Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

ADOPCIÓN DE UN ENTORNO DEVOPS UTILIZANDO GITLAB-CI, KUBERNETES Y SERVICIOS DE AWS PARA UNA APLICACIÓN WEB CON ANGULAR Y NODEJS

Erick Alexander Lemus Morales

Asesorado por Ing. Juan Ernesto de León Pineda

Guatemala, noviembre 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

ADOPCIÓN DE UN ENTORNO DEVOPS UTILIZANDO GITLAB-CI, KUBERNETES Y SERVICIOS DE AWS PARA UNA APLICACIÓN WEB CON ANGULAR Y NODEJS

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

ERICK ALEXANDER LEMUS MORALES
ASESORADO POR ING. JUAN ERNESTO DE LEÓN PINEDA

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, NOVIEMBRE 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANA	Inga. Aurelia Anabela Cordova Estrada
VOCAL I	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Kevin Vladimir Armando Cruz Llorente
VOCAL V	Br. Fernando José Paz González
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANA	Inga. Aurelia Anabela Cordova Estrada
EXAMINADOR	Ing. Marlon Francisco Orellana López
EXAMINADOR	Ing. Edgar Estuardo Santos Sutuj
EXAMINADOR	Ing. Gabriel Alejandro Díaz López
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

ADOPCIÓN DE UN ENTORNO DEVOPS UTILIZANDO GITLAB-CI, KUBERNETES Y SERVICIOS DE AWS PARA UNA APLICACIÓN WEB CON ANGULAR Y NODEJS

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha 26 de agosto de 2021.

Erick Alexander Lemus Morales

Guatemala, 21 de octubre de 2022

Ingeniero
Carlos Alfredo Azurdia
Coordinador de Privados y Trabajos de Tesis
Escuela de Ingeniería en Ciencias y Sistemas
Facultad de Ingeniería - USAC

Respetable Ingeniero Azurdia:

Por este medio hago de su conocimiento que en mi rol de asesor del trabajo de investigación realizado por el estudiante **ERICK ALEXANDER LEMUS MORALES** con carné **201612097 y CUI 3186 71581 0501** titulado **"ADOPCIÓN DE UN ENTORNO DEVOPS UTILIZANDO GITLAB-CI, KUBERNETES Y SERVICIOS DE AWS PARA UNA APLICACIÓN WEB CON ANGULAR Y NODEJS",** luego de corroborar que el mismo se encuentra finalizado, lo he revisado y doy fé de que el mismo cumple con los objetivos propuestos en el respectivo protocolo, por consiguiente, procedo a la aprobación correspondiente.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

Ing. Juan Ernesto de León Pineda
Colegiado No. 7123

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala 25 de octubre de 2022

Ingeniero
Carlos Gustavo Alonzo
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Alonzo:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **ERICK ALEXANDER LEMUS MORALES** con carné **201612097** y CUI **3186 71581 0501** titulado **"ADOPCIÓN DE UN ENTORNO DEVOPS UTILIZANDO GITLAB-CI, KUBERNETES Y SERVICIOS DE AWS PARA UNA APLICACIÓN WEB CON ANGULAR Y NODEJS"** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo aprobado.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

LNG.DIRECTOR.238.EICCSS.2022

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del Asesor, el visto bueno del Coordinador de área y la aprobación del área de lingüística del trabajo de graduación titulado: **ADOPCIÓN DE UN ENTORNO DEVOPS UTILIZANDO GITLAB-CI, KUBERNETES Y SERVICIOS DE AWS PARA UNA APLICACIÓN WEB CON ANGULAR Y NODEJS**, presentado por: **Erick Alexander Lemus Morales**, procedo con el Aval del mismo, ya que cumple con los requisitos normados por la Facultad de Ingeniería.

"ID Y ENSEÑAD A TODOS"

Msc. Ing. Carlos Gustavo Alonzo
Director
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, noviembre de 2022

Decanato
Facultad de Ingeniería
24189101- 24189102
secretariadecanato@ingenieria.usac.edu.gt

LNG.DECANATO.OI.772.2022

La Decana de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al Trabajo de Graduación titulado: **ADOPCIÓN DE UN ENTORNO DEVOPS UTILIZANDO GITLAB-CI, KUBERNETES Y SERVICIOS DE AWS PARA UNA APLICACIÓN WEB CON ANGULAR Y NODEJS,** presentado por: **Erick Alexander Lemus Morales,** después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANA
FACULTAD DE INGENIERÍA

Inga. Aurelia Anabela Cordova Estrada
Decana

Guatemala, noviembre de 2022

AACE/gaoc

ACTO QUE DEDICO A:

Dios	Por ser quien ha guiado mi camino, quien cuida de mi familia y de mí en todo momento, y quien me ha hecho aprender cada día de mis errores para ser una mejor persona y profesional.
Mis padres	Brenda de Lemus y Antonio Lemus. Por ser quienes me han brindado tanto amor, y todo lo necesario para crecer y ser un hombre de bien. Siempre serán mi mayor inspiración.
Mis hermanas	Brenda, Saira, Wendy, Claudia (q. e. p. d) y Marely Lemus. Por ser parte fundamental en mi vida.
Mis hermanos	Antonio y Carlos Lemus, por todo el apoyo brindado, y ser parte fundamental a lo largo de mi carrera universitaria.
Mi abuela	Rosa Rodríguez. Por ser quien me ha brindado mucho amor desde que era niño.

Mi tía Elvira de Soto. Por ser una de las personas que más han creído en mí, y quien ha estado muy pendiente sobre cada paso que he dado.

Mi novia Ariana Pérez. Por creer en mí, apoyarme, animarme y darme tanto amor en todo momento. Agradezco a Dios porque la colocó en mi camino en el momento correcto de mi vida.

AGRADECIMIENTOS A:

Universidad de San Carlos de Guatemala	Por ser la institución que me ha visto crecer como persona y como futuro profesional.
Universidad Silesiana de Tecnología	Por haberme dado la oportunidad de estudiar en Polonia, como estudiante de intercambio del programa Erasmus+ KA107.
Facultad de Ingeniería	Por ser parte fundamental en mi formación como profesional, y por ser quien me ha brindado cada uno de los conocimientos que hoy he aprendido
Mi asesor	Juan Ernesto de León Pineda. Por brindarme consejos en el inicio de mi carrera universitaria que me fueron de ayuda en el camino, así como su asesoría y apoyo en la realización del presento trabajo de graduación.
Mi amigo	Raúl Reyna. Por brindarme su ayuda desde que llegué a Polonia. Estaré muy agradecido por el apoyo brindado y que hizo mi adaptación más sencilla.
Mi amigo	José Aguilar. Por brindarme su ayuda y apoyo para poder estudiar en Europa, que era un sueño desde que entre a la universidad.

Mi amigo

Vartan Babayan. Por ser un gran amigo con quien compartí numerosos momentos y viajes en Europa, durante el programa Erasmus.

Mis amigos

Herlindo Corona, Carlos García, Pablo Colindres y Henry Gálvez. Por ser con quienes hemos compartido innumerables momentos como risas, éxitos y fracasos; gracias a ellos por hacerme sentir el camino más corto de lo esperado

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

LISTA DE SÍMBOLOS

Símbolo	Significado
AWS	Amazon Web *Services*.
ACM	Amazon *Certificate Manager*.
CD	*Continuous Delivery* (Entrega Continua).
CI	*Continuous* Integration (Integración Continua).
EC2	Elastic *ComputeCloud*.
ECR	*Elastic Container Registry*.
ECS	*Elastic Container Service.*
EKS	*Elastic Kubernetes Service.*
JWT	JSON Web *Token*.
ORM	*Object Relational Mapping.*
RDS	*Relational Database Service.*
S3	*Simple Storage Service.*
SSL	*Secure Sockets Layer.*
TLS	*Transport Layer Security.*

GLOSARIO

AWS	Amazon Web *Services* (por sus siglas en inglés), es una plataforma en la nube, propiedad de Amazon.
ACM	Servicio de AWS que permite crear certificados SSL/TLS.
AWS-SDK	Kit de desarrollo de AWS.
Backend	Es quien se encarga de realizar todos los procesos lógicos y de negocio en una aplicación web.
Bucket	Es un espacio donde se almacenan archivos en el servicio de S3.
Commit	Función que permite confirmar los cambios dentro de una base de datos.
CD	Entrega continua.
Certificado SSL/TLS	Es un certificado digital que asegura la identificación de las aplicaciones web.
CI	Integración continua.

CLI	Interfaz de línea de comandos.
Contenedor	Es un paquete liviano que contiene únicamente lo necesario para poder ejecutar una aplicación, y que se ejecuta dentro de la plataforma de docker.
DevOps	Es el acrónimo de *Development* (Desarrollo) y *Operations* (Operaciones).
Docker	Es una plataforma open source, que es utilizada para crear contenedores.
Dominio	Nombre o identificador de una aplicación en la web.
EC2	Servicio de AWS que permite crear instancias o máquinas virtuales.
ECR	Servicio de AWS que permite almacenar contenedores de Docker.
ECS	Servicio de AWS que permite crear un orquestador de contenedores.
EKS	Servicio de AWS que permite crear y administrar una aplicación web en un clúster de Kubernetes.
Freelance	Es un trabajador independiente que realiza trabajos para terceros, de forma autónoma.

Frontend	Es la parte en la cual los usuarios finales interactúan con el sistema.
JWT	Estándar que permite crear un *token* para el intercambio de información de forma segura en la web.
Kubeconfig	Archivo que almacena los datos para conexión a un clúster de Kubernetes.
Kubernetes	Plataforma que permite administrar un automatizar de despliegue continuo y auto escalado.
ORM	Modelo que permite mapear las estructuras de una base de datos de tipo relacional.
Payload	Sección que corresponde a los nuevos datos que se utilizarán para modificar el estado de la aplicación en Redux.
Pipeline	Es la parte en la cual los usuarios finales interactúan con el sistema.
Rollback	Es una función que permite deshacer los cambios realizados en una base de datos.
S3	Servicio de AWS que permite crear un *bucket* para almacenar archivos.

Stage	Es la parte en la cual los usuarios finales interactúan con el sistema.
Template	Hace referencia a una plantilla.
Test	Pruebas sobre algún código de aplicación.
Quality Assurance	Departamento que se encarga de asegurar la calidad del *software*.

RESUMEN

El presente trabajo de graduación contiene las bases teóricas y prácticas para poder adoptar un entorno DevOps haciendo uso de tecnologías como Gitlab-Ci, AWS, Docker, Kubernetes, Angular y Node.js.

El primer capítulo contiene temas sobre DevOps y la importancia de su implementación en las organizaciones, así como la descripción de sus fases y algunas herramientas para cada una de estas.

En el segundo capítulo se encuentran generalidades de Docker, Microservicios y Kubernetes. En cada uno de estos se detallan aspectos importantes a considerar.

El tercer capítulo contiene temas de tecnologías de la nube, las opciones disponibles (más importantes hasta el momento) y los servicios de AWS que tienen relación con el presente documento.

El cuarto capítulo habla sobre el desarrollo de aplicaciones web con frameworks basados en Javascript. Acá se encuentran detalles de Angular y Node.js.

En el quinto capítulo se describe cómo implementar e integrar cada herramienta tecnológica, con el fin de adoptar la metodología DevOps en un equipo de desarrollo ágil.

OBJETIVOS

General

Elaborar una propuesta para guiar, a todos los equipos de desarrollo, departamentos de IT y empresas pequeñas, medianas y grandes, a adoptar un entorno DevOps con herramientas específicas, buscando la mayor optimización y el uso de las últimas tecnologías que se encuentran en el mercado, ante el incremento exponencial del uso de aplicaciones web desplegadas en la nube. Cabe destacar que es importante la automatización de los procesos que conciernen a un ciclo de vida, con el objetivo de mejorar la manera de desarrollar y desplegar aplicaciones.

Específicos

1. Adoptar la metodología DevOps como un marco de trabajo específico dentro de un equipo de trabajo u organización.

2. Automatizar los procesos de ciclo DevOps a través del uso de la herramienta Gitlab-Ci.

3. Conocer los beneficios que provee Amazon Web Services al momento de utilizar los servicios ofrecidos.

4. Conocer la importancia del uso de microservicios con Kubernetes y AWS.

5. Aprender a utilizar los servicios de bases de datos en la nube, Kubernetes, almacenamiento de objetos y reconocimiento facial de AWS.

6. Desplegar una aplicación web con Angular con Kubernetes y AWS.

7. Dividir una aplicación monolítica, creada con Nodejs, en diferentes microservicios y desplegarlos haciendo uso de Kubernetes y AWS.

8. Proporcionar una guía final, para la adopción del entorno DevOps e implementación de las tecnologías de las tecnologías mencionadas.

MARCO TEÓRICO

- Origen de Devops:

La historia de esta metodología se remonta en el año 2007, con Patrick Debois (quien era un consultor *freelance*) como primer actor. Todo inicia durante la realización de un proyecto en Bélgica, el cuál consistía en migrar una *data center* (o centro de datos). Durante la ejecución de este, Debois se da cuenta de un gran problema; conflictos entre desarrolladores y administradores de operaciones, lo que lo llevó a analizar el departamento de IT y su cadena de valor.

En la Agile2008 *Conference*, se habla sobre el tema Infraestructura *Agile*, que era impartido por Andrew Clay Shafer. En este mismo evento, Debois es el único que atiende. Shafer, al ver que no existía interés, decide saltar el tema. Más adelante, al finalizar Shafer, Debois se acerca a él y hablan sobre este tema, además de otros como conflicto entre desarrollo y operaciones. A partir de ello, ambos crean un grupo llamado *Agile Systems Administrators*, con el objetivo de debatir y obtener conclusiones que serían de gran ayuda para formalizar el término DevOps.

Llegado el año 2009, O'Reilly organiza la *Velocity* 09 *Conference*, en el cual se da a conocer el tema 10 *Deploy per Day*, que tenía como objetivo principal asegurar que Desarrollo y Operaciones trabajen juntos haciendo uso de buenas prácticas y metodologías ágiles. Es en ese momento que Patrick Debois toma los conceptos abordados y decide crear un evento llamado DevOpsDays (DOD). Dado que temas de Integración Continua y Entrega Continua estaban tomando

fuerza dentro del ambiente *Agile*, llegan a tener su participación dentro del marco DevOps.

Dado que el tema empezó a ser de interés, grandes empresas como IBM y HP inician a aplicar el concepto.

Cabe destacar que, según Debois, el nombre no se plasmó como título final, sino que se obtuvo de la contracción, dando como resultado DevOpsDays, y por consiguiente DevOps.

- Docker y contenedores:

Era el año 2000, cuando se conoció, por primera vez, los contenedores. Este primer acercamiento, se dio gracias a FreeBSD Jail, que permitía dividir un sistema FreeBSD en diferentes subsistemas.

Posteriormente, en el sistema operativo Linux, durante el año 2001, inició la implementación de subsistemas o entornos separados (uno del otro), esto a través del proyecto llamado VServer. Este proyecto consistía en que cada entorno o partición del sistema, pudieran compartir los mismos recursos, sin afectarse los unos con los otros. A pesar de que cada partición tenía una dependencia con un sistema principal, eran independientes en sus propios procesos. Es así como surgieron las bases de contenedores, que hoy en día son muy utilizados.

Aunque el concepto de Docker haya nacido muchos años atrás, este se formalizó por la empresa francesa dotCloud, la cual tenía como fortaleza PaaS (*Platform as a Service* o Plataforma como Servicio), servicios de alojamiento, entre otros. A lo largo de las investigaciones, se empezar a agregar

características como los CGroups, dentro del *kernel* de Linux, que tiene como función principal controlar el uso de los recursos del sistema para un determinado proceso o diferentes grupos. Gracias a los diferentes cambios y nuevas características implementadas, no solo a nivel del *kernel*, se procedió a crear un estándar para poder simplificar el uso de la característica Linux *Container* (LXC).

Por lo tanto, esta API que permitía construir contenedores, fue nombrado Docker, y no fue hasta el año 2013 que dotCloud lo hizo público en la Pycon (Python *Conference*), para que después fuese liberado como *open source* o código abierto, para que la comunidad pudiera iniciarse en el mundo de la virtualización.

- Amazon Web *Services* (AWS):

En sus inicios, Amazon se dio a conocer en el mercado de libros (convirtiéndose en el líder), que a través de los años fue incluyendo nuevos servicios, hasta llegar a ser lo que hoy se conoce en el comercio electrónico. Llegado el año 2003, Jeff Bezos, Jeff Barr, Andy Jassy y Al Vermeulen se reúnen en casa del primero. En esta reunión comienzan a enlistar la lluvia de ideas que han tenido, que años más tarde, se convertiría en lo que conocemos como Amazon Web *Services* (AWS).

No es hasta el año 2006, cuando Amazon lanza de manera oficial AWS, que es una plataforma en la nube objetiva y dedicada para desarrolladores. Cabe destacar que Amazon migró su infraestructura hacia servidores Linux, con el objetivo de reducir los costos hasta un 80%. En 2009, AWS arrasó por completo con el mercado de servicios en la nube, muy por encima de su más cercano competidor que era Oracle.

Pasados los años, AWS empezó a tener competidores mayores, como lo son Google Cloud Platform y Microsoft Azure. A pesar de la competencia generada por estos grandes de la tecnología, AWS sigue estando muy por encima de ellos, lo que lo convierte como líder indiscutible en el mercado de *Cloud Computing*.

Hoy en día, AWS ofrece diferentes servicios a los desarrolladores, resaltando su plataforma como Infraestructura como Servicios (IaaS), lo que permite generar escalabilidad a bajos costos, teniendo presencia en 190 países en el mundo.

- *Frameworks* y el desarrollo web:

Cuando se habla de aplicaciones web, se puede encontrar muchas maneras de desarrollar este tipo de *software*, entre los cuales se tiene la posibilidad de conocer diferentes tecnologías; desde PHP, JavaScript, Python, hasta llegar a los tan famosos *Frameworks*, que hoy en día son muy utilizados en diferentes organizaciones y que son categorizados como un estándar dentro de ellas.

Pero ¿por qué utilizar un *Framework* para desarrollar una aplicación web? Este tipo de tecnologías permite a los desarrolladores crear aplicaciones que puedan ser escalables, de la misma manera que proveen diferentes herramientas (completamente evaluadas) que permiten construir, ejecutar y mantener el código generado, de manera muy sencilla.

Muchos de estos *Frameworks*, ya incluyen herramientas que el desarrollador puede utilizar para automatizar diferentes procesos; tales como pruebas unitarias, pruebas de integración y pruebas e2e (*end to end*).

Ahora bien, conociendo los alcances que pueden proveer este tipo de tecnologías ¿qué herramientas se encuentran disponibles para hacer uso de ellas? Existen muchas herramientas para cada tecnología en específico, por ejemplo; en PHP, se encuentra Laravel; en JavaScript, React o Angular para desarrollo *Frontend*, y Nodejs para desarrollo *Backend*; mientras que, en Python, tenemos Django y Flask.

Cada tecnología es utilizada conforme a las necesidades que pueden surgir a lo largo de un proyecto; mientras que, en otros casos, se han estandarizado tanto en el desarrollo web, que es importante conocerlos en la actualidad.

Aun así ¿son los *Frameworks* la mejor solución para un proyecto de *software*? Esto realmente dependerá del alcance y la complejidad del proyecto, así como del gusto o facilidad que pueda sentir la organización, o equipo, al hacer uso de estos.

INTRODUCCIÓN

En la actualidad, el mundo de la tecnología crece a grandes escalas debido a la gran demanda que existe por parte de las empresas y organizaciones. Sin embargo, muchas de estas organizaciones requieren que el desarrollo de sus aplicaciones sea lo más rápido posible, con el objetivo tener disponibilidad de sus servicios en todo momento.

Aquí es donde entra en juego la pregunta ¿Cómo seguir desarrollando un *software* de calidad sin detener la aplicación, darle valor continuamente al mismo de manera automatizada utilizando diferentes herramientas? Para poder solventar la problemática planteada, existen diferentes metodologías ágiles y marcos de trabajo que se pueden implementar dentro del flujo de desarrollo del *software* para mejorar los avances del desarrollo.

Algunas metodologías ágiles, como Scrum y Kanban, permiten darle continuamente valor al *software*, pero en sus prácticas no permiten automatizar muchos procesos necesarios como pruebas, entrega y despliegue continuo, ya que está definido como implementarlos.

Una solución para el planteamiento inicial es la adopción de DevOps, que es una metodología que permite agilizar el proceso de desarrollo, además que define diferentes fases que permite automatizar muchos procesos repetitivos, implementando diferentes herramientas que, sin duda alguna, acelerará el flujo de desarrollo, pruebas, entrega y despliegue continuo del *software*.

1. DEVOPS, UNA METODOLOGÍA PARA LA AUTOMATIZACIÓN

1.1. ¿Qué es DevOps?

DevOps se define como una metodología que ha venido a revolucionar el marco de desarrollo de aplicaciones, gracias a la aplicación de diferentes prácticas y herramientas de automatización que permiten producir *software* más rápido. Además de lo mencionado, esta metodología permite la unificación de Desarrollo y Operaciones, lo que facilita el trabajo y la distribución la infraestructura aplicativa.

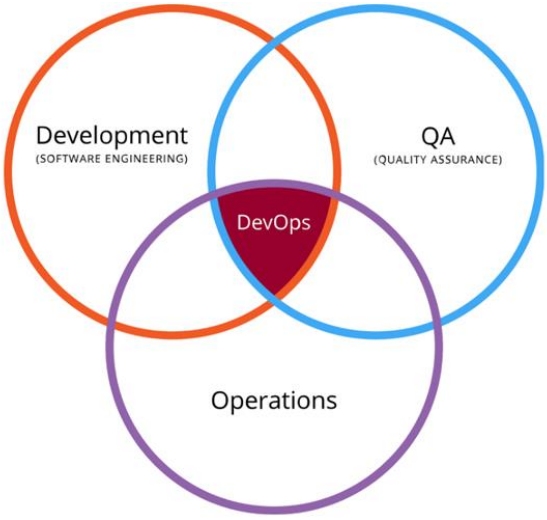
1.2. ¿Por qué la necesidad de DevOps?

Hay que decir que la necesidad de los clientes es lo más importante, ya que estas se ven en aumento conforme un proyecto se encuentra en desarrollo. Además de lo anterior, la parte de desarrollo y operaciones, pueden convivir en un mismo proyecto, solucionando cada una de las necesidades requeridas por ambas partes, y para el cliente principalmente.

En algunos casos, las necesidades de los clientes cambian y evolucionan con tanta frecuencia, lo que hace que las empresas se encuentren en el dilema de elegir entre: cambios rápidos en un entorno inestable o entorno rancio, pero estable. Y es en estos casos que DevOps es la solución, ya que permite crear un sistema y aplicar prácticas, que pueden equilibrar cada una de las partes.

Al adoptar un entorno DevOps, no solo se equilibra desarrollo y operaciones, sino que también se encuentra un equilibrio con QA *(Quality Assurence*, por sus siglas en inglés), lo que permite controlar la calidad del *software* desarrollado, dentro del mismo entorno.

Figura 1. **Equilibrio entre Desarrollo, Operaciones y QA**



Fuente: Sánchez (2021), *Introducción a DevOps y cómo implementarlo*. Consultado 7 de septiembre 2021. Recuperado de https://tech.tribalyte.eu/blog-introduccion-devops.

Tal como se puede observar en la figura 1, gracias al entorno DevOps, se puede mantener un equilibrio entre estos tres importantes grupos durante el desarrollo de una nueva aplicación, lo que es importante dentro de un equipo.

1.3. Beneficios de la adopción de un entorno DevOps

Esta metodología no solo permite una mejor colaboración entre distintos grupos y rapidez de desarrollo, sino que también implica diferentes beneficios

que hacen el producto sea de mayor calidad, y que el equipo tenga mayor productividad a la hora de producir *software.*

Cabe destacar, que DevOps fomenta la colaboración, comunicación e integración, lo que hace que el proceso de producción sea más ágil. Estos son solo algunos de los beneficios que ofrece la adopción de DevOps, pero existen muchos más, tales como:

- Mayor velocidad de producción.
- Entrega rápida.
- Seguridad.
- Escalabilidad.
- Satisfacción del cliente.
- Mayor calidad de *software*.
- Estabilidad del *software*.

Estos beneficios son solo algunos de los más importantes observados al momento de adoptar un entorno DevOps, pero se pueden agregar más conforme se vayan utilizando las herramientas y aplicando las prácticas que ofrece esta metodología.

1.4. Fases del ciclo DevOps

DevOps no solo es una metodología de desarrollo de *software*, sino que también es un ciclo en el cual se puede ir aplicando cada una de las prácticas que permita desarrollar *software* de manera rápida, fiable y escalable.

A lo largo de este ciclo, se pueden aplicar herramientas que permitan automatizar ciertos pasos, lo que agiliza el proceso de desarrollo, entrega y despliegue de una aplicación.

Por lo tanto, DevOps se caracteriza por tener cierta cantidad de fases que permiten desarrollar *software* continuamente, y que se inicia al finalizar el último. Estas fases son:

1.4.1. Planificación

Como primera fase del ciclo, es importante definir cada una de las ideas para construir el *software* requerido por el cliente, además se especifica el alcance, características y capacidad que tendrá el *software* para poder resolver las necesidades por las cuales se está desarrollando.

En esta fase también es importante definir las herramientas que se han de utilizar, con el objetivo de definir las tareas, definir plazos de entrega, en algunos casos considerar la capacitación de algunos miembros del equipo con dichas herramientas, entre otros.

1.4.2. Codificación

En esta fase, se desarrolla el código y diseño necesario para poder satisfacer las necesidades de los clientes. Esto se lleva a cabo en repositorios de control de versiones, y en los cuales se realizan cambios continuamente.

Algunas de las herramientas de control de versiones más utilizadas dentro de la fase de codificación son Github, Gitlab y Bitbucket.

1.4.3. Construcción

Es en esta fase donde se lleva a cabo el control de las nuevas versiones del *software* producido, así como la compilación del código generado.

Además de realizar lo anterior, se hace uso de herramientas que permitan automatizar los procesos que son repetitivos, tanto para genera las nuevas versiones, compilar, entre otros. Entre las herramientas más utilizadas podemos encontrar Docker, Kubernetes, Ansible, Fabric, Maven, entre otros.

La mayoría de las herramientas mencionadas, hacen uso de un repositorio de control de versiones, donde toman el código, y generan sus procesos de manera automatizada, con el objetivo de agilizar el proceso de entrega continua

1.4.4. Pruebas

Tal como lo dice su nombre, esta fase se encarga de verificar la calidad del *software*, a través de la realización de pruebas continuas (la mayoría de las veces automatizadas).

En esta fase se pueden realizar diferentes tipos de pruebas, tales como.

- Pruebas unitarias.
- Pruebas de Integración.

Existen diferentes herramientas que permiten desarrollar este tipo de pruebas, y estas dependen del lenguaje que se esté utilizando en el desarrollo del proyecto. Algunos ejemplos son: Selenium, Mocha, Jasmine, Junit, Karma, entre otros.

Cabe destacar, que estas herramientas devuelven el estado de una prueba, lo que permite verificar la calidad y rendimiento de una porción del código, o del sistema completo.

1.4.5. Lanzamiento

Durante esta fase se inicia el proceso de integración de todo el código existente, y se realiza inmediatamente después de que el nuevo código generado pasa todas las pruebas a las que ha sido expuesto.

Además, se debe asegurar que, al integrar las nuevas funcionalidades, esta no afecte el funcionamiento del resto de la aplicación, al momento de ser puesto en marcha en producción.

De igual manera, existen diferentes herramientas, como Kubernetes, Ansible y Fabric, que permiten gestionar y lanzar la nueva versión.

1.4.6. Operación

Esta fase se realiza en los entornos de producción, y se debe garantizar el funcionamiento adecuado de la aplicación, así como el mantenimiento y resolución de problemas.

En el caso de que exista algún problema, automáticamente se debe realizar un *rollback* de la versión, hasta encontrar y reparar el fallo de la o las funcionalidades que están afectando. Ante esto, se deben tomar medidas e implementarlas antes de que se devuelva nuevamente al entorno de producción.

Es importante considerar que, para los sistemas de alta disponibilidad, no se debe ver afectado durante la puesta en marcha del *rollback* o del nuevo *commit* que arregla los errores y *bugs* del sistema, además se debe garantizar la seguridad al momento de la realización de estos.

1.4.7. Monitoreo

Esta es la última fase del ciclo, y se centra en evaluar el comportamiento del sistema, a través de métricas de interés que se definen como importantes y de gran importancia para realizar observar la aplicación.

Al monitorizar, se recopila la información necesaria, que permite evaluar el rendimiento y estado de la aplicación, lo que permite realizar mejoras continuas al sistema.

También existen herramientas que permiten obtener este tipo de métricas para diferentes partes del *software*. Entre estas podemos encontrar Prometheus y Grafana, Linkerd, Nagios, entre otros. El uso de estas herramientas queda completamente ligado a las herramientas de desarrollo utilizados a lo largo del ciclo.

Figura 2. **Fases del ciclo DevOps**

Fuente: Sánchez (2021). *Introducción a DevOps y cómo implementarlo.* Consultado 7 de septiembre 2021. Recuperado de https://tech.tribalyte.eu/blog-introduccion-devops.

1.5. Conceptos de CI/CD

CI (Integración Continua) y CD (Entrega Continua) son dos conceptos que se han unificado en un solo término (CI/CD), y que tienen en común automatizar procesos repetitivos para evitar gastos operacionales, y entre otros temas, con el objetivo de agilizar los procesos de desarrollo

1.5.1. Integración Continua (CI)

La integración continua, es el proceso de integración de código de una rama diferente, a la rama principal del repositorio de versiones. En este proceso se encuentran pruebas automatizadas (unitarias y de integración), que se ejecutan al momento de fusionar los nuevos cambios a la rama de desarrollo principal.

1.5.2. Entrega Continua (CD)

La entrega continua es el proceso de despliegue de manera automatizada del *software* con las nuevas funcionalidades adoptadas y funcionales. Este despliegue se realiza en el entorno de producción que se posee, y se realiza con diferentes tipos de herramientas que automatizan el proceso, reduciendo, de esta manera, el tiempo de espera para disponer de los nuevos cambios realizados al *software*.

1.6. ¿Qué es la cultura DevOps, y qué implica?

La cultura DevOps es una parte fundamental dentro de la adopción de un entorno DevOps. Dentro de este, se engloban distintas prácticas y valores que le dan sentido, objetivo y valor agregado al proceso de desarrollo, tal como el trabajo y colaboración entre equipos.

Al adoptar esta cultura, se debe considerar:

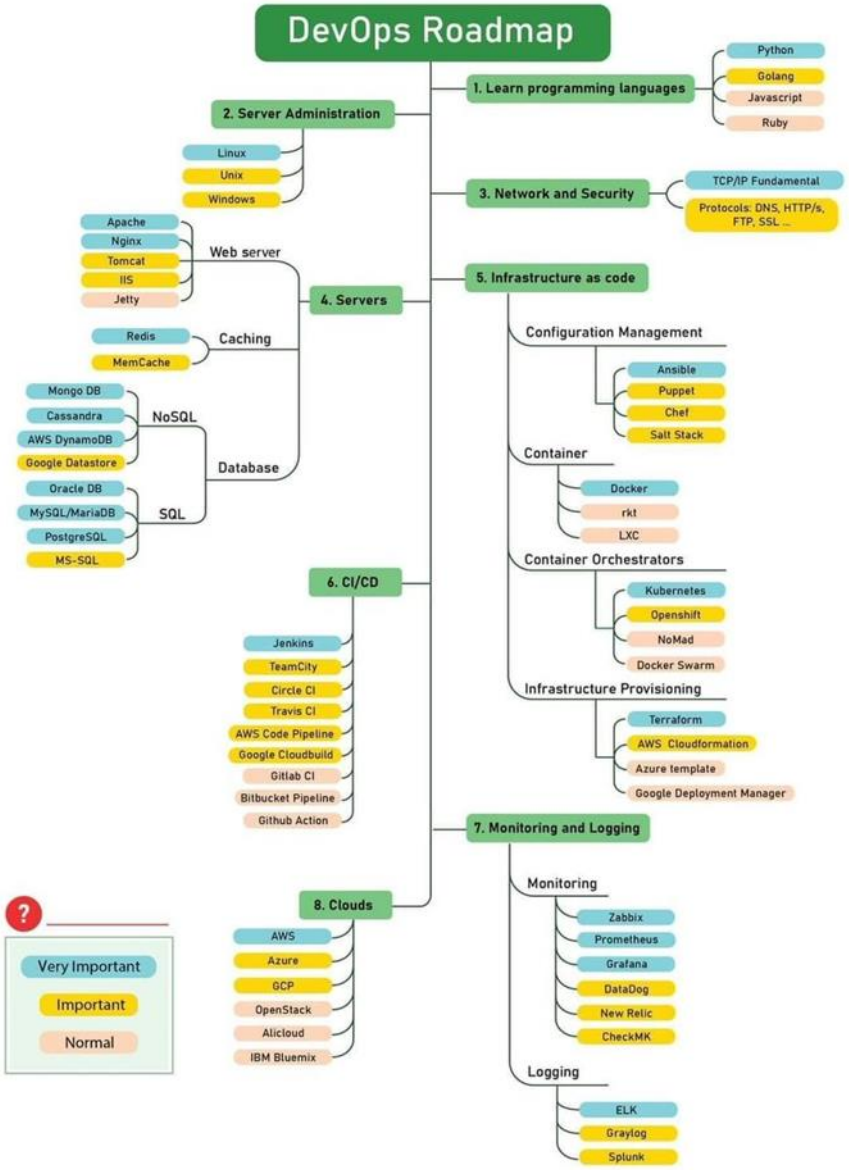
- Ciclos de Despliegue cortos
- Rendir cuentas
- Visibilidad, colaboración y alineación.
- Mentalidad de crecer.

Este proceso, es un completo compromiso, que requiere que el equipo este enfocado para que el proceso de desarrollo sea altamente productivo.

1.7. Ruta de aprendizaje de la cultura DevOps

Es importante conocer que ruta debemos seguir para adquirir un nuevo conocimiento y habilidad. En DevOps no es la excepción, ya que tenemos una amplia gama de caminos y tecnologías, que podríamos seguir y aplicar. En la Figura 3 se muestra una ruta de aprendizaje que se puede utilizar como guía:

Figura 3. **Ruta de aprendizaje de Devops**



Fuente: Twitter (2021). *DevOps.* Consultado 8 de septiembre 2021. Recuperado de https://twitter.com/orhanergunccde/status/1349700336609615875.

1.8. ¿A quién va dirigido DevOps, y para qué?

Hoy en día, DevOps está dirigido y es utilizado por diferentes grupos u organizaciones, entre las cuales se pueden mencionar:

- Empresas con departamento de IT.
- Facebook, Amazon, Netflix.
- Sony Pictures.
- Organizaciones estadounidenses, para el gobierno.
- Empresas pequeñas, mediana y grandes.
- Equipos de desarrollo individuales o *freelancer*.

Cabe destacar, que DevOps es utilizado para agilizar el proceso de desarrollo de *software*, además para automatizar y gestionar de mejor manera los recursos de los que se disponen.

1.9. Herramientas DevOps

Para poder gestionar un entorno DevOps de la mejor manera, existen diferentes herramientas que permiten realizar diferentes acciones, lo que facilita el manejo de los proyectos y los recursos.

Para ello, se tienen las siguientes herramientas, divididas por su finalidad:

1.9.1. De planificación

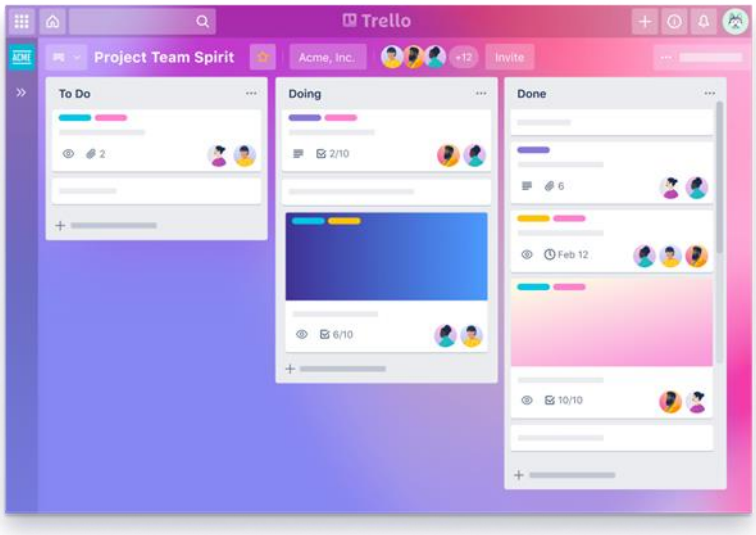
Estas herramientas son utilizadas para planificar un ciclo de desarrollo. En estas se pueden plasmar las ideas y trazar las metas que se desean alcanzar al finalizar el período. Existen diferentes herramientas, pero las más utilizadas son:

1.9.1.1. Trello

Es una herramienta, que permite gestionar la panificación de un proyecto, a través de tableros y tarjetas en los cuáles se asignan tareas y plazos dentro del calendario. Además, las tarjetas plasmadas en los tableros permiten compartir el estado de una tarea, desde su inicio, hasta el final.

Trello también posee diferentes características, que cada equipo de desarrollo puede descubrir y utilizar conforme a sus necesidades.

Figura 4. **Tablero y tarjetas en Trello**



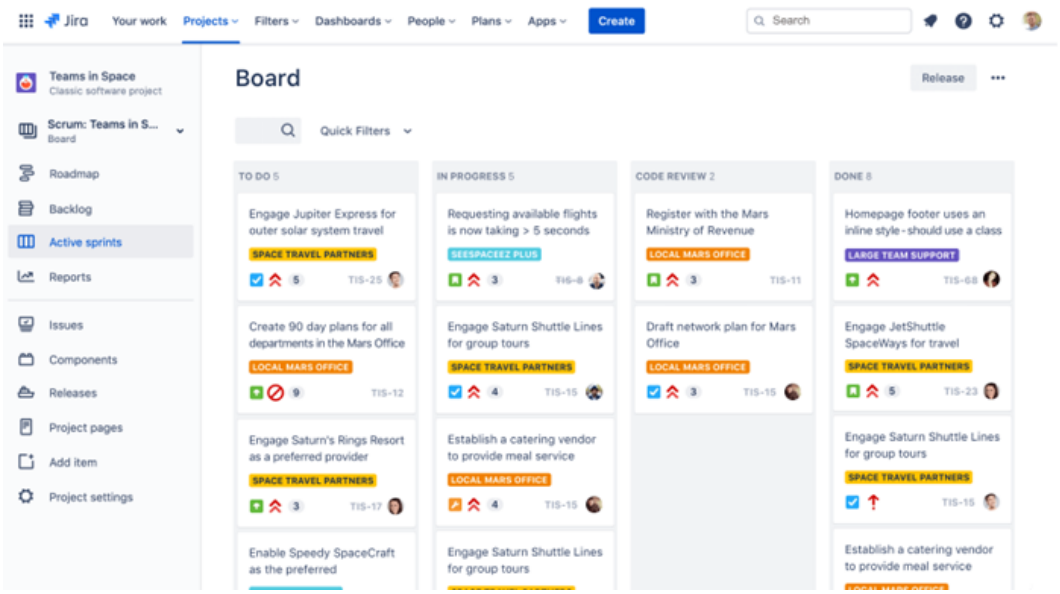
Fuente: Trello (2021). *Tablero en trello.* Consultado 10 de septiembre 2021. Recuperado de https://trello.com/es.

1.9.1.2. Jira

Es una herramienta muy similar a la anterior, con la diferencia que permite generar informes del equipo de desarrollo, con el objetivo de mejorar el rendimiento del proceso de desarrollo.

También, permite elegir un flujo de trabajo, el cuál es utilizado por el equipo de desarrollo para producir el *software*.

Figura 5. **Tablero y tarjetas en Jira**



Fuente: Atlassian (2021). *Tablero en Jira.* Consultado 10 de septiembre 2021. Recuperado de https://www.atlassian.com/es/software/jira.

1.9.2. Control de versiones

Un sistema de control de versiones permite almacenar y controlar el código fuente de una aplicación. Algunos de los sistemas de control de versiones son:

1.9.2.1. Gitlab

Es una de las herramientas que se encuentra en constante crecimiento y que actualmente es una de las plataformas de control de versiones más completas en la actualidad. Cuenta con diferentes servicios, entre los cuales se pueden encontrar Gitlab-Ci para gestionar un entorno DevOps y automatizar, además de servicios como Kubernetes y su propio *Container Registry* (Registro de contenedores). Además, tiene su propio gestor para DevOps, conocido como Gitlab *Runners*, que son utilizados para comunicar diferentes máquinas virtuales.

1.9.2.2. Github

Esta es una herramienta de las más utilizadas y famosas de los últimos años. Es muy común escuchar su nombre cuando se inicia en el mundo de control de versiones, ya que es de las primeras herramientas que se utilizan. Actualmente, se encuentra en manos de Microsoft, quienes están iniciando a hacer mejoras al sistema. Además, se están implementando nuevas características, tales como su propio sistema de CI/CD, que es conocido como Github *Actions*.

1.9.3. De automatización

Este tipo de herramientas es utilizado para automatizar, empaquetar, y luego desplegar aplicaciones de manera continua. Existen diferentes

herramientas, pero hay una en especial que es muy utilizada por millones de desarrolladores alrededor del mundo, por su facilidad y flexibilidad.

1.9.3.1. Docker

Es una herramienta de código abierto, que es muy utilizado para construir, empaquetar y desplegar las aplicaciones de manera rápida, continua y confiable. Además de ser multiplataforma, se adapta fácilmente al trabajo distribuido.

También, cabe resaltar, que es fácilmente adaptable a plataformas de la nube, tales como Azure, Amazon Web *Services* y Google *Cloud Platform*.

Por último, es importante mencionar, que es fácil de integrar a herramientas como Gitlab-Ci, Github, Ansible, Fabric, entre otros.

1.9.4. *Pipeline*s (CI/CD)

Una herramienta de *Pipeline* permite gestionar de mejor manera la Integración y Despliegue Continua de las aplicaciones, dado que se puede agregar un conjunto de instrucciones que automatizará diferentes procesos que conciernen a estas diferentes acciones. Algunas de las herramientas son:

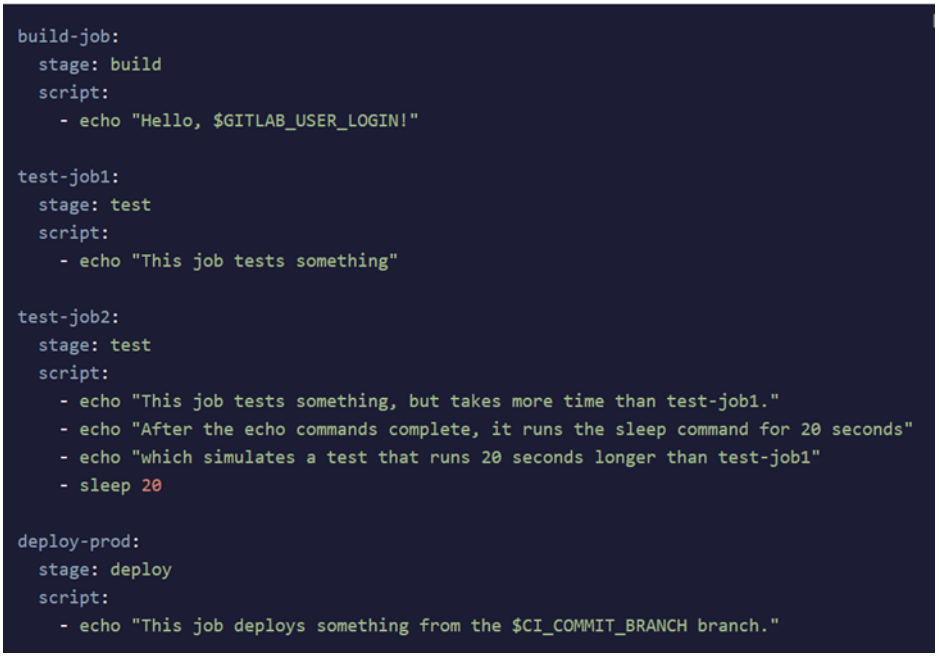
1.9.4.1. Gitlab-Ci

Esta es una moderna herramienta, creada por Gitlab, que tiene como objetivo automatizar todos los procesos involucrados en el ciclo devops, utilizando el repositorio de control de versiones de su misma propiedad. Utiliza el lenguaje YAML, por lo que facilita especificar cada uno de los *stages* y tareas que debe ejecutar.

Además, que facilita la integración con cualquier herramienta, ya que permite hacer *pre-scripts*, y los *scripts* principales.

Una de sus mayores virtudes, es que es fácil de integrar con los servicios que también tiene Gitlab, ya que este actualmente posee una amplia gama de diferentes servicios para los desarrolladores. De igual manera, es fácil de configurar con proveedores como AWS, Azure y GCP.

Figura 6. **Ejemplo de *pipeline* en Gitlab-Ci**



```
build-job:
  stage: build
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

test-job1:
  stage: test
  script:
    - echo "This job tests something"

test-job2:
  stage: test
  script:
    - echo "This job tests something, but takes more time than test-job1."
    - echo "After the echo commands complete, it runs the sleep command for 20 seconds"
    - echo "which simulates a test that runs 20 seconds longer than test-job1"
    - sleep 20

deploy-prod:
  stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
```

Fuente: Gitlab (2021). *Popeline en Gitlab-Ci.* Consultado 12 de septiembre 2021. Recuperado de https://docs.gitlab.com/ee/ci/quick_start/.

1.9.4.2. Jenkins

Esta es una de las herramientas más conocidas, y de las más antiguas en lo que concierne a CI/CD. Una de las principales características, es que permite crear nodos, que se conectan a los servidores en la nube, y que pueden ser configurados como maestro esclavo. Cabe destacar, que esta herramienta es compatible con muchas herramientas antiguas y nuevas.

El archivo de configuración se llama Jenkinsfile, y tiene su propio lenguaje de especificación, donde se definen cada uno de los *stages* y *scripts* que se desean implementar. Además, que permite elegir con que nodo se desea ejecutar las instrucciones actuales.

Figura 7. **Ejemplo de *pipeline* en Jenkins**

Fuente: Jenkins.io (2020). *Pipeline en Jenkins.* Consultado 12 de septiembre 2021. Recuperado de https://www.jenkins.io/doc/book/pipeline/.

1.9.5. Monitorización

Hoy en día, es importante monitorizar los sistemas, ya que permiten evaluar el compartimiento de estos a partir de métricas definidas, que facilitan la toma de decisiones para la mejora continua del *software*. Algunas de estas herramientas son:

1.9.5.1. Prometheus

Esta es una herramienta que permite obtener métricas de casi cualquier cosa que se necesite, desde bases de datos, *frameworks*, hasta herramientas que se disponen en la nube, como máquinas virtuales. Estas métricas se recopilan y se pueden utilizar en otras herramientas para visualizarlas de manera gráfica.

1.9.5.2. Grafana

Es una herramienta que permite mostrar las métricas obtenidas con Prometheus. Así mismo, permite crear *dashboard*s, o utilizar los que ya se encuentran disponibles para mostrar los datos.

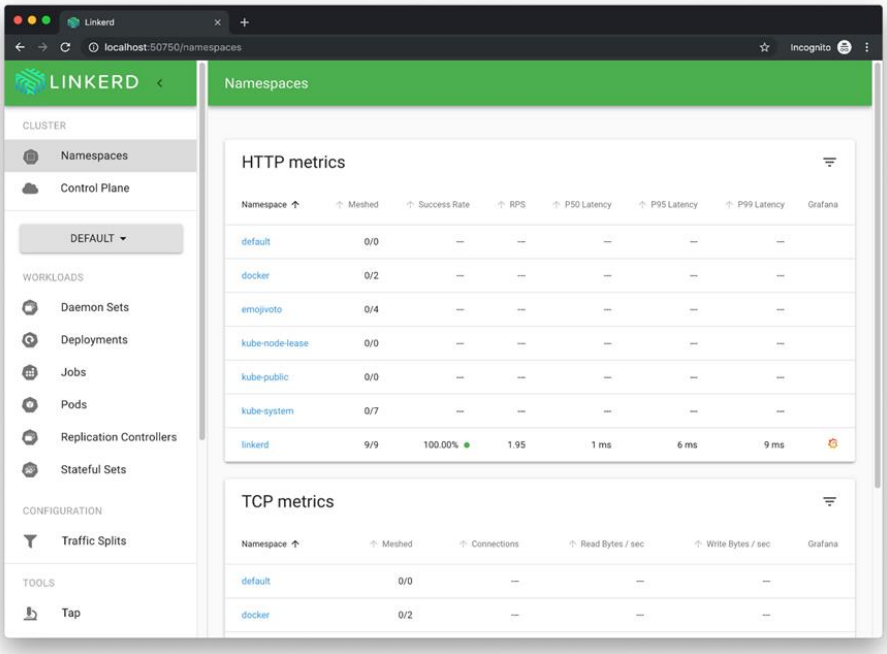
Figura 8. ***Dashboard* en Grafana para visualizar métricas**

Fuente: Grafana (2020). *Dashboard en Grafana.* Consultado 12 de septiembre 2021. Recuperado de https://grafana.com/.

1.9.5.3. Linkerd

Esta herramienta de monitoreo está dirigida para Kubernetes, ya que esta se instala directamente en el clúster implementado, lo que permite evaluar el comportamiento de los *pods*, *deployments*, *services*, *namespaces*, entre otros. Este ya tiene integrado grafana, por lo que no es necesario utilizarlo por aparte para visualizar las métricas obtenidas.

Figura 9. ***Dashboard* en Linkerd para visualizar métricas**



Fuente: Storti (2020). *How to install and use Linkerd with Kubernetes.* Consultado 12 de septiembre 2021. Recuperado de https://www.digitalocean.com/community/tutorials/how-to-install-and-use-linkerd-with-kubernetes.

1.9.6. De pruebas

Son utilizadas para generar entornos de prueba, y poder ejecutar las diferentes pruebas que se permiten, unitarias o de integración. Entre estas se pueden encontrar:

1.9.6.1. Selenium

Esta herramienta permite generar pruebas automatizadas de integración a las aplicaciones basadas en un navegador (mejor conocidas como *frontend*), a través de su *WebDriver* que se utiliza en un servidor en la nube. También se pueden crear para pruebas locales.

1.9.6.2. Mocha

Es una herramienta de pruebas para JavaScript que se ejecuta en Nodejs. Permite crear pruebas automatizadas, unitarias y de integración, que se ejecutan de manera síncrona, lo que ayuda a evaluar cada prueba en el orden que se han descrito.

1.9.6.3. Jasmine

Esta herramienta es muy similar a Mocha, con la diferencia que, al ser una herramienta con más antigüedad, tienen aserciones o *expects* que pueden ser utilizados. Además, dispone de su propia elaboración de *Mocks* y *Spies*, para generar pruebas a datos, o funciones *fake*.

2. CONTEINERIZACIÓN DE UNA APLICACIÓN WEB

2.1. Conceptos generales

A continuación, se describen los conceptos más importantes a considerar al momento de trabajar con Docker y Kubernetes.

2.1.1. Virtualización ligera

También conocida como Virtualización a Nivel de Sistema Operativo, es un tipo de virtualización que permite aislar, en diferentes espacios, múltiples instancias, viéndose cada una de estas como un servidor. Este tipo de virtualización mejora en velocidad y rendimiento a las tradicionales máquinas virtuales que se basan en un hipervisor.

2.1.2. ¿Qué es un contenedor?

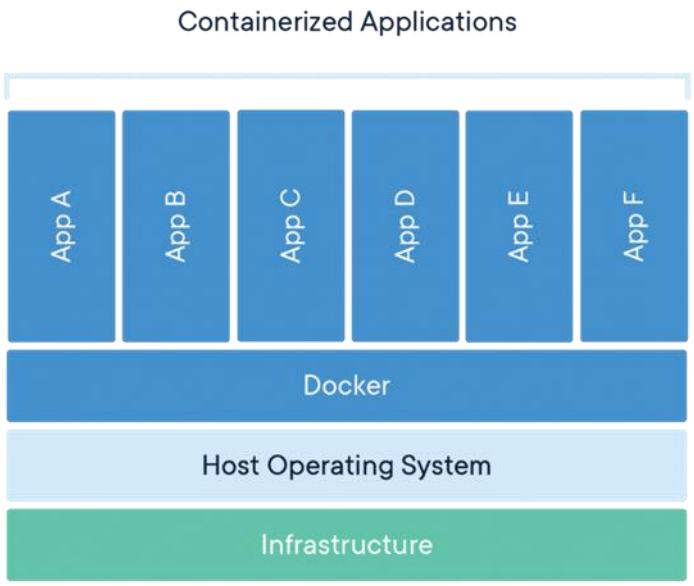
Es una ligera unidad de *software*, que contiene dependencias y código ejecutable de una aplicación. Además, posee todo lo necesario para que la aplicación se ejecute de manera más rápida. Este paquete es independiente y ligero, ya que únicamente contiene los archivos necesarios para el entorno de ejecución.

2.1.3. ¿Qué es Docker?

Docker es un *software* que permite crear, y desplegar aplicaciones basadas en contenedores. Docker puede ejecutar aplicaciones

independientemente del sistema operativo, ya que únicamente se necesita tener instalado este *software*. Esto supone una gran ventaja al momento de crear aplicaciones multiplataforma.

Figura 10. **Vista de la arquitectura de Docker**



Fuente: Docker (2021). *What is a container?* Consultado el 3 de octubre de 2021. Recuperado de https://www.docker.com/resources/what-container.

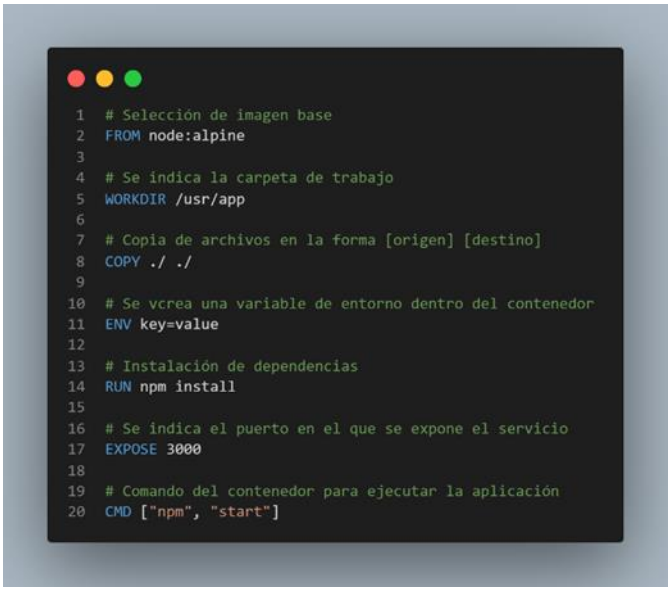
2.2. Dockerfiles

Es un archivo de texto plano, el cual contiene las instrucciones necesarias para poder generar una nueva imagen, en base de otra. En este archivo, se tienen instrucciones para copiar, correr y exponer puertos en los cuales se puede acceder a los recursos de estos. El archivo no tiene extensión, únicamente se llama Dockerfile.

2.2.1. Estructura de un Dockerfile

Un Dockerfile tiene una estructura que es utilizada con frecuencia para poder crear nuevas imágenes. El contenido de este archivo puede variar, dependiendo de las necesidades de la aplicación.

Figura 11. **Estructura de un archivo Dockerfile**



Fuente: elaboración propia, realizado con Visual Studio Code.

2.3. Docker-Compose

Es una tecnología muy utilizada junto a Docker, ya que facilita diferentes acciones. A continuación, se encuentra información con mayor detalle sobre esta tecnología.

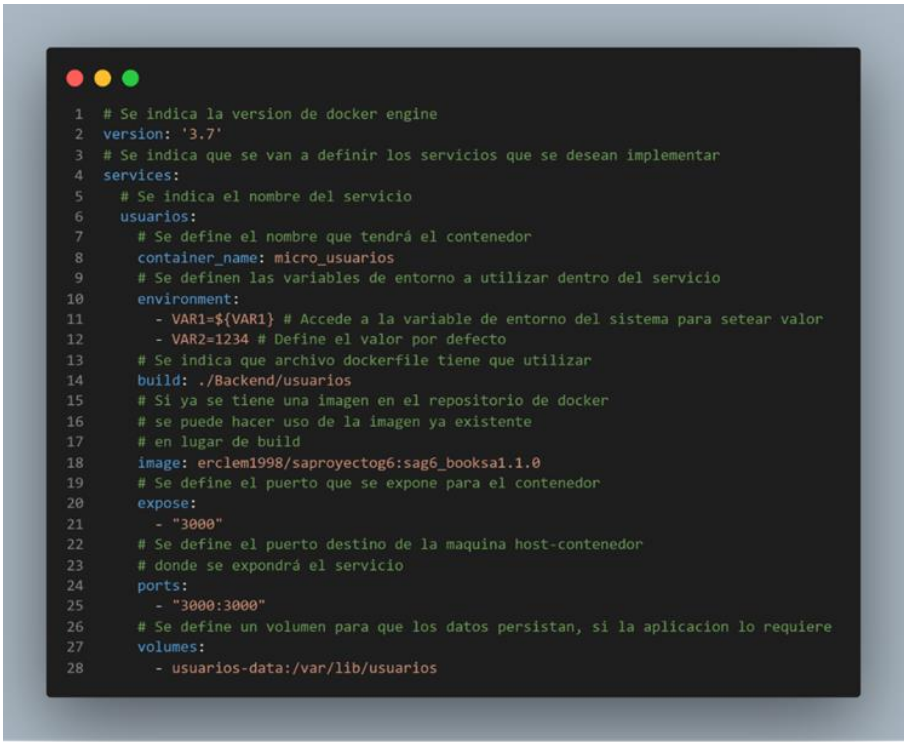
2.3.1. ¿Qué es Docker-Compose?

Es una herramienta que es utilizada para ejecutar aplicaciones. Estas aplicaciones se encuentran dentro de los archivos dockerfile, y luego se definen cada uno de los servicios que se desean ejecutar dentro del archivo docker-compose.yml. Finalmente se ejecuta un comando propio de docker-compose, que se encarga de levantar y poner en marcha los servicios definidos previamente.

2.3.2. Estructura de un archivo Docker-Compose

A diferencia de los archivos dockerfile, los archivos docker-compose si tienen una extensión que es .yml o .yaml. Así mismo, estos archivos deben respetar la sintaxis del lenguaje YAML, además que tienen una estructura que permiten definir cada uno de los servicios que se desean implementar.

Figura 12. **Estructura de un archivo docker-compose.yml**



```yaml
# Se indica la version de docker engine
version: '3.7'
# Se indica que se van a definir los servicios que se desean implementar
services:
  # Se indica el nombre del servicio
  usuarios:
    # Se define el nombre que tendrá el contenedor
    container_name: micro_usuarios
    # Se definen las variables de entorno a utilizar dentro del servicio
    environment:
      - VAR1=${VAR1} # Accede a la variable de entorno del sistema para setear valor
      - VAR2=1234 # Define el valor por defecto
    # Se indica que archivo dockerfile tiene que utilizar
    build: ./Backend/usuarios
    # Si ya se tiene una imagen en el repositorio de docker
    # se puede hacer uso de la imagen ya existente
    # en lugar de build
    image: erclem1998/saproyectog6:sag6_booksa1.1.0
    # Se define el puerto que se expone para el contenedor
    expose:
      - "3000"
    # Se define el puerto destino de la maquina host-contenedor
    # donde se expondrá el servicio
    ports:
      - "3000:3000"
    # Se define un volumen para que los datos persistan, si la aplicacion lo requiere
    volumes:
      - usuarios-data:/var/lib/usuarios
```

Fuente: elaboración propia, realizado con Visual Studio Code.

2.4. Comandos de Docker

Docker tiene la ventaja de realizar diferentes tipos de acciones, tales como verificar imágenes, contenedores, entrar al sistema de archivos del contendor, ejecutar y detener un servicio, obtener la IP de un contenedor, entre otros.

Tabla I. **Comandos para ejecutar Docker**

Comando	Función
docker images	Muestra las imágenes de docker que se tienen en el sistema.
docker ps	Devuelve una lista de contenedores en ejecución.
docker ps -a	Devuelve una lista de contenedores en ejecución y detenidos.
docker start id_container	Inicia un contenedor detenido.
docker stop id_container	Detiene un contenedor.
docker rmi id_imagen	Elimina una imagen de docker a partir de su id.
docker rm id_container	Elimina un contenedor a partir de su id.
docker exec -t id_container /bin/bash	Entra al sistema de archivos de un contenedor.
docker run -p portPc:portContainer -t nombre_container /bin/bash	Corre un contenedor.

Fuente: elaboración propia, realizado con Microsoft Word 2016.

Además, se tiene como complemento docker-compose, que facilita el despliegue de los servicios.

Tabla II. **Comandos para ejecutar Docker-Compose**

Comando	Función
docker-compose up	Levanta todos los servicios descritos en el archivo .yml.
docker-compose up –build	Levanta todos los servicios descritos en el archivo .yml, y construye todos los contenedores.
docker-compose up –build -d	Levanta todos los servicios descritos en el archivo .yml, y construye todos los contenedores. Oculta el proceso, ejecutando todo en 2do plano.

Fuente: elaboración propia, realizado con Microsoft Word 2016.

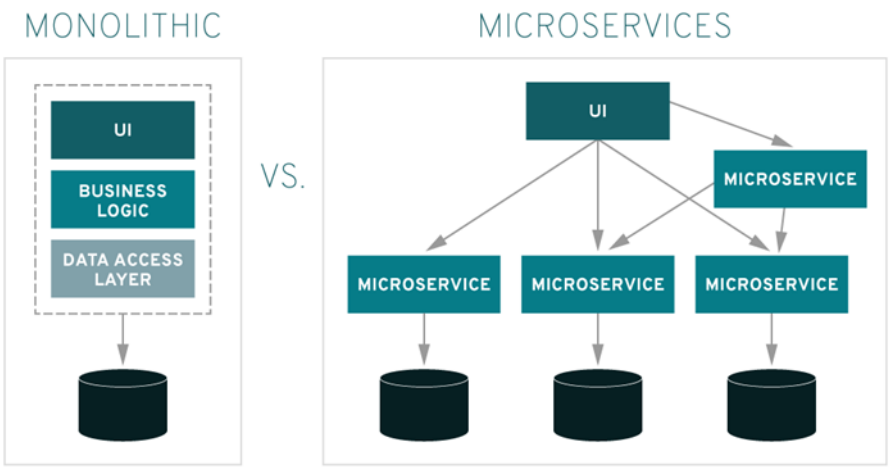
2.5. Microservicios

Es un concepto muy utilizado en el diseño de arquitecturas de *software*. A continuación, se detalla con más información.

2.5.1. Concepto de microservicios

Es un tipo de arquitectura de *software* que se centra en dividir una aplicación en partes más pequeñas e independientes, que son llamadas servicios. Al unir cada una de estas pequeñas funcionalidades, se obtiene la aplicación completa. La idea de los microservicios es tener porciones de código que sean fáciles de mantener y desplegar, además de continuar con el funcionamiento de la aplicación sin tener que dar de baja todo el servicio en caso se realice un mantenimiento o se lance una nueva versión del *software*.

Figura 13. **Arquitectura Monolítica vs Microservicios**



Fuente: Redhat (2018). *¿Qué son los microservicios?* Consultado el 5 de octubre de 2021. Recuperado de https://www.redhat.com/es/topics/microservices/what-are-microservices.

2.6. Kubernetes

Es un concepto muy utilizado en conjunto con Docker para poder escalar y mantener aplicaciones basadas en esta última. A continuación, se detalla más información relacionada.

2.6.1. ¿Qué es Kubernetes?

Es un orquestador de contenedores que permite automatizar muchos procesos manuales para desplegar una aplicación. De la misma manera, permite administrar y escalar dichas aplicaciones, implementando de esta manera Balanceadores de Carga para todos los servicios expuestos.

Kubernetes provee muchas ventajas al momento de desplegar aplicaciones, entre estas se encuentran:

- Escalamiento horizontal de los nodos (servicios y *deployments*).
- Automatizar procesos de despliegue.
- Optimizar recursos.
- Administrar servicios.
- Replicación automática.
- Control y automatización de actualizaciones.
- *Rollback* de actualizaciones.

2.6.2. Conceptos generales

Los siguientes conceptos, son muy importantes y considerados clave para implementar de manera exitosa una aplicación con esta tecnología.

2.6.2.1. Clúster

Es el conjunto de nodos o máquinas que ejecutan una aplicación. Este contiene un plano de control que se encarga de administrar el estado del clúster, y también contiene uno o más nodos.

2.6.2.2. *Namespace*

Un *namespace* se define como un clúster virtual dentro del clúster principal de Kubernetes. El objetivo de este es separar los recursos de una aplicación para diferentes tipos de usuarios.

2.6.2.3. *Pod*

Es la unidad más pequeña que se puede crear y administrar con *Kubernetes*. Está constituido por uno o más contenedores, en los cuales se define la manera de ejecutar. Dentro del mismo *pod*, los contenedores pueden compartir recursos, para mayor eficacia de la aplicación.

2.6.2.4. *ReplicaSets*

Un *ReplicaSet* es el encargado de manejar réplicas de los *Pods*, con el objetivo de garantizar la alta disponibilidad de la aplicación.

2.6.2.5. *Deployments*

Es el encargado de administrar las actualizaciones dentro de los *Pods* y *ReplicaSets*.

Un *deployment* se utiliza para:

- Desplegar nuevos *ReplicaSets*.
- Actualizar *Pods*.
- *Rollback* de un *Deployment* anterior.
- Escalamiento automático para soporte de cargas mayores.
- Eliminar *ReplicaSets*.

2.6.2.6. *Services*

Define a un conjunto de *Pods* y la política para acceder a sus recursos. Para poder dirigir el servicio a los *Pods*, el servicio define un selector.

2.6.2.7. Balanceador de carga

Es un *software* que permite repartir la carga de trabajo entre los diferentes nodos dentro del clúster de Kubernetes. De esta manera, optimiza el flujo de la aplicación, evitando congestiones.

2.6.2.8. *Ingress*

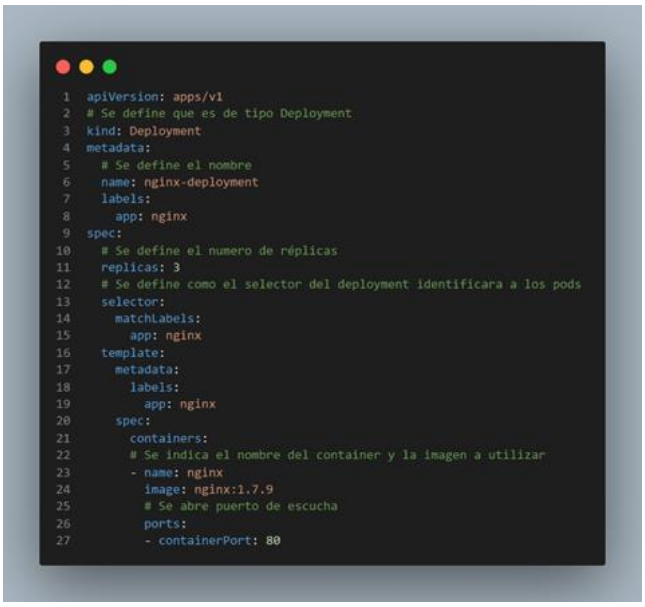
Es un objeto que administra el acceso a los servicios del clúster desde el exterior de este, a través de los protocolos HTTP. Al funcionar como un *proxy* inverso, dirige las peticiones a los servicios destino deseados. El más utilizado es NGINX *Ingress Controller*.

Al utilizar un *Ingress* junto al Balanceador de Carga, permite reducir costos, ya que en este se pueden definir más servicios destino que pueden ser utilizar por el balanceador.

2.6.3. Estructura de un archivo YAML

La estructura de un archivo para Kubernetes dependerá del propósito de este. Los más comunes son los referidos en las siguientes imágenes:

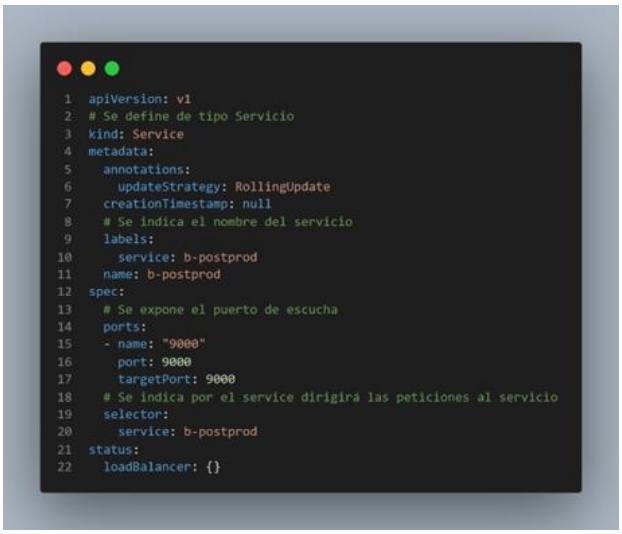
Figura 14. **Definición de un *Deployment***



```yaml
apiVersion: apps/v1
# Se define que es de tipo Deployment
kind: Deployment
metadata:
  # Se define el nombre
  name: nginx-deployment
  labels:
    app: nginx
spec:
  # Se define el numero de réplicas
  replicas: 3
  # Se define como el selector del deployment identificara a los pods
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      # Se indica el nombre del container y la imagen a utilizar
      - name: nginx
        image: nginx:1.7.9
        # Se abre puerto de escucha
        ports:
        - containerPort: 80
```

Fuente: elaboración propia, realizado con Visual Studio Code.

Figura 15. **Definición de un *Service***



```yaml
apiVersion: v1
# Se define de tipo Servicio
kind: Service
metadata:
  annotations:
    updateStrategy: RollingUpdate
  creationTimestamp: null
  # Se indica el nombre del servicio
  labels:
    service: b-postprod
  name: b-postprod
spec:
  # Se expone el puerto de escucha
  ports:
  - name: "9000"
    port: 9000
    targetPort: 9000
  # Se indica por el service dirigirá las peticiones al servicio
  selector:
    service: b-postprod
status:
  loadBalancer: {}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

Figura 16. **Definición de un *Ingress***

```yaml
# Se define la version del API de K8S a utilizar
apiVersion: networking.k8s.io/v1beta1
# Se indica el tipo, en este casi ingress
kind: Ingress
# Se define el nombre del ingress y el namespace destino
metadata:
  name: my-ingress
  namespace: name_namespace
spec:
  # Se dfine el conjunto de reglas HTTP que contituyen
  # el ingress
  rules:
  - http:
      paths:
      # Se definen los paths para acceder al servicio.
      - path: /productos/crear-producto
        # Se define como tip obackend
        backend:
          # Se define el nombre del servicio
          serviceName: b-postprod
          # Se define el puerto de acceso
          servicePort: 9000
      # Más servicios
      - path: /categoria/obtener-categorias
        backend:
          serviceName: b-postprod
          servicePort: 9000
```

Fuente: elaboración propia, realizado con Visual Studio Code.

2.6.4. Comandos de Kubernetes

Los comandos de Kubernetes, permiten interactuar con el entorno del clúster que se está administrando. Con estos comandos se puede realizar:

- Desplegar nuevos *ReplicaSets*.
- Crear *Namespaces*.
- Desplegar *Deployments*.

- Desplegar Servicios.
- Implementar *Ingress*.
- Desplegar un *Ingress*.
- Ver *Pods*.

Tabla III. **Comandos de Kubernetes**

Comando	**Función**
kubectl get namespaces	Obtiene todos los *namespaces* del clúster.
kubectl create namespace nombre_namespace	Crea un nuevo *namespace* dentro del clúster.
kubectl apply -f nombre-deployment.yaml	Crea un nuevo *deployment* a partir de un archivo de especificación .yaml.
kubectl apply -f nombre-service.yaml	Crea un nuevo servicio a partir de un archivo de especificación .yaml.
kubectl apply -f services-ingress.yaml	Crea un nuevo *ingress* a partir de un archivo de especificación .yaml.
kubectl get services	Obtiene todos los servicios que se encuentran en el clúster.
kubectl get pods	Obtiene todos los *pods* dentro del clúster.
kubectl rollout undo deployment nombre_deployment -n nombre_namespace	Realiza *rollback* de los *pods* de un *deployment* dentro de un *namespace*.
kubectl describe ingress nombre_ingress	Describe un *ingress*.

Fuente: elaboración propia, realizado con Microsoft Word 2016.

3. TECNOLOGÍAS DE LA NUBE

3.1. ¿Qué es la nube?

La nube, también conocida como *Cloud*, es una red de servicios informáticos, en servidores, que funcionan como un todo y de manera remota. Cada nodo, o servidor, que se encuentra dentro de esta red mundial, tiene diferentes funciones y objetivos.

Cada uno de los servidores, están diseñados para:

- Administrar datos.
- Generar servicios de *streaming*.
- Ejecutar aplicaciones.
- Redes sociales.
- Correos electrónicos.

Los servicios en la nube no acceden a datos o archivos locales, sino que al estar todo en línea, puede acceder a estos recursos desde cualquier dispositivo.

3.2. Ventajas de la implementación de servicios en la nube

Implementar servicios en la nube provee diferentes ventajas a todas las organizaciones que se mudan hacia esta. Entre las principales están:

- Cero almacenamientos locales: cada organización puede almacenar todos sus datos de manera virtual, es decir, no tiene que preocuparse por implementar más almacenamiento en caso de que se necesite más, ya que la nube provee el almacenamiento que el cliente desee

- Disminución en costos de mantenimiento y almacenamiento: al no tener recursos físicos en la organización, o solo los mínimos que son recursos privados, los gastos disminuyen en cierto porcentaje en beneficio de la organización.

- Restricción de acceso a los datos: cada nube implementa diferentes niveles de accesos a los datos, que son configurables a los tipos de usuarios que se encuentran registrados en el proveedor. Con esto, se puede restringir el acceso a información sensible, además que se pueden configurar diferentes políticas para mayor seguridad de los datos.

- Capacidad de almacenamiento: con la nube no es necesario preocuparse por la cantidad de almacenamiento, ya que permite aumentar o disminuir su capacidad, conforme las necesidades del cliente. Así mismo, permite administrar la estructura, para tener mayor facilidad de acceso a los datos.

- Accesibilidad a los datos: gracias a que los datos se encuentran en diferentes servidores, los datos siempre se encuentran accesibles desde cualquier lugar, lo que permite el fácil intercambio y actualización de la información.

3.3. Principales proveedores de la nube

Los servicios en la nube también son un modelo de negocio, en el cual existe competencia. Cada proveedor ofrece una amplia gama de servicios, algunos pueden ser similares, y otros muy diferentes. La selección de un proveedor dependerá, principalmente, de las necesidades que se tengan en la organización, así como los costos que cada una pueda ofrecer.

Actualmente existen muchos proveedores que se han clasificado entre los grupos Líderes, Aspirantes, Operadores Especializados y Visionarios, en el Cuadrante Mágico de Gartner. Este cuadrante, es actualizado anualmente.

Figura 17. **Cuadrante mágico de Gartner 2021 para servicios de infraestructura y plataforma en la nube**

Fuente: Gartner (2021). *Magic Quadrant para servicios de infraestructura y plataforma en la nube*. Consultado el 8 de octubre de 2021. Recuperado de https://www.gartner.com/technology/media-products/reprints/AWS/1-271W1OTA-ESP.html.

3.3.1. Amazon Web *Services*

Actualmente es el proveedor de servicios en la nube líder y más completa en el mundo. Uno de sus puntos fuertes es que reduce costos, aumenta agilidad y permite innovar las soluciones, gracias a que ofrece más de 200 servicios con diferentes tecnologías.

Cabe destacar que AWS es el pionero en el área de servicios en la nube, con más de 15 años de experiencia operacional, lo que lo convierte en el proveedor más importante, ya que implementa herramientas de seguridad avanzadas, conformidad y gobernanza.

3.3.2. Microsoft Azure

Es el segundo proveedor en la nube que se encuentra entre los líderes del mercado. Esta plataforma ofrece más de 200 productos y servicios que son utilizados para diseñar nuevas soluciones de *software*. Esta plataforma se mantiene en constante innovación, lo que permite utilizar nuevos servicios que están destinados a innovar las soluciones.

En términos de precios, Azure es 5 veces más barato que AWS para los servicios de Windows Server y SQL Server. Asimismo, ofrece flexibilidad de precios, pagando únicamente por lo que se utiliza, además de tener la ventaja de ser una nube híbrida, permitiendo de esta manera la adición de medios físicos

3.3.3. Google *Cloud Platform*

Google *Cloud Platform* (GCP), es una nube abierta que está comprometida con el código abierto, esto quiere decir que permite el uso de

múltiples nubes y la incorporación de nubes híbridas, permitiendo ejecutar aplicaciones en cualquier entorno.

Esta plataforma permite optimizar los gastos de infraestructura de TI, además qué aumenta la eficiencia operacional. De igual manera que las nubes anteriores, únicamente se paga por lo que se utiliza.

En términos de seguridad, es una plataforma que encripta la información en tránsito y en reposo, lo que maximiza la confiabilidad de accesos autorizados.

3.3.4. Ventajas de cada nube

En resumen, la elección de un proveedor de la nube dependerá, no solo de los precios, sino que también de las necesidades y lo que pueda ofrecer cada proveedor.

A continuación, se muestra una tabla con las ventajas que posee cada una de las nubes:

Tabla IV. **Proveedores de la nube**

Proveedor	Ventajas
Amazon Web Services	• Flexibilidad y facilidad de uso. • Rentabilidad. • Escalabilidad. • Elasticidad. • Seguridad. • Velocidad de Organización. • Innovación
Microsoft Azure	• Reducción de costes. • Ambientes *Multi-Cloud* • Flexibilidad. • Innovación. • Uso operativo con sus productos. • Apto para todo tipo de empresas. • Facilidad de desarrollo de aplicaciones.
Google Cloud Platform	• Mayor rendimiento. • Rápida restauración de información. • Seguridad mejorada. • Rendimiento mejorado. • Accesibilidad de precios. • Innovación.

Fuente: elaboración propia, realizado con Microsoft Word 2016.

3.4. ¿Por qué utilizar AWS?

AWS es la plataforma más completa y utilizada del mundo, además de ser líder en servicios de *Cloud Computing*, tiene una amplia gama de servicios que van desde servidores, almacenamiento y bases de datos, hasta servicios de aprendizaje automático.

A continuación, se detallan el manejo de usuarios y servicios utilizados en el ámbito de DevOps.

3.4.1. *Identity & Access Management* (IAM)

Es un servicio que permite controlar el acceso seguro a los recursos de AWS. Además, permite administrar usuarios y grupos a quienes, por medio de políticas, se les puede asignar permisos para poder acceder a ciertos servicios de la nube.

El servicio administra de manera centralizada a los usuarios, y generando claves de acceso que permiten controlar el acceso a los recursos de AWS.

3.4.2. Servicios en la nube

En el presente trabajo de investigación, se utilizan las siguientes tecnologías de la nube de AWS:

3.4.2.1. *Elastic Compute Cloud* (EC2)

Amazon EC2, es un servicio que permite administrar máquinas virtuales, o servidores, que son utilizados para compilar y hospedar las aplicaciones desarrolladas o en proceso de desarrollo.

También, simplifica el uso de la informática para los desarrolladores. Algunas de sus ventajas son:

- Confiabilidad y escalabilidad bajo demanda.
- Flexibilidad de costos operacionales.
- Flexibilidad de migración y creación de aplicaciones.
- Permite seleccionar los componentes de las máquinas virtuales.

3.4.2.2. *Simple Storage Service* (S3)

Es un servicio de almacenamiento físico. En este se pueden almacenar diferentes tipos de datos y archivos, y que pueden ser accesibles desde cualquier lugar.

Sus beneficios son:

- Rendimiento, disponibilidad y escalabilidad.
- Capacidad de seguridad y auditoría.
- Administración de los datos.
- Disminución de costos.
- Fácil de utilizar.

3.4.2.3. *Relational Database Service* (RDS)

Es un servicio web que permite configurar y escalar bases de datos relacionales en la nube de AWS. Además de ofrecer rentabilidad y ajustar el tamaño de una base de datos, permite administrar todas las tareas en relación de esta.

Los beneficios que provee son:

- Fácil administración de la información.
- Escalabilidad.
- Disponibilidad y durabilidad.
- Seguridad y rapidez.
- Bajas tarifas por los recursos utilizados.

3.4.2.4. Amazon *Rekognition*

Es un servicio de *Machine Learning* de AWS que permite el análisis y procesamiento de videos e imágenes. También ofrece reconocimiento facial de alta precisión. Así mismo, permite identificar diferentes tipos de objetos, texto, personas, entre otros.

Entre los beneficios de uso podemos encontrar:

- Potente análisis de video e imágenes.
- Análisis basado en aprendizaje profundo.
- Análisis escalable.
- Bajo costo.
- Fácil integración con otros servicios de AWS.

3.4.2.5. *Elastic Container Registry* (ECR)

Es un servicio de registro de contenedores de Docker que permite almacenar, administrar e implementar imágenes de Docker. Además, elimina la necesidad de operar repositorios propios de contenedores.

Los beneficios que otorga a los desarrolladores son:

- Alta disponibilidad.
- Flujo de trabajo simplificado.
- Seguro y confiable.
- Totalmente gestionado.

3.4.2.6. *Elastic Container Service* (ECS)

Es un servicio que permite implementar cargas de trabajo en contenedores. Además, permite administrar una amplia gama de contenedores que ejecutan aplicaciones empresariales.

Los beneficios que ofrece son:

- Reducción de costos computacionales.
- Cumple con los requisitos de seguridad.
- Utiliza herramientas de automatización.

3.4.2.7. *Elastic Kubernetes Service* (EKS)

Es un servicio que permite ejecutar Kubernetes en la nube de AWS. Una de sus ventajas es que no hay necesidad de instalar u operar clústeres de Kubernetes.

Al ser un servicio administrado, permite administrar automáticamente la disponibilidad y escalabilidad de los nodos y aplicaciones programados.

Los beneficios que provee son:

- Seguridad de forma predeterminada.
- Plano de control administrado.
- Integración de servicios de AWS.
- Consola de Kubernetes alojada.
- Grupos de nodos gestionados.

4. DESARROLLO DE APLICACIONEWS WEB CON *FRAMEWORKS* BASADOS EN JAVASCRIPT

4.1. Angular como *framework frontend*

Angular es un *framework* que es utilizado para crear aplicaciones web de una sola vista o página. Esto quiere decir que todos los recursos de la aplicación se cargan en el instante y se van mostrando de manera dinámica, conforme se vayan requiriendo con la interacción de los usuarios.

Al ser así, la aplicación denota una experiencia como las aplicaciones de escritorio y móviles. Cabe destacar que este *framework* utiliza el lenguaje *TypeScript*.

4.1.1. Esencia de Angular

Angular como tal, posee diferentes conceptos importantes. Sin embargo, se detallas los más importantes a considerar al trabajar con esta tecnología:

4.1.1.1. Componentes

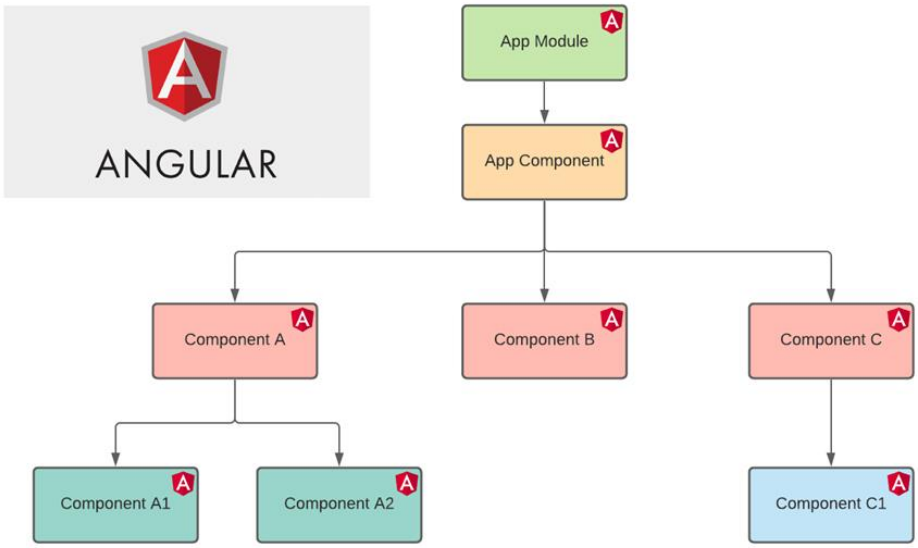
Los componentes son los bloques que componen una aplicación. Al crear un componente, este automáticamente crea los siguientes ficheros.

- Clase de *TypeScript*.
- Plantilla HTML.
- Archivo para estilos (puede ser css, sass, entre otros).

- Clase para *testing* con *TypeScript*.

Al utilizar los componentes mencionados, se va creando un árbol de componentes, en el cual se puede ir observando que recursos se van cargando conforme la interacción de los usuarios.

Figura 18. **Árbol de componentes de Angular**

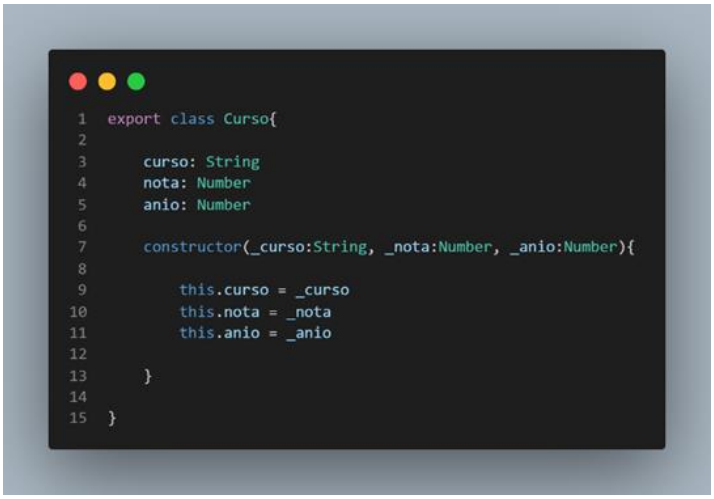


Fuente: elaboración propia, realizado con lucid.app.

4.1.1.2. Modelos

Son fragmentos de código independientes que ayudan a modelar algún tipo de objeto. Se puede decir que es una plantilla. Para mayor facilidad, son clases que se generan para crear los objetos, mencionados anteriormente, en diferentes tipos de acciones.

Figura 19. **Ejemplo de modelo o clase en Angular**



```
export class Curso{

    curso: String
    nota: Number
    anio: Number

    constructor(_curso:String, _nota:Number, _anio:Number){

        this.curso = _curso
        this.nota = _nota
        this.anio = _anio

    }

}
```

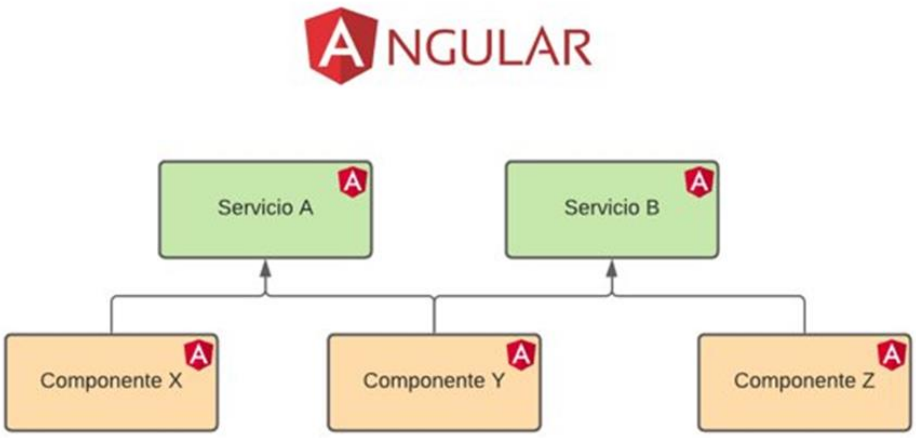
Fuente: elaboración propia, realizado con Visual Studio Code.

4.1.1.3. Servicios

Es una definición de código que permitirán acceder a la información de un servidor a través de diferentes peticiones HTTP. Los servicios serán consumidos por los componentes conforme estos los vayan necesitando.

Es importante destacar que es necesario tener habilitados los CORS (Intercambio de Recursos de Origen Cruzado) en el *backend*, para poder acceder a los recursos que este último proporciona.

Figura 20. **Llamadas HTTP desde diferentes componentes**

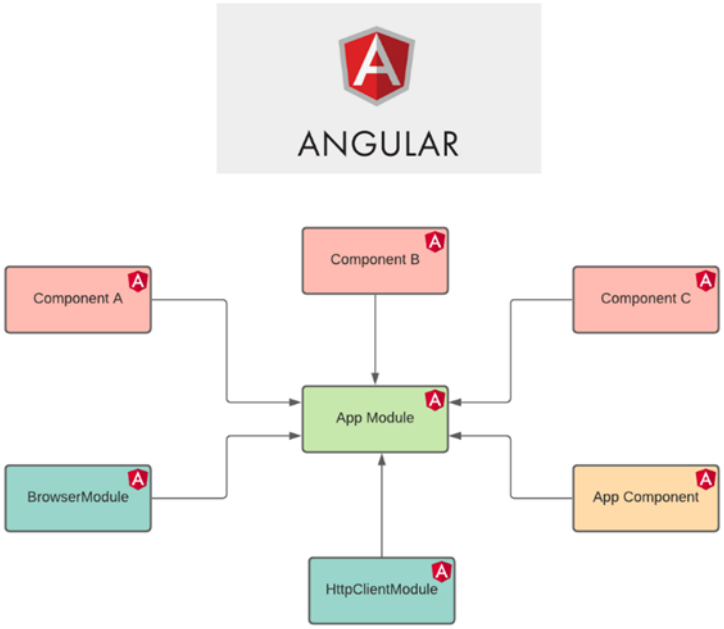


Fuente: elaboración propia, realizado con lucid.app.

4.1.2. *AppModule*

Es la raíz del proyecto de Angular, y es donde se importan todos los módulos y se declaran los componentes que utilizará la aplicación para su funcionamiento.

Figura 21. **Llamadas HTTP desde diferentes componentes**



Fuente: elaboración propia, realizado con lucid.app.

4.1.3. *AppRoutingModule*

Es un módulo donde se definen las rutas que tendrá la aplicación. Este es importado por el *App Module* para acceder a cada ruta configurada.

4.1.4. Directivas

Una directiva es un tipo de instrucción que tiene como objetivo extender la funcionalidad de HTML. Gracias a estas directivas, se puede definir lógica que será ejecutada en el DOM.

Las directivas, se dividen en tres grupos principales que son:

4.1.4.1. Directivas de atributo

Este tipo de directivas permiten modificar la vista de elementos del DOM, y se utilizan como atributos

Tabla V. **Tipos de directivas de atributo**

Directiva	Función
ngModel	Implementar *binding* o contenidos dinámicos.
ngClass	Implementar clases como elementos.
ngStyle	Implementar estilos.

Fuente: elaboración propia, realizado con Microsoft Word 2016.

4.1.4.2. Directivas estructurales

Tienen como objetivo agregar, editar, eliminar y manipular los objetos del DOM. Algunas de estas tienen asociadas funcionalidades muy parecidas a instrucciones de otros lenguajes de programación.

Tabla VI. **Tipos de directivas estructurales**

Directiva	Función
ngIf	Controlar cuando mostrar algo en código HTML.
ngFor	Iterar sobre una lista, para mostrar de forma dinámica los datos almacenados o recibidos.
ngSwitch	Similar al *ngIf*, permite administrar diferentes casos, para mostrar elementos. Su lógica es como la de los lenguajes de programación.

Fuente: elaboración propia, realizado con Microsoft Word 2016

4.1.4.3. Directivas de componente

Son directivas con un *template*, que tiene objetivo administrar partes HTML como un elemento de este.

4.1.5. Comandos

Como cualquier otro controlador, Angular posee comandos que son importantes para su funcionalidad. En la siguiente tabla se muestran los comandos más utilizados.

Tabla VII. **Comandos para ejecutar Angular CLI**

Comando	Función
npm install -g @angular/cli	Permite instalar angular en los ordenadores. Es necesario tener instalado Node.js previamente.
ng new nombre-proyecto	Permite crear un nuevo proyecto de Angular.
ng serve	Permite correr el proyecto de angular, por defecto se ejecuta en el puerto 4200.
ng generate component [ruta]	Permite crear un nuevo componente, crea un archivo html, *typescript*, estilos, y para *test*.
ng g component [ruta]	Permite crear un nuevo componente, crea un archivo html, *typescript*, estilos, y para *test*.
ng generate service [ruta]	Permite crear un nuevo archivo para generar servicios HTTP, además crear un archivo para *test*.
ng g service [ruta]	Permite crear un nuevo archivo para generar servicios HTTP, además crear un archivo para *test*.
ng generate class [ruta]	Permite generar modelos o plantillas que se pueden utilizar en el proyecto.
ng g class [ruta]	Permite generar modelos o plantillas que se pueden utilizar en el proyecto.

Fuente: elaboración propia, realizado con Microsoft Word 2016.

4.2. Node.js como *framework backend*

Node.js es un *framework* que permite crear y programas con JavaScript. Gracias a su amplia variedad de librerías, permite generar programas independientes, que pueden ejecutarse fuera de un navegador. Algunas de estas

librerías objetivos como generar API Rest, conexiones con otros servidores, y realizar pruebas en las aplicaciones. Algunas importantes son:

4.2.1. Express.js para creación de servicios web

Es una librería o módulo disponible en el repositorio de NPM (Node.js), que permite generar un API Rest de manera fácil y rápida. Esta librería posee diferentes características que ayuda al desarrollo de peticiones HTTP.

Para instalar Express.js basta con ejecutar el comando *npm install express*.

Figura 22. **API Rest básico con Express.js**

Fuente: Expressjs. Consultado el 10 de noviembre de 2021. Recuperado de https://expressjs.com/en/starter/hello-world.html.

4.2.2. AWS-SDK para conexión con la nube

Es un módulo que proporciona una API para utilizar los servicios de AWS desde JavaScript. Mediante el uso de llaves de acceso generadas para los usuarios, se puede conectar a los diferentes servicios proporcionados por esta plataforma en la nube.

Para instalarlo se debe ejecutar *npm install aws-sdk*.

Figura 23. **Ejemplo de importación del SDK de AWS en Node.js**

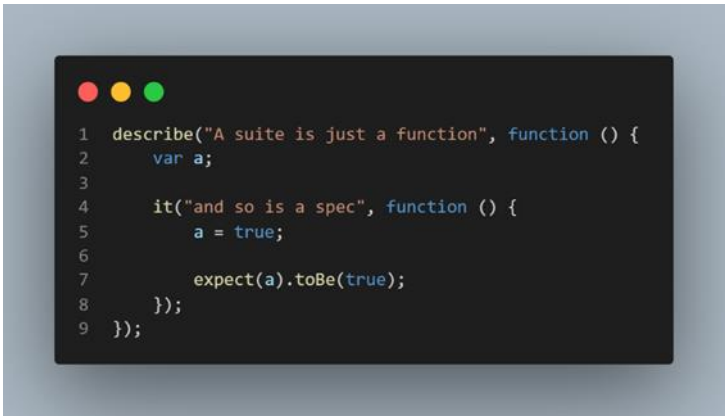
Fuente: AWS (2020). *SDK de AWS.* Consultado el 10 de noviembre de 2021. Recuperado de https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/getting-started-nodejs.html.

4.2.3. *Testing* con Jasmine

Esta es una librería que es utilizada para crear archivos de especificaciones para hacer pruebas en una aplicación creada con JavaScript. Contienen una amplia gama de *asserts* que ayudan a controlar respuestas o valores esperados luego de ejecutar una funcionalidad específica.

Su instalación es muy sencilla, y solo se debe ejecutar el comando *npm install jasmine*.

Figura 24. **Ejemplo de *test* con Jasmine**



Fuente: Jasmine (2021). *Test en comando npm install.* Consultado el 10 de noviembre de 2021. Recuperado de https://jasmine.github.io/.

5. ADOPCIÓN E IMPLEMENTACIÓN

El presente capítulo, tiene como objetivo demostrar, al lector, la adopción de un entorno DevOps en un proyecto de *software*, así como distintas configuraciones de herramientas, entre otros, que permitirán desarrollar de manera flexible y rápida el producto final.

5.1. Primeros pasos

Antes de iniciar con el desarrollo de la aplicación, es necesario preparar las herramientas y entornos en los que se deberá desarrollar cada módulo de esta. Para realizar lo mencionado anteriormente, se define a continuación los primeros pasos a realizar previo a iniciar el proyecto.

5.1.1. Descripción y arquitectura del *software*

Al momento de iniciar un nuevo proyecto de *software*, es importante definir la arquitectura de esta, ya que esto permitirá construir el producto esperado con base en los componentes, relaciones y tareas que se tiene que realizar dentro de la aplicación final.

El presente trabajo de graduación tiene como objetivo generar una aplicación web funcional que permita, a distintos usuarios, almacenar archivos en la nube de manera segura, así como manejar una estructura archivos a su conveniencia.

La misma permitirá registrar y *logguear* nuevos usuarios, quienes tendrán acceso a las siguientes opciones:

- Crear, actualizar, copiar, mover y eliminar directorios.
- Cargar, actualizar, descargar, copiar, mover y eliminar archivos.
- Historial de directorios y archivos.
- Detalles de directorios y archivos, tales como nombre, fecha de creación, entre otros.
- Control de directorios y archivos eliminados.
- Marcar directorios o archivos como destacados.
- Vista de directorios y archivos.

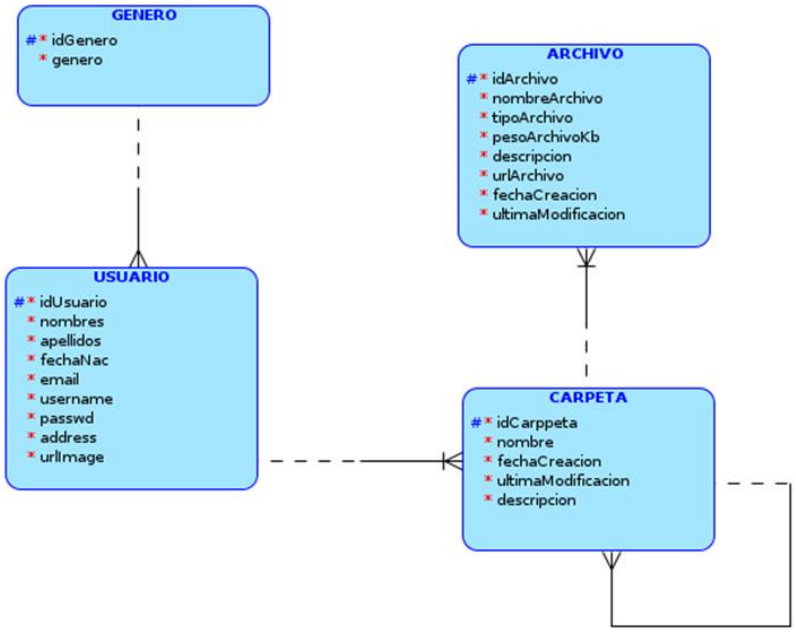
Dados los requerimientos de la aplicación, se puede continuar y definir los aspectos importantes que son el diseño de base de datos y la arquitectura de *software*.

5.1.1.1. Diseño de Base de Datos

Para poder representar la aplicación dentro del entorno de base de datos, se hace uso de un Modelo Entidad-Relación (ER), el cual es un diagrama que permite representar como las entidades se relacionan entre sí, formando, de esta manera, un todo.

El modelo ER, en el cual se basará la aplicación, es el siguiente:

Figura 25. **Modelo entidad-relación**



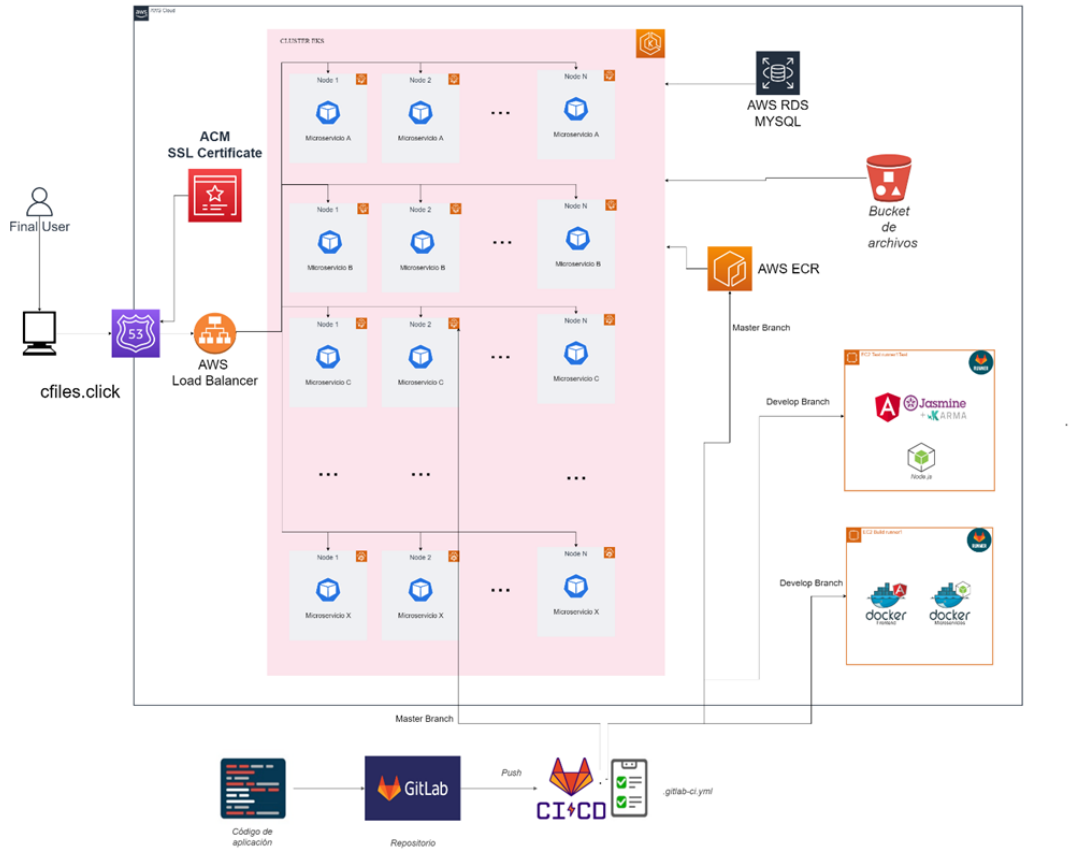
Fuente: elaboración propia, realizado con Oracle Datamodeler.

5.1.1.2. Arquitectura de microservicios

Conociendo los requerimientos de la aplicación, se puede observar que se incluyen diferentes funcionalidades que, al trabajar juntas, hacen un todo. Sin embargo, se puede tomar cada una de estas funcionalidades y dividirlas en pequeñas porciones de código independientes que al unirlas seguirán funcionando como un todo. Cada pequeña porción de código es llamada microservicio.

Entonces, si la aplicación se divide en microservicios y se unifican con todas las herramientas y entornos que se han de utilizar, se obtiene una Arquitectura basada en Microservicios, que se detalla a continuación:

Figura 26. **Arquitectura de Microservicios en AWS**



Fuente: elaboración propia, realizado con draw.io.

En el diagrama anterior, se puede observar todos los elementos que conforman la arquitectura de microservicios las cuales son:

- Clúster de Kubernetes en Amazon *Elastic Kubernetes Service* (EKS).
- Gama de microservicios.
- Amazon *Elastic Container Registry* (ECR).
- Balanceador de carga, para nivelar el tráfico entrante.
- *Bucket* de S3, para almacenamiento de archivos.

- Base de Datos MySQL en Amazon RDS.
- Amazon *Certificate Manager* (ACM), para certificados SSL.
- *Route* 53 para asignación del dominio de la aplicación.
- Máquinas virtuales en Amazon EC2 para entornos de *testing*.
- Repositorio en Gitlab.

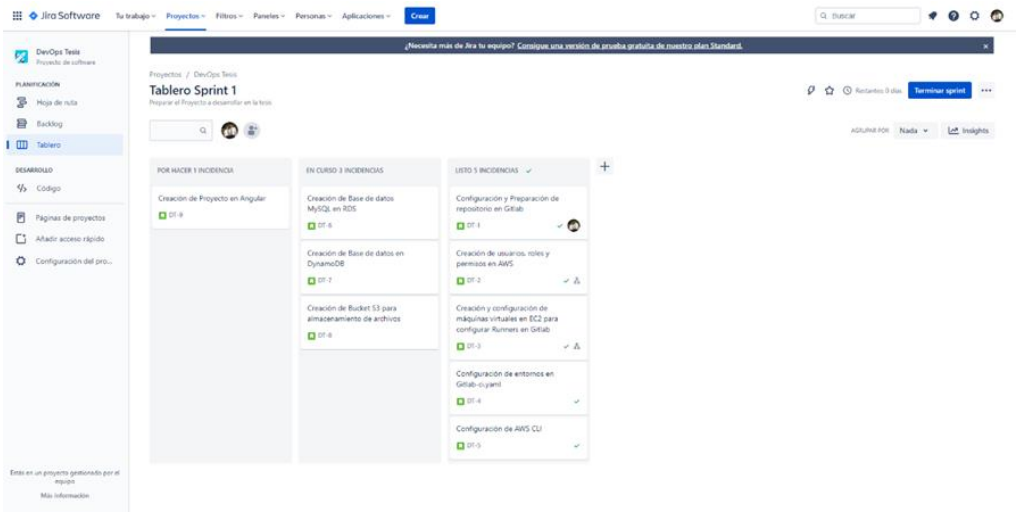
5.1.2. *Software* de Planificación

En la 1.9.1 se hace referencia a dos diferentes *softwares* que son utilizados para planificar y llevar el control de tareas y actividades. Para este capítulo se hará uso del *software* Jira.

Además de esta tecnología, se implementará la metodología ágil Scrum para llevar el control de las distintas fases del proyecto de manera flexible.

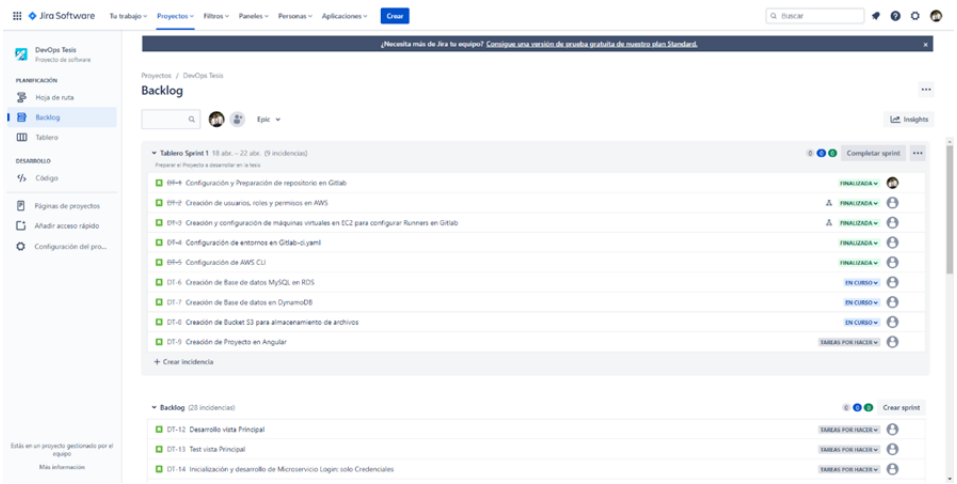
A continuación, se presenta el tablero implementado en Jira, utilizando un tablero de Scrum.

Figura 27. **Tablero de Scrum en Jira**



Fuente: elaboración propia, realizado con Microsoft Edge.

Figura 28. **Tablero Scrum Backlog en Jira**



Fuente: elaboración propia, realizado con Microsoft Edge.

5.1.3. *Software* de control de versiones

Para el manejo y control de versiones del *software* a implementar, se utilizará Gitlab, ya que permite la configuración del archivo .gitlab-ci el cual procesará las tareas definidas para el entorno DevOps.

5.1.4. Gitflow como flujo de trabajo

Gitflow es un flujo de trabajo que permite el trabajo en equipo de manera descentralizada, además que brinda flexibilidad en el desarrollo, permite la integración Desarrollo, Control de Calidad y Operaciones.

5.1.4.1. Nomenclatura de ramas

Cuando se selecciona Gitflow como el flujo de trabajo, es importante definir el proceso de ramificación. A continuación, se presenta la propuesta para el nombramiento de ramas, ya que muchos de estos dependen de los estándares de las organizaciones.

Tabla VIII. **Propuesta de nombramiento de ramas**

Tipo de Rama	Nombre	Función
Máster	máster	Contendrá la última versión de producción
Desarrollo	*develop*	Integrará todos los cambios y nuevas funcionalidades de las ramas *feature*.
Feature	*feature*/Nombre_Funcion	Representará cada nueva funcionalidad creada y que será unida a la rama *develop*.
Bugfix	*bugfix*/Nombre_Corrección	Representará cada error corregido antes de pasar a producción.
Release	*release*/X.Y.Z	Representará cada versión del *software*.
Hotfix	*hotfix*/Nombre_Corrección	Representará cada error corregido y que fue encontrado en producción. Se origina de *master* y se termina en la misma.

Fuente: elaboración propia, realizado con Microsoft Edge.

5.1.5. Preparación de *pipeline*

En este paso, se debe preparar el *pipeline* en el cual se agregarán diferentes *Jobs* (pequeñas unidades de trabajo) que estarán ligadas directamente a cada *stage* (*build*, *test*, *delivery* y *deploy*), y que tendrán como objetivo realizar una acción dentro del ciclo DevOps.

A continuación, se define la plantilla de nuestro *pipeline* (archivo llamado .gitlab-ci.yml), donde agregaremos algunos *Jobs* de prueba y se definirán los *stages* mencionados. Es importante mencionar que el archivo debe ser creado en la raíz del repositorio.

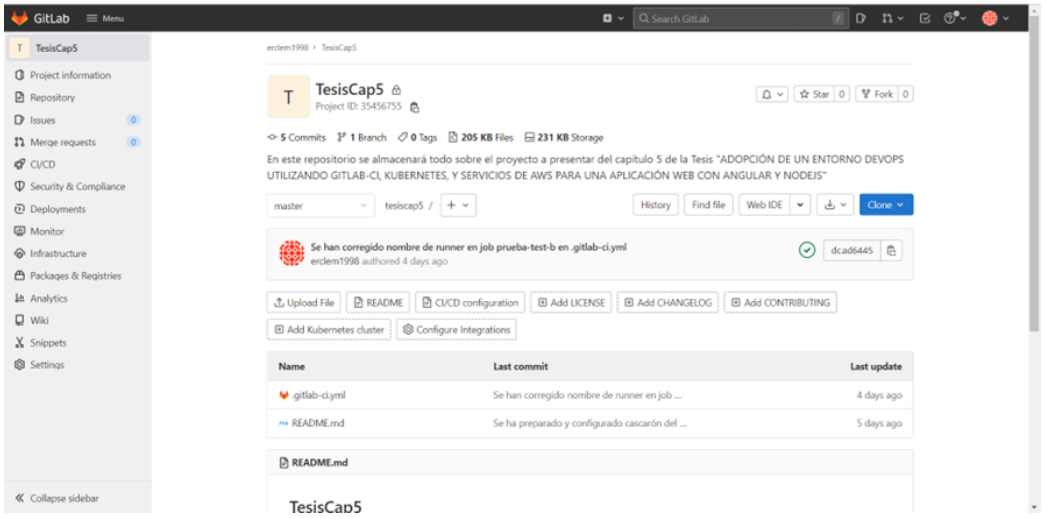
Figura 29. **Plantilla de *pipeline* .gitlab-ci.yml**

```yaml
# ------------------DEFINIMOS LOS STAGES------------
stages:
  - build
  - test
  - delivery
  - deploy

# -------------------DEFINIMOS LOS JOBS--------------
# - Job: es una unidad de trabajo que permite ejecutar ciertas acciones
#        de manera automática. Cada Job está ligado a un stage, el cuál
#        representa un paso del ciclo DevOps

prueba-build:       # Este Job se ejecuta en el stage de build.
  stage: build
  script:
    - echo "Compilando el código..."
    - echo "Compilación completada."


prueba-test-a:   # Este Job se ejecuta en el stage de test.
  stage: test    # Iniciará unicamente si el stage de build se completó exitosamente.
  script:
    - echo "Ejecutando tests..."
    - echo "Tests completados con éxito"

prueba-delivery:      # Este Job se ejecuta en el stage de delivery.
  stage: delivery  # Inicia unicamente cuando el stage de test finaliza.
  script:
    - echo "Subiendo dockerfiles..."
    - echo "Dockerfiles subidos exitosamente."

deploy-job:      # Este Job se ejecuta en el stage de deploy.
  stage: deploy  # Se ejecutará cuando la entrega se realice exitosamente.
  script:
    - echo "Desplegando aplicación..."
    - echo "Aplicación desplegada exitosamente."
```

Fuente: elaboración propia, realizado con Visual Studio Code.

Después de generar la plantilla de *pipeline*, es necesario probar su funcionamiento. Para eso se realizan las acciones *commit* y *push* a la única rama existente del momento (*master*) del repositorio, y se dirige al repositorio en Gitlab para verificar que los cambios han sido cargados.

Figura 30. **Cambios cargados en repositorio de Gitlab**

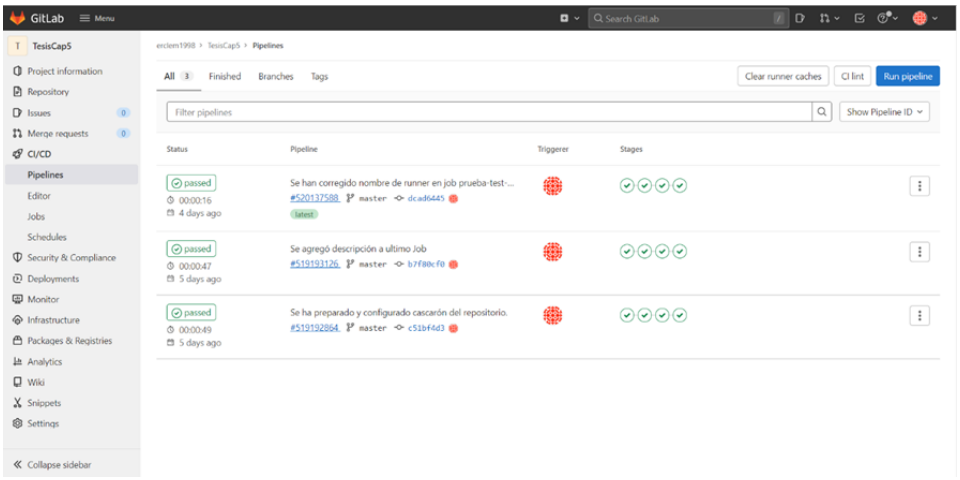


Fuente: elaboración propia, realizado con Visual Studio Code.

Por último, se verifica si el *pipeline* se ha ejecutado correctamente. Para observar el compartimiento del *pipeline* se realizan los siguientes pasos:

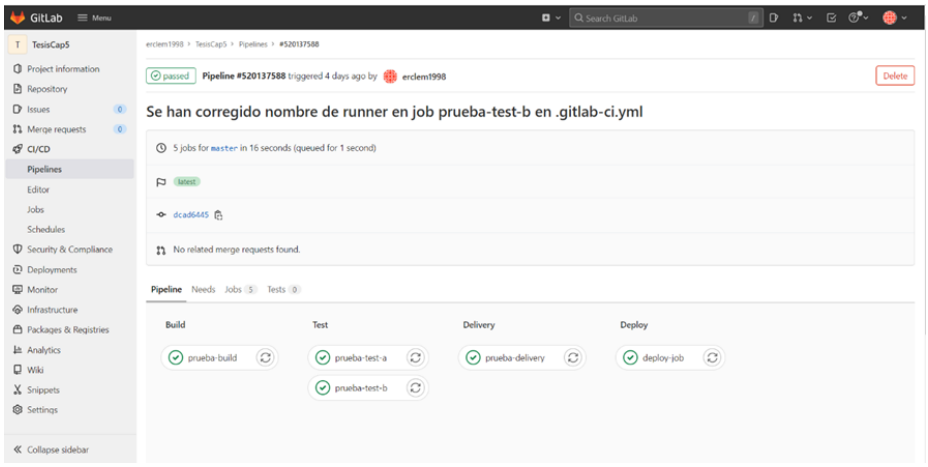
- Dirigirse a la parte lateral izquierda del repositorio y buscar la opción CI/CD.
- Seleccionar la opción *Pipelines*.
- En la vista mostrada, se puede observar verificar lo siguiente:
  - Estado.
  - *Commit* que lanzó el *pipeline*.
  - Persona que realizó el *commit*.
  - *Stages* ejecutados. Este último se verá distribuido cuando los *stages* se dividan por las diferentes ramas que se trabajarán.
- Si se presiona el botón *passed*, se dirigirá a una vista donde se puede ver más detallada la información anterior, además que permitirá ejecutar manualmente el *pipeline*.

Figura 31. **Vista sin detalles de la ejecución del *pipeline***



Fuente: elaboración propia, realizado con Visual Studio Code.

Figura 32. **Vista detallada de la ejecución del *pipeline***



Fuente: elaboración propia, realizado con Visual Studio Code.

En la 5.7, se definen opciones más complejas, tales como referencias a ramas y ejecución de *runners* (que se encargarán de realizar ciertos *jobs*),

además de las instrucciones que se ejecutarán de manera automatizada de cada *job* para cada *stage*.

5.1.6. Preparación de cuenta en AWS

Antes de iniciar a utilizar los servicios que la nube de AWS ofrece, es importante considerar lo siguiente:

- Se debe crear una cuenta en AWS que será el usuario *root* (raíz).
- Crear un usuario administrador, a partir del usuario *root*, quien será el encargado de crear usuarios y asignar grupos y permisos necesarios. Este usuario tendrá la mayor parte de permisos (o todos).

Lo anterior tiene como objetivo asegurar la cuenta raíz, y evitar cualquier acceso no deseado que pueda poner en peligro la información de la organización.

5.1.6.1. Creando usuarios, grupos y permisos

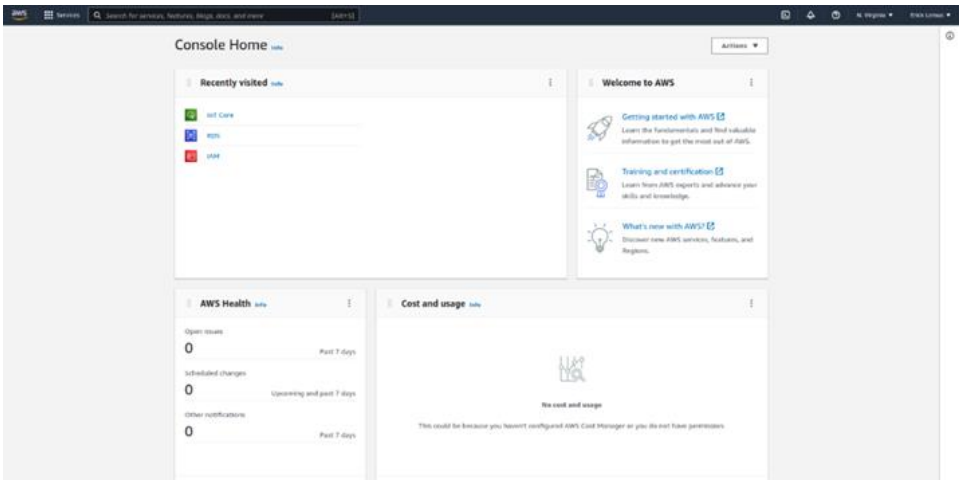
Crear diferentes usuarios y asignar grupos y permisos, permite generar una capa de seguridad, ya que la información no se encuentra en una única cuenta, lo que da como resultado la distribución de las acciones e información en cada uno de estos.

Para este paso, se creará un usuario administrador, el cual se encargará de crear el resto de los usuarios que se han de utilizar dentro del proyecto. Es importante recordar que para crear el usuario administrador se debe iniciar sesión con el usuario *root*.

Retomando la sección, se creará el usuario administrador siguiendo los siguientes pasos:

- Iniciar sesión en la consola de AWS con el usuario *root*.
- En el buscador, escribir IAM, que es el servicio que permite administrar y controlar el acceso a los recursos que ofrece AWS, y seleccionarlo.

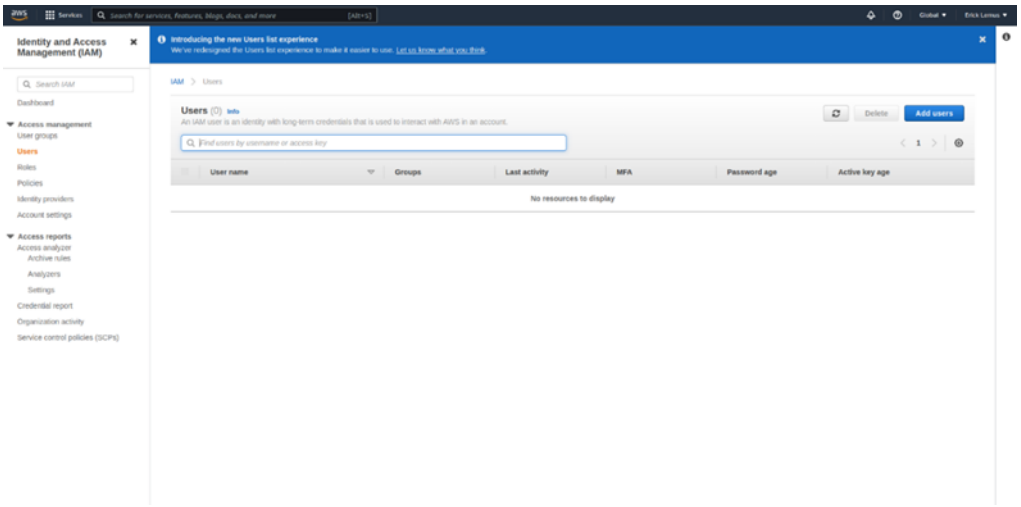
Figura 33. **Panel principal de la consola de AWS**



Fuente: elaboración propia, realizado con Microsoft Edge.

- Seleccionar la opción *Users* en la barra lateral izquierda. Se mostrará una tabla de usuarios vacía.
- Seleccionar *Add users*, para crear un nuevo usuario.

Figura 34. **Tabla de usuarios en IAM**



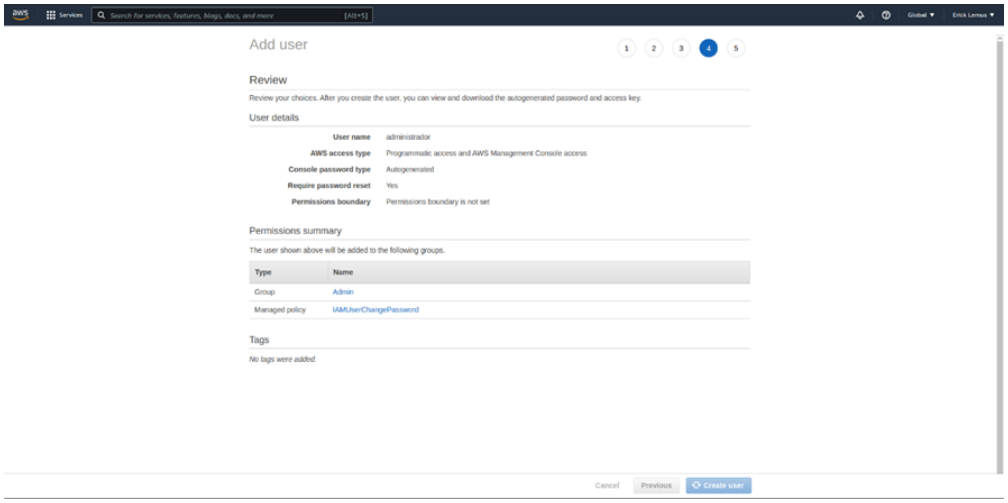
Fuente: elaboración propia, realizado con Microsoft Edge.

- Asignar un nombre de usuario, en este caso administrador.
- En tipo de credenciales de AWS, seleccionar:
  - *Access key – Programmatic access* para habilitar una clave de acceso encriptada que permitirá utilizar herramientas de AWS de desarrollo como API's, AWS CLI, SDK, entre otros.
  - *Password – AWS Managament Control Acces* para habilitar el uso de contraseñas que permitirá al usuario iniciar sesión en la Consola de AWS.
- Al presionar siguiente se mostrará una vista donde se debe crear un grupo de usuario y asignar permisos para finalmente otorgárselos al usuario. Para eso:
  - Seleccionar *Add User to group*, seguidamente seleccionar *Create group*.
  - Asignar un nombre al grupo.

- Para asignar permisos, en el buscador se debe escribir el tipo de permisos que se desear agregar al grupo, en este caso *AdministratorAccess*.
- Seleccionar el grupo del paso anterior o crear uno nuevo.
- Seleccionar *Create group*.

- Presionar *Next: Tags*.
- Si se desea agregar algún valor representativo, se pueden crear *tags* por clave-valor, de lo contrario presionar *Next: Review*.
- Se mostrará los detalles del usuario a crear, si los datos están correctos, presionar *Create user*.

Figura 35. **Detalles del usuario a crear**

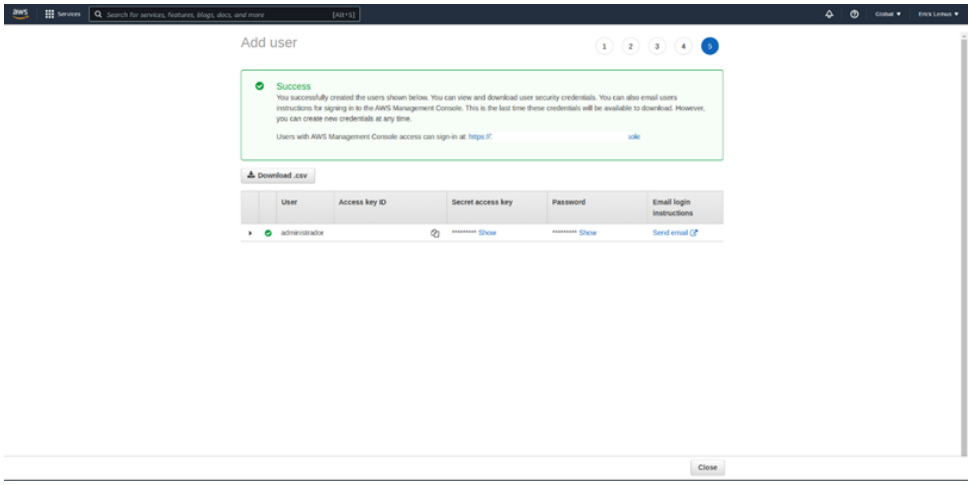


Fuente: elaboración propia, realizado con Microsoft Edge.

- Finalmente se mostrará una tabla con datos como la llave de acceso, llave secreta, contraseña, nombre del usuario, *link* de acceso para la organización y la posibilidad de enviar los datos por correo.
- Copiar los datos o descargar un archivo .csv con los datos mencionados.

- Presionar *Close*, y retornará a la tabla de usuarios donde se puede ver al usuario administrador creado.

Figura 36. **Tabla de datos finales de usuario**



Fuente: elaboración propia, realizado con Microsoft Edge.

- Para crear nuevos usuarios, se recomienda realizarlo con el usuario administrador, siguiendo los pasos anteriores.

A continuación, se muestra una tabla con los usuarios a utilizar, así como sus grupos y permisos correspondientes.

Tabla IX. **Usuarios, grupos y permisos**

Usuario	Grupo	Permisos
administrador	*Admin*	*AdminstratorAccess*
DbAdmin	DB	RDS *Full Access*
TestAdmin	*Test*	EC2 *Full Access*
StorageAdmin	*S3Group*	S3 *Full Access*

Fuente: elaboración propia, realizado con Microsoft Word.

5.1.7. Configurando AWS

AWS ofrece flexibilidad, al momento de trabajar con sus servicios, al cliente, dando la posibilidad de trabajar con la interfaz de usuario conocida como Consola de Administración de AWS, así como con la AWS CLI (AWS *Command Line Interface* o Interfaz de Linea de Comandos). La diferencia entre ambas opciones es la interfaz de uso. La primera opción se caracteriza porque se puede acceder a todos los servicios por medio de la interfaz de usuario que provee el sitio web de AWS, y la segunda opción se caracteriza por el uso de una línea de comandos de consola.

Para fines del presente trabajo de graduación, se hace uso de la AWS CLI, ya que permite configurar de manera más detallada cada servicio que se desea utilizar.

5.1.7.1. Configuración de AWS CLI

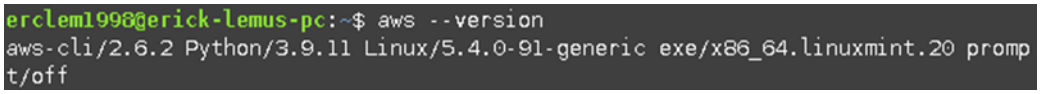
Antes de hacer uso de la AWS CLI, es necesario realizar la instalación de esta dentro de la computadora local y configurar usuarios y llaves secretas que permitirán hacer uso de los servicios. Cabe destacar que, para el proceso de

desarrollo de la aplicación, se está haciendo uso de una distribución Linux, por lo tanto, los comandos pueden cambiar dependiendo de esta.

Para poder instalar esta herramienta, se debe seguir la siguiente secuencia de comandos:

- Abrir la línea de comandos de la distribución Linux.
- Ejecutar las siguientes instrucciones en el orden especificado:
  - *curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"*
  - *unzip awscliv2.zip*
  - *sudo ./aws/install*
- Para verificar la instalación de la herramienta, ejecutar el siguiente comando:
  - *aws --version*
  - Si se ha instalado correctamente, se mostrará un resultado similar a la siguiente figura.

Figura 37. **Verificación de instalación de AWS CLI**



Fuente: elaboración propia, realizado con Linux *Command Line*.

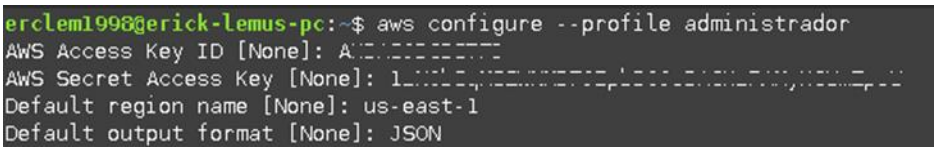
Al finalizar la instalación, es necesario configurar los usuarios que se han creado anteriormente dentro del entorno de AWS CLI. En este caso se debe configurar el usuario administrador. Es importante mencionar que es el mismo proceso para los usuarios creados en IAM que se muestran en la Tabla IX, y los

datos requeridos, en los posteriores pasos, se encuentran en el archivo descargado al momento de crear el usuario.

La configuración mencionada, se obtiene siguiendo los siguientes pasos:

- Abrir la línea de comandos de nuestra distribución Linux.
- Ejecutar la siguiente secuencia de comandos.
    - *aws configure -- profile administrador*
        - Este comando configura el perfil de administrador.
        - Ingresar el AWS *Access Key*, *Secret Access Key*, la región *default* y el formato de resultado, como se muestra en la siguiente figura.

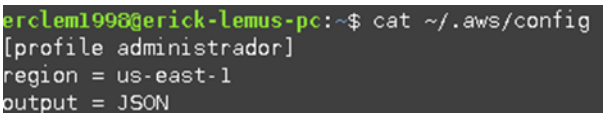
Figura 38. **Configuración de usuario administrador**



Fuente: elaboración propia, realizado con Linux *Command Line*.

- Para verificar que el perfil de usuario ha sido correctamente creado, ejecutar el siguiente comando:
    - *cat ~/.aws/config*
        - Se obtendrá un resultado similar a la siguiente figura.

Figura 39. **Configuración de perfil administrador**



Fuente: elaboración propia, realizado con Linux *Command Line*.

5.1.8. Inicialización de Proyecto de Angular

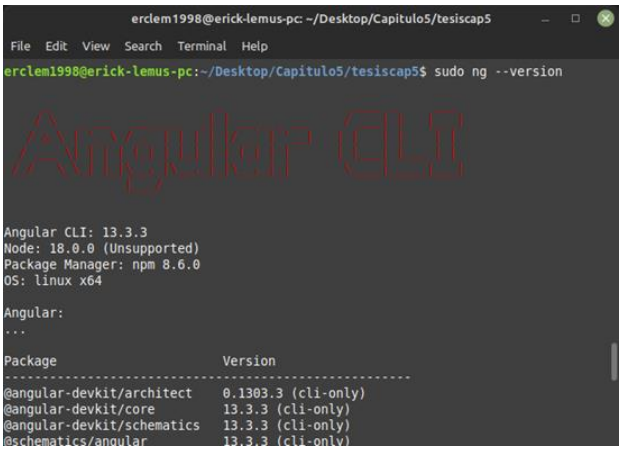
Antes de iniciar, es necesario instalar la interfaz de línea de comandos de este Angular. Posteriormente, verificar la instalación, y, por último, generar el nuevo proyecto de angular.

5.1.8.1. Instalación de Angular CLI

Para instalar y verificar la última versión de Angular CLI, es necesario seguir la siguiente secuencia de comandos:

- Instalación:
  - *npm install -g @angular/cli*
- Verificación:
  - *ng --version*

Figura 40. **Verificación de instalación de Angular CLI**



Fuente: elaboración propia, realizado con Linux *Command Line*.

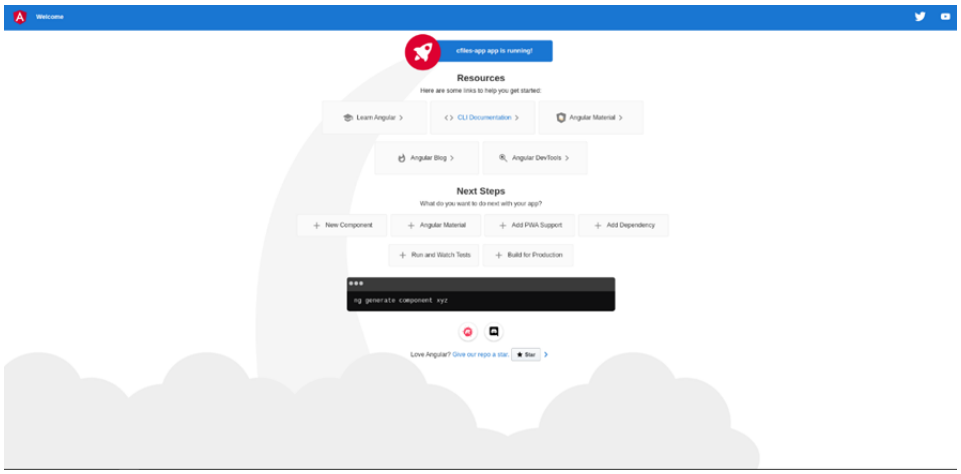
5.1.8.2. Creación de proyecto de Angular

Luego de instalar y verificar la instalación de Angular CLI, se puede crear el proyecto y ejecutar la plantilla por defecto.

Para lograr lo anterior, es necesario seguir la siguiente secuencia de comandos:

- Creación de proyecto:
  - *ng new nombre-app*
- Ejecución de la aplicación:
  - *cd nombre-proyecto*
  - *ng serve*
    - Dirigirse al navegador web e ingresar a localhost:4200.

Figura 41. **Resultado de ejecución de proyecto de Angular**



Fuente: elaboración propia, realizado con Microsoft Edge.

5.1.9. Configuración de Docker y Kubernetes

Durante la ejecución del *pipeline* en Gitlab-Ci, se debe hacer referencia a una instancia EC2 para realizar los procesos de entrega y despliegue continuo. Cada uno de estos procesos se hará con Docker y Kubernetes respectivamente.

Por lo tanto, de las máquinas que se han creado, se elegirá una para instalar las dos herramientas mencionadas. El proceso de instalación es el siguiente:

- Conectarse a la instancia vía SSH (ver 5.2.1.1).
- Instalar Docker.
  - *sudo apt-get update.*
  - *sudo apt-get install docker.io*.
  - Verificar instalación y estado de Docker.

        - *sudo systemctl status docker*.
- Instalar Kubernetes.
    - *curl -o kubectl https://s3.us-west-2.amazonaws.com/amazon-eks/1.23.7/2022-06-29/bin/linux/amd64/kubectl.*
    - *chmod +x ./kubectl.*
    - *mkdir -p $HOME/bin && cp ./kubectl $HOME/bin/kubectl && export PATH=$PATH:$HOME/bin.*
    - *echo 'export PATH=$PATH:$HOME/bin' >> ~/.bashrc*.
    - Verificar instalación.
        - *kubectl version --short –client.*
- Instalar AWS CLI y configurar el usuario administrador (ver 5.1.7.1).
- Crear/actualizar kubeconfig con el clúster de Kubernetes en EKS.
    - *aws eks --region region update-kubeconfig --name cluster_name --profile administrador*.
- Verificar conexión al clúster de EKS.
    - *kubectl get svc*.

5.2. Implementación de la nube de AWS

Antes de utilizar los servicios que ofrece AWS, es importante realizar las configuraciones, sobre esta tecnología, mencionadas en la 5.1.7.

5.2.1. Servicios en la nube

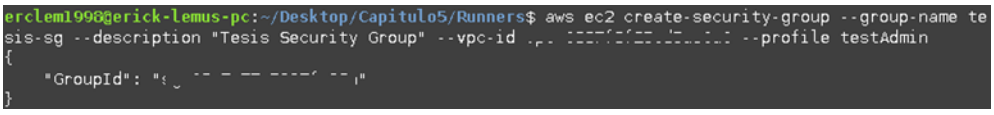
La nube de AWS ofrece una gran gama de servicios que se pueden utilizar conforme a las necesidades de las empresas y sus lógicas de negocio. Para el actual trabajo de investigación, se tiene lo siguiente.

5.2.1.1. Entorno para *test* con EC2

El servicio de Amazon EC2, permite crear máquinas virtuales (instancias) con diferentes sistemas operativos. En el caso actual, se hará uso de instancias de Linux con Ubuntu y se utilizará el usuario *testAdmin*. Para crear una máquina virtual es necesario realizar lo siguiente:

- Crear un par de llaves para acceder a las instancias se crea en el directorio actual).
  - *aws ec2 create-key-pair --key-name NOMBRE --output text > NOMBRE.pem --profile testAdmin*.
- Seleccionar una VPC (en este caso se usará la de por defecto del usuario).
  - *aws ec2 describe-vpcs --profile testAdmin*.
  - Copiar el id de la VPC, que se utilizará en el siguiente paso.
- Crear *security group* (para manejar reglas de entrada y salida).
  - *aws ec2 create-security-group --group-name NOMBRE-SG --description "DESCRIPCION" --vpc-id ID-VPC --profile testAdmin*.
  - Copiar el *id* del *security* group creado, se utilizará en el siguiente paso.

Figura 42. **Creación de *security group***



Fuente: elaboración propia, realizado con Linux *Command Line*.

- Crear reglas de entrada.

    - *aws ec2 authorize-security-group-ingress --group-id ID-SG --protocol tcp --port 22 --cidr x.x.x.x --profile testAdmin*.
- Crear instancia.
    - *aws ec2 run-instances --image-id ami-0eea504f45ef7a8f7 --instance-type TIPO-INSTANCIA --security-group-ids sg-03c7e77c5997fa98a --key-name runner1 --profile testAdmin*.
- Obtener IP de instancia.
    - *aws ec2 describe-instances --profile testAdmin*.
    - Buscar la IP pública de la instancia creada, la cual se utilizará para conectarse via SSH.
- Conectarse a instancia via SSH (debe estar creada la regla de entrada en el puerto 22 del *security group*).
    - *sudo ssh -i /PATH/NOMBRE.pem ubuntu@IP*.

En la siguiente tabla se definen los puertos y cidr (redes permitidas) por cada instancia definida en la arquitectura del presente trabajo de investigación, mostrado en la Figura 26.

Tabla X. **Puertos y CIDR por instancia EC2**

Instancia	Puertos	CIDR
runnerTest	22, 80, 443	0.0.0.0/0
runner	22, 80, 443, 3000 - 5000	0.0.0.0/0

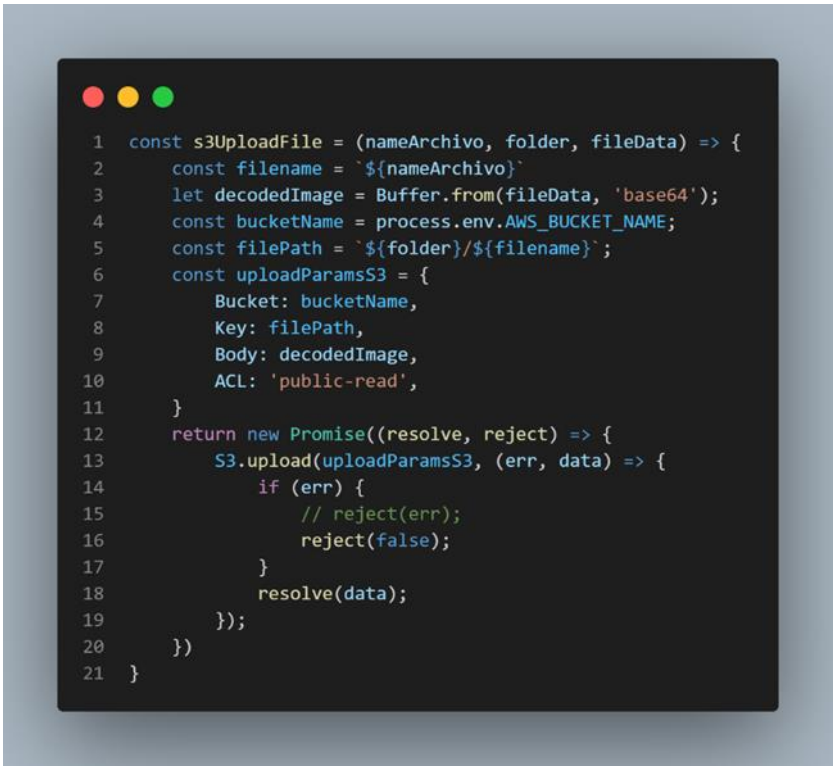
Fuente: elaboración propia, realizado con Microsoft Word 2016.

5.2.1.2. Almacenamiento de archivos en S3

AWS ofrece el servicio de S3, que permite almacenar archivos dentro de un *bucket*, el cual se puede personalizar con base en las necesidades del usuario. Este servicio será utilizado, en el presente trabajo de investigación, para almacenar una amplia gama de archivos que los usuarios de la aplicación podrán subir para acceder a ellos en cualquier momento. Para poder hacer uso del servicio, se debe configurar de la siguiente manera desde la interfaz de usuario de AWS:

- Crear un *bucket*.
  - Dirigirse al servicio de S3.
  - Seleccionar Crear *Bucket*.
  - Asignar un nombre.
  - En la opción Configuración de bloqueo de acceso público para este *bucket*, deseleccionar la opción Bloquear todo el acceso público.
  - Presionar Crear *Bucket*.
- Subir archivos al *bucket*.
  - Implementar AWS-SDK en Node.js (ver 5.3.2).
  - Crear una función, en la carpeta *helpers*, para poder subir un archivo al *bucket*.

Figura 43. **Función para subir archivos a *bucket* de S3**



```
const s3UploadFile = (nameArchivo, folder, fileData) => {
    const filename = `${nameArchivo}`
    let decodedImage = Buffer.from(fileData, 'base64');
    const bucketName = process.env.AWS_BUCKET_NAME;
    const filePath = `${folder}/${filename}`;
    const uploadParamsS3 = {
        Bucket: bucketName,
        Key: filePath,
        Body: decodedImage,
        ACL: 'public-read',
    }
    return new Promise((resolve, reject) => {
        S3.upload(uploadParamsS3, (err, data) => {
            if (err) {
                // reject(err);
                reject(false);
            }
            resolve(data);
        });
    })
}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.2.1.3. Manejo de Base de Datos con RDS

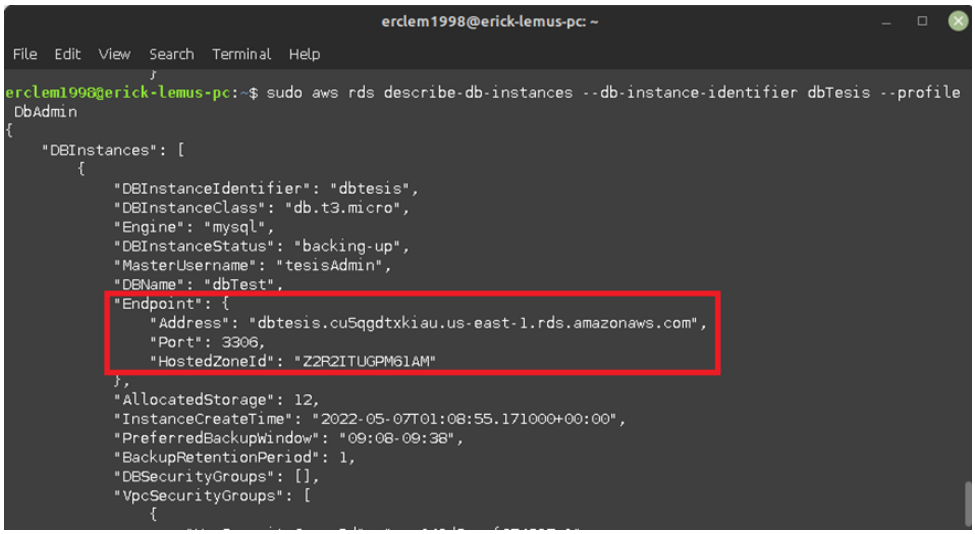
Cuando se desea hacer uso de una base de datos en producción, AWS RDS es una muy buena opción. Para crear una base de datos MySQL en RDS se deben ejecutar las siguientes instrucciones (se hará uso del usuario *DbAdmin*):

- Crear un par de llaves para acceder a las instancias (se crea en el directorio actual).

- *sudo aws rds créate-db-instance --db-name* NOMBRE *–db-instance-identifier DB-ID –master-user-password PASSWORD --engine mysql --db-instance-class* TIPO *–allocated-storage* CAPACIDAD *--profile DbAdmin.*

- Obtener puerto y url de acceso.
  - *sudo aws rds describe-db-instances --db-instance-identifier DB-ID --profile DbAdmin*.

Figura 44. **Puerto y URL de la Base de Datos**



Fuente: elaboración propia, realizado con Linux *Command Line*.

Al finalizar el proceso de creación en el servicio de RDS, se puede utilizar la url y puertos asignados, junto a la contraseña escrita al momento de la creación, para poder conectarse a la instancia de base de datos.

5.2.1.4. Reconocimiento con *Rekognition*

AWS *Rekognition*, es un servicio que permite realizar diferentes tipos de análisis con imágenes y videos. En el actual trabajo de graduación, se utilizarán algunas funciones que permiten almacenar los rostros y analizarlos posteriormente para realizar un reconocimiento facial exitoso.

Antes de utilizar cada función de *Rekognition*, es necesario implementar AWS-SDK en Node.js y realizar las configuraciones para usar el servicio (ver 5.3.2).

Para poder hacer un reconocimiento facial, es necesario realizar lo siguiente:

- Crear colección de rostros.

Figura 45. **Función para crear una colección de rostros**

```
const createCollection = async (collectionName) => {
    return new Promise((resolve, reject) => {
        var colparams = {
            CollectionId: collectionName
        }
        Rekognition.createCollection(colparams, function (err, data) {
            if (err) {
                reject(false);
            } // an error occurred
            else {
                resolve(data);
            };
        });
    })
}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

- Agregar rostro a la colección de imágenes.

Figura 46. **Función para agregar rostros a una colección**

```
const addFaceToCollection = async (Key) => {
    var par = {
        CollectionId: "tesisphotocollection",
        Image: {
            S3Object: {
                Bucket: process.env.AWS_BUCKET_NAME,
                Name: Key //lugar donde se encuentra almacenada la foto dentro del bucket
            }
        }
    }
    try {
        return new Promise((resolve, reject) => {
            Rekognition.indexFaces(par, (err, data) => {
                if (!err) {
                    resolve(data)
                }
                console.log(err)
                reject(false)
            });
        })
    } catch (error) {
        console.log(error)
    }
}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

- Comparar rostro de inicio de sesión con rostros de la colección.

Figura 47. **Función para comparar rostros**

```
const compareFacesLogin = async (Key) => {
    try {
        var params = {
            CollectionId: process.env.AWS_REKOGNITION_COLLECTION,
            FaceMatchThreshold: 95,
            Image: {
                S3Object: {
                    Bucket: process.env.AWS_BUCKET_NAME,
                    Name: Key //utilizamos el key de la imagen subida a S3
                }
            },
            MaxFaces: 1//queremos un solo faceID que es el de la cara reconocida
        };
        return new Promise((resolve, reject) => {
            Rekognition.searchFacesByImage(params, function (err, data) {
                if (err) {
                    reject(false)
                }
                resolve(data)
            })
        })
    } catch (error) {
        console.log(error)
    }
}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

Nota: Es importante destacar que para usar las funciones de reconocimiento facial debemos subir las imágenes a S3 y almacenar sus referencias (propiedad *Key*) en la base de datos, en este caso es en la tabla de usuarios.

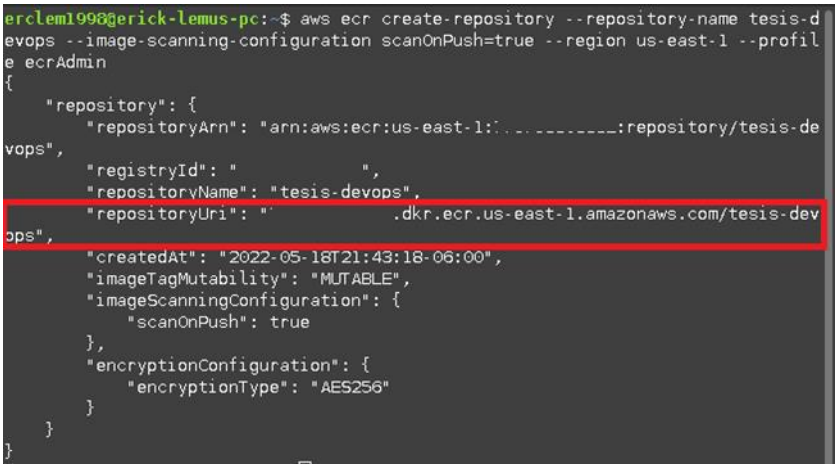
5.2.1.5. Versionamiento de contenedores en ECR

Amazon ECR, es un servicio que permite almacenar los contenedores que se generan para la aplicación. En este servicio existen repositorios públicos y privados. La elección del tipo de repositorio quedará sujeto a las necesidades de cada usuario. El presente trabajo de investigación hace uso de un repositorio

privado, el cual se crea y se inicia sesión (utilizando el usuario *ecrAdmin*), respectivamente, de la siguiente manera:

- Crear repositorio.
  - *sudo aws ecr create-repository --repository-name NOMBRE --image-scanning-configuration scanOnPush=true –region us-east-1 --profile ecrAdmin*.
  - Copiar el uri de acceso al repositorio, se utilizará en el siguiente paso.

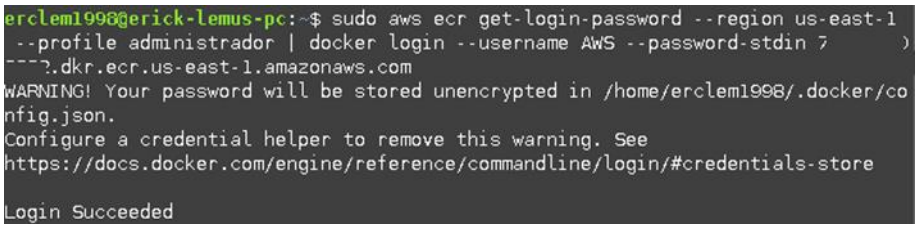
Figura 48. **Creación de repositorio de contenedores en ECR**



Fuente: elaboración propia, realizado con Linux *Command Line*.

- Iniciar sesión en repositorio, utilizando docker (se debe tener instalado previamente).
  - *sudo aws ecr get-login-password --region us-east-1 --profile ecrAdmin | docker login -username AWS --password-stdin URI-REPOSITORIO*.

Figura 49. **Inicio de sesión en repositorio ECR utilizando Docker**



Fuente: elaboración propia, realizado con Linux *Command Line*.

5.2.1.6. Clúster de Kubernetes con EKS

Crear un clúster de Kubernetes en el servicio de EKS es muy sencillo, ya que gracias a la herramienta de línea de comandos para administrar clústeres en EKS (eksctl), solo basta ejecutar una sola instrucción.

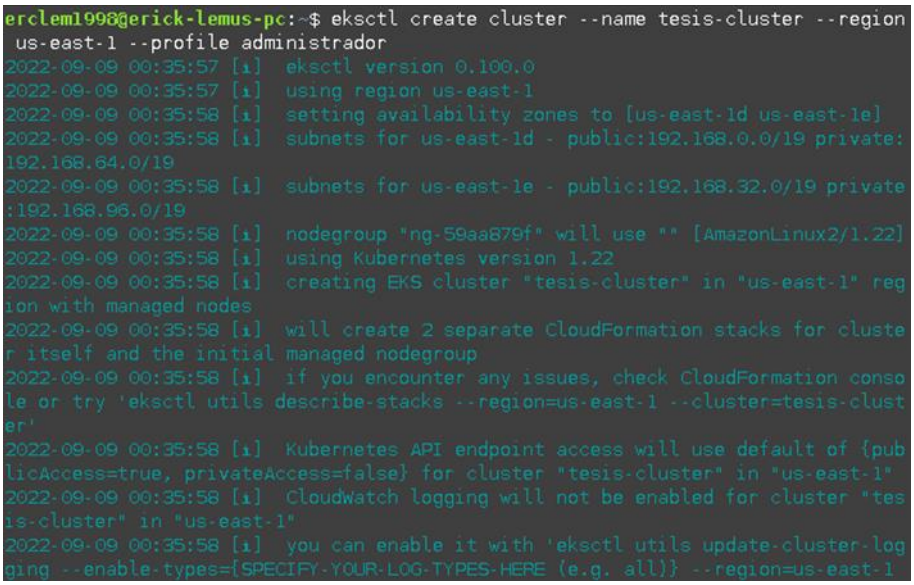
Es importante resaltar que para utilizar la herramienta eksctl se debe instalar previamente kubernetes y eksctl, como se indica en la 5.1.9, que también es válido para utilizar en la computadora personal.

Para crear un clúster en EKS, se debe realizar lo siguiente:

- Tener instalado Kubernetes (ver 5.1.9).
- Instalar eksctl.
  - *curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp*.
  - *sudo mv /tmp/eksctl /usr/local/bin.*
  - *eksctl version*.
- Crear clúster de Kubernetes utilizando eksctl.

- *eksctl create cluster --name tesis-cluster --region us-east-1 –profile administrador*.
- El comando anterior configurará automáticamente lo necesario para ejecutar comandos de kubectl sobre el clúster de Kubernetes en EKS.

Figura 50. **Creación de clúster de Kubernetes en EKS con eksctl**



Fuente: elaboración propia, realizado con Linux *Command Line*.

5.2.1.7. Uso de dominios con *Route* 53

Route 53 es un servicio de DNS que permite asignar un dominio a la aplicación web. Para poder obtener un dominio, es primordial comprarlo por medio de la interfaz de AWS y luego realizar las configuraciones pertinentes con respecto del balanceador de carga, la aplicación web y el certificado SSL. Para esto se realizará lo siguiente:

- Registrar dominio desde la interfaz de usuario de AWS.
    - Seleccionar el servicio de *Route* 53.
    - Seleccionar Registrar dominio.
    - Escribir el nombre del dominio, seleccionar la extensión (el costo depende del tipo de la extensión) y presionar comprobar.
    - Si el dominio se encuentra disponible, añadir al carrito y presionar en siguiente.
    - Rellenar el formulario de datos del titular del dominio y presionar en siguiente.
    - Aceptar los términos y condiciones del AWS para el registro de dominios, y seleccionar en finalizar pedido.
    - Después de finalizar el pedido, el registro del dominio tardará entre 15 a 60 minutos.
- Registrar zona alojada.
    - Al comprar el dominio con *Route* 53, este crea automáticamente la zona alojada del mismo.
- Configurar registros en zona alojada.
    - Seleccionar la zona alojada de nuestro dominio.
    - Presiona Crear un registro.
    - En política de direccionamiento, seleccionar Direccionamiento sencillo.
    - Seleccionar Definir un registro simple.
    - Agregar el nombre del subdominio solo si es necesario, si se deja en blanco este hace referencia al nombre del dominio.
    - En Tipo de registro, seleccionar A: dirige el tráfico a una dirección IPv4 y a algunos recursos de AWS.
    - Seleccionar el servicio de referencia, puede ser S3, un Balanceador de carga, entre otros.
    - Presionar en Definir un registro simple.

- Realizar el proceso para los registros mostrados en la Tabla XI.

Tabla XI. **Registros necesarios en zona alojada**

Subdominio	Tipo de registro	Servicio
nombre.ext	A	Balanceador de carga
backend.nombre.ext	A	Balanceador de carga
www.nombre.ext	A	Balanceador de carga
test.nombre.ext	A	Dirección IP

Fuente: elaboración propia, realizado con Microsoft Word 2016.

- Configurar certificado SSL.
  - Se realiza al solicitar un certificado en ACM (ver 5.2.1.8).

Nota: cada servicio mostrado en la Tabla XI ya debe estar funcional para hacer uso de ellos

5.2.1.8. SSL con Amazon Certification Manager

AWS ofrece el servicio de ACM que permite crear un certificado SSL/TLS, público o privado, para asegurar la aplicación. En el caso del presente trabajo de investigación, se creará un certificado público realizando lo siguiente:

- Crear certificado.
  - Dirigirse al servicio de ACM.
  - Seleccionar Solicitar un certificado.
  - En tipo de certificado, Seleccionar un certificado público, y presionar siguiente.
  - Escribir el nombre de dominio y subdominios a utilizar.

- En método de validación, seleccionar Validación de DNS.
- Seleccionar Solicitar.

- Asignar certificado SSL/TLS al dominio en *Route* 53.
  - Dirigirse al apartado Enumerar certificados.
  - Seleccionar el certificado recién creado.
  - Seleccionar Crear registros en *Route* 53
  - Presionar Crear registros.
  - Al finalizar, el certificado se agregará como un registro en la zona alojada del dominio.

5.3. Implementación de microservicios con Node.js

Los microservicios en Node.js, no son más que un directorio que contienen una única funcionalidad de la aplicación, se encuentra aislada del resto de microservicios y que al unirlas funcionan como un todo. Es importante que, dentro de estos directorios, se encuentre un archivo Dockerfile que permitirá crear un contenedor de Docker para el manejo y control de microservicios en Kubernetes.

5.3.1. Creación y estructura de un microservicio

Para crear un microservicio en Node.js, es suficiente con crear un proyecto de este *framework* utilizando el comando *npm init*, el cual genera una plantilla con los archivos necesarios para ejecutarlo.

Luego de crear el proyecto del microservicio a desarrollar, es importante utilizar una estructura que permita mantener ordenado y legible el código de este. Una estructura recomendada es la siguiente:

5.3.1.1. *Controller*

Contendrá únicamente las llamadas a los métodos y funciones que realizarán ciertas acciones del microservicio.

5.3.1.2. *Database*

Usualmente contiene los archivos necesarios para realizar la lógica de la conexión a la base de datos.

5.3.1.3. *Middlewares*

Este directorio contendrá funciones que permitirán realizar validaciones de las peticiones HTTP antes de llegar al controlador del microservicio.

5.3.1.4. *Helpers*

Contendrá todas las funciones que realizarán cierta acción o paso del microservicio. Estas funciones son llamadas en el controlador.

5.3.1.5. *Routes*

En este directorio se agregará el archivo con la ruta del microservicio. Algo importante, es que en este archivo se realiza la llamada al controlador y se hace uso de los *middlewares*.

5.3.1.6. *Specs*

Directorio que contiene todas las pruebas (*tests*) necesarias del microservicio.

5.3.1.7. Carpeta de variables de entornos

Usualmente, esta carpeta es nombrada como .env, y es donde se agregarán archivos con extensión .env, que permitirá controlar, localmente, las variables de entorno de la aplicación durante su desarrollo. Estas variables serán configuradas, posteriormente, en Gitlab-Ci para el entorno de producción.

5.3.1.8. AWS

Se crea únicamente si se hace uso de los servicios de AWS, y este contiene las configuraciones necesarias para utilizar los servicios.

5.3.2. Implementación de AWS-SDK

El AWS-SDK es un paquete que contiene las funciones necesarias para conectarse a los servicios de AWS. Para poder instalarlo se debe ejecutar *npm install aws-sdk --save*.

Luego de su instalación, se debe crear los archivos que contienen las llaves secretas y de acceso para conectarse a un servicio de AWS. Las llaves mencionadas se obtienen al momento de crear los usuarios (ver 5.1.6).

5.3.3. Integración de Base de Datos

Para conectar a la base de datos, se puede realizar de dos maneras; creando consultas o utilizando un ORM. A continuación, se detalla cada opción.

5.3.3.1. Creación de consultas

Para este caso, se puede crear una función que reciba una consulta o *query* SQL, que es enviado por el transportador (librería) para realizar cierta acción en la base de datos.

Figura 51. **Consultas SQL para verificar si un usuario existe**

```
const userEmailExists = (userEmail = '') => {
    return new Promise((resolve, reject) => {
        pool.getConnection((err, conn) => {
            conn.query(`SELECT U.idUsuario
                        FROM
                            USUARIO AS U
                        WHERE
                            U.username='${userEmail}' OR U.email='${userEmail}'`, (err, data) => {
                conn.release()
                const userEmailList = data
                if (userEmailList.length === 0) {
                    reject( new Error(`El usuario o correo no existe`) );
                }
                resolve(true)
            });
        })
    });
}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

El método de conexión puede variar por cada librería.

5.3.3.2. Uso de ORM Sequelize

Un ORM es un modelo que permite mapear estructuras relacionales de las bases de datos de este tipo. Esto quiere decir que no es necesario hacer las consultas manualmente, ya que los realiza por medio de la relación de estas.

Sequelize es un ORM para Javascript, y para poder utilizarlo se debe realizar lo siguiente:

- Instalar Sequelize.
  - *npm install sequelize –save*
- Crear un modelo para la tabla o tablas objetivo.

Figura 52. **Modelo de tabla Carpeta**

Fuente: elaboración propia, realizado con Visual Studio Code.

- Realizar una función para realizar la petición a la base de datos.

Figura 53. **Función para crear carpetas utilizando Sequelize**

```
const createFolder = async (nombre = '', descripcion = '', idPadre = 0, idUsuario=0) => {
    try {
        const carpeta = new Carpeta({
            nombre: nombre,
            descripcion: descripcion,
            idPadre: idPadre,
            idUsuario: idUsuario,
        })

        const newCarpeta= await (await carpeta.save()).reload();

        return {
            carpeta: newCarpeta
        }

    } catch (error) {

    }
}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.3.4. Uso de *token* con JWT

Un JSON Web *Token* es un estándar de seguridad de información que permite crear un *token*, el cual es utilizado para transportar información entre aplicaciones y servicios en la web. Este necesita una llave secreta única para poder encriptar y desencriptar la información transportada. Para instalar la librería, es necesario ejecutar *npm install jsonwebtoken –save*.

5.3.4.1. Creación

La creación es muy sencilla, y únicamente necesita hacer uso de la librería, una llave secreta única y crear un *payload* (cuerpo con la información a encriptar).

Figura 54. **Función para generar un *token* con JWT**

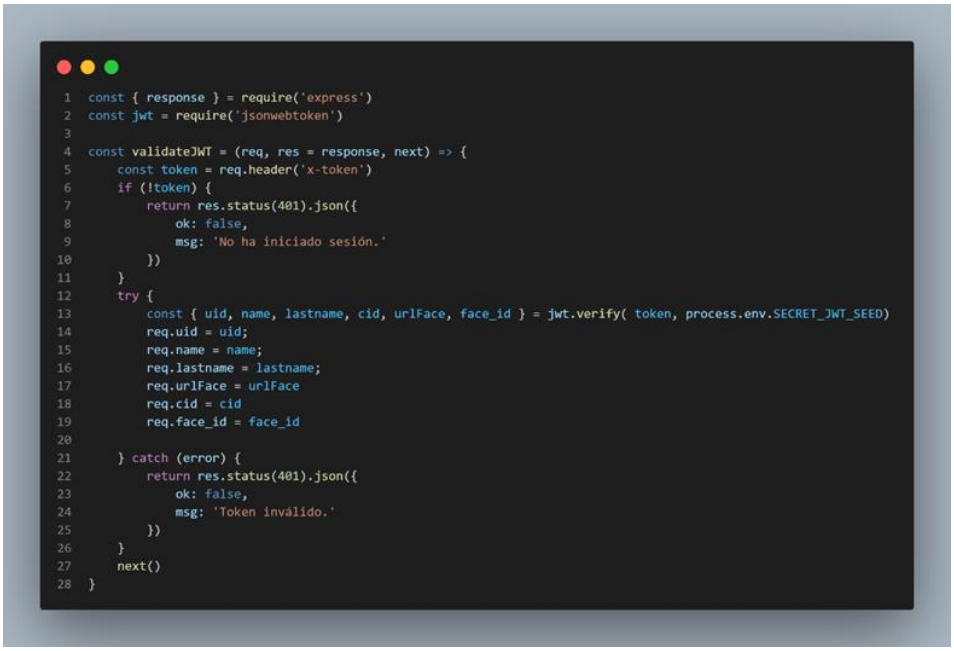
```
const jwt = require ('jsonwebtoken')

const generarJWT = (idUsuario, nombres, apellidos, idMainFolder, urlFace, face_id) => {

    const payload = {
        uid: idUsuario,
        name: nombres,
        lastname: apellidos,
        cid: idMainFolder,
        urlFace: urlFace,
        face_id: face_id
    }

    return new Promise((resolve, reject)=>{
        jwt.sign(payload, process.env.SECRET_JWT_SEED, {
            expiresIn: '2h'
        }, (err, token)=>{
            if(err){
                console.log(err)
                reject(err)
            } else{
                resolve(token)
            }

        })
    })

}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.3.4.2. Validación

Para validar un *token*, únicamente se necesita recibir el *token* a validar y la llave secreta.

Figura 55. **Función para validar un *token* con JWT**



```
const { response } = require('express')
const jwt = require('jsonwebtoken')

const validateJWT = (req, res = response, next) => {
    const token = req.header('x-token')
    if (!token) {
        return res.status(401).json({
            ok: false,
            msg: 'No ha iniciado sesión.'
        })
    }
    try {
        const { uid, name, lastname, cid, urlFace, face_id } = jwt.verify( token, process.env.SECRET_JWT_SEED)
        req.uid = uid;
        req.name = name;
        req.lastname = lastname;
        req.urlFace = urlFace
        req.cid = cid
        req.face_id = face_id

    } catch (error) {
        return res.status(401).json({
            ok: false,
            msg: 'Token inválido.'
        })
    }
    next()
}
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.3.5. Conteinerizando un microservicio

Se debe recordar que, para poder crear un contenedor de Docker, y poder utilizarlo en Kubernetes, se necesita crear un archivo llamado Dockerfile en la raíz del proyecto del microservicio. Este archivo contiene una serie de instrucciones que permiten crear un contenedor para ejecutarlos. Posteriormente, el contenedor es utilizado en un archivo docker-compose para realizar las pruebas pertinentes en el ambiente de *testing* y en el entorno de producción con Kubernetes.

Figura 56. **Instrucciones del archivo Dockerfile para *backend***

Fuente: elaboración propia, realizado con Visual Studio Code.

5.4. Implementación de una aplicación web con Angular

Antes de iniciar la implementación de la aplicación haciendo uso de Angular, es importante considerar lo siguiente:

5.4.1. Creación de componentes, interfaces y servicios

Al momento de crear una aplicación con el *framework* Angular, se tiene que considerar el uso correcto de los componentes, así como de interfaces y servicios. Se debe recordar que un componente es un elemento pequeño con cierta funcionalidad en las aplicaciones. De la misma forma, una interfaz (*interface*) es una porción de código que permite modelar, mediante un tipado estricto, un dato de la aplicación. Y por último un servicio es el elemento que permite realizar peticiones hacia un *backend* (API's, REST API´s, entre otros).

La creación y uso de estos elementos es realmente sencilla, ya que basta con una simple instrucción para que genere una plantilla donde se indicarán las instrucciones necesarias para cada uno de los mencionados.

El proceso para cada elemento en cuestión se detalla a continuación:

- Uso de componentes.
  - Crear componente.
    - *ng generate component* RUTA.
- Uso de interfaces.
  - Crear interfaz.
    - *ng generate interface* RUTA.

Figura 57. **Creación de una interfaz para países**

Fuente: elaboración propia, realizado con Visual Studio Code.

- Uso de servicios.
  - Crear servicio.
    - ng *generate service* RUTA.

Figura 58. **Servicio para obtener lista de países**

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';
import { GetCountriesResponse } from '../../interfaces/country.interface';

@Injectable({
  providedIn: 'root'
})
export class CountryServicesService {

  private baseUrl: string = environment.baseUrlCountry!

  constructor(private http: HttpClient) { }

  getCountries(){

    const url = `${this.baseUrl}/api/country/getCountries`;
    const httpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json',
      }),
    };

    return this.http.get<GetCountriesResponse>(url, httpOptions);
  }

}

```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.4.2. Configuración e implementación de *RoutingModule*

El *routing module* es un elemento esencial dentro de las aplicaciones en angular, ya que permite crear las rutas para la aplicación. Este módulo es creado automáticamente al generar el proyecto en angular.

Su configuración y utilización es muy sencilla, y consta de los siguientes pasos:

- Configurar rutas.
  - Para esto se debe indicar la ruta y el componente destino.

Figura 59. **Configuración de *RoutingModule***

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './components/home/home.component';
import { RegisterComponent } from './components/register/register.component';
import { LoginComponent } from './components/login/login.component';

const routes: Routes = [
  {
    path:'',
    component: HomeComponent
  },
  {
    path:'registrarse',
    component: RegisterComponent
  },
  {
    path:'login',
    component: LoginComponent
  },
  {
    path: '**',
    redirectTo: ''
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Fuente: elaboración propia, realizado en Visual Studio Code.

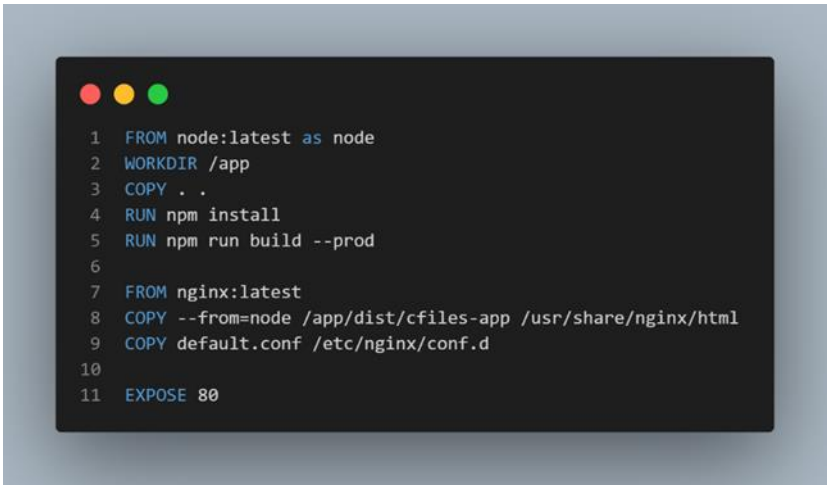
- Implementar rutas, utilizando *routerLink* en HTML.
  - Para esto, se puede implementar de dos maneras:
    - Utilizando una ruta fija. Si es el caso, se utiliza *routerLink*.
    - Utilizando una ruta variable. Si es el caso, se utiliza [*routerLink*].
- Implementar navegación mediante *navigateByUrl*(RUTA).
- Si en dado caso se necesita implementar rutas internas, se puede crear un nuevo *routing module*.

- *ng generate module --routing=true*.

5.4.3. Conteinerización *backend*

El contenedor se crea de manera similar al *backend*, con la diferencia que se hace uso de instrucciones para angular y se implementa nginx.

Figura 60. **Instrucciones del archivo Dockerfile para *frontend***



Fuente: elaboración propia, realizado con Visual Studio Code.

5.5. *Testing*

Actualmente, es importante realizar las pruebas automatizadas respectivas a las aplicaciones que se están creando, ya que permiten verificar la calidad de esta. A continuación, se detallan las pruebas respectivas.

5.5.1. *Testing* con Jasmine para microservicios

Para realizar pruebas automatizadas de los microservicios, se debe realizar lo siguiente:

- Instalar Jasmine.
  - *npm install jasmine –save-dev*.
- Configurar set de pruebas automáticas.
  - Dirigirse al archivo *package*.json.
  - En la sección de “*scripts*”, agregar "*test*": "jasmine".
- Crear pruebas.
  - Crear una carpeta llamada *spec* en la raíz del proyecto del servicio.
  - En la carpeta del paso anterior, crear un archivo con extensión.spec.js.

Figura 61. **Pruebas en Node.js con Jasmine**

```
const { usernameExists, emailExists } = require('../../helpers/db-validator');
const { mockDb } = require('../mockDb/mockDb');

describe('Mock Test to helpers', function () {

    let helpers = {
        usernameExists: usernameExists,
        emailExists: emailExists
    }

    const fakeCheckEmail = (email='') => {
        for(let i = 0; i<mockDb.length; i++){
            if(mockDb[i].email===email){
                throw new Error(`El correo ${email} ya existe`)
            }
        }
        return true
    }

    it('Should check if email exist on database', async () => {


        spyOn(helpers,'emailExists').and.callFake(fakeCheckEmail)
        expect(()=>{
            helpers.emailExists('erc1@gmail.com')
        }).toThrow();
        expect(helpers.emailExists).toHaveBeenCalled();

    });

});
```

Fuente: elaboración propia, realizado con Visual Studio Code.

- Ejecutar pruebas.
    - *npm run test*.

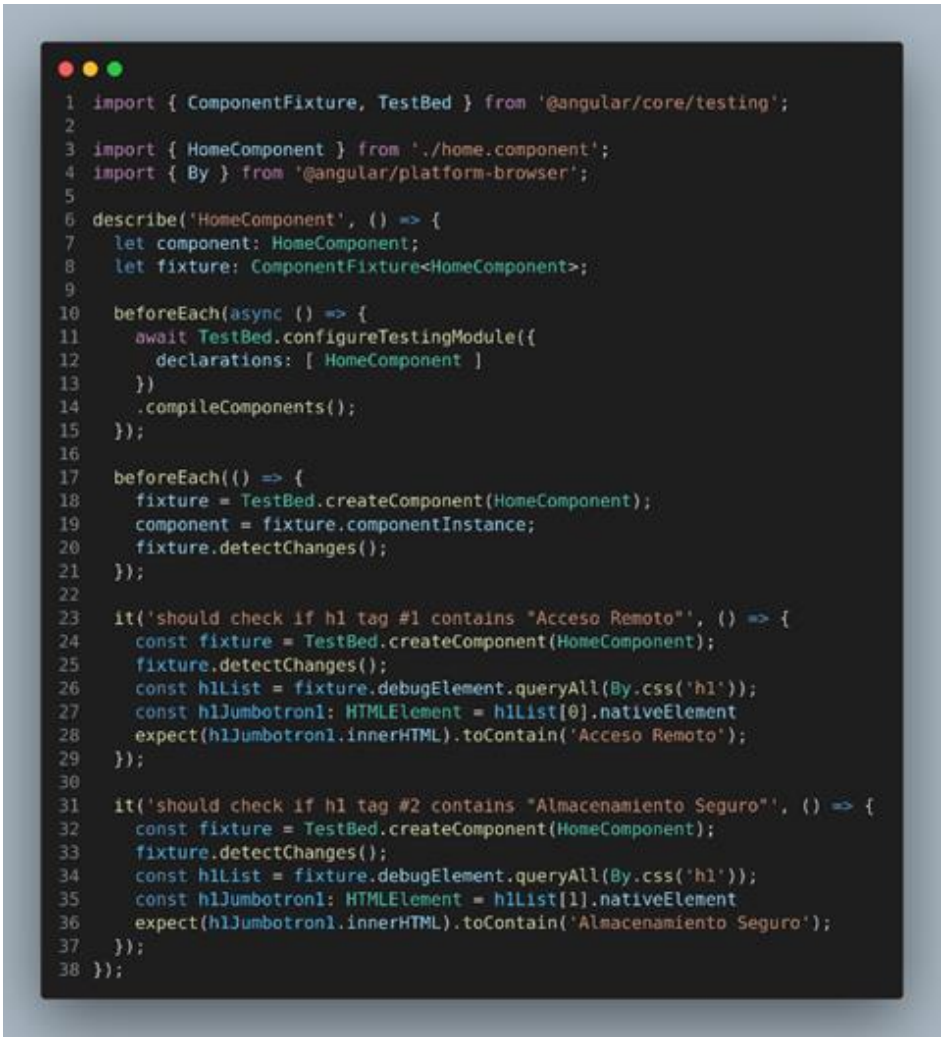
5.5.2. *Testing* con Karma y Jasmine para Angular

Las pruebas en componentes de Angular también se ha de realizar con Jasmine, pero también utiliza Karma para los componentes de este *framework*. Al momento de crear un proyecto de Angular, se agregan sus dependencias.

Para realizar pruebas en angular, es necesario realizar lo siguiente:

- Crear pruebas.
  - Al momento de crear un nuevo componente o servicio, automáticamente se añade el archivo de pruebas inicial.
  - Crear las pruebas necesarias para el componente o servicio.

Figura 62. **Pruebas en Angular con Jasmine y Karma**



```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { HomeComponent } from './home.component';
import { By } from '@angular/platform-browser';

describe('HomeComponent', () => {
  let component: HomeComponent;
  let fixture: ComponentFixture<HomeComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ HomeComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(HomeComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should check if h1 tag #1 contains "Acceso Remoto"', () => {
    const fixture = TestBed.createComponent(HomeComponent);
    fixture.detectChanges();
    const h1List = fixture.debugElement.queryAll(By.css('h1'));
    const h1Jumbotron1: HTMLElement = h1List[0].nativeElement
    expect(h1Jumbotron1.innerHTML).toContain('Acceso Remoto');
  });

  it('should check if h1 tag #2 contains "Almacenamiento Seguro"', () => {
    const fixture = TestBed.createComponent(HomeComponent);
    fixture.detectChanges();
    const h1List = fixture.debugElement.queryAll(By.css('h1'));
    const h1Jumbotron1: HTMLElement = h1List[1].nativeElement
    expect(h1Jumbotron1.innerHTML).toContain('Almacenamiento Seguro');
  });
});
```

Fuente: elaboración propia, realizado con Visual Studio Code.

- Ejecutar pruebas.
    - *npm run test*.
- Para entornos automatizados en Gitalb-Ci, se debe añadir la configuración *Headless* que simulará un navegador.

Figura 63. **Configuración *Headless* en archivo karma.conf**

```
module.exports = function (config) {
  process.env.CHROME_BIN = require('chromium').path
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular-devkit/build-angular'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage'),
      require('@angular-devkit/build-angular/plugins/karma')
    ],
    client: {
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    coverageIstanbulReporter: {
      dir: require('path').join(__dirname, './coverage/frontend'),
      reports: ['html', 'lcovonly', 'text-summary'],
      fixWebpackSourcePaths: true
    },
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: [
      "ChromeHeadlessNoSandbox"
    ],
    customLaunchers: {
      ChromeHeadlessNoSandbox: {
        base: "ChromeHeadless",
        flags: [
          "--no-sandbox"
        ]
      }
    },
    singleRun: false,
    restartOnFileChange: true
  });
};
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.6. Implementación y despliegue en un clúster de Kubernetes

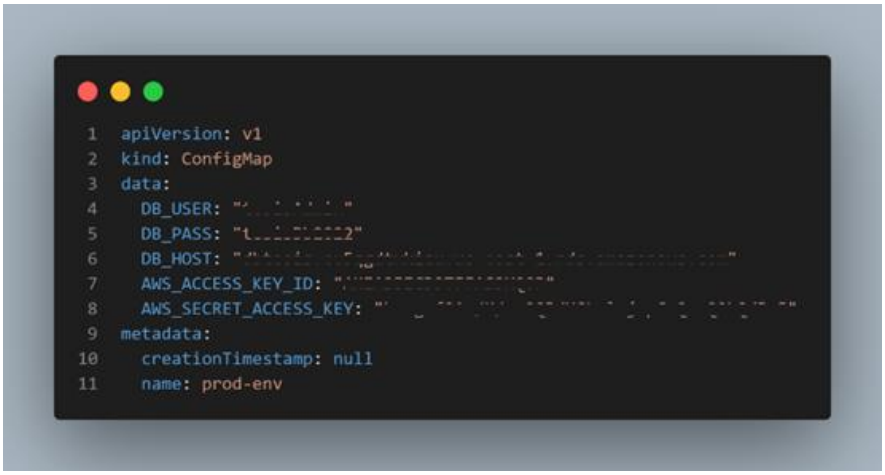
Para el proceso de despliegue en un clúster de Kubernetes, solo se deben ir añadiendo los nuevos cambios en el archivo de Gitlab-Ci (ver secciones *5.7.2* y *5.7.3*) para que su despliegue se realice de manera continua.

Los cambios mencionados, se realizan en las siguientes implementaciones:

5.6.1. Variables de entorno en *ConfigMap*

- Un *ConfigMap*, ayuda a establecer pares de clave/valor, que permite almacenar datos secretos (como API *keys*, contraseñas, entre otros.). Estos son solicitados por los *deployments* para construir el contenedor de Docker y únicamente se necesita un archivo llamado prod-env.*configmap*.yaml, en este caso, para agregar todos los elementos secretos (ver Figura 64). Además, debido a la sensibilidad de la información, debe ejecutarse previamente con la instrucción *kubectl apply -f prod-env.configmap.yaml*.

Figura 64. **Configuración de un *ConfigMap***

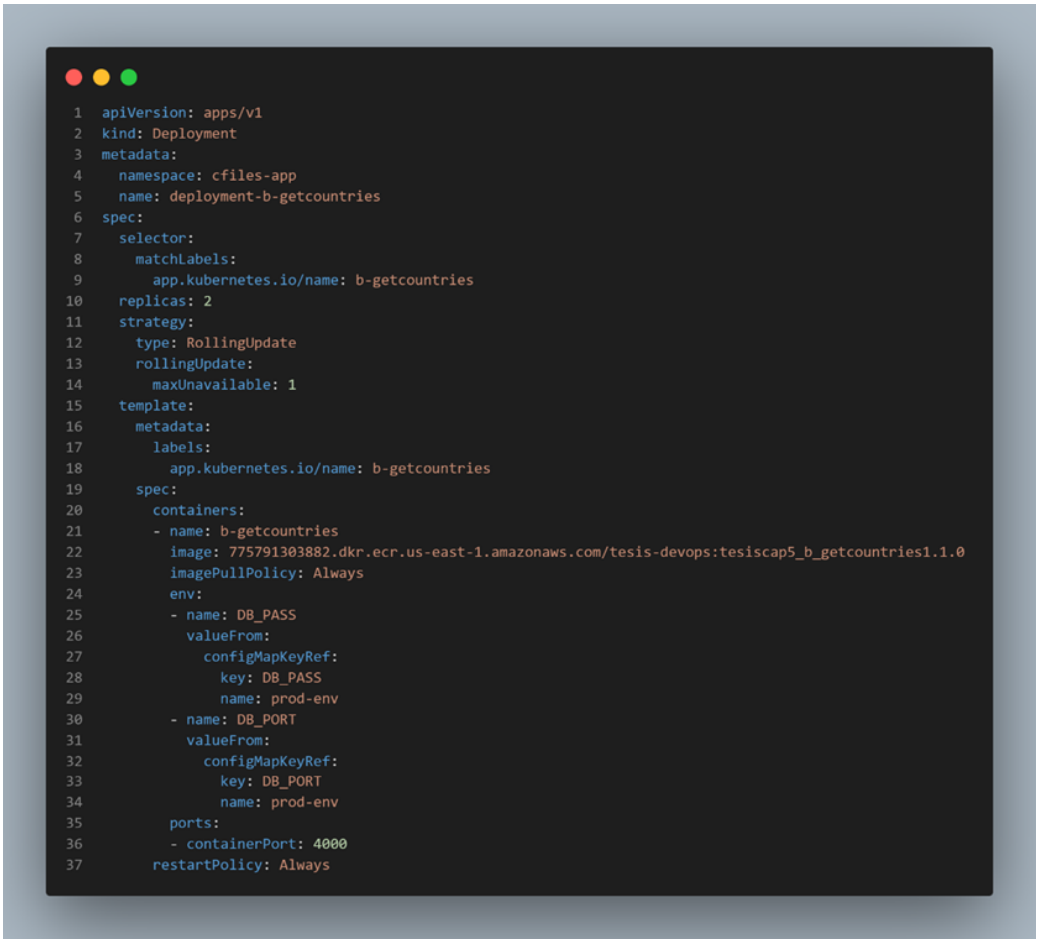


Fuente: elaboración propia, realizado con Visual Studio Code.

5.6.2. Implementando *Deployments*

Un *deployment* no es más que un archivo de configuración, en el cual se indica la cantidad de réplicas, identificador, estrategias de actualización, puerto de escucha, contenedor de Docker a asignar y sus variables de entorno (si aplica).

Figura 65. **Configuración de un *Deployment***



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: cfiles-app
  name: deployment-b-getcountries
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: b-getcountries
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app.kubernetes.io/name: b-getcountries
    spec:
      containers:
      - name: b-getcountries
        image: 775791303882.dkr.ecr.us-east-1.amazonaws.com/tesis-devops:tesiscap5_b_getcountries1.1.0
        imagePullPolicy: Always
        env:
        - name: DB_PASS
          valueFrom:
            configMapKeyRef:
              key: DB_PASS
              name: prod-env
        - name: DB_PORT
          valueFrom:
            configMapKeyRef:
              key: DB_PORT
              name: prod-env
        ports:
        - containerPort: 4000
      restartPolicy: Always
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.6.3. Configuración de *services*

A diferencia del anterior, un *service* permite indicar un identificador de servicio, el tipo de nodos a utilizar, protocolo de transferencia de datos (recurrentemente TCP) y los puertos de origen y destino. Este archivo es llamado en el *ingress* para redirigir el tráfico entrante y saliente a los *deployments*.

Figura 66. **Configuración de un *service* para países**

```yaml
apiVersion: v1
kind: Service
metadata:
  namespace: cfiles-app
  name: service-b-getcountries
spec:
  ports:
    - port: 4000
      targetPort: 4000
      protocol: TCP
  type: NodePort
  selector:
    app.kubernetes.io/name: b-getcountries
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.6.4. Uso de un Ingress *Frontend* y *Backend*

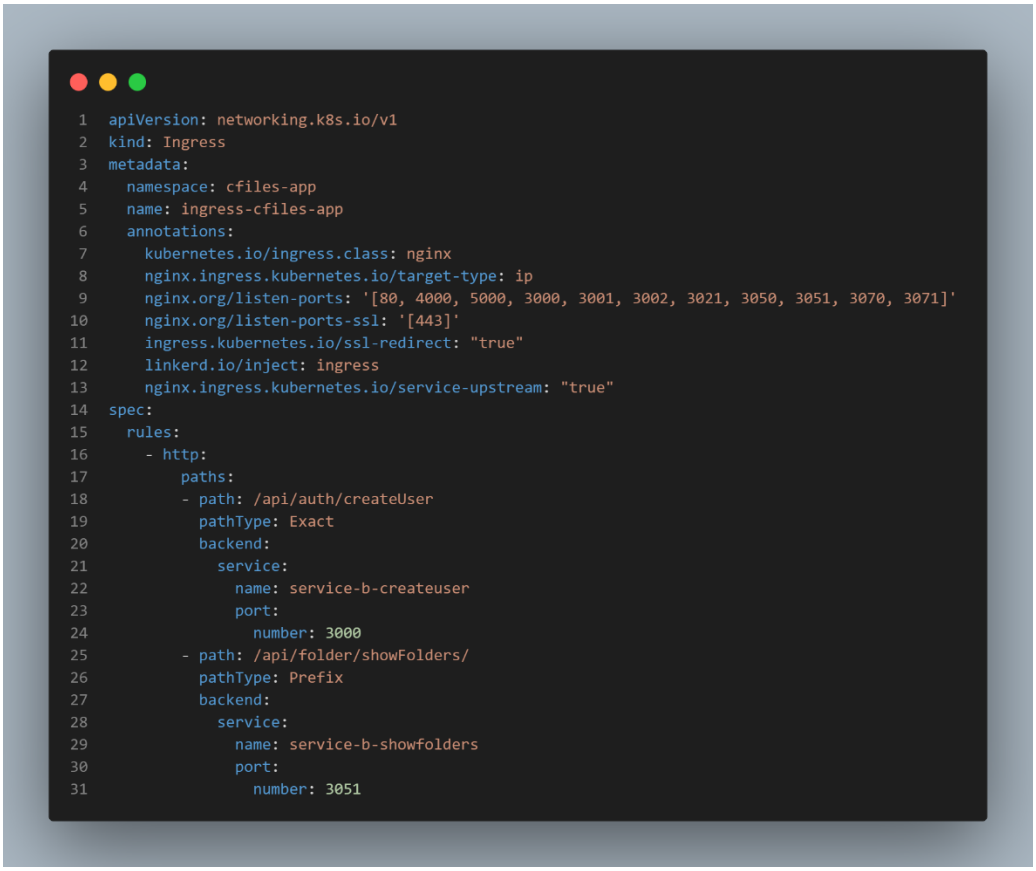
Un archivo *ingress* permite definir las rutas de la aplicación, así como enlazarlo a un servicio para que este pueda enviar el tráfico al *deployment*. Al momento de ejecutarlo, este retorna una URL que pertenece al balanceador de carga que se crea automáticamente a partir de todas las configuraciones indicadas. En el presente trabajo de investigación, se ha dividido en dos; una definición para *frontend* y otra para *backend*.

Figura 67. **Configuración de *ingress* para *frontend***

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: cfiles-app
  name: ingress-frontend
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/target-type: ip
    nginx.org/listen-ports: '[80]'
    nginx.org/listen-ports-ssl: '[443]'
    ingress.kubernetes.io/ssl-redirect: "true"
    linkerd.io/inject: ingress
    nginx.ingress.kubernetes.io/service-upstream: "true"
spec:
  rules:
    - http:
        paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: service-cfiles-app
              port:
                number: 80
```

Fuente: elaboración propia, realizado con Visual Studio Code.

Figura 68. **Configuración de *ingress* para *backend***



```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: cfiles-app
  name: ingress-cfiles-app
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/target-type: ip
    nginx.org/listen-ports: '[80, 4000, 5000, 3000, 3001, 3002, 3021, 3050, 3051, 3070, 3071]'
    nginx.org/listen-ports-ssl: '[443]'
    ingress.kubernetes.io/ssl-redirect: "true"
    linkerd.io/inject: ingress
    nginx.ingress.kubernetes.io/service-upstream: "true"
spec:
  rules:
    - http:
        paths:
        - path: /api/auth/createUser
          pathType: Exact
          backend:
            service:
              name: service-b-createuser
              port:
                number: 3000
        - path: /api/folder/showFolders/
          pathType: Prefix
          backend:
            service:
              name: service-b-showfolders
              port:
                number: 3051
```

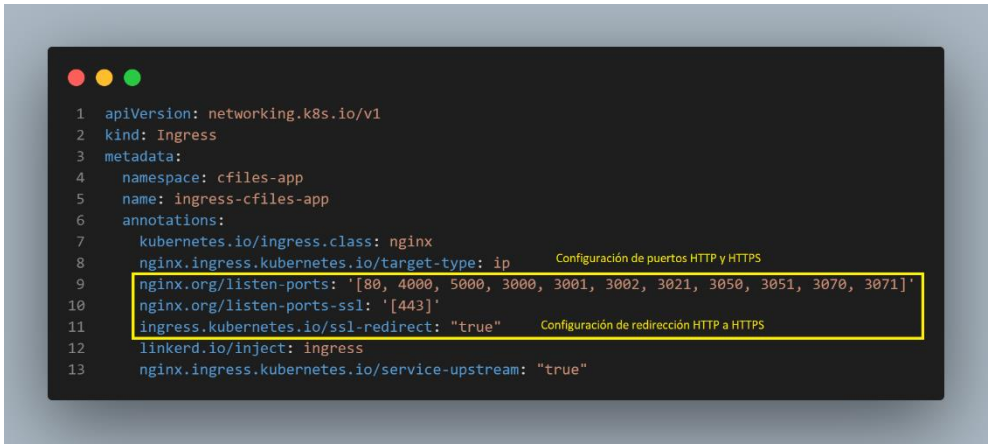
Fuente: elaboración propia, realizado con Visual Studio Code.

Nota: para la creación del balanceador de carga, se está utilizando el controlador de Nginx (disponible para descargar en el siguiente enlace https://gist.github.com/erclem1998/df974cf64060418c2b55235113342954), el cual debe ser previamente instalado en el clúster de EKS. Tras la descarga, se debe ejecutar *kubectl apply -f nginx-ingress-controller.yaml* para poder instalarlo.

5.6.5. Configuración HTTP y HTTPS

Estas configuraciones son relativamente sencillas, ya que únicamente se debe indicar el puerto y el protocolo de comunicación (HTTPS o HTTPS). Su definición se realiza en el archivo del *ingress*, en la sección de anotaciones.

Figura 69. **Configuración de HTTP y HTTPS**

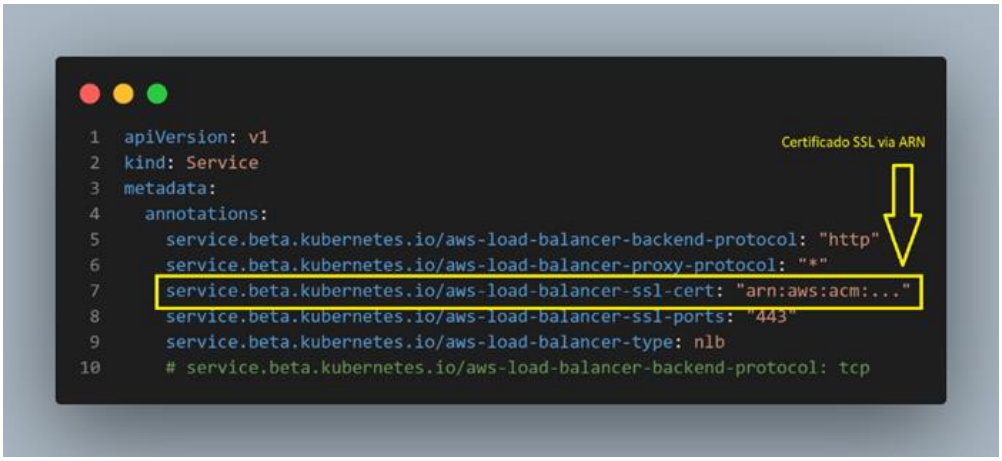


Fuente: elaboración propia, realizado con Visual Studio Code.

5.6.6. Implementación de certificado SSL/TLS

Este certificado permite que el tráfico de datos viaje de manera segura en internet. La configuración del certificado SSL, emitido en ACM, se realiza en la línea 357 del archivo mencionado en la 5.6.4.

Figura 70. **Configuración de certificado SSL/TLS**



Fuente: elaboración propia, realizado con Visual Studio Code.

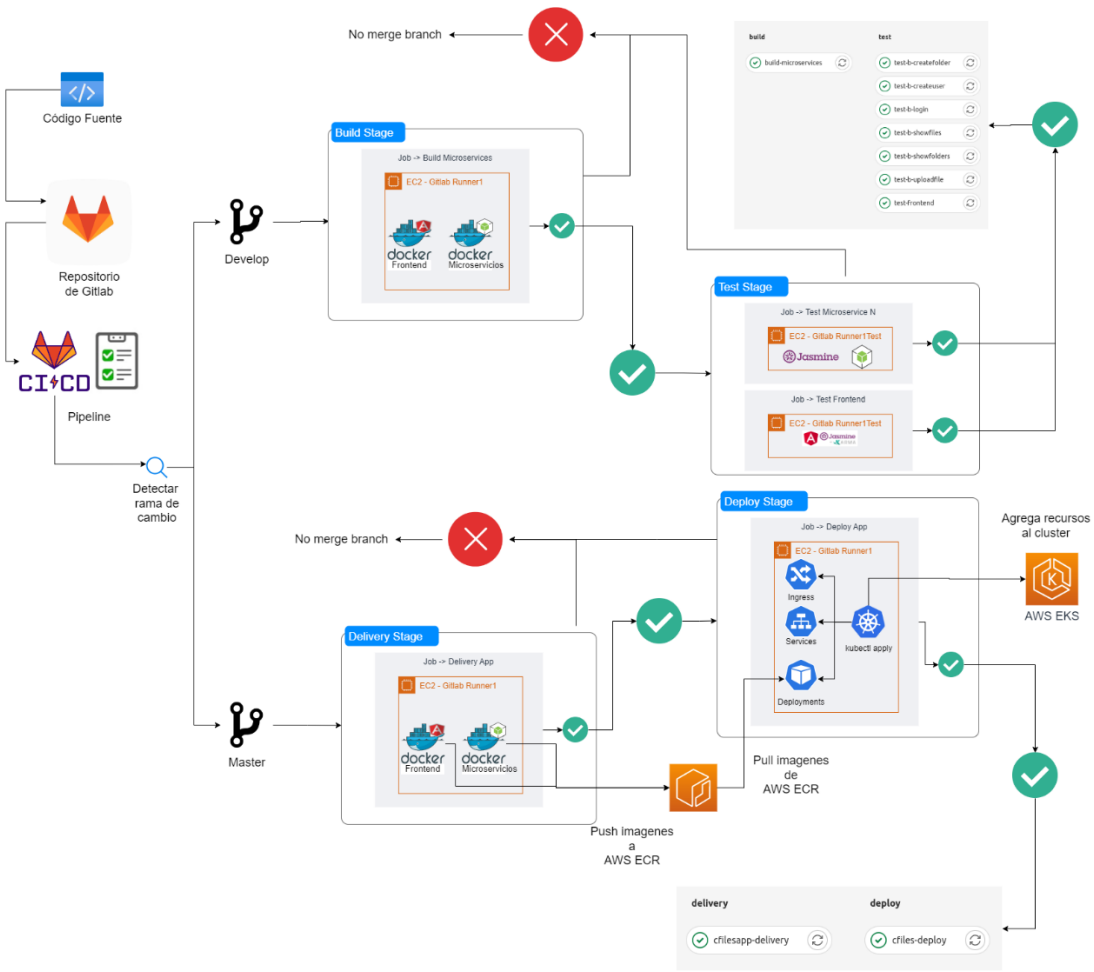
5.7. Adopción de DevOps con Gitlab-Ci

Con todas las herramientas y tecnologías configuradas, es momento de configurar la automatización de los procesos en el *pipeline* que se ejecutará en Gitlab-Ci. Para alcanzar la configuración deseada, es importante realizar las siguientes subsecciones.

5.7.1. Diagrama de flujo del *pipeline*

A continuación, se presenta un diagrama que representa el flujo para ejecutar las acciones dentro de un *pipeline* de Gitlab-Ci.

Figura 71. **Diagrama de flujo de *pipeline***



Fuente: elaboración propia, realizado con draw.io.

El diagrama mostrado en la Figura 71, demuestra las siguientes acciones:

- El código fuente se envía al repositorio de control de versiones, que se encuentra en Gitlab.
- Gitlab detecta un archivo *pipeline*.gitlab-ci.yml.

- El *pipeline* detecta la rama en la que se está realizando el cambio, considerando dos caminos posibles:
    - Se detecta cambio en la rama *Develop*.
        - Se inicia la ejecución del *stage* de *Build*.
            - ✓ Crea los microservicios utilizando Docker.
            - ✓ Si finaliza correctamente continúa al siguiente *stage*, de lo contrario finalizará con un error.
        - Se inicia la ejecución del *stage* de *Test*.
            - ✓ Instala las dependencias necesarias para el *backend* y *frontend*.
            - ✓ Ejecuta los *test* automatizados.
            - ✓ Si finaliza correctamente, finaliza el proceso para *develop*, de lo contrario finaliza con error,
        - Si los procesos finalizan correctamente, se puede pasar el código estable a una rama para crear una nueva versión del *software* estable.
    - Se detecta cambio en la rama *Master*. Para estar en este punto, el código debió pasar por la rama *develop* para verificación automatizada del código.
        - Se inicia *stage* de *Delivery*.
            - ✓ Se nombran las imágenes de Docker conforme a la versión de lanzamiento.
            - ✓ Las imágenes se envían al repositorio de control de versiones de ECR.
            - ✓ Si finaliza correctamente, el proceso continúa con el siguiente *stage*, de lo contrario finaliza con error.
        - Inicia *stage* de *Deploy*.
            - ✓ Se ejecutan los manifiestos de *deployments*.
            - ✓ Se descargan las imágenes contenidas en ECR.

- ✓ Se ejecutan los manifiestos de *services*, para generar un nuevo servicio.
- ✓ Se ejecutan los archivos *ingress* para mapear y actualizar los microservicios.
- ✓ Si finaliza correctamente, se obtiene como resultado la aplicación montada en un entorno de producción, de lo contrario finaliza con error.

Nota: los procesos descritos anteriormente pueden variar, dependiendo la cantidad de *jobs* y *stages* utilizados en el *pipeline*.

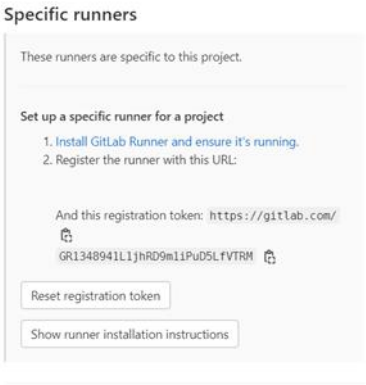
5.7.2. Configuración de *runners*

Para poder utilizar Gitlab-Ci, se deben implementar los *runner*s dentro de las instancias EC2, quienes se encargarán de ejecutar los *jobs* y cada *stage* descrito en el *pipeline*. Antes de realizar los pasos descritos, debe existir una instancia EC2 (ver 5.2.1.1).

- Abrir la línea de comandos de la distribución Linux.
- Conectarse vía SSH a nuestra instancia EC2.
- Instalar Gitlab-*Runner*.
  - *curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh* | *sudo bash.*
  - *sudo apt-get install gitlab-runner*.
- Dirigirse a repositorio del proyecto en Gitlab.
  - Seleccionar Ajustes y a continuación CI/CD.
  - Desactivar la opción Habilitar corredores compartidos para este proyecto.
  - Expandir la opción *Runners*.

- Copiar el *token* de registro.

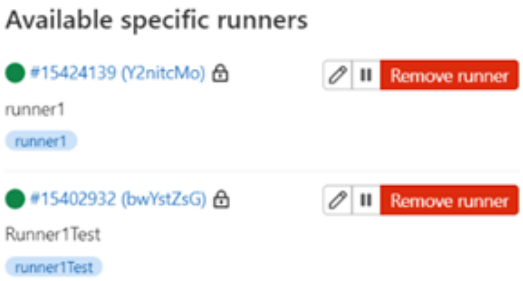
Figura 72. ***Token* de registro de *runners***



Fuente: elaboración propia, realizado con Visual Studio Code.

- Registrar *runner*.
  - *sudo gitlab-runner register*.
  - Ingresar URL de instancia: https://gitlab.com
  - Ingresar el *token* obtenido previamente.
  - Ingresar una descripción del *runner*.
  - Agregar un tag para el *runner* (se utilizará dentro del *pipeline*), para este caso *runner1*.
  - Agregar una nota para el *runner*, puede ser el mismo *tag*.
  - Seleccionar *Shell* para ejecutar comandos.
- Verificar registro del *runner*.
  - Dirigirse a repositorio del proyecto en Gitlab.
  - Seleccionar Ajustes y luego CI/CD.
  - En la sección Corredores específicos disponibles, verificar que se ha registrado correctamente el *runner*.

Figura 73. ***Runners* registrados en Gitlab-Ci**



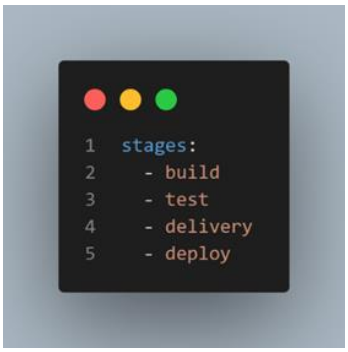
Fuente: elaboración propia, realizado con Visual Studio Code.

5.7.3. Estructura del *pipeline*

Hasta este punto, ya se tiene todo preparado para construir el *pipeline* automatizado, por lo que es importante resaltar que debe estar configurado el o los *runners* a utilizar para automatizar las tareas.

En el presente trabajo, se utilizarán 4 *stages* o fases personalizadas que se utilizarán dentro del *pipeline* junto a sus respectivos *jobs*. Para esto, es importante definir los *stages* en el encabezado del *pipeline* (archivo .gitlab-ci.yaml). Luego de definirlo, se procederá a definir las acciones (*jobs*) a realizar por cada fase (*stage*).

Figura 74. **Definición de *stages* dentro del *pipeline***



Fuente: elaboración propia, realizado con Visual Studio Code.

Figura 75. **Plantilla de definición de *job*s por *stage***

Fuente: elaboración propia, realizado con Visual Studio Code.

5.7.3.1. *Stage* de *Build*

Este *stage* se encargará de crear los contenedores de Docker de la aplicación, así como levantarla en un ambiente de pruebas dentro de la EC2, que puede ser accedido por medio de la IP pública de la misma. Este se ejecutará dentro del entorno de desarrollo (dev_env) para que utilice las variables de entorno definidas para este en la sección de Variables de la opción Ajustes. Se ejecutará cada vez que existan cambios en la rama *Develop*.

Figura 76. ***Job* para *stage* de *build***

```
build-microservices:
  stage: build
  environment:
    name: dev_env
  before_script:
    - cp /home/ubuntu/.env-vars/getcountries/.env -R ./backend/country/getCountries
  script:
    - echo "Generando contenedores..."
    - docker-compose -f docker-compose-build.yml up --build -d
    - echo "Contenedores generados."
  tags:
    - runner1
  only:
    refs:
      - develop
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.7.3.2. *Stage* de *Test*

Tiene por objetivo ejecutar los *tests* automatizados tanto para *backend* como para *frontend*. En el caso actual, se debe realizar un *job* para cada

microservicio enlazado al *stage* de *test*. Se ejecutará cada vez que existan cambios en la rama *Develop*.

Figura 77. ***Job* para *stage* de *test* de un microservicio**

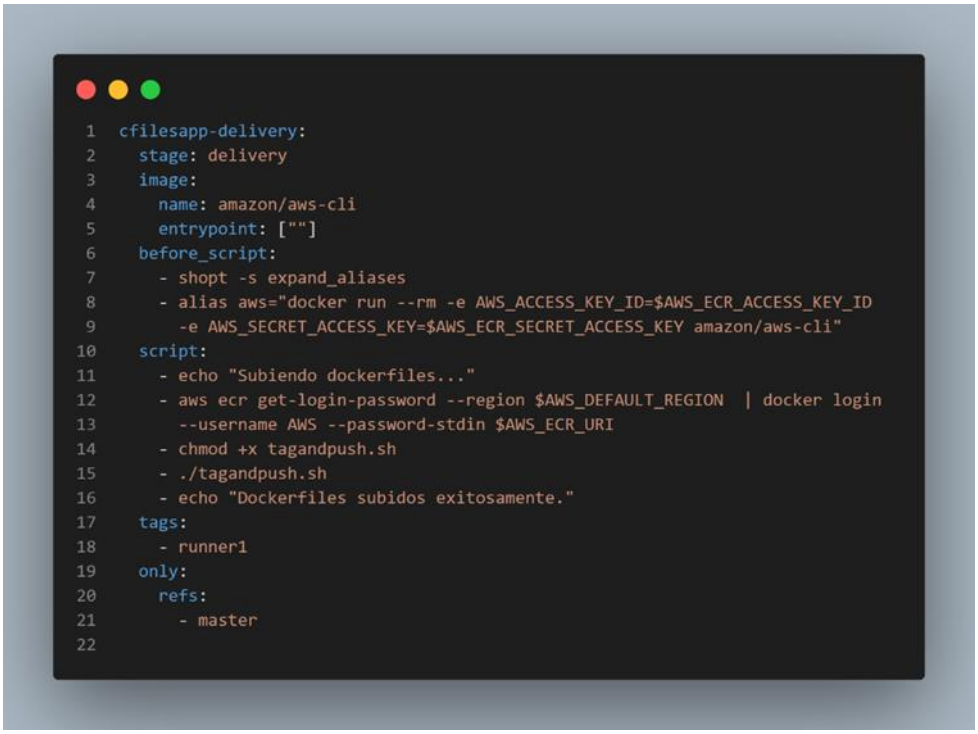
```
test-b-createuser:
  stage: test
  script:
    - echo "Ejecutando tests..."
    - cd ./backend/users/createUser
    - npm install
    - npm test
    - echo "Tests completados con éxito"
  tags:
    - runner1Test
  only:
    refs:
      - develop
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.7.3.3. *Stage* de *Delivery*

El objetivo de este *stage* es entregar el *software* continuamente, siempre y cuando los *stages* anteriores se ejecuten correctamente. Su función es subir las imágenes de Docker al repositorio de AWS ECR con nombres específicos. Estas imágenes serán consumidas por los *deployments* para poder desplegar la aplicación a producción en el último *stage*. Se ejecutará únicamente en la rama *master*.

Figura 78. ***Job* para *stage* de *delivery***



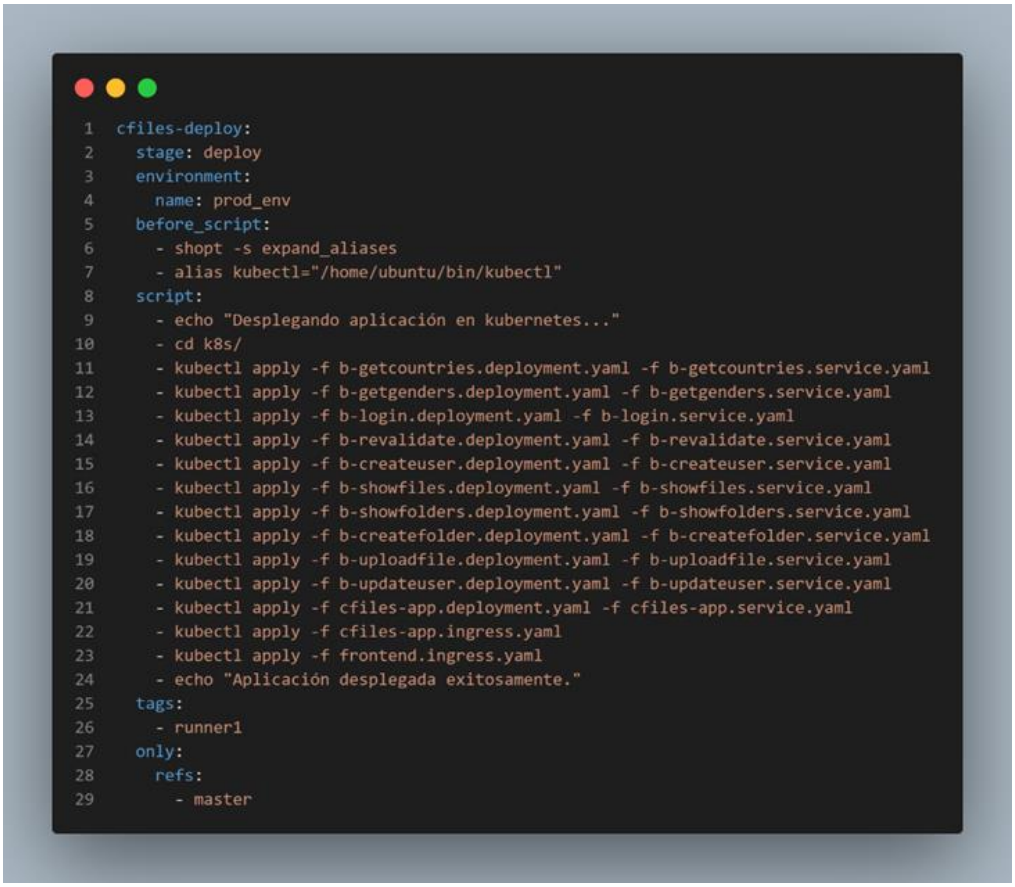
```
cfilesapp-delivery:
  stage: delivery
  image:
    name: amazon/aws-cli
    entrypoint: [""]
  before_script:
    - shopt -s expand_aliases
    - alias aws="docker run --rm -e AWS_ACCESS_KEY_ID=$AWS_ECR_ACCESS_KEY_ID
      -e AWS_SECRET_ACCESS_KEY=$AWS_ECR_SECRET_ACCESS_KEY amazon/aws-cli"
  script:
    - echo "Subiendo dockerfiles..."
    - aws ecr get-login-password --region $AWS_DEFAULT_REGION  | docker login
      --username AWS --password-stdin $AWS_ECR_URI
    - chmod +x tagandpush.sh
    - ./tagandpush.sh
    - echo "Dockerfiles subidos exitosamente."
  tags:
    - runner1
  only:
    refs:
      - master

```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.7.3.4. *Stage* de *Deploy*

Es el ultimo *stage* a configurar, y que tiene como objetivo desplegar la aplicación en el clúster de kubernetes en producción. Sus instrucciones se ejecutarán únicamente para la rama *master*, justo después de realizar el *stage* anterior.

Figura 79. ***Job* para *stage* de *deploy***



```
cfiles-deploy:
  stage: deploy
  environment:
    name: prod_env
  before_script:
    - shopt -s expand_aliases
    - alias kubectl="/home/ubuntu/bin/kubectl"
  script:
    - echo "Desplegando aplicación en kubernetes..."
    - cd k8s/
    - kubectl apply -f b-getcountries.deployment.yaml -f b-getcountries.service.yaml
    - kubectl apply -f b-getgenders.deployment.yaml -f b-getgenders.service.yaml
    - kubectl apply -f b-login.deployment.yaml -f b-login.service.yaml
    - kubectl apply -f b-revalidate.deployment.yaml -f b-revalidate.service.yaml
    - kubectl apply -f b-createuser.deployment.yaml -f b-createuser.service.yaml
    - kubectl apply -f b-showfiles.deployment.yaml -f b-showfiles.service.yaml
    - kubectl apply -f b-showfolders.deployment.yaml -f b-showfolders.service.yaml
    - kubectl apply -f b-createfolder.deployment.yaml -f b-createfolder.service.yaml
    - kubectl apply -f b-uploadfile.deployment.yaml -f b-uploadfile.service.yaml
    - kubectl apply -f b-updateuser.deployment.yaml -f b-updateuser.service.yaml
    - kubectl apply -f cfiles-app.deployment.yaml -f cfiles-app.service.yaml
    - kubectl apply -f cfiles-app.ingress.yaml
    - kubectl apply -f frontend.ingress.yaml
    - echo "Aplicación desplegada exitosamente."
  tags:
    - runner1
  only:
    refs:
      - master
```

Fuente: elaboración propia, realizado con Visual Studio Code.

5.7.4. Uso de *runners* y ramas en un *pipeline*

A continuación, se definen los *runners* y ramas a referenciar para cada *stage*.

Tabla XII. ***Runners* por cada rama**

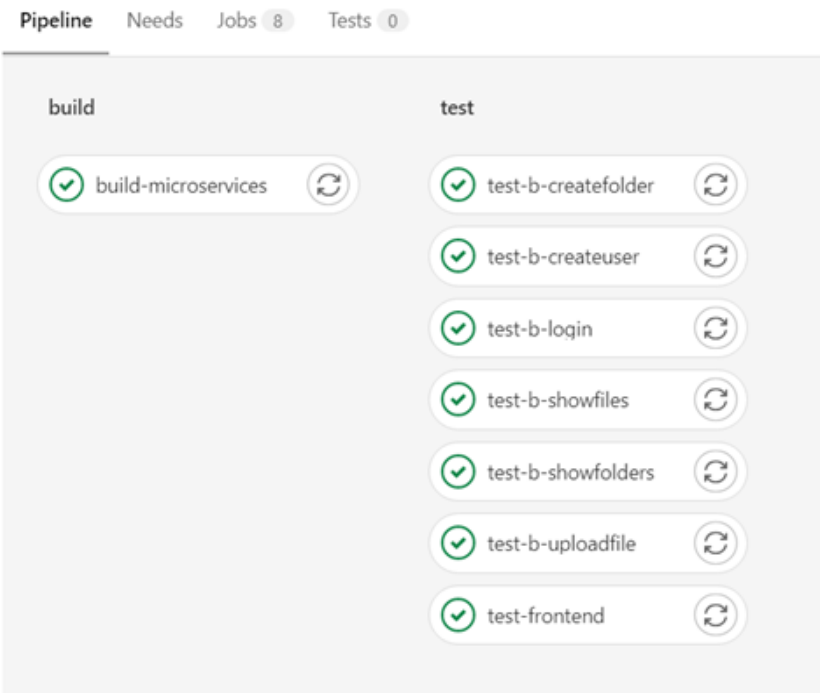
Stage	Runner	Rama
Build	*Runner1*	*Develop*
Test	*Runner1Test*	*Develop*
Delivery	*Runner1*	*Master*
Deploy	*Runner1*	*Master*

Fuente: elaboración propia, realizado con Microsoft Word 2016.

5.7.5. Resultados de ejecución de un ciclo DevOps

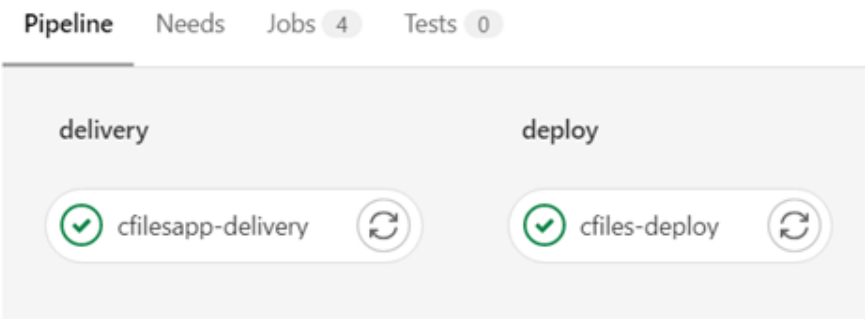
Si la ejecución de los *stages* definidos en el *pipeline* son ejecutados correctamente, se obtendrán resultados similares a las siguientes figuras:

Figura 80. **Ejecución de *stage* de *build* y *test***



Fuente: elaboración propia, realizado con Visual Studio Code.

Figura 81. **Ejecución de *stage* de *delivery* y *deploy***



Fuente: elaboración propia, realizado con Visual Studio Code.

Nota: puede visualizar el *pipeline* accediendo al siguiente enlace https://gist.github.com/erclem1998/8dbeaa46e2b831111ba121280fc39701.

5.8. Implementación de herramientas de monitoreo

El uso de estas herramientas es muy importante dentro del ciclo DevOps, ya que permite verificar el funcionamiento en tiempo real. Al finalizar esta sección, se podrán explorar todas las opciones ofrecidas por las herramientas descritas a continuación.

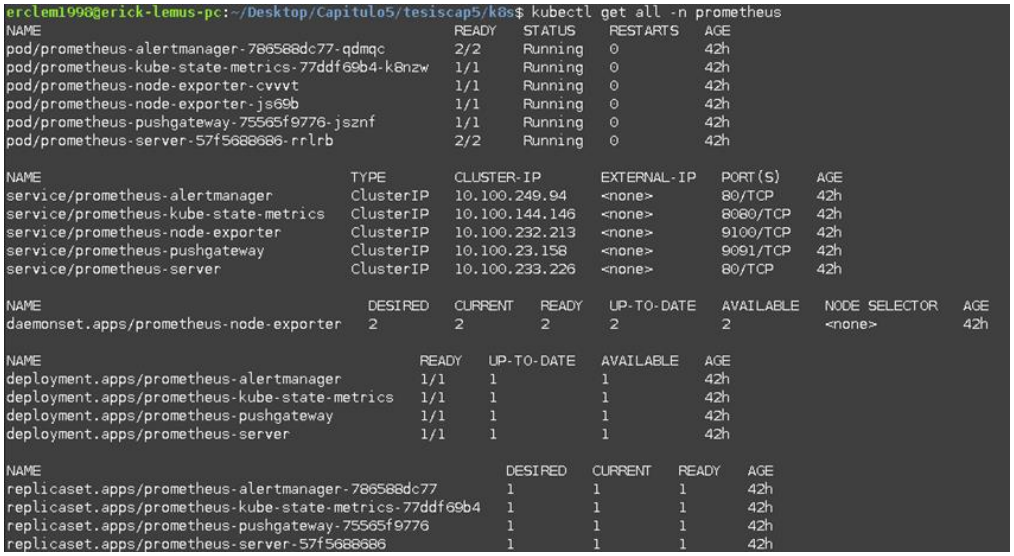
5.8.1. Métricas con Prometheus

Esta es una herramienta que proporciona diferentes tipos de métricas del clúster de Kubernetes alojado en AWS EKS. Para su instalación se deben seguir las siguientes instrucciones:

- Crear un *namespace* para los componentes de Prometheus.
  - kubectl create namespace prometheus.
- Instalar Helm, un paquete de herramientas para Kubernetes.
  - *curl -sSL https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash.*
  - *helm version –short.*
  - *helm repo add stable https://charts.helm.sh/stable.*
  - *helm search repo stable.*
  - *helm completion bash >> ~/.bash_completion*
    . /etc/profile.d/bash_completion.sh
    . ~/.bash_completion
    source <(helm completion bash)

- Instalar Prometheus.
  - *helm repo add prometheus-community https://prometheus-community.github.io/helm-charts.*
  - *helm install prometheus prometheus-community/prometheus *
    *--namespace prometheus *
    *--set alertmanager.persistentVolume.storageClass="gp2" *
    --set server.persistentVolume.storageClass="gp2"
- Verificar que los componentes de Prometheus se han desplegado.
  - *kubectl get all -n prometheus*.

Figura 82. **Componentes de Prometheus desplegados**

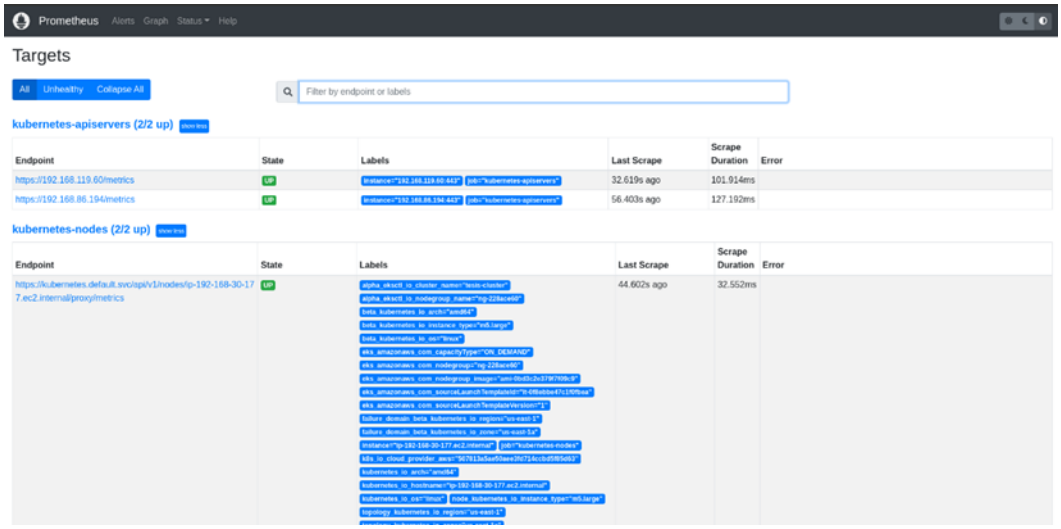


Fuente: elaboración propia, realizado con Linux *Command Line*

- Acceder al servidor de Prometheus.
  - *kubectl port-forward -n prometheus deploy/prometheus-server 8080:9090*.

- Dirigirse a http://localhost:8080/targets, en el navegador, para visualizar las métricas disponibles para el clúster.

Figura 83. **Métricas en servidor de Prometheus**



Fuente: elaboración propia, realizado con Microsoft Edge.

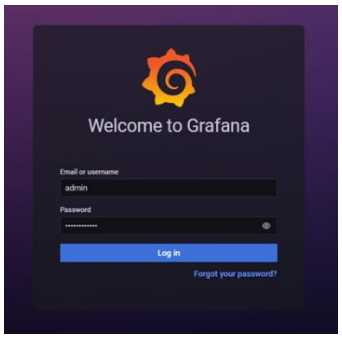
5.8.2. *Dashboards* para Prometheus con Grafana

Grafana es una herramienta que permite visualizar de manera gráfica las métricas de Prometheus por medio de *dashboards* que se pueden obtener del sitio de grafana (https://grafana.com/grafana/dashboards/). Para su instalación, se deben realizar los siguientes pasos:

- Crear un *namespace* para los componentes de Grafana.
  - *kubectl create namespace grafana.*
- Crear un archivo llamado grafana.yaml, que contendrá la configuración de Prometheus.

- Instalar Grafana.
  - *helm install grafana grafana/grafana *
    *--namespace grafana *
    *--set persistence.storageClassName="gp2" *
    *--set persistence.enabled=true *
    *--set adminPassword='ContraseniaAdmin' *
    *--values grafana.yaml *
    --set service.type=LoadBalancer.
- Verificar que los componentes de Grafana han sido desplegados.
  - *kubectl get all -n grafana.*
- Obtener URL de acceso a Grafana.
  - *export ELB=$(kubectl get svc -n grafana grafana -o jsonpath='{.status.loadBalancer.ingress[0].hostname}').*
  - *echo "http://$ELB".*
- Ir al URL obtenido en el paso anterior
- Iniciar sesión en Grafana con las siguientes credenciales.
  - Usuario: admin.
  - Contraseña: ContraseniaAdmin

Figura 84. **Inicio de sesión en grafana**



Fuente: elaboración propia, realizado con Microsoft Edge.

- Importar un *dashboard* para visualizar las métricas de Prometheus.
    - Dirigirse al panel izquierdo y seleccionar *Dashboards -> Import*.
    - En el campo *Import* vía grafana.com, agregar 3119, que es el *ID* de un *dashboard* existente, y presionar *load*.
    - En el campo prometheus, seleccionar Prometheus (*Default*), y presionar *import*.
    - Al finalizar, se podrán visualizar gráficas, y otros componentes, que muestran las métricas de Prometheus en tiempo real.

Figura 85. ***Dashboards* en grafana**

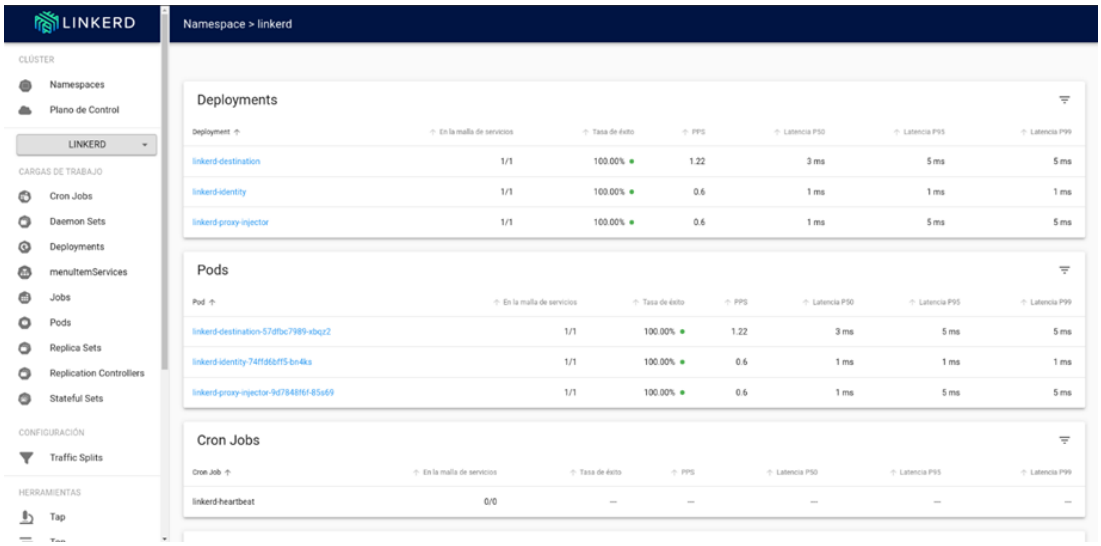
Fuente: elaboración propia, realizado con Microsoft Edge.

5.8.3. Linkerd para kubernetes

Linkerd es una herramienta que tiene diferentes objetivos, entre los cuales se encuentra el monitoreo de aplicaciones. Para instalarlo, se deben seguir las siguientes instrucciones:

- Instalar la línea de comandos de Linkerd.
  - *curl --proto '=https' --tlsv1.2 -sSfL https://run.linkerd.io/install | sh.*
  - Exportarlo como variable de entorno.
  - *linkerd version.*
- Validar el clúster de kubernetes con Linkerd.
  - *linkerd check –pre.*
- Instalar Linkerd en el clúster.
  - *linkerd install --crds | kubectl apply -f –*
  - *linkerd install | kubectl apply -f –*
  - *linkerd check.*
- Inyectar el proyecto objetivo.
  - *kubectl get -n namespaceProyecto deploy -o yaml *
    *| linkerd inject - *
    | kubectl apply -f -.
- Instalar el *dashboard* de Linkerd.
  - *linkerd viz install | kubectl apply -f -.*
  - *linkerd check.*
- Abrir *Dashboard*.
  - *linkerd viz dashboard.*

Figura 86. ***Dashboards* en Linkerd**



Fuente: elaboración propia, realizado con Microsoft Edge.

CONCLUSIONES

1. Adoptar un entorno DevOps permite agilizar los procesos de desarrollo e implementación de las aplicaciones, ya que unifica los equipos de desarrollo y operaciones.

2. El ciclo DevOps permite determinar, con mayor precisión, que procesos se pueden mejorar, ya que evalúa por cada ciclo la eficiencia de cada uno.

3. Para todo departamento de desarrollo, es necesario que tengan conocimiento sobre las tecnologías que están a la vanguardia porque proveen nuevas herramientas que son útiles para la construcción de un *software* robusto y confiable.

4. Dentro del ciclo DevOps, es de gran importancia la implementación de *test* automatizados, ya que permite evaluar la eficiencia del *software* desarrollado, además que verifica si la funcionalidad a agregar al código fuente cumple con los requerimientos de calidad.

5. Las herramientas en la nube permiten desplegar una infraestructura de *software* rápidamente permitiendo el ahorro de recursos físicos y de mantenimiento de una solución *onPremise*.

6. La implementación de una arquitectura de microservicios permite mantener el código fuente de la aplicación lo más atómico posible, ya que la aplicación se divide en segmentos pequeños que permiten manejar de

manera más eficiente los errores y futuros mantenimientos de una funcionalidad sin necesidad de dar de baja la aplicación.

7. Kubernetes permite manejar automáticamente la escalabilidad de nuestra aplicación, ya que solo se debe definir en qué condiciones puede la aplicación escalar y cuántos recursos debe agregar.

8. Implementar herramientas de monitoreo ayuda a evaluar el comportamiento de nuestra infraestructura en Kubernetes, dadas las condiciones a las que el *software* se encuentre expuesto.

RECOMENDACIONES

1. Adoptar DevOps como una metodología, provee grandes beneficios al equipo de desarrollo sin importar el tamaño de su departamento, sin embargo, se recomienda mayormente para equipos que operan con proyectos de mediana y larga escala, ya que permitirá agilizar y automatizar diferentes procesos a través de un *pipeline*.

2. Conocer acerca de Docker y Kubernetes es muy importante hoy en día, ya que son tecnologías que se encuentran en constante crecimiento, además de ser altamente utilizadas en infraestructuras de la nube.

3. Aplicar los diferentes tipos de *test* automatizados para una aplicación, tales como *test* unitarios y de integración.

4. Utilizar Gitlab que, además de ser un repositorio de control de versiones, es una tecnología que es recomendable de utilizar para adoptar un entorno DevOps, ya que permite integrar fácilmente las diferentes tecnologías con las que el equipo se encuentra trabajando.

5. Implementar *frameworks* basados en JavaScript es altamente recomendable hoy en día, dado que proveen herramientas que automatizan el proceso de manejo del DOM de una aplicación *frontend*, y la construcción de aplicaciones *backend*.

6. Implementar la nube es un enfoque al cual se está migrando de forma masiva en la actualidad, dada la facilidad de su implementación y reducción de costos operacionales.

7. Implementar balanceadores de carga, ya que la cantidad de usuarios que se manejan en la actualidad debe ser dividida de manera adecuada, para que el tráfico de datos realice la función o petición que requiere el usuario.

8. Utilizar certificados SSL/TLS en las aplicaciones web, ya que agregan una capa más de seguridad para el envío de datos de un *frontend* a un *backend*.

REFERENCIAS

1. Amazon EKS Workshop. *Monitoring using Prometheus and Grafana*. Recuperado de https://www.eksworkshop.com/intermediate/240_monitoring/.

2. Andritz, J. (2012). *Manual de operación y mantenimiento para turbina Pelton en Hidroeléctrica El Recreo*. Austria: Prentice Hall.

3. Arroyave, H. (2018). *Model del comportamiento de presas en cascada y visualización de software.* Bogotá, Colombia: Prentice Hall. Recuperado de http://www.andritz.com/no-index/pf-detail?productid=9224.

4. Chávez, I. (2013). *Reacondicionamiento de turbina Francis de hidroeléctrica Zunil, Municipalidad de Quetzaltenango*. (tesis de licenciatura). Universidad de San Carlos de Guatemala. Guatemala.

5. Equipo de AWS. *AWS Command Line Interface.* Recuperado de https://docs.aws.amazon.com/es_es/cli/latest/userguide/cli-chap-welcome.html.

6. Equipo de AWS. *Documentación de AWS.* Recuperado de https://docs.aws.amazon.com/es_es/.

7. Equipo de AWS. *EKS Documentation.* Recuperado de https://docs.aws.amazon.com/eks/latest/userguide/getting-started.html.

8. Equipo de AWS. *Connect to your Linux instance using SSH*. Recuperado de https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html.

9. Equipo de AWS. *EC2 commands*. Recuperado de https://awscli.amazonaws.com/v2/documentation/api/latest/reference/ec2/index.html#cli-aws-ec2.

10. Equipo de AWS. *Installing the AWS Load Balancer Controller add-on.* Recuperado de https://docs.aws.amazon.com/eks/latest/userguide/aws-load-balancer-controller.html.

11. Equipo de Gitlab. *Gitlab CI/CD.* Recuperado de https://docs.gitlab.com/ee/ci/.

12. Equipo de Google. *Angular Docs.* Recuperado de https://angular.io/docs.

13. Equipo de Google. *Angular CLI.* Recuperado de https://angular.io/cli.

14. Equipo de Kubernetes. (Actualizado 2022) *Kubernetes Documentation.* Recuperado de https://kubernetes.io/docs/home/.

15. Equipo de Linkerd. *Getting Started*. Recuperado de https://linkerd.io/2.12/getting-started/.

16. Equipo de Nginx. *Installation*. Recuperado de https://docs.nginx.com/nginx-ingress-controller/installation/.

17. Equipo de Red hat. (2018). *El concepto de DevOps.* Recuperado de https://www.redhat.com/es/topics/devops.

18. Hasib, S. (2020). *CI/CD using Gitlab, AWS, EKS, Kubernetes, and ECR.* Recuperado de https://faun.pub/cicd-using-gitlab-aws-eks-kubernetes-and-ecr-a55ff757b921.

19. Mehedi. S. (2021). *The Complete DevOps Engineer RoadMap 2021.* Recuperado de https://blog.devgenius.io/devops-roadmap-2021-c9d77482804f.

20. Node.js. *Documentación de Node.js.* Recuperado de https://nodejs.org/es/docs/.

21. Pathak. A. (2021). *Explora las 30 Mejores Herramientas de DevOps a Tener en Cuenta en 2022.* Recuperado de https://kinsta.com/es/blog/herramientas-devops/.

22. Rawdat, A. (2021). *Deploying Nginx Ingress Controller on Amazon EKS: How We Tested.* Recuperado de https://www.nginx.com/blog/deploying-nginx-ingress-controller-on-amazon-eks-how-we-tested/#iamserviceaccount-step.

23. Rawdat, A. (2021). *How to Simplify Kubernetes Ingress and Egress Traffic Management?* Recuperado de

https://www.nginx.com/blog/deploying-nginx-ingress-controller-on-amazon-eks-how-we-tested/#iamserviceaccount-step.

APÉNDICES

Apéndice 1. **Costos de AWS EC2 obtenidos por mes**

Recurso	Tipo	Cantidad	Costo Unidad
Instancia Linux	t3.*small*	2	14.00 USD
		Total	28.00 USD

Fuente: elaboración propia, realizado con Microsoft Word 2016.

Apéndice 2. **Costos de AWS EKS**

Recurso	Tipo	Cantidad	Costo Unidad
Clúster	-	1	73.00 USD
Nodo	m5.*large*	2 (inicial)	45.30 USD
		Total	166.30 USD

Fuente: elaboración propia, realizado con Microsoft Word 2016.

Apéndice 3. **Costos de AWS *Route* 53**

Recurso	Cantidad	Costo
Dominio	1	3.00 USD
Zona alojada	4	1.00 USD
	Total	4.00 USD

Fuente: elaboración propia, realizado con Microsoft Word 2016.

Apéndice 4. **Costos de otros recursos de AWS**

Recurso	Tipo	Cantidad	Costo Unidad
S3	-	1	0.00 USD
ACM	-	1	0.00 USD
RDS	db.t3.*micro*	1	0.00 USD
ECR	-	1	0.00 USD
Rekognition	-	1	0.00 USD
		Total	0.00 USD

Fuente: elaboración propia, realizado con Microsoft Word 2016.