled
lerna info Looking for changed packages since @trainscoding/prettier-config@1.0.0
lerna info ignoring diff in paths matching [ 'services/**' ]
lerna info getChangelogConfig Successfully resolved preset "conventional-changelog-angular"

Changes:
 - @trainscoding/prettier-config: 1.0.0 => 1.1.0

lerna info auto-confirmed
lerna info execute Skipping releases
lerna info git Pushing tags...
lerna info publish Publishing packages to npm...
lerna notice Skipping all user and access validation due to third-party registry
lerna notice Make sure you're authenticated properly ¯\_(ツ)_/¯
lerna http fetch PUT 200 https://registry.npmjs.com/@trainscoding%2fprettier-config 4030ms
lerna success published @trainscoding/prettier-config 1.1.0
lerna notice
lerna notice 📦  @trainscoding/prettier-config@1.1.0
lerna notice === Tarball Contents ===
lerna notice 11.4kB LICENSE
lerna notice 151B   lib/prettier-config.js
lerna notice 1.0kB  package.json
lerna notice 551B   README.md
lerna notice === Tarball Details ===
lerna notice name:          @trainscoding/prettier-config
lerna notice version:       1.1.0
lerna notice filename:      trainscoding-prettier-config-1.1.0.tgz
lerna notice package size:  4.9 kB
lerna notice unpacked size: 13.1 kB
lerna notice shasum:        bcf5b085912724910dc394829917ef68ca799f3b
lerna notice integrity:     sha512-xCEOEImMY1VCd[...]lGnsqcK61/aCw==
lerna notice total files:   4
lerna notice
lerna info lifecycle trainscoding@0.0.1~publish: trainscoding@0.0.1

> trainscoding@0.0.1 publish /home/runner/work/trainscoding/trainscoding
> lerna publish

lerna notice cli v4.0.0
lerna info versioning independent
lerna info ci enabled
lerna notice Current HEAD is already released, skipping change detection.
lerna success No changed packages to publish
Successfully published:
 - @trainscoding/prettier-config@1.1.0
lerna success published 1 package
```

Como proceso de confirmación, NPM envía un email al equipo de desarrollo designado a esta librería en cuestión para informar de que una nueva versión ha sido publicada en el registro:

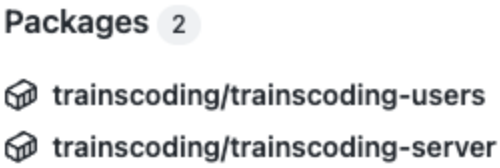


El workflow está accesible desde este enlace.

Workflows para entregar (publicar) servicios

Los servicios se empaquetan en imágenes Docker para posteriormente ser desplegadas en un clúster Kubernetes. En Trainscoding se ha seguido el modelo de desarrollo TBD, por lo que cada cambio en la rama principal está destinada a ser una versión final en producción. Esto elimina el concepto de RC (release candidate). Por tanto, cada cambio produce una nueva imagen a la que se añade una etiqueta que referencia el cambio en el sistema de control de versiones (git[23]) y el momento temporal en el que se produjo (timestamp). Una vez se ha generado el nombre de la imagen de forma dinámica se procede a su publicación en los registros Dockerhub[24] (registro más utilizado en la comunidad de contenedores) y en el registro de paquetes de Github[25]. El primero es utilizado para albergar las imágenes que posteriormente serán desplegadas en producción, y el segundo es utilizado con la única finalidad de dotar al proyecto en Github de una mayor estructura y consistencia, pues es posible ver de forma rápida qué imágenes se están produciendo en el proyecto cuando se accede a la página principal del repositorio.

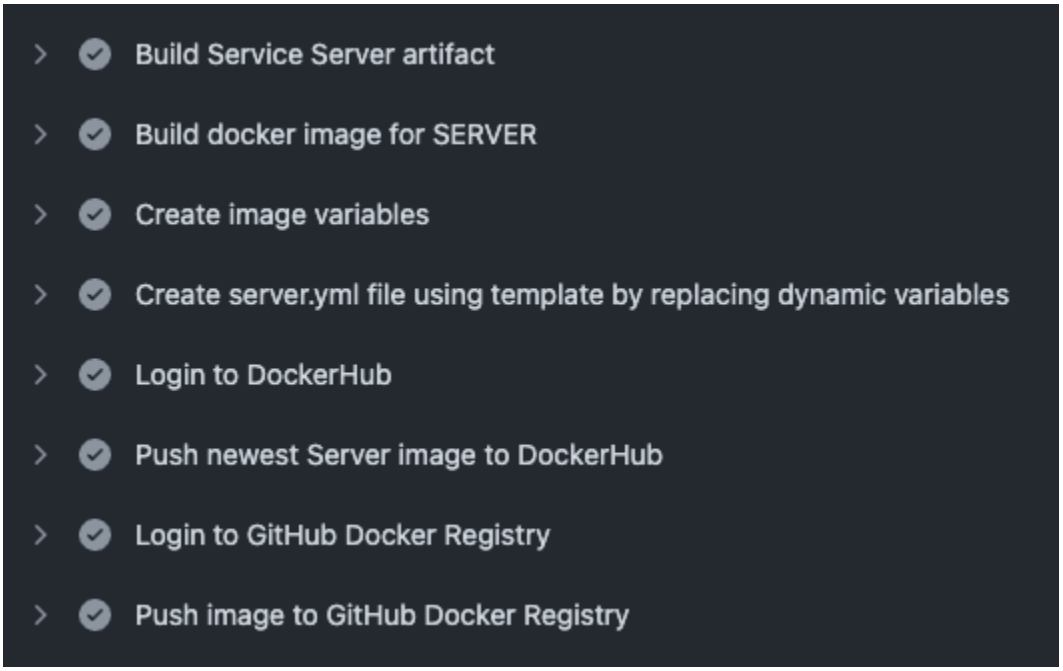
[23] https://git-scm.com/
[24] https://hub.docker.com/
[25] https://docs.github.com/packages/learn-github-packages/introduction-to-github-packages



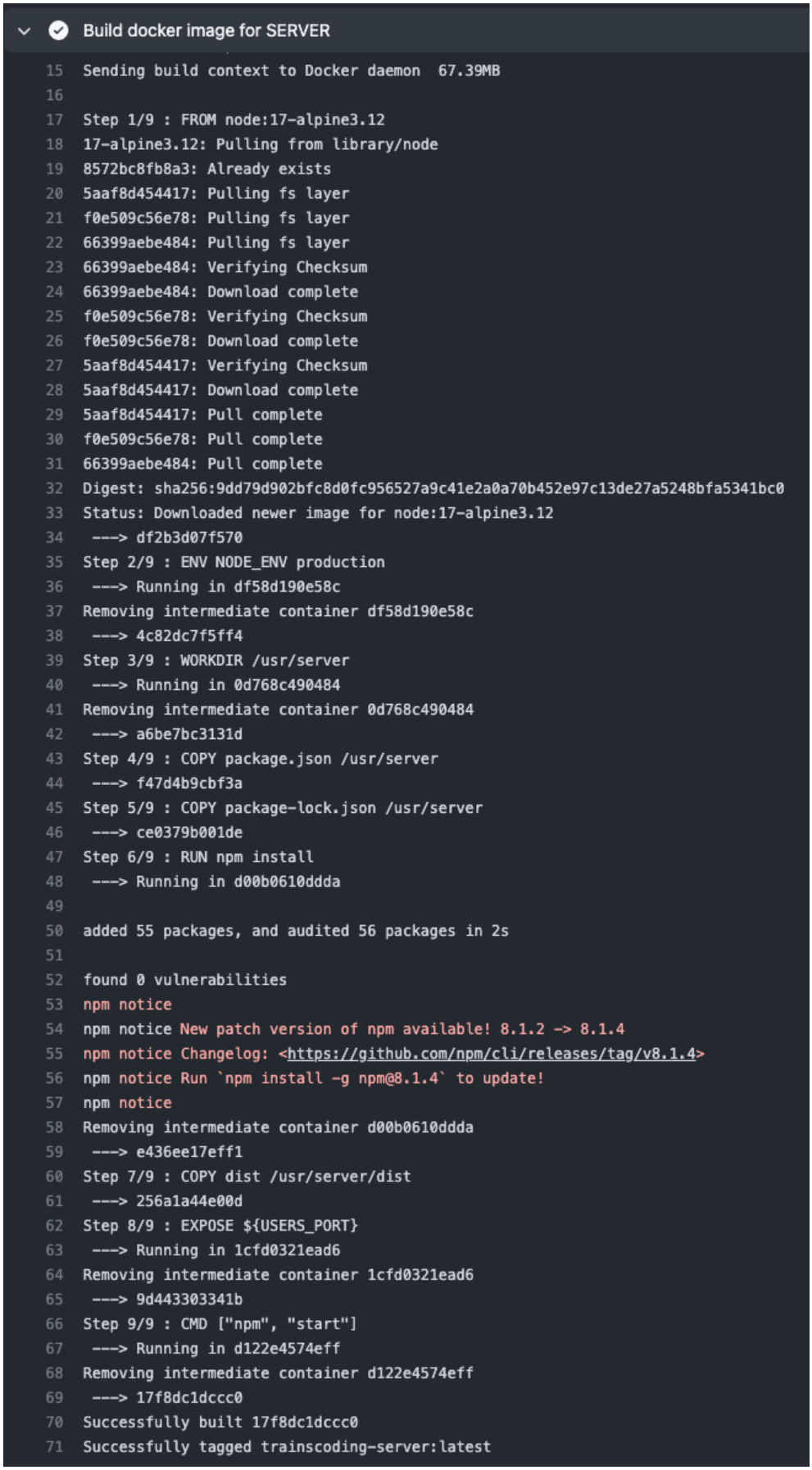
Las ejecuciones de estos workflows para los servicios de server y users pueden ser consultados en estos enlaces.

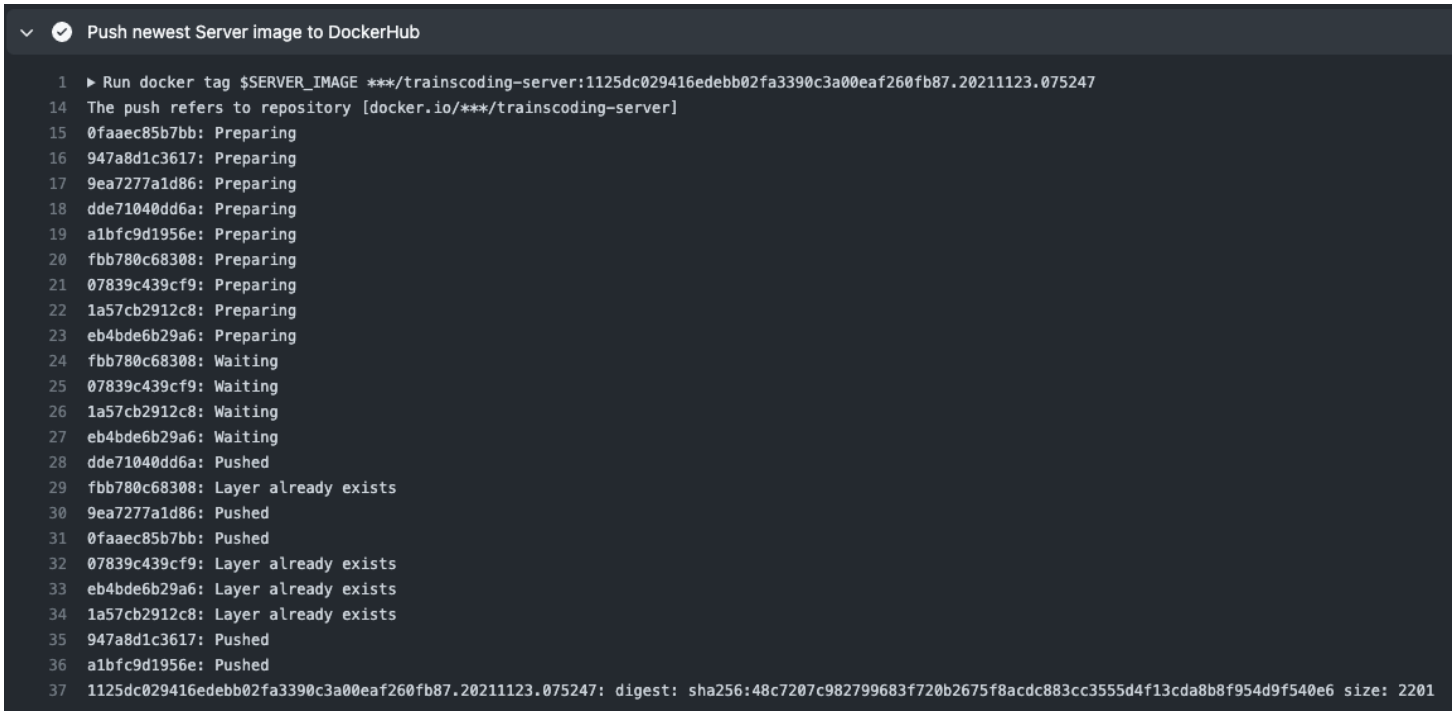
Ejemplo de un commit que contiene cambios en el servicio Server

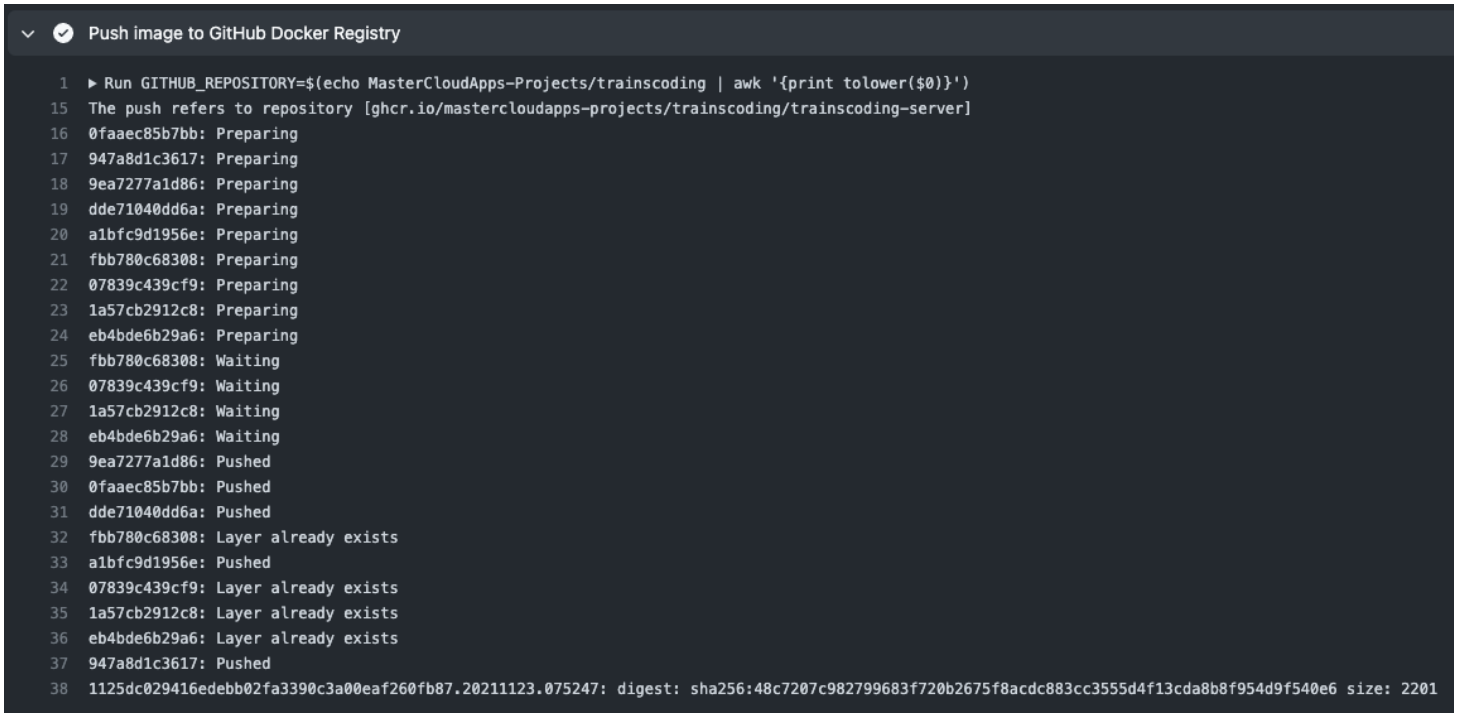
En este ejemplo se detalla la sección del workflow en la que se genera la nueva imagen del servicio que contiene una etiqueta que relaciona esta versión del servicio en particular con el commit que incluye los cambios y el momento en el tiempo en el que se produjo.



Los pasos más relevantes son la creación de la imagen (2º paso en la lista anterior) y las publicaciones en los registros Dockerhub y Github (antepenúltimo y último paso respectivamente).


Build docker image for SERVER
Sending build context to Docker daemon 67.39MB
Step 1/9 : FROM node:17-alpine3.12
17-alpine3.12: Pulling from library/node
8572bc8fb8a3: Already exists
5aaf8d454417: Pulling fs layer
f0e509c56e78: Pulling fs layer
66399aebe484: Pulling fs layer
66399aebe484: Verifying Checksum
66399aebe484: Download complete
f0e509c56e78: Verifying Checksum
f0e509c56e78: Download complete
5aaf8d454417: Verifying Checksum
5aaf8d454417: Download complete
5aaf8d454417: Pull complete
f0e509c56e78: Pull complete
66399aebe484: Pull complete
Digest: sha256:9dd79d902bfc8d0fc956527a9c41e2a0a70b452e97c13de27a5248bfa5341bc0
Status: Downloaded newer image for node:17-alpine3.12
---> df2b3d07f570
Step 2/9 : ENV NODE_ENV production
---> Running in df58d190e58c
Removing intermediate container df58d190e58c
---> 4c82dc7f5ff4
Step 3/9 : WORKDIR /usr/server
---> Running in 0d768c490484
Removing intermediate container 0d768c490484
---> a6be7bc3131d
Step 4/9 : COPY package.json /usr/server
---> f47d4b9cbf3a
Step 5/9 : COPY package-lock.json /usr/server
---> ce0379b001de
Step 6/9 : RUN npm install
---> Running in d00b0610ddda
added 55 packages, and audited 56 packages in 2s
found 0 vulnerabilities
npm notice
npm notice New patch version of npm available! 8.1.2 -> 8.1.4
npm notice Changelog: <https://github.com/npm/cli/releases/tag/v8.1.4>
npm notice Run `npm install -g npm@8.1.4` to update!
npm notice
Removing intermediate container d00b0610ddda
---> e436ee17eff1
Step 7/9 : COPY dist /usr/server/dist
---> 256a1a44e00d
Step 8/9 : EXPOSE ${USERS_PORT}
---> Running in 1cfd0321ead6
Removing intermediate container 1cfd0321ead6
---> 9d443303341b
Step 9/9 : CMD ["npm", "start"]
---> Running in d122e4574eff
Removing intermediate container d122e4574eff
---> 17f8dc1dccc0
Successfully built 17f8dc1dccc0
Successfully tagged trainscoding-server:latest





La ejecución de este workflow en particular puede ser consultado en este enlace.

Despliegue continuo (CD)

Para la fase de despliegue sólo se va a tener en cuenta a los servicios que componen Trainscoding puesto que las librerías no se despliegan, únicamente se publican.

Se ha elegido Okteto como proveedor de servicios en la nube porque ofrece la capacidad de usar un clúster de Kubernetes compuesto por un nodo y un máximo de 10 pods (para la capa gratuita), que es suficiente para la demostración de este proyecto.

Okteto ofrece además una interfaz de la línea de comandos (CLI) para realizar las operaciones necesarias para desplegar Trainscoding puesto que permite usar la herramienta Kubectl[26] (para manejar clústers Kubernetes), de modo que la configuración no difiere de un escenario en el que se tiene acceso a las máquinas de forma física.

En tercer lugar, dentro de los workflows de Github Actions para los servicios, ocurre el despliegue del mismo. Para ello, es necesario usar la imagen que ha sido generada en la fase de entrega continua. En el repositorio existe un template por cada servicio que se utiliza como base para posteriormente incluir la referencia a la nueva imagen de forma automatizada. Este template contiene toda la información necesaria para configurar un servicio y un despliegue en Kubernetes, a excepción del nombre de la imagen a desplegar que contiene la etiqueta más reciente y otras variables de entorno relevantes, como por ejemplo, el puerto por el que el servicio va a estar atendiendo dentro de la red interna del clúster.

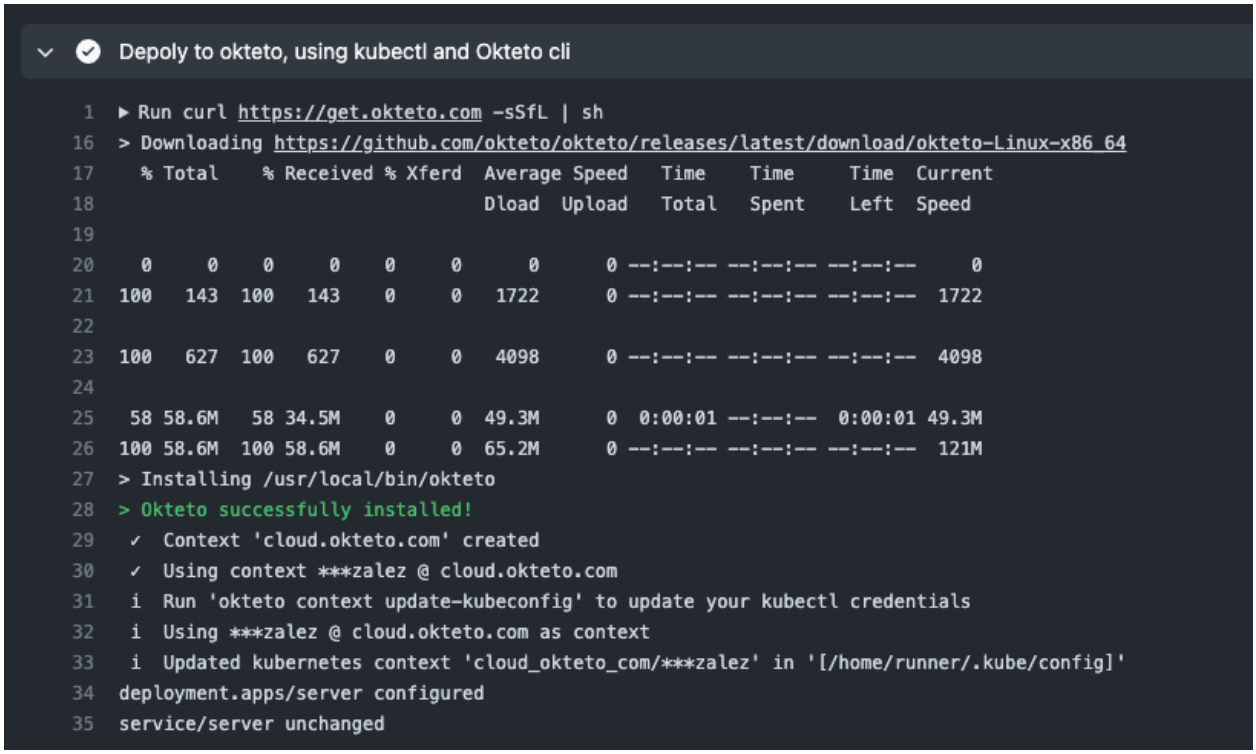
Una vez se ha creado el archivo "yaml" que contiene la información actualizada se utiliza la CLI de Okteto para aplicar los cambios sobre el servicio y el despliegue. En cuestión de segundos la nueva versión del software está disponible y accesible desde un endpoint público.

Ejemplo de un commit que contiene cambios en el servicio Server

Utilizando el ejemplo usado en la sección anterior, a continuación se muestra una imagen de la parte del workflow donde ocurre el despliegue del servicio Server a la plataforma Okteto.

Este es el siguiente paso a la entrega de la imagen a los registros de Dockerhub y Github.

[26] https://kubernetes.io/docs/reference/kubectl/kubectl/



La ejecución de este workflow puede ser consultado en este enlace.

Conclusiones

Tras haber profundizado en los diferentes aspectos de desarrollo y despliegue de un proyecto monorepo que combina la publicación de librerías y servicios distribuidos en Javascript he podido ver algunas de las ventajas y desventajas de seguir este modelo. Para mí, una de las ventajas fundamentales que ofrece el monorepo es que permite desarrollar arquitecturas basadas en servicios independientes ahorrando el coste de mantener tu código descentralizado. Se puede ver como un equilibrio entre un monolito y una arquitectura de microservicios.

Otra de las ventajas fundamentales es que permite reutilizar herramientas a lo largo de todos los servicios independientes sin tener que configurar cada una tantas veces como servicios existan en la aplicación.

Sin embargo, también existen otros compromisos que ponen en cuestión si el monorepo es el enfoque más óptimo para todo tipo de proyectos. A medida que crece la diversidad y diferencias entre los servicios (o paquetes) que componen un monorepo se vuelve más y más complejo aplicar acciones genéricas que funcionen de forma transversal.

A lo largo del desarrollo de este proyecto me he encontrado en muchas ocasiones en la necesidad de rehacer una solución que creía que se podría aplicar de forma genérica en el momento de haber añadido un nuevo servicio que necesitaba de características especiales, o simplemente tenía una estructura ligeramente diferente. Mientras que en un multi-repo (cada servicio está contenido en su propio repositorio) se pueden aplicar soluciones específicas por servicio y estas no necesitan cambiar o adaptarse cuando se añaden nuevos servicios a la aplicación, con los mono-repos no siempre es así. Para mí, la peor parte es cuando necesitas aplicar excepciones en un proyecto que tiende a utilizar soluciones genéricas siempre que sea posible.

Gracias a Lerna, muchos de los problemas más comunes ya están resueltos (gestión de dependencias entre servicios y ejecución de tareas de forma simultánea para todos los servicios desde un punto centralizado), por lo que considero esencial el uso de herramientas de este tipo a la hora de gestionar un proyecto monorepo. De hecho, la infraestructura ha sido la parte del desarrollo que ha resultado más compleja porque permite al desarrollador tomar más soluciones por cuenta propia que pueden llevarte a problemas en el futuro.

Dentro del ecosistema de Javascript, veo que el enfoque monorepo es el más acertado a la hora de construir librerías de componentes o módulos que comparten un cometido común. Por ejemplo, en proyectos front-end, el uso de monorepos está muy extendido en el desarrollo de librerías de componentes de interfaz de usuario, donde todos comparten una estructura similar y precisan de una gestión similar (mismas tecnologías de tests, linters, etc…). Sin embargo, a lo que el desarrollo de servicios se

refiere, tengo mis dudas, sobre todo cuando se pretende desarrollar servicios que no tienen procesos automatizables similares.

Bibliografía

- Material proporcionado en las asignaturas del Master Cloud Apps 2020/2021.
- Libro Open Source "CI/CD for Monorepos" - Semaphore. https://semaphoreci.com/blog/new-book-cicd-for-monorepos
- Documentación oficial Github Actions - https://docs.github.com/es/actions
- Documentación oficial Lerna - https://github.com/lerna/lerna/
- Artículo "Speeding up your CI/CD build times with a custom Docker image" de Sébastien Dubois - https://itnext.io/speeding-up-your-ci-cd-build-times-with-a-custom-docker-image-3bfaac4e0479