UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas

Carrera de Software

CREACIÓN DE UNA APLICACIÓN BACKEND DEL MÓDULO DE TRANSFERENCIAS (CUENTAS Y MONTOS), CON MICROSERVICIOS GRAPHQL Y REST EN CONTENEDORES DOCKER, PARA FOMENTAR EL LEVANTAMIENTO DE UNA ARQUITECTURA DEVOPS

Trabajo de grado previo la obtención del título de Ingeniera en Software

Autora:

Zamia Marlene Guitarra De la Cruz

Director:

MSc. José Antonio Quiña Mera

Ibarra – Ecuador

2022

UNIVERSIDAD TÉCNICA DE NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DE CONTACTO			
CÉDULA DE IDENTIDAD:	1003699566		
APELLIDOS Y NOMBRES:	GUITARRA DE LA CRUZ ZAMIA MARLENE		
DIRECCIÓN:	COTACACHI, SAGRARIO		
EMAIL:	zmguitarrad@utn.edu.ec zamy2308guit@gmail.com		
TELÉFONO FIJO:	N/A	**TELÉFONO MÓVIL:**	0967846144
DATOS DE LA OBRA			
TÍTULO:	CREACIÓN DE UNA APLICACIÓN BACKEND DEL MÓDULO DE TRANSFERENCIAS (CUENTAS Y MONTOS), CON MICROSERVICIOS GRAPHQL Y REST EN CONTENEDORES DOCKER, PARA FOMENTAR EL LEVANTAMIENTO DE UNA ARQUITECTURA DEVOPS		
AUTOR (ES):	GUITARRA DE LA CRUZ ZAMIA MARLENE		
FECHA:	21/11/2022		
PROGRAMA:	■ PREGRADO	☐ POSTGRADO	
TÍTULO POR EL QUE OPTA:	INGENIERA EN SOFTWARE		
ASESOR / DIRECTOR:	MSC. ANTONIO QUIÑA		

2. CONSTANCIAS

El autor (es) manifiesta (n) que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto, la obra es original y que es el (son) titular (es) de los derechos patrimoniales, por lo que asume (n) la responsabilidad sobre el contenido de la misma y saldrá (n) en defensa de la Universidad en caso de reclamación por partes de terceros.

Ibarra, a los 21 días del mes de noviembre del 2022

EL AUTOR:

ESTUDIANTE
Zamia Guitarra
C.I: 1003699566

CERTIFICADO DEL DIRECTOR DE TRABAJO DE GRADO

UNIVERSIDAD TECNICA DEL NORTE

FACULTAD DE INGENIERIA EN CIENCIAS APLICADAS

CERTIFICACIÓN DEL ASESOR

Certifico que la Tesis previa a la obtención del título de Ingeniera en Software con el tema: "CREACIÓN DE UNA APLICACIÓN BACKEND DEL MÓDULO DE TRANSFERENCIAS (CUENTAS Y MONTOS), CON MICROSERVICIOS GRAPHQL Y REST EN CONTENEDORES DOCKER, PARA FOMENTAR EL LEVANTAMIENTO DE UNA ARQUITECTURA DEVOPS" ha sido desarrollada y terminada en su totalidad por la Srta. Guitarra De la Cruz Zamia Marlene, con cédula de identidad Nro. 100369956-6 bajo mi supervisión para lo cual firmo en constancia.

MSc. Antonio Quiña

DIRECTOR DE TESIS

Dedicatoria

El presente trabajo de titulación dedico con mucho cariño y amor a mi padre Rafael Guitarra, a mi madre Susana De la Cruz quienes me han apoyado siempre a lo largo de mi vida, por los consejos, enseñanzas, por todos los valores que me han inculcado para crecer como persona. También por siempre motivarme a no rendirme hasta lograr mis objetivos, por todos sus esfuerzos, sacrificios estoy infinitamente agradecida.

A mis hermanos por todo el cariño y apoyo, por siempre estar ahí cuando los necesito, por todas las locuras, risas, por los momentos felices y tristes que hemos pasado.

A todas las personas que me han brindado su apoyo y han permitido que el desarrollo de mi trabajo de grado se realice con éxito.

Agradecimientos

Agradezco a Dios por la vida, por sus bendiciones, a mis queridos padres y hermanos por todo el apoyo que me dieron para que pueda culminar mis estudios universitarios, pese a que tuve muchos tropiezos me han ayudado a seguir adelante. Son mi motivo de inspiración

A mi tutor, MSc. Antonio Quiña por la guía y por todo el apoyo brindado para la culminación de mi trabajo de tesis, de igual manera al MSc. Juan Carlos Esteves por compartir su experiencia y ayuda en el planteamiento del tema.

A mis opositores, PhD. Irving Reascos y MSc. Diego Trejo por todas sus observaciones y recomendaciones en la elaboración de mi trabajo de titulación.

A todos mis docentes de la carrera por compartirme sus conocimientos para mi formación profesional.

A los amigos por el compañerismo, la amistad brindada en el transcurso de la carrera universitaria.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Resumen

Actualmente la tecnología va evolucionando constantemente y existe la necesidad de implementar nuevos servicios en los sistemas informáticos, pero, al momento de integrar estos servicios a menudo presentan problemas en el despliegue de las aplicaciones. Para ello, se hará uso de las nuevas tecnologías a través de contenedores representada por Docker y la arquitectura de microservicios, que permitan integrar y obtener una mejor eficiencia en las aplicaciones.

El presente trabajo de investigación permitió el despliegue de la aplicación Backend dentro de contenedores Docker, a través de la construcción de las imágenes de PostgreSQL, Nginx y la configuración del archivo Dockerfile. Para validar el uso de la tecnología Docker se realizó una experimentación computacional basada en la guía Wholin, en donde se comparó la eficiencia de rendimiento de la arquitectura Docker y del Entorno Localhost con respecto a la calidad del producto del software. Para medir la eficiencia de rendimiento se basó en la métrica *"tiempo medio de respuesta"* definida en la ISO/IEC 25010 e ISO/IEC 25023. Mediante los resultados obtenidos se comprobó que el valor medio de respuesta que presenta la arquitectura Docker es menor al Entorno Localhost, y de la misma manera se verificó que GraphQL presenta una eficiencia mayor al API REST. Por lo cual, se concluyó que la arquitectura Docker es más eficiente para el despliegue de los servicios especialmente en GraphQL.

Palabras claves: contenedores, Docker, REST, GraphQL, experimentación computacional, guía Wholin, eficiencia de rendimiento, calidad del producto del software.

Abstract

Currently, technology is constantly evolving and there is a need to implement new services to computer systems, but when integrating these new services, they often present problems in the deployment. Therefore, new technologies will be applied, through containers, represented by Docker and micro - services architecture which allow integrate and obtain better efficiency in the applications.

The following Research Work allowed the deployment of Backend Application within Docker Containers, through the construction of PostgreSQL images, Nginx and Dockerfile archive configuration. To validate the use of Docker Technology, a computational experimentation was carried out based on the Wholin guide, where efficiency of the Docker Architecture performance and Localhost Environment were compared regarding software product quality. To measure the performance efficiency was based on the metric *"average response time"* defined in ISO/IEC 25010 and ISO/IEC 25023. Through the results obtained it was proved that the "average response time" that Docker Architecture presents is shorter than Localhost Environment, and in the same way, it was proved that GraphQL presents a greater efficiency compared to API REST. Consequently, it was concluded that Docker Architecture is more efficient for the deployment of the services, especially in GraphQL.

Keywords: containers, Docker, REST, GraphQL, computational experimentation, Wholin guide, performance efficiency, software product quality.

INTRODUCCIÓN

Antecedentes

Hoy en día la tecnología avanza a grandes pasos, requiriendo nuevas formas de comunicación y desarrollo de las aplicaciones. Una de las arquitecturas más utilizadas es la monolítica. Pero al momento de realizar actualización a la aplicación han existido problemas al desplegarlos nuevamente. Esto ocurre debido a que sus componentes están fuertemente acoplados y si algún componente falla existe el riesgo de que todo el sistema se detenga (De Paz, 2017).

Una de las arquitecturas que se caracteriza por su escalabilidad son los Microservicios donde representa a un conjunto de servicios que se pueden desplegar de manera independiente. Cada uno de ellos se comunican entre sí a través de las APIs como: GraphQL que es un lenguaje de consulta que permite implementar arquitecturas de software basadas en servicios y la cual es una alternativa a las aplicaciones basadas en REST. Dado que al implementar GraphQL los clientes pueden definir exactamente los datos que se requiere; en cambio REST devuelve un archivo JSON con todos los campos, aunque el cliente solo requiera uno de ello (Brito & Valente, 2020).

Por otra parte, durante los proyectos de aula los estudiantes de CSOFT-UTN, presentan inconvenientes al momento de integrar y desplegar los sistemas de trabajos grupales, ya sea por recursos del computador, versionamiento de los lenguajes, dependencia de los servicios, entre otros. Para mitigar estas inconsistencias, existe la virtualización a nivel del sistema operativo, a lo que se le conoce como Docker, que tiene como objetivo compilar y ejecutar aplicaciones sobre contendores de software, donde separa las aplicaciones de la infraestructura, y agiliza el proceso de despliegue, reduciendo de forma significativa el tiempo desde que se escribe el código de la aplicación, hasta que esta se pone en producción (Docker, 2022).

Situación Actual

Actualmente, varias empresas que se dedican al desarrollo de las aplicaciones de software, sus arquitecturas son cada vez más grandes, la lógica del código más compleja y el grado de unificación entre códigos y módulos son cada vez mayores. De igual manera al realizar cualquier modificación se requiere una recopilación y el despliegue de toda la aplicación, la cual produce un ciclo de iteración largo y desfavorable. Por lo tanto, el uso de microservicios mitiga estos inconvenientes, donde se desarrolla una sola aplicación con un

conjunto de pequeños servicios que se pueden compilar, implementar y operar de forma independiente de acuerdo con las necesitades de cada organización (Qian et al., 2020).

Las diferentes arquitecturas de software se han convertido en un punto crucial dentro la Ingeniería de Software, partiendo de las dificultades que atraviesa el enfoque monolítico tradicional, donde al existir un cambio de tecnología continuo, el software se vuelve frágil, complicado de actualizar y por consiguiente propenso a errores. Por ende, la arquitectura de microservicios como tal, se encuentra en constante evolución y crecimiento en cuanto al problema que surge en la escalabilidad (Gouigoux & Tamzalit, 2017).

Los estudiantes de la CSOFT-UTN al matricularse en la materia de Fabrica de Software tienen un limitado conocimiento de las nuevas herramientas tecnológicas como microservicios, Docker o incluso sobre el manejo de DevOps (Development - Operations), la cual proporciona prácticas que ayudan a mejorar el proceso de entrega del producto. Estas herramientas permiten a la solución de grandes inconvenientes que se producen al momento de desarrollar o desplegar un software. Docker es una solución de virtualización ligera que reduce el consumo de memoria y recursos al compartir el kernel entre el contenedor y el anfitrión. De igual manera los microservicios se adaptan bien para implementarse en contenedores, puesto que cada contenedor aloja un servicio, la cual al ejecutarse varios de ellos al mismo tiempo facilita la simulación de una arquitectura compleja de microservicios (Qian et al., 2020).

Prospectiva

La presente investigación se basa en el despliegue de la aplicación Backend dentro de contenedores Docker. A través de una experimentación computación se realizará una comparativa entre la Arquitectura Docker y el Entorno Localhost para la ejecución de los servicios APIs, con el fin de seleccionar la mejor alterativa para el desarrollo y despliegue de las aplicaciones de software. De manera que permita la disminución de las dificultades que presentan los estudiantes de la carrera CSOFT-UTN en el desarrollo de sus trabajos de aula.

Planteamiento del Problema

En el entorno del desarrollo de software, se encuentran problemas como la baja eficiencia de ejecución, una distribución desequilibrada de los recursos, fallas de reutilización de software causadas por los diferentes entornos de desarrollo. De igual forma los estudiantes de la CSOFT-UTN presentan estos inconvenientes al efectuar las integraciones de código con otros estudiantes; ya sea por incrementar o configurar nuevas librerías sin ninguna documentación, o alguna de ellas se encuentre obsoleto que puede provocar problemas en

todo el proyecto. Docker puede resolver eficazmente estos problemas, ya que puede crear el mismo ambiente mediante las configuraciones dentro de contenedores, la cual permite que el desarrollo, pruebas y despliegue ocurran en el mismo entorno (You & Sun, 2022). De igual manera el desconocimiento, la falta de información en gran parte dificulta la implementación de esta estas herramientas tecnológicas como Docker, microservicios. A continuación, en la Fig. 1 se detalla el árbol del problema del proyecto.

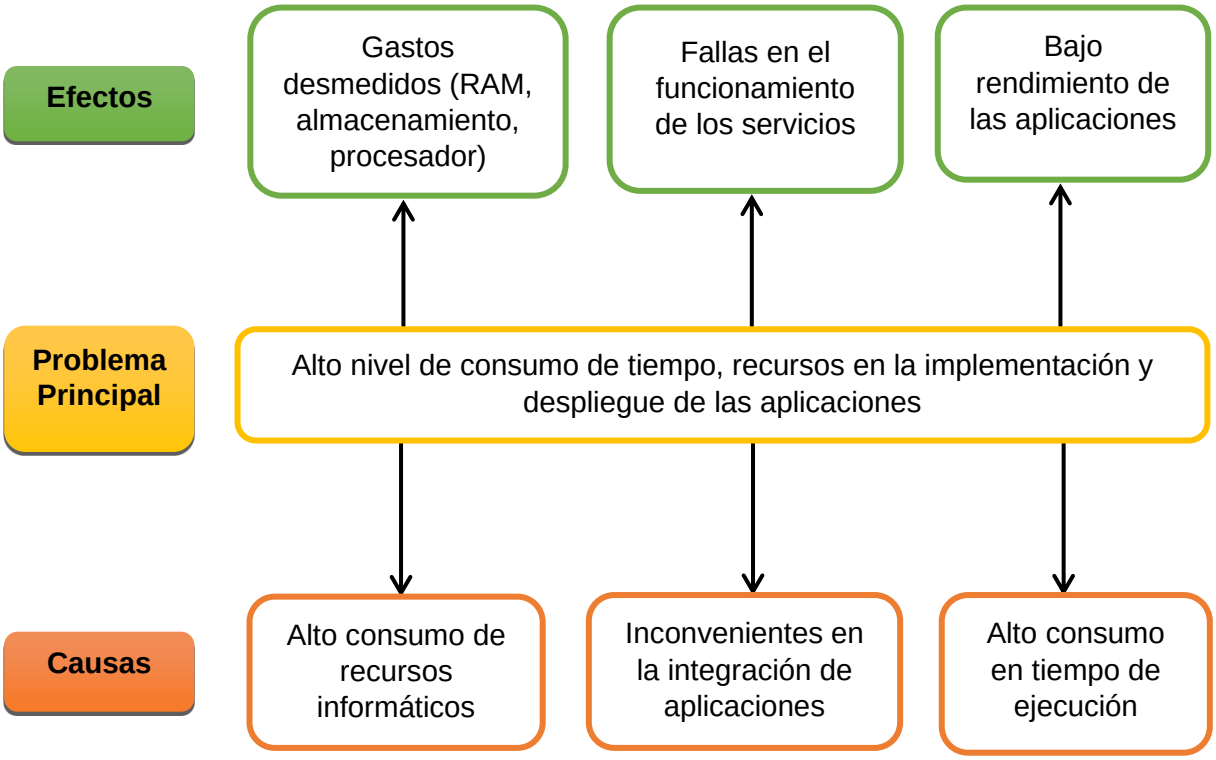


Fig. 1. Diagrama de problemas

Fuente: Propia

Objetivos

Objetivo General

Desarrollar el microservicio del módulo de transferencia (cuentas y montos) con GraphQL y Rest en contenedores Docker, para fomentar el levantamiento de una arquitectura DevOps, bajo la norma ISO/IEC 25010.

Objetivos Específicos

- Establecer el estado del arte en el uso de microservicios y Docker aplicados a un módulo de transferencias bancarias.

- Construir un ambiente virtualizado basado en Docker donde permita el despliegue de los servicios desarrollado con GraphQL y REST.

- Validar los resultados de la investigación aplicando el estándar ISO/IEC 25010 mediante la característica: Eficiencia del desempeño.

Alcance

La implementación de contenedores Docker está orientada a la materia de Fábrica de Software de la Universidad Técnica del Norte, donde el enfoque que se da es dejar sentado las bases necesarias que permita el levantamiento de una arquitectura DevOps.

La aplicación Backend se efectuará en base a una prueba de concepto de un sistema bancario del módulo transferencia bancarias. En donde se implementará cuatro casos de uso en cada arquitectura (GraphQL y REST). Para lo cual, serán desarrolladas mediante NestJS que es un framework con abstracciones sobre NodeJs, que permite crear aplicaciones eficientes y escalables (NestJS, 2022). Para almacenar los datos se realiza la conexión a la Base de Datos PostgreSQL, y por medio de los clientes Postman para REST y GraphQL Playground se verificará su correcta funcionalidad.

Con el fin de simplificar el despliegue de la aplicación se aplicará la tecnología Docker mediante la construcción de varios contenedores: el primer contenedor contendrá la base de datos con la imagen PostgreSQL obtenida del repositorio Docker Hub, el segundo contenedor se construirá mediante la configuración de un archivo Dockerfile de la aplicación y el tercer contenedor se aplicará para el servidor Nginx. Finalmente se creará un archivo Docker-compose con extensión yml, que permite la ejecución de varios contenedores. Tanto los códigos, como los archivos de configuración de Docker serán gestionados por el repositorio Bitbuckend.

Para la validación de los resultados se llevará a cabo una experimentación computacional, donde se comparará los casos de uso en las diferentes arquitecturas, y se analizará los resultados por medio de la métrica *“tiempo medio de respuesta”* de la característica: Eficiencia del desempeño establecido en la norma ISO/IEC 25010.

En la Fig. 2 se detalla la arquitectura de software propuesta.

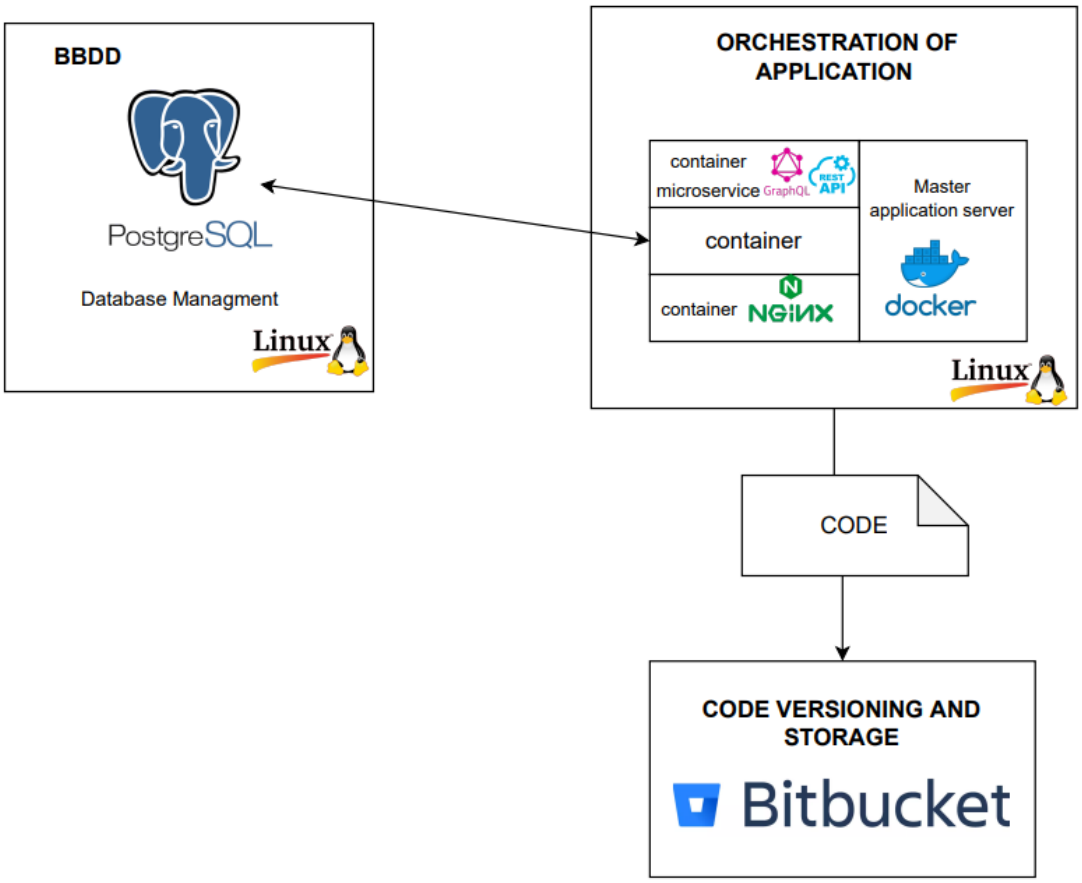


Fig. 2. Arquitectura Tecnológica propuesta

Fuente: Propia

Justificación

El presente trabajo de titulación aporta al cumplimiento de uno de los Objetivos de Desarrollo Sostenible elaborados por la ONU y la UNESCO.

Objetivo 9: Industria, innovación e infraestructura. Los avances tecnológicos son también esenciales para encontrar soluciones permanentes a los desafíos económicos (Naciones Unidas, 2022).

Los microservicios son una alternativa a los sistemas monolíticos, las cuales son aquellas en las que una única aplicación está compuesta por un conjunto de pequeños servicios, cada cual ejecutándose en su propio proceso y comunicándose por mecanismos ligeros, a menudo una API HTTP. Con el uso de contenedores Docker ayuda a la ejecución de dichos servicios de una manera rápida y sin problemas (Arcidiácono et al., 2021).

Perfil de Egreso CSOFT UTN: Aplica técnicas, lenguajes de programación y herramientas actuales tecnológicas necesarias para la práctica profesional laboral (Universidad Técnica del Norte, 2022).

Justificación Tecnológica. – Al momento de implementar un software siempre se presenta una serie de inconvenientes, ya sea durante el desarrollo o despliegue del software. Para solucionar estas fallas surgen los contenedores que empaquetan todas las dependencias necesarias para ejecutar una aplicación en cualquier entorno y con ello el uso de arquitecturas microservicios nos permite realizar servicios más escalables (Docker, 2022). Por lo cual es importante utilizar tecnologías actuales para hacer frente a la competencia actuales.

Justificación Académica: Uno de los enfoques de la materia de Fabrica de Software CSOFT-UTN, es la gestión de la arquitectura de software, por lo tanto, el uso de microservicios como la herramienta Docker permiten al desarrollo de las buenas prácticas en la Ingeniería de Software (Universidad Técnica del Norte, 2022).

CAPÍTULO 1

Marco Teórico

1.1. Arquitectura de Software

El término arquitectura de software aparece en el año 1992, y fue declarada una disciplina por Dewayne Perry y Alexander Wolf's en su artículo *"Foundations for the Study of Software Architecture"* donde en un inicio el propósito se centraba en *"resolver problemas básicos del diseño de estructuras de software estáticas",* pero hoy en día, existe sistemas globales conectados a Internet con arquitecturas en constante cambio (Woods, 2016). La arquitectura de software es considerada una de las principales disciplinas para la creación de softwares exitosos, debido a que proporciona patrones que permiten establecer la estructura, funcionamiento, confiabilidad, y el rendimiento del software (Barrios, 2018).

1.1.1. Evolución de la Arquitectura de Software

En las cinco últimas décadas, la arquitectura de software ha evolucionado de manera significativamente, debido a las nuevas tecnologías de desarrollo. Sin embargo, algunos de ellos aún se siguen utilizando en diferentes proyectos (Woods, 2016). A continuación, se detalla la evolución de las distintas arquitecturas de software al paso de cada década:

- **Año 1980:** La arquitectura predominante es la monolítica, donde la mayoría de los sistemas de software solían ejecutarse en computadoras individuales, ya sean en grandes mainframes centrales o en computadoras de un solo usuario; se puede decir que en esta época se destaca el estilo de las aplicaciones de desarrollo de escritorio (Woods, 2016).

- **Año 1990:** Se integra el concepto de sistema distribuidos como en el principal procesamiento compartido, es decir el inicio de la arquitectura orientada a servicios. Esta estructura de tres niveles (MVC) se convirtió en un estándar para los sistemas empresariales (Woods, 2016).

- **A inicio de año 2000:** Se estable el Internet como una tecnología principal, donde las organizaciones requerían que sus sistemas siempre se encuentren disponibles. Estos sistemas iniciaron con los sitios Web incorporando interfaces de usuarios públicas. Sin embargo, la incorporación de estos sistemas, a una red mundial tenía que soportar varios cambios de requerimientos no funcionales, como firewall, servidores,

balanceadores de carga de red siendo la principal preocupación de los proveedores (Woods, 2016).

- **Desde el año 2010:** El Internet se convirtió en un servicio básico para muchas personas, y esto provocó que los sistemas informáticos estén disponibles en la red para poder usarlos desde cualquier lugar y en cualquier momento. Esta arquitectura permitió la interconexión de las aplicaciones mediante la utilización de estilos arquitectónicos (microservicios), incluso los servicios pasan a formar parte del Internet a través de las APIs (Application Programming Interfaces). A partir de esta década se incorpora el protocolo de servicios web SOAP Y RESTFUL (Woods, 2016).

- **A partir del año 2020:** Se le considera a la arquitectura orientada a microservicios la más moderna y como base para el desarrollo de sistemas futuros. En esta quinta era, los sistemas inteligentes será uno de los principales avances en la actualidad. Con el surgimiento de la computación en la nube, la Inteligencia Artificial (IA), servicios con capacidades de aprendizaje automático permitirán la conexión de las "cosas" a los sistemas informáticos, mediante las plataformas PaaS y plataformas de Internet de las cosas (IoT) (Woods, 2016).

En la Fig. 3 se observa las cinco etapas de la evolución de la arquitectura de software.

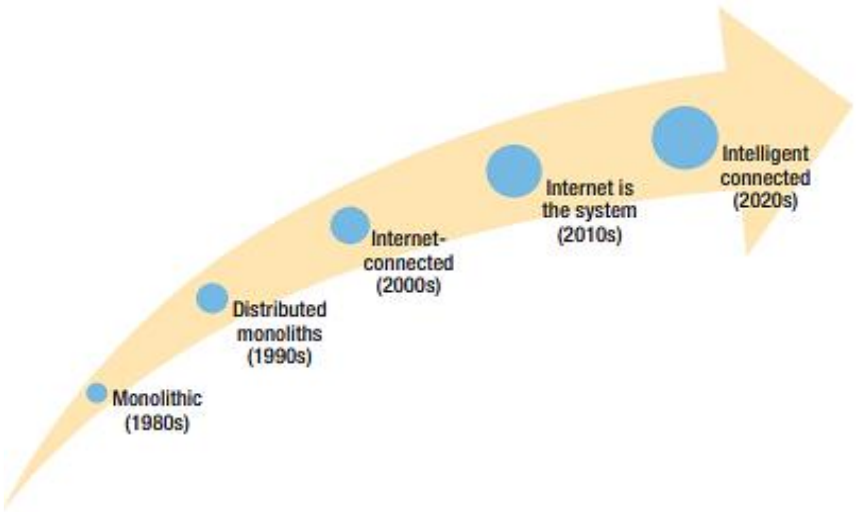


Fig. 3. Evolución de la Arquitectura de Software
Fuente: (Woods, 2016)

1.1.2. Beneficios de la Arquitectura

Según (Carballo & Barrientos, 2018), manifiestan que al no analizar cuidadosamente la calidad de la arquitectura antes de la construcción del sistema pueden presentarse problemas, y mientras más tarde se detecten estos, mayor será el impacto. Se considera que

el 50% de los proyectos atrasados y el 35% de los excedidos en costo de producción tienen problemas en la arquitectura.

El propósito principal de la arquitectura es apoyar al ciclo de vida del software, donde una buena arquitectura hace que el sistema sea fácil de entender, desarrollar, mantener e implementar. Por lo cual su objetivo final es minimizar el costo de por vida del sistema y maximizar la productividad del desarrollador (Martin, 2018). En la Fig. 4 muestra los principales beneficios que aportan en el desarrollo de sistemas informáticos a través de la calidad, tiempo y costos.

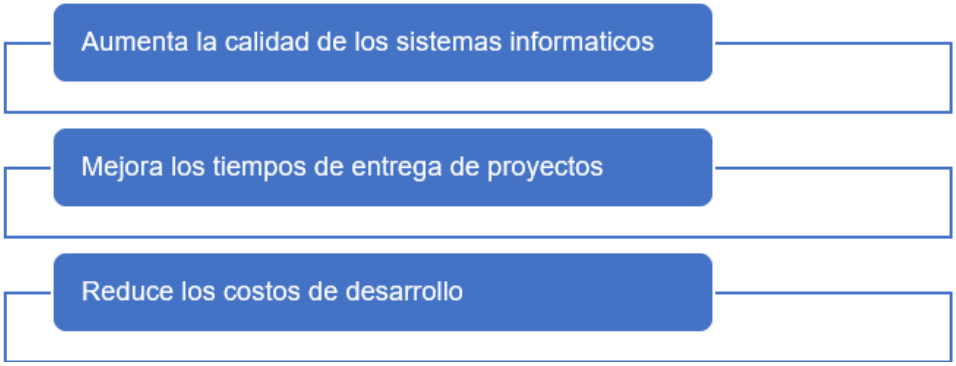


Fig. 4. Beneficios de la arquitectura

Fuente: (Cervantes et al., 2016)

1.1.3. Tipos de Arquitectura de Software

A nivel empresarial como en el sector privado y público se realiza desarrollo de software para automatizar procesos internos. Hoy en día, los avances tecnológicos han evolucionado rápidamente, de manera que las empresas deben estar en la capacidad de adaptarse a los cambios para poder tener éxito con el entorno competitivo actual. Para el desarrollo de sistemas informáticos es conveniente adoptar una arquitectura acorde a las necesidades de las aplicaciones (Barrios, 2018). A continuación, se enlista los tipos de arquitecturas.

Arquitectura Monolítico

Se considera una arquitectura monolítica debido a que todos los módulos del sistema se encuentran empaquetados en un único ejecutable. Estos tipos de aplicaciones están desarrolladas a la medida, por lo que pueden ser rápidas y eficientes; sin embargo, carecen de flexibilidad debido a su fuerte acoplamiento arquitectónico (Roldán Martínez et al., 2018).

En una arquitectura monolítica la capa de la vista, lógica de negocio y la capa de acceso a la base de datos, se encuentran juntos en un mismo sistema como se observa en la

Fig. 5. Al iniciar un producto de software este tipo de arquitectura es excelente ya que simplifican el mantenimiento y facilitan el testing, pero a medida que los requerimientos aumentan la aplicación se vuelve muy grande, siendo difícil de mantener (Guimarey, 2020).

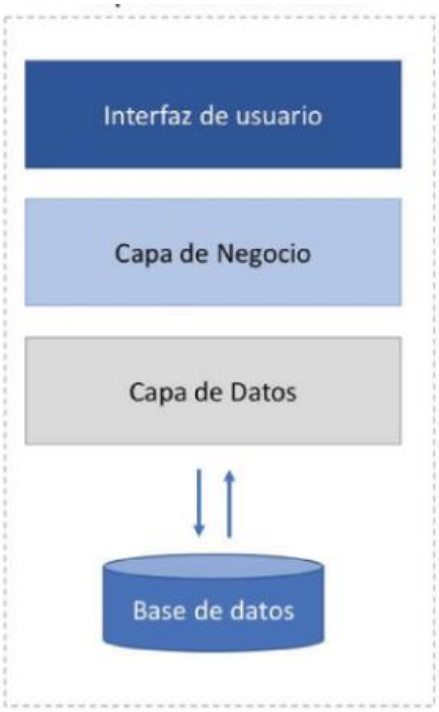


Fig. 5. Arquitectura Monolítico
Fuente: (Roldán Martínez et al., 2018)

Mediante los problemas presentados por la arquitectura monolítica nace un nuevo paradigma para resolver estos inconvenientes y el paso se da a la evolución de las aplicaciones distribuidas representadas por SOA (Service Oriented Architecture) (Roldán Martínez et al., 2018).

Arquitectura Orientada a Servicios

SOA es un estilo arquitectónico basadas en la construcción de servicios para construir soluciones empresariales, puesto que al ser independiente permite adaptarse a los cambios de manera sencilla que las demás arquitecturas. Varias empresas han optado en esta arquitectura debido a su mayor flexibilidad (Rosado & Jaimes, 2018).

Pertenece a la subclase de los sistemas distribuidos y se le considera a la arquitectura SOA como predecesor de los microservicios debido a su escalabilidad, flexibilidad y tolerancia a fallos. La Fig. 6 muestra como los microservicios forman parte de la implementación de SOA (Yarygina & Bagge, 2018).

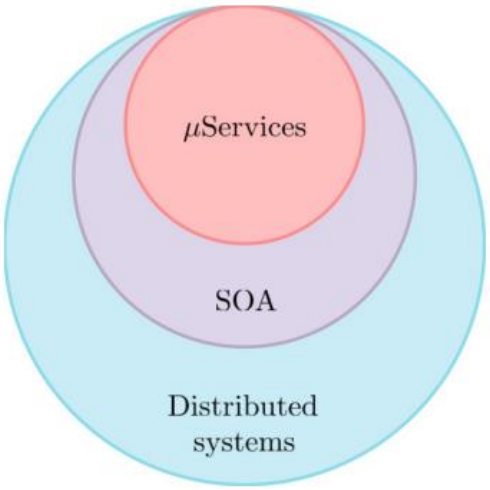


Fig. 6. Arquitectura de Microservicios en perspectiva
Fuente: (Yarygina & Bagge, 2018)

Arquitectura de Microservicios

Lewis y Fowler ven a los microservicios como un enfoque para desarrollar una sola aplicación con pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, a menudo un API de recursos HTTP. Una de las características de estos servicios es que puede estar escrito en diferentes lenguajes de programación y utilizar diferentes tecnologías de almacenamiento de datos (Yarygina & Bagge, 2018). En la siguiente sección se especifica a profundidad sobre esta arquitectura.

1.2. Arquitectura Orientada a Microservicios

El término "microservicios" fue discutido por primera vez en un taller para arquitectos de software cerca de la ciudad de Venecia en el año 2011 y a partir del año 2014 se convirtió en el término más popular entre los desarrolladores, debido a su nueva manera de estructurar las aplicaciones (Barrios, 2018).

Según Jamshidi et al (2018), menciona que la arquitectura de microservicios es la última tendencia en el diseño, desarrollo y entrega de servicios de software. Actualmente, constituye un enfoque bien establecido de modularización y agilidad, puesto que cada microservicio se convierte en una unidad independiente de desarrollo.

De acuerdo con el concepto de microservicio el escalado funcional, se basa en la separación de la aplicación en unidades independientes, de manera que se puedan escalar sin la necesidad de perjudicar al resto de componentes, ver Fig. 7. De igual forma las pruebas, el despliegue se realiza de forma separada (Roldán Martínez et al., 2018).

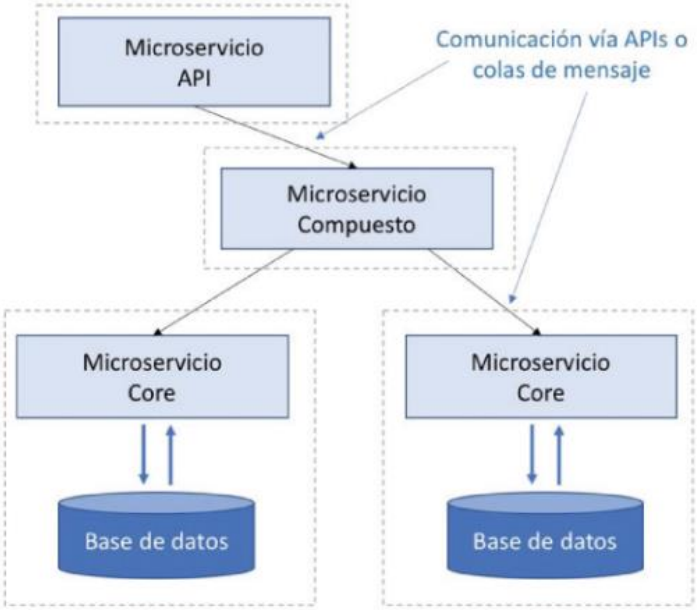


Fig. 7. Arquitectura de Microservicios
Fuente: (Roldán Martínez et al., 2018)

1.2.1. Beneficios de los microservicios

Los microservicios pueden ser testeados y desplegados individualmente, debido a que cada uno tiene su propio ciclo de vida, también se pueden combinar entres si, por la ventaja de la independencia de lenguaje y tecnología (Guimarey, 2020). A continuación, se mencionan algunos beneficios de los microservicios que aportan al desarrollo de software. Ver Fig. 8.

- **Gestión de servicios:** el código fuente de un microservicio es menor a comparación con la arquitectura monolítico, lo que permite tener una mejor comprensión y gestión de las aplicaciones.
- **Independencia:** cada microservicio contiene su propia lógica, lo que permite realizar actualizaciones sin afectar al sistema global.
- **Modularidad:** son flexibles y mejoran la escalabilidad de las aplicaciones, debido a su enfoque modular.
- **Gestión de datos:** un microservicio puede tener su base de datos propia, incluso en tecnologías distintas.
- **Tolerancia a fallos:** se encuentran débilmente acoplados, por lo que permite realizar cambios sin la necesidad de interrumpir la funcionalidad de un sistema completo.

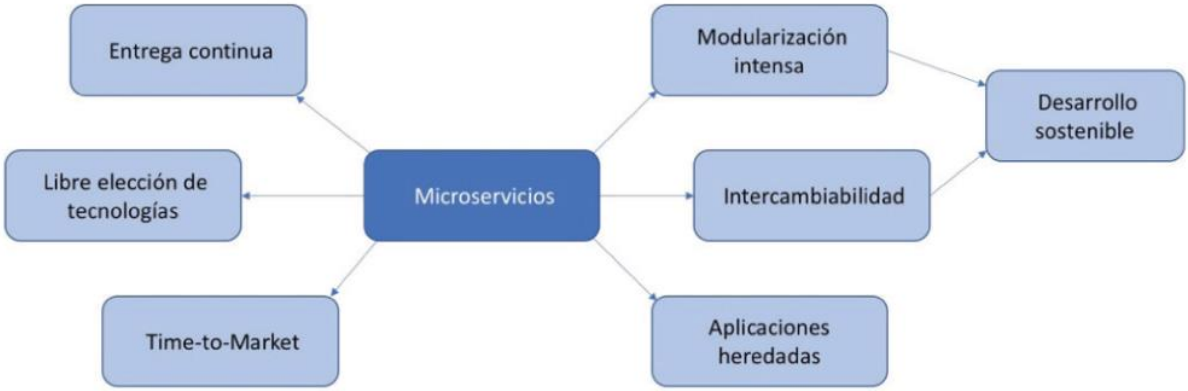


Fig. 8. Beneficios de los Microservicios
Fuente: (Roldán Martínez et al., 2018)

1.2.2. Comparación entre arquitecturas

La arquitectura de microservicios surge debido a los problemas que presentaban los sistemas monolíticos al momento de añadir nuevas funcionalidades, sin embargo, existe un alto porcentaje de aplicaciones desarrollados en este enfoque monolítico y a medida que surgen nuevos requerimientos varias organizaciones empresariales han decidido migrar sus sistemas a una arquitectura moderna con infraestructuras en la nube (Rodríguez et al., 2020). En la Tabla 1, se detalla las diferencias que existe entre estas dos arquitecturas monolíticas y microservicios.

Tabla 1: Comparación entre Arquitecturas Monolítica y Microservicio

Categoría	Arquitectura Monolítica	Arquitectura Microservicio
Codificación	Una sola base de código para toda la aplicación	Múltiples bases de código cada microservicio tiene su propio código base
Comprensibilidad	A menudo confuso y difícil de mantener	Mejor flexibilidad y fácil de mantener
Despliegue	Despliegues complejos	Implementación simple, debido a que cada microservicio se puede implementar individualmente
Lenguaje de Programación	Desarrollado en su totalidad en un tipo de lenguaje de programación	Cada microservicio puede ser desarrollado en diferentes lenguajes de programación y tecnología

Escalabilidad	Se requiere que toda la aplicación escale	Se despliega de manera independiente, por tanto, no existe la necesidad de escalar toda la aplicación

Fuente: (Rodríguez et al., 2019)

1.2.3. Ventajas y desventajas de los microservicios

Una aplicación construida en base a microservicios mejora la implementación y el despliegue de las aplicaciones. Sin embargo, existen algunas desventajas como se muestra en la Tabla 2.

Tabla 2. Cuadro de Ventajas y Desventajas

Ventajas	Desventajas
Equipo de trabajo mínimo, donde el programador tiene la libertad de desarrollar y desplegar servicios	Necesidad de varios desarrolladores para la solución de los problemas
Permite la escalabilidad, debido a su enfoque modular	Demora en fragmentar distintos microservicios
Uso de contenedores para el despliegue y el desarrollo de la aplicación rápidamente	Complejidad en gestionar un gran número de servicios
Implementar servicios en diferentes lenguajes y tecnología	Tener conocimiento en varios tipos de lenguajes para su mantenimiento

Fuente: (Rodríguez et al., 2019)

1.2.4. Comunicación entre microservicios

Al momento de construir aplicaciones, uno de los principales puntos es decidir como accederán los clientes al software. En la arquitectura monolítica los clientes se conectan a la página web y envían peticiones HTTP a la aplicación, obteniendo respuestas según la lógica de la aplicación. En cambio, en la arquitectura de microservicios se ha distribuido funcionalidades independientes, donde el cliente puede enviar peticiones a cualquier microservicio, puesto que cada uno expone un punto final al público (Roldán Martínez et al., 2018). En la siguiente sección se detalla las diferentes arquitecturas APIs que permiten la comunicación entre servicios.

1.3. Interfaz de programación de aplicaciones (API)

Las API (Application Programming Interfaces) son conjuntos de definiciones que facilitan la integración de las aplicaciones de software, es decir le permiten comunicar con un

sistema informático para obtener información, de manera que la aplicación entienda la solicitud y lo devuelva. Las aplicaciones de software desarrolladas en distintos lenguajes y tecnologías pueden comunicarse a través de los servicios Web como SOAP, REST y GraphQL que brindan interoperabilidad entre los sistemas. En entornos distribuidos los procesos se ejecutan en distintas máquinas y mediante las mensajerías asíncronas se logra la comunicación entre ellos (Brito & Valente, 2020).

1.3.1. Arquitectura REST

La arquitectura REST (Representational State Transfer) se introdujo en el año 2000 por Thomas Fielding. Según los principios las interfaces Rest se fundamentan en identificadores uniformes de recursos (URI), que proporciona un identificador de recurso único y en el protocolo de transferencia de hipertexto (HTTP) que define el tipo de operación que va a realizarse en el recurso (Neumann et al., 2018). Es un estilo arquitectónico basado en el paradigma cliente-servidor que permite definir la escalabilidad, disponibilidad y rendimiento en los sistemas distribuidos. Las APIs basadas en REST se exponen mediante endpoints y cada endpoint devuelve la información definida de acuerdo con la operación (Brito & Valente, 2020). En la Fig. 9 se define algunas características de la arquitectura REST.

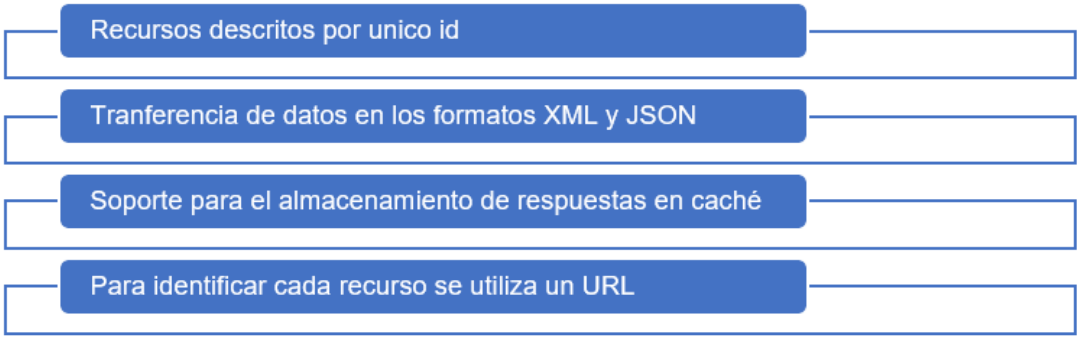


Fig. 9. Características - Arquitectura REST
Fuente:(Sayago & Flores, 2019)

Estado

La arquitectura REST se define un protocolo sin estado, debido a que no se guarda el estado de la información en el servidor, sino que toda la información se encuentra en la consulta del cliente. Este estilo arquitectónico expone sus funciones a través de URI (Universal Resource Identifier), donde se debe evitar el uso de verbos que estén ligados a una acción, puesto a que las acciones se especifican a través de comandos HTTP. Por otra parte, se debe tomar en cuenta los estados de código HTTP que se recibe frente a una petición (Roldán Martínez et al., 2018). En la Tabla 3 muestra los códigos de estado.

Tabla 3. Código de Estado HTTP

Código de estado	Resultado
1XX	Respuestas informativas
2XX	Peticiones correctas
3XX	Redirecciones
4XX	Errores del cliente
5XX	Error del servidor

Fuente: (Roldán Martínez et al., 2018)

Representaciones

Corresponde a la representación del estado del recurso, que el servidor expone a través del formato JSON, aunque la arquitectura REST es compatible con cualquier formato como: JSON, XML, entre otros (Sayago & Flores, 2019). En la Fig. 10 se puede apreciar un ejemplo del estado del recurso: api/users.


```
{
    "identificationCard": "7290799483",
    "name": "Jaren",
    "lastname": "Bode",
    "phone": "(951) 840-8157 x238",
    "email": "Whitney_Daniel80@yahoo.com",
    "password": "tQqGNhkFIgOkb8X",
    "state": false,
    "id": 1,
    "createAt": "2022-10-25T05:16:30.610Z",
    "updateAt": "2022-10-25T05:16:30.610Z"
},
```


Fig. 10. Representación del estado del recurso

Fuente: Propia

Operaciones

La Tabla 4 muestra los principales comandos HTTP que son utilizadas en la arquitectura REST, mediante las operaciones: GET, POST, PUT, DELETE.

Tabla 4. Comandos HTTP

Acción	Comando HTTP	Utilidad
READ	GET	Consultar registros
CREATE	POST	Crear nuevos registros
UPDATE	PUT	Actualizar registros
UPDATE	PATCH	Actualizar partes concretas
DELETE	DELETE	Eliminar registro

Fuente: Propia

1.3.2. Arquitectura GraphQL

La arquitectura GraphQL es un lenguaje de consulta para implementar servicios web. Fue desarrollado internamente en la empresa Facebook, como una alternativa a las aplicaciones basadas en REST. En 2015, Facebook implementó GraphQL y como resultado, el lenguaje comenzó a ganar impulso entre los desarrolladores. GraphQL pueden definir exactamente los datos que se requieren de los servicios, en cambio con REST el servidor devuelve un documento JSON, con todos los campos, aunque el cliente solo requiera uno, como se muestra en el ejemplo de la Fig. 11 (Brito & Valente, 2020).

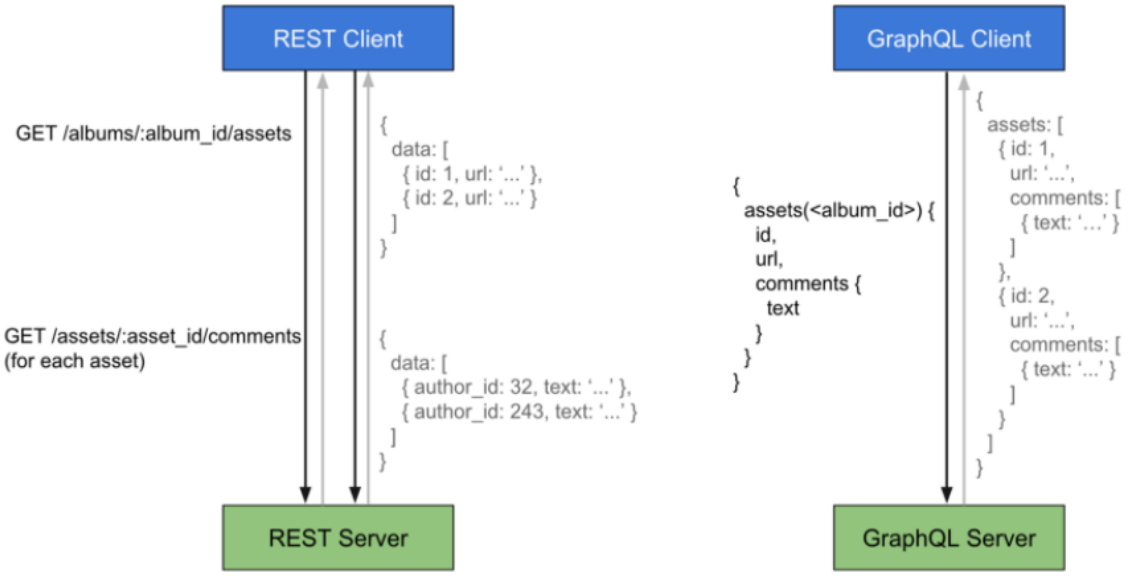


Fig. 11. REST API vs GraphQL

Fuente: https://wesecureapp.com/blog/what-is-devsecops-and-its-role-in-devops-architecture/

A diferencia de REST, la arquitectura de GraphQL tiene la ventaja de presentar sus datos dinámicamente, también proporciona una único endpoint para poder acceder a los datos del servidor. Algunos términos que maneja GraphQL son: tipos de datos, consultas y mutaciones (Mas et al., 2021).

Esquema (Schema)

En esta sección GraphQL define los esquemas (schema) donde se describe las relaciones y los tipos disponibles para realizar consultas, suscripciones y mutaciones a la base de datos. En la Fig. 12 se observa el tipo User que tiene una serie de campos y el signo de exclamación, refiere a que ese campo no debe ser nulo. El signo de corchetes alrededor de un tipo hace referencia a otro tipo de objeto. Finalmente se observa tipos predefinimos de GraphQL llamados Query, y Mutation (Eizinger, 2017).

Fig. 12. Representación de esquemas (schema)

Fuente: Propia

Resolutores (Resolver)

En este componente se realiza instrucciones para convertir una operación (query, mutation, subscription) en datos, la cual se retorna con la misma forma que se especificó en el esquema, ya sea de manera sincrónico o por medio de una promesa (NestJS, 2022).

Operaciones

a) Query

Para obtener los datos de los servicios, GraphQL define un lenguaje de consulta, la cual se inicia con el campo de tipo query (Brito et al., 2019). Como se ha mencionado anteriormente GraphQL tiene la facilidad de presentar datos dinámicos. Por ello en la Fig. 13 se muestra tres ejemplos de consulta. En la primera consulta (getAllUser), el cliente solicita solo un campo de identificationCard del objeto, la segunda consulta es la misma, pero con solicitudes de dos campos: identificationCard y name y en la tercera consulta solicita tres campos, pero como el campo de account, pertenece a otro objeto se especifica sus campos, en este caso accountNumber.

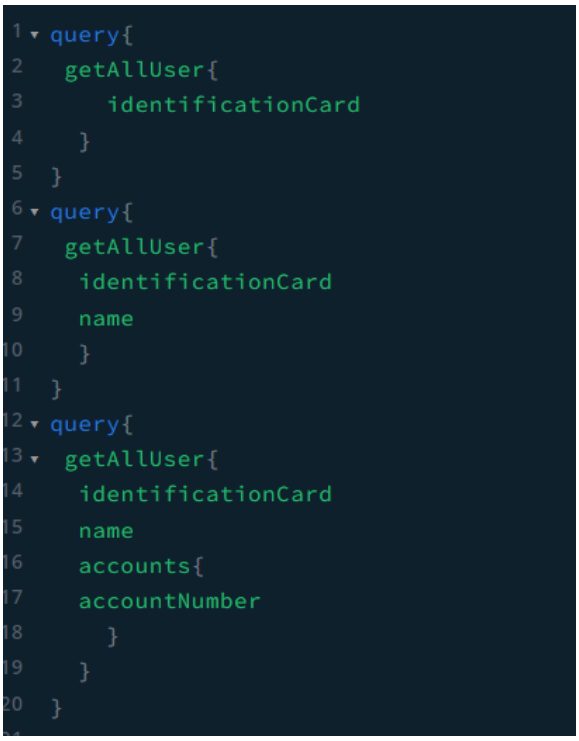


Fig. 13. Consulta de distintos datos de un objeto

Fuente: Propia

En la Fig. 14 muestra el resultado de la tercera consulta con los campos solicitados en el formato JSON.

Fig. 14.Vista de la consulta en formato JSON

Fuente: Propia

b) Mutation

Son Queries, con la única diferencia que su objetivo es insertar o modificar datos en el servidor. En la Fig. 15 muestra un ejemplo de mutación createUser, donde recibe el objeto y lo devuelve solamente para confirmar que su ejecución se realizó con éxito (Brito et al., 2019). Se debe tomar en cuenta que cada mutación debe tener su propia función, en este caso el valor de la función es de insertar un usuario.

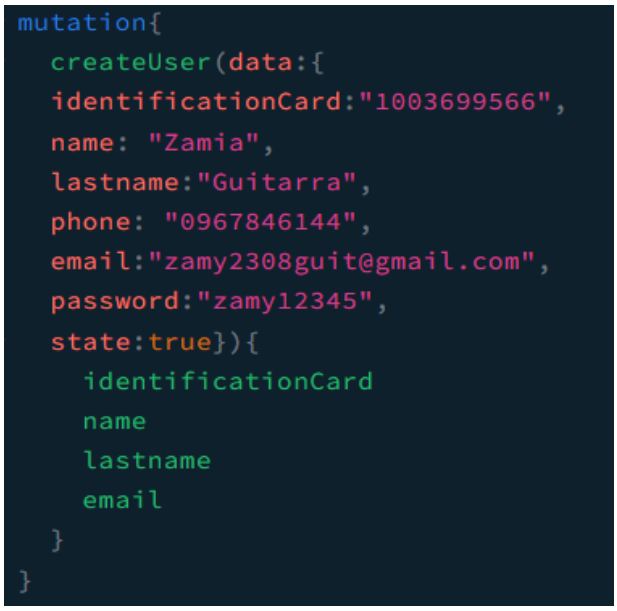


Fig. 15. Operación de mutación

Fuente: Propia

c) Subscriptions

GraphQL tiene una tercera operación llamado subscription, la cual permite que un servidor envíe datos a sus clientes cuando ocurra un evento. El caso más común de las subscripciones es notificar sobre los eventos particulares al cliente (NestJS, 2022).

1.3.3. Diferencia entre REST y GraphQL

La arquitectura REST y GraphQL son dos enfoques de diseño de API que cumplen la misma función el de transmitir datos a través de protocolos de Internet HTTP (Brito et al., 2019). No obstante, la Tabla 5 detalla algunas diferencias que existe entre estas arquitecturas.

Tabla 5. Diferencias entre Rest y GraphQL

Componente	Rest	GraphQL
Popularidad	82% de las empresas usan esta arquitectura	19 % de las empresas usan esta arquitectura
Usabilidad	Los resultados son más difíciles de predecir, ya que carece pautas sobre el control de versiones	Facilidad para enviar consultas personalizadas
Rendimiento	Almacenamiento en caché	No tienen almacenamiento en caché
Seguridad	Existen muchos métodos de autenticación para garantizar la seguridad	Proporcionan sus propios métodos de autenticación y autorización

Fuente: (Brito et al., 2019)

1.3.4. Modelo de Despliegue

Una vez que se encuentran implementados y testeado los microservicios se realiza el despliegue y la puesta en producción. Para ello, existen dos enfoques principales: las máquinas virtuales y contenedores. Los contenedores son paquetes ejecutables ligeros que incluyen todo lo necesario para ejecutarse como: código, entorno de ejecución, herramientas del sistema, librerías, configuraciones, entre otros. Por otro lado, una máquina virtual (VM, Virtual Machine) sigue una filosofía distinta, donde inicia con un sistema operativo completo y, dependiendo de la aplicación, los desarrolladores pueden o no eliminar componentes no deseados (Roldán Martínez et al., 2018). En la siguiente sección se especifica las tecnologías utilizadas para el despliegue de las aplicaciones.

1.4. Contenerización de servicios en Docker

1.4.1. Docker

Salomon Hykes inicio Docker como un proyecto interno dentro de dotCloud, empresa enfocada en el entorno Paas (Plataforma como servicio), la cual fue liberada como código abierto en el año 2013. Actualmente es uno de los proyectos con más estrellas en Github. Docker es una plataforma “open source”, que sigue una arquitectura cliente – servidor y cuya finalidad es automatizar el despliegue de las aplicaciones distribuidas basadas en contenedores. Un contenedor se encarga de empaquetar todo lo indispensable para que el sistema funcione, por lo tanto, no se debe preocuparse del vesionamiento del software o de los recursos del computador puesto que se encuentra dentro de un contenedor. Esta nueva manera de trabajar cambia el paradigma del desarrollo, de aplicaciones monolíticas a las aplicaciones de microservicios (Guijarro et al., 2019).

Según Singh et al (2019) menciona que Docker tiene la facilidad de empaquetar, compilar, probar y desplegar software en un entorno Linux aislado llamado contenedor. Debido a este aislamiento se puede ejecutar varios contenedores en una sola máquina. La Fig. 16 muestra todo el flujo de trabajo de Docker, como el Docker Client se comunica con el Docker Daemon, la cual extrae las imágenes del Docker Registry en base de un archivo Dockerfile, que posterior será desplegada en un contenedor.

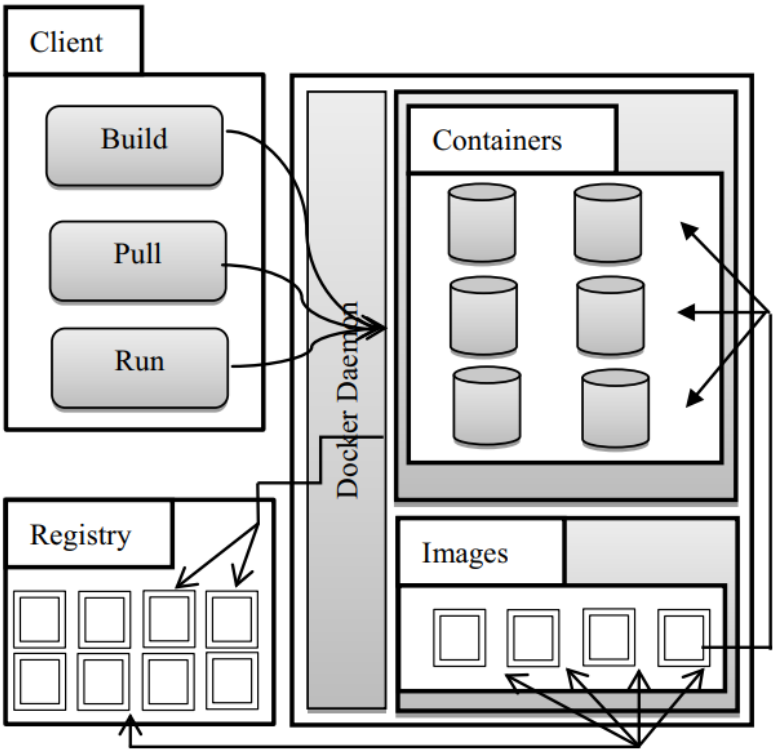


Fig. 16. Diagrama de flujo de Docker

Fuente: (Singh et al., 2019)

La tecnología Docker es una solución de virtualización ligera, que mejora la ejecución de un contenedor y disminuye el consumo de memoria al compartir el mismo kernel entre el host y un contenedor virtual. Un contenedor Docker es semejante a una máquina virtual, puesto que puede tener las funciones de un sistema operativo especifico. A comparación de las máquinas virtuales comunes, una de las ventajas principales de Docker es su velocidad de crear, detener, eliminar las aplicaciones (Qian et al., 2020).

1.4.2. Virtualización

La virtualización se puede realizar con máquinas virtuales o contenedores. Como se observa en la Fig. 17 las máquinas virtuales utilizan diferentes hipervisores y dentro de ella existe un sistema operativo invitado donde se ejecuta las librerías, los archivos binarios y las aplicaciones; en cambio, en la tecnología de contenedores crea un entorno donde se ejecuta aplicaciones independientes del sistema operativo, por lo tanto, supera a las máquinas virtuales en términos de tiempo de arranque y escalabilidad. El objetivo de la virtualización y la contenerización se describe con el slogan “Escribir una vez, ejecutar en cualquier lugar (WORA)” (Sollfrank et al., 2021).

Máquina Virtual

Representa la virtualización a nivel del hardware, sobre el que habitualmente se ejecuta un sistema operativo completo. Esta arquitectura se encuentra conformado por una máquina física, es decir, algún tipo de hardware, pero la infraestructura por sí misma no es nada, si no contiene un sistema operativo, posteriormente se encuentra el hipervisor por ejemplo Virtualbox y, por encima de todo esto se encuentra las diferentes aplicaciones, las cuales deben disponer de su propio sistema operativo para poder funcionar (Álvaro & Martínez, 2021). En otras palabras, es una máquina dentro de otra máquina con los mismos componentes que tiene una maquina real y al ejecutarse uno de ellos, consume demasiada RAM, CPU colocando a la máquina lenta.

Contenedores

Los contenedores son procesos aislados, pero en lugar de alojar un sistema operativo completo, lo hacen compartiendo los recursos del propio host sobre el que se ejecutan. Los contenedores también necesitan una máquina física, así como un sistema operativo instalado en el host, pero a diferencia de las máquinas virtuales no necesitan de un hipervisor que corra los diferentes sistemas operativos, sino por el denominado Docker Engine (Álvaro & Martínez, 2021). La tecnología de contenedores es una alternativa a las máquinas virtuales, donde permite mejorar el uso de los recursos del hardware. Por lo tanto, al formar parte de un proceso más del sistema utiliza una cantidad mínima de RAM y CPU.

La principal diferencia es que una máquina virtual necesita tener virtualizado todo el sistema operativo, mientras que el contenedor aprovecha el sistema operativo sobre el cual se ejecuta, es decir la virtualización de las máquinas virtuales es a nivel del hardware y la virtualización de Docker se manifiesta a nivel del sistema operativo, ya que, para el sistema operativo anfitrión, cada contenedor no es más que un proceso que corre sobre el kernel (Guijarro et al., 2019). La ventaja de estos contenedores es que se pueden recrearlo muy fácil.

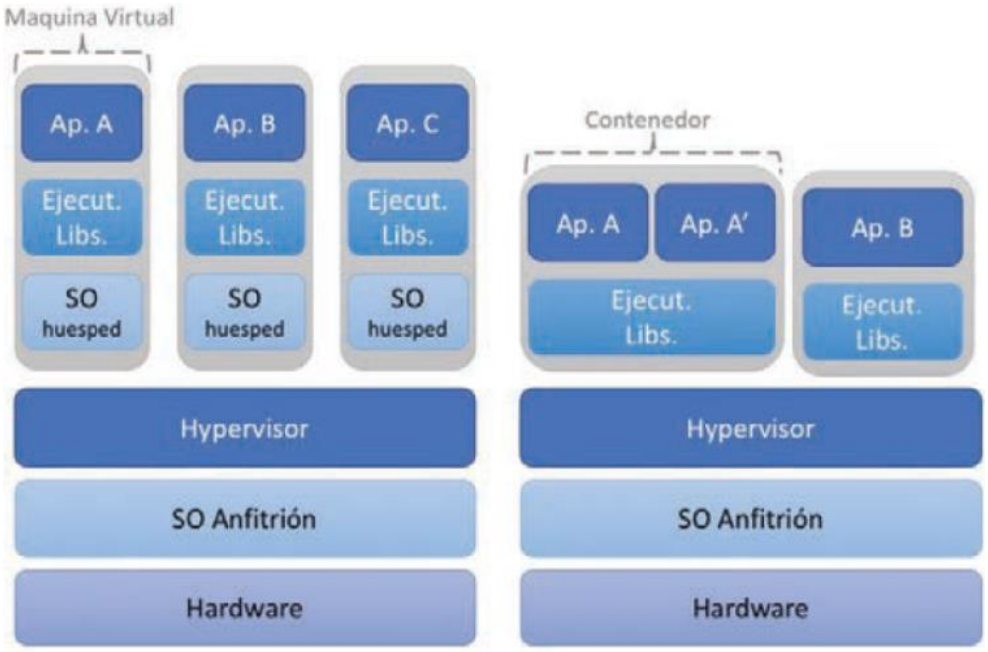


Fig. 17. Máquinas virtuales vs contenedores

Fuente: (Roldán Martínez et al., 2018)

1.4.3. Características

En el desarrollo de las aplicaciones, a menudo se encuentran fallas en la reutilización de software causadas por los diferentes entornos de desarrollo. Sin embargo, la herramienta Docker puede resolver eficazmente estos problemas, puesto que todas las configuraciones se encuentran dentro de un contenedor permitiendo así que el desarrollo, la prueba y el despliegue del software ocurran en el mismo entorno. Por lo tanto, Docker es una herramienta que permite automatizar el despliegue de aplicaciones dentro de contenedores, de forma rápida y portable (You & Sun, 2022). En la Tabla 6 se detalla las principales características de Docker.

Tabla 6: Características de Docker

Atributos	Características
Portabilidad	Despliegue en cualquier sistema, sin necesidad de volver a configurarlo para que la aplicación funcione, esto se debe a que todas las dependencias están empaquetadas con la aplicación en el contenedor.
Ligereza	No es necesario instalar un sistema operativo completo por cada contenedor
Autosuficiencia	Un contenedor Docker solo contiene aquellas librerías, archivos y configuraciones necesarias para desplegar la aplicación.

Fuente: (Guijarro et al., 2019)

1.4.4. Ventajas y Desventajas

Al emplear contenedores Docker permiten a los administradores y desarrolladores de software ejecutar aplicaciones en un entorno seguro, con una mejor adaptación entre los entornos de prueba y producción, obteniendo una optimización de recursos, no obstante, también tiene su contra (Guijarro et al., 2019). En la Tabla 7 muestra las principales ventajas y desventajas de usar contenedores Docker.

Tabla 7. Ventajas y Desventajas de Docker

Ventajas	Desventajas
Las instancias son replicables y se inician en pocos minutos	Para su ejecución requiere como mínimo un Kernel de 3.8
Facilidad en la automatización e integración en entornos de integración continua	En sistemas operativos Windows no se puede utilizar de forma nativa
Consume menos recursos del hardware debido a que se ejecuta directamente sobre el kernel teniendo mayor rendimiento	Soporta arquitecturas de 64 bits, en Ubuntu de 16.04 en adelante
Ocupan menos espacios que las máquinas virtuales	Soporta en sistemas operativos de 64 bits en Windows 8 en adelante
con Contiene varias imágenes que pueden ser descargadas libremente	Se necesita tener acceso a Internet para la descarga de imágenes
Aislada de la maquina física, por ende, el contenedor es fácil y seguro de administrar	Por ser nuevo, puede existir errores de código entre versiones

Fuente: (Pacheco, 2018)

1.4.5. Arquitectura de Docker

Docker se basa en la arquitectura cliente-servidor, donde el Servidor Docker es aquel que presta todos los servicios por medio de Rest API, que es el canal de la comunicación entre Docker CLI-Client con el Servidor Docker. A través de Docker CLI-Client se puede manejar contenedores, imágenes, volúmenes y redes (You & Sun, 2022). A continuación, se detalla el funcionamiento de cada uno de los componentes que forman parte de la arquitectura de la herramienta Docker definida en la Fig. 18.

a) **Docker Daemon:** se ejecuta dentro del sistema operativo, donde expone una API para la gestión de los componentes de Docker.
b) **Docker Client:** permite administrar los entornos de desarrollo local como los servidores de producción.

c) **REST API:** permite la comunicación entre el Docker Daemon y el Cliente.
d) **Imágenes:** es un paquete que contine toda la configuración necesaria para que se ejecute la aplicación.
e) **Contenedores:** es una capa de lectura y escritura que instancia la ejecución de una imagen temporal.
f) **Volúmenes:** los volúmenes de Docker son herramientas muy utilizadas y útiles para garantizar la persistencia de los datos mientras se trabaja en contenedores.
g) **Redes:** las redes en Docker permiten la comunicación entre diferentes contenedores.

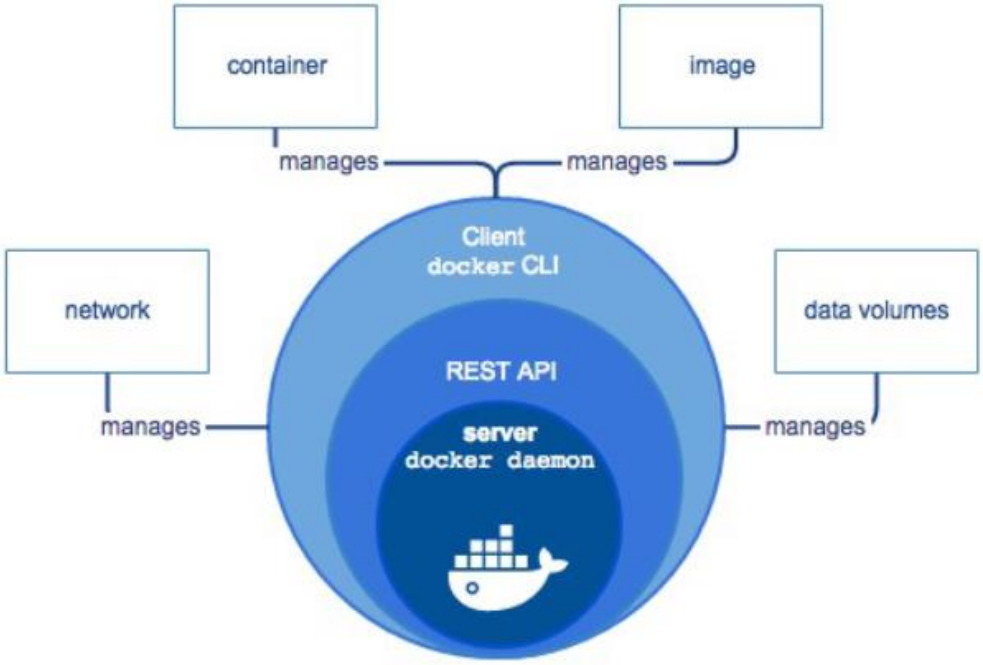


Fig. 18. Arquitectura Docker
Fuente: (Álvaro & Martínez, 2021)

1.4.6. Tipos de componentes en Docker

Imágenes

Son plantillas de modo lectura, que se utiliza como base para ejecutar un contenedor, por lo tanto, los cambios que se realiza en el contenedor solo persisten en ese contenedor y si se requiere que los cambios sean permanentes se debe crear una nueva imagen con un contendedor personalizado. Existen varias imágenes ya configuradas que se pueden descargarse mediante el repositorio de Docker (Docker Hub), pero también se puede construir uno propio a través de un archivo Dockerfile, como se ilustra en la Fig. 19 (Guijarro et al., 2019).

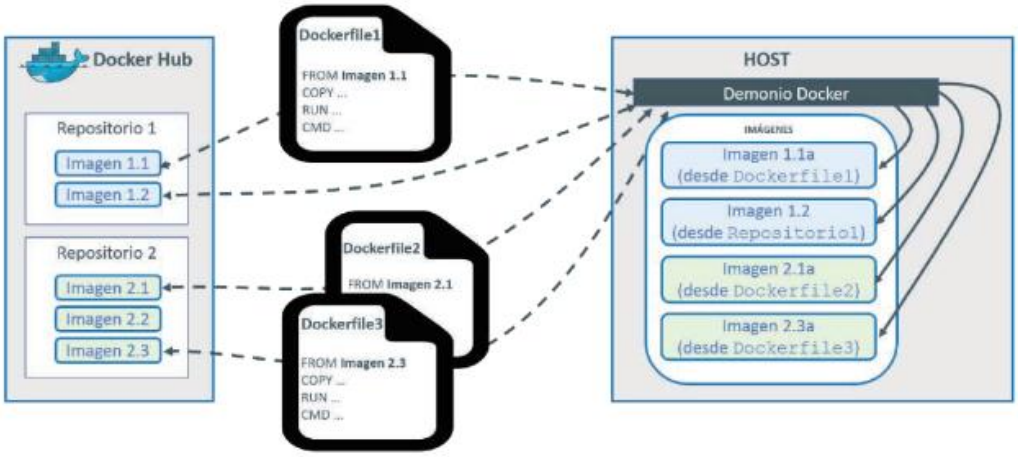


Fig. 19. Crear imágenes de Docker

Fuente: (Roldán Martínez et al., 2018)

Dockerfile

Es un archivo de texto plano que contiene una serie de instrucciones necesarias para construir una imagen y automatizar el proceso de la creación de un contenedor como muestra en la Fig. 20. El Daemon de Docker es el encargado de crear, las imágenes siguiendo las instrucciones del archivo Dockerfile y va proyectando los resultados línea por línea en la pantalla (Guijarro et al., 2019).

Fig. 20. Proceso de construcción de un contenedor

Fuente: (Álvaro & Martínez, 2021)

Contenedores

Es una instancia ejecutable de una imagen que integra todo lo necesario para desplegar una aplicación. De tal manera que se logra realizar las diferentes acciones de crear, iniciar, detener, eliminar sobre los contenedores. Cada contenedor se ejecuta en un entorno independiente, con sus propias variables, puertos, tal como se muestra en la Fig. 21. Por otra parte, al borrar un contenedor toda la información es eliminado, excepto si se haya construido volúmenes para la persistencia de los datos (Álvaro & Martínez, 2021).

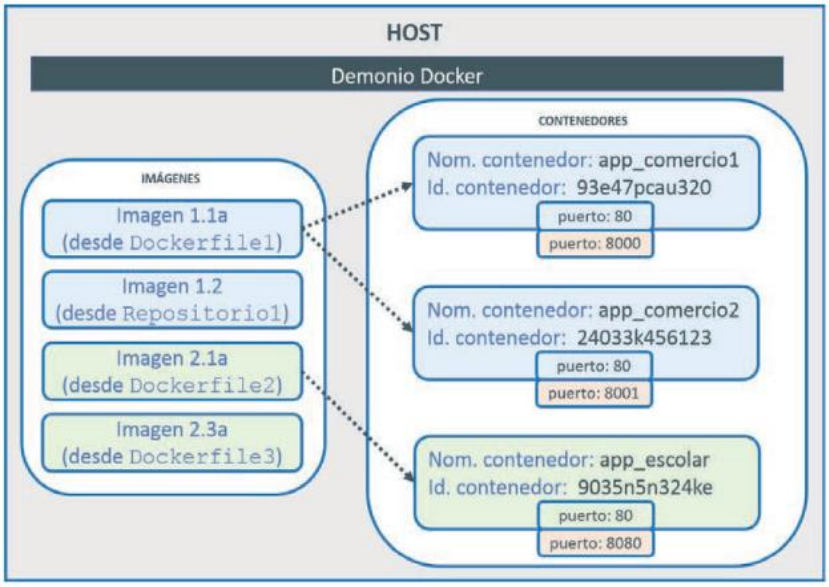


Fig. 21. Crear imágenes de Docker

Fuente: (Roldán Martínez et al., 2018)

Volúmenes

Es un mecanismo que proporciona automáticamente la persistencia de datos a través de los volúmenes, la cual pueden ser instalados directamente dentro de contenedores. Para realizar esta persistencia de los datos, los volúmenes permiten separar los datos de los contenedores, donde estos volúmenes son almacenados como carpetas en un espacio determinado para Docker, particularmente en la ruta /var/lib/docker/volumes, como se ilustra en la Fig. 22 (Roldán Martínez et al., 2018).

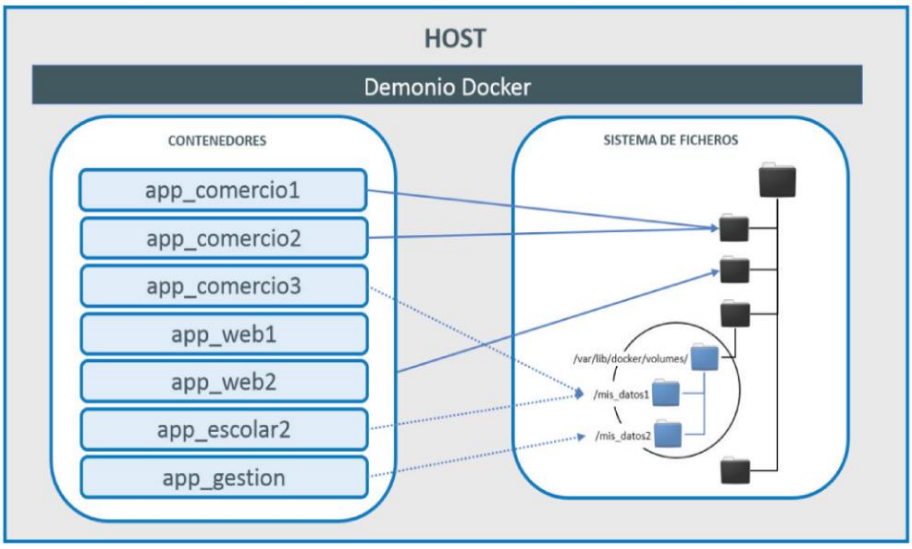


Fig. 22. Crear imágenes de Docker

Fuente: (Roldán Martínez et al., 2018)

Redes

Son las diferentes maneras de configuración de la red, donde se encarga que todos los contenedores aislados se comuniquen entre sí. Estos contenedores se pueden comunicarse entre ellos a través de varios métodos, algunos de ellos se asignan de manera predeterminada por el sistema y dentro de las redes de Docker más principales se encuentra: None, Brigde, Host (Álvaro & Martínez, 2021).

- **None:** indica cuando el contenedor no contiene una interfaz de red. Este tipo de red se puede utilizar en contenedores de prueba que no necesita tener una comunicación con nadie.
- **Brigde:** el modo brigde es una red estándar que toma por defecto al iniciar una plataforma de Docker y todos sus contenedores usarán esta red.
- **Host:** excluye el aislamiento que existe en la red y se usan las mismas interfaces de la red de host, es decir un contenedor utilizará la misma dirección IP del servidor.

Orquestador: Docker Compose

El objetivo de Docker Compose es agrupar un conjunto de contenedores mediante un archivo de formato YML, es decir permite definir y ejecutar aplicaciones con varios contenedores, por lo tanto, crea e inicia todos los servicios desde su configuración a través de un solo comando. Esta herramienta de Docker es muy útil en el desarrollo de las aplicaciones, pruebas y como también en los flujos de trabajo de integración continua (Raj & Jasmine, 2021). Al ejecutar el comando *docker-compose up*, pone en marcha todo lo necesario para desplegar la aplicación. Trabajar con este tipo de archivo implica seguir estos tres pasos (Roldán Martínez et al., 2018).

1. Definir el entorno de la aplicación a través del archivo *Dockerfile*, de tal manera que se pueda reproducir en cualquier lugar.
2. Definir en el archivo *docker-compose.yml*, los servicios que determinan la aplicación.
3. Ejecutar el archivo YAML para iniciar la herramienta y desplegar la aplicación.

1.4.7. Dockerización de Microservicios

Según (Qian et al., 2020), manifiesta que la combinación de la tecnología Docker y la arquitectura de microservicios en gran medida mejora el rendimiento y la eficiencia de los sistemas de información. De manera que una computadora despliega varios contenedores al mismo tiempo, lo que facilita la simulación de una arquitectura de microservicios. Como se

observa en la Fig. 23 cada contenedor ejecuta diferentes instancias de microservicio, la cual deben estar empaquetados en imagen de Docker. Además, con la utilización de DevOps se puede aligerar la implementación, monitoreo de las aplicaciones. En la siguiente sección se profundizará este concepto de DevOps.

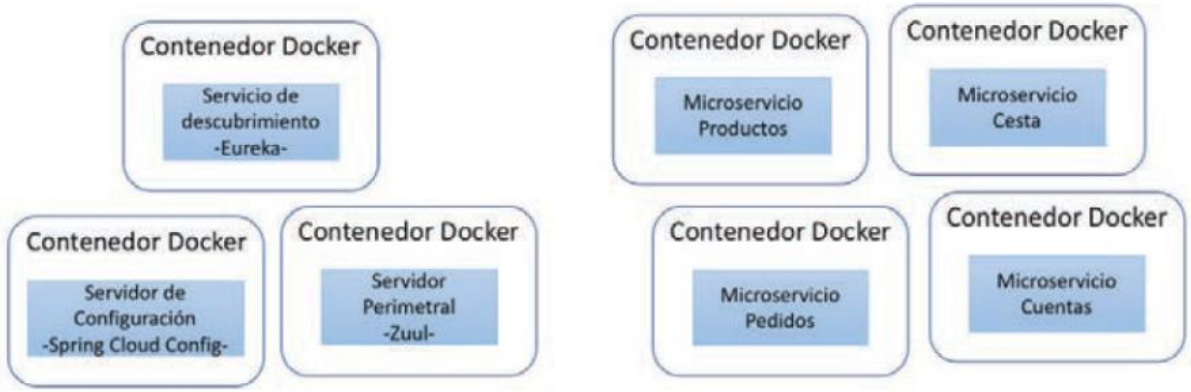


Fig. 23. Microservicios en contenedores
Fuente: (Roldán Martínez et al., 2018)

1.5. Developer Operation (DevOps)

El termino DevOps proviene de una combinación de las palabras desarrollador y operaciones, la cual define un conjunto de prácticas de valores culturales basado en los principios de desarrollo de softwares ágiles. Se centra en la integración y entrega continuas (CI/CD) de software, lo que significa una nueva cultura de desarrollo e implementación de software (Rodríguez et al., 2020).

DevOps se define como un movimiento que nace de la necesidad de reducir las barreras entre los equipos de desarrollo y operaciones de una empresa, para ello, se introduce prácticas como la integración y entrega continua, la cual ayudan a reducir el tiempo de salida al mercado y a producir un software de calidad. Para tener éxito DevOps debe ser promovido al más alto nivel de la empresa y aceptado por cada uno de sus departamentos (Riti, 2018).

Por otro parte, para satisfacer las necesidades los operadores prefieren utilizar un enfoque basado en imágenes (imágenes Docker) para administrar el ciclo de vida del código, como se ilustra en la Fig. 24. Cuando ocurre un problema en una nueva versión, los operadores no necesitan depurar y corregir en el entorno de producción, simplemente recurren a la imagen del contenedor anterior y esperan a que esté disponible una nueva imagen con correcciones. Este enfoque basado en imágenes combinados con un diseño de estilo microservicio, amplía los beneficios de desacoplar los componentes del sistema, evitando la interferencia entre si durante el mantenimiento (Kang et al., 2016).

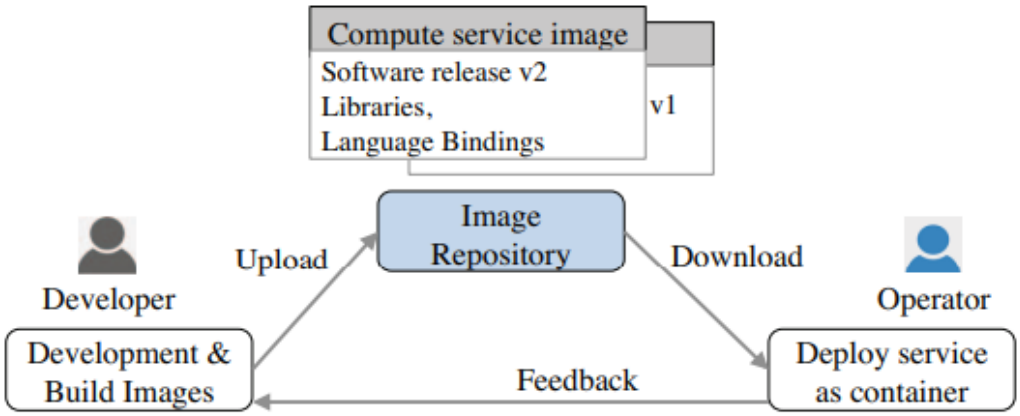


Fig. 24. Arquitectura basada en Microservicios y DevOps

Fuente: (Kang et al., 2016)

Actualmente DevOps se está convirtiéndose más popular, debido a la entrega rápida de aplicaciones al mercado con bajo costo y corta duración. Por ello, varias empresas han adoptado masivamente los principios de DevOps como Google, Netflix, Amazon, LinkedIn, y Spotify para desplegar software con mayor velocidad y mejor calidad. A pesar de la creciente popularidad de esta tecnología, es un desafío adoptar estas técnicas, ya que no existe una visión general clara de estos procedimientos. El cambio cultural es una barrera difícil de transformar, dado que afecta a todos, desde la parte gerencial hasta cada uno de los departamentos de la empresa (Khan et al., 2022). Además, varias fábricas de software han implementado DevOps, con el fin de optimizar sus procesos y crear servicios de calidad.

Fábrica de Software

El término fábrica de software está determinada por un factor de tamaño, que se encarga no solamente del componente del desarrollo sino también en las pautas de la creación, innovación, en factores de adquisición de software desarrollados y en las actualizaciones de los procesos. Una fábrica de software es un entorno donde se emplea actividades relacionadas con el ciclo de vida del software, de acuerdo con los procesos, estándares y modelos de calidad (Bernardi et al., 2017).

1.5.1. Beneficios de DevOps

Varias empresas deciden adoptar DevOps, para obtener una mejora en la calidad del software y en la gestión del despliegue, para ello se opta procedimientos en base a la integración y entrega continua, la cual permite identificar fácilmente errores cuando el código es integrado en el repositorio y por medio de la entrega continua se puede desplegar el software directamente en el control de calidad varias veces (Riti, 2018). Según estudios realizados por Sánchez et al (2020), las organizaciones han identificado algunos beneficios

después de la adopción de las prácticas de DevOps en sus empresas, la cual se ilustra en la Fig. 25.

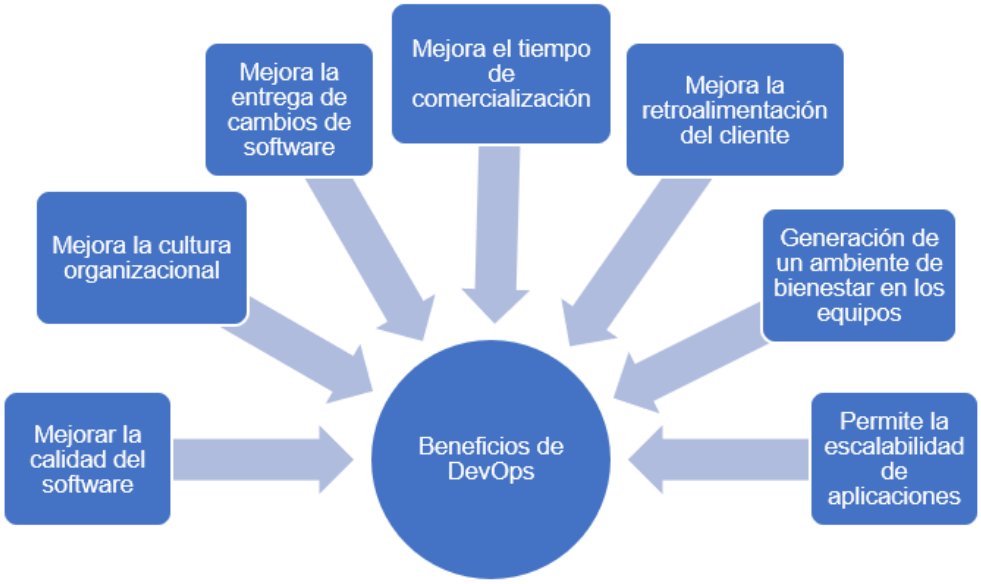


Fig. 25. Beneficios de DevOps

Fuente: (Sánchez et al., 2020)

1.5.2. Modelos de DevOps

El uso de las políticas de integración, entrega y despliegue continuo (CI/CD), ayudan a las organizaciones a identificar más rápido los problemas, permitiendo así acelerar el desarrollo y entrega de software sin afectar su calidad (Shahin et al., 2017). Estos procesos implican un cambio en todos los equipos de desarrollo y operaciones, puesto que para construir un sistema se realizarán tareas compartidas entre estos equipos (Guijarro et al., 2019). En la Fig. 26 muestra la relación que existe entre estos tres procesos de la ingeniería de software continua.

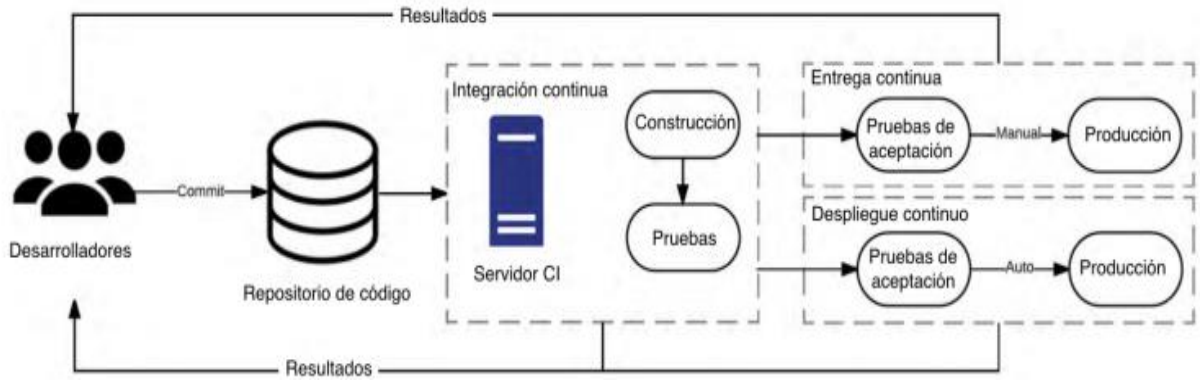


Fig. 26. Relación entre integración, entrega y despliegue continuo

Fuente: (Shahin et al., 2017)

a) Integración Continua

El proceso de Integración Continua en sus siglas en inglés CI (Continuos Integration), afecta principalmente al proceso de desarrollo. Por ejemplo, cuando los desarrolladores integran nuevas versiones dentro del código original. El principal objetivo de la integración continua es procesar la nueva versión creada, luego compilarla, desplegarla y testearla de forma automática, de manera que devuelva un reporte a los desarrolladores con los posibles errores que se susciten durante la fase de testeo; siendo así, los desarrolladores realizan un proceso de retroalimentación del sistema a través de un bucle de cambios (versión – compilación – test – corrección). Una vez realizados los cambios y superado las pruebas, se procede a integrarlos en el control de versiones de manera definitiva para posterior ser puesta en producción. En cuanto a la herramienta, se puede utilizar Jenkins que permite realizar la integración continua de manera flexible (Guijarro et al., 2019).

b) Entrega Continua

La entrega continua o CD (Continuous delivery), es la continuidad de la integración continua, como se ilustra en la Fig. 26, puesto que los cambios que se realizaron en la parte del desarrollo puedan ser entregados al entorno de producción en menor tiempo. Con la ayuda de este proceso, se puede evitar que haya demasiada incertidumbre en la entrega del producto (Guijarro et al., 2019). Por otra parte, Shahin et al (2017) menciona que la entrega continua emplea un conjunto de prácticas de Integración continua (CI) la cual automatiza la entregar el software a producción, es decir, garantiza que la aplicación se encuentre listo para el entorno de producción después de pasar con éxito las pruebas automatizadas y los controles de calidad.

c) Despliegue Continuo

El despliegue continuo o CD (Continuous Deployment), se basa en un proceso de pasos que se ejecutan de manera ordenada y correcta sin intervención humana, es decir realiza el despliega de la aplicación de forma automática y continua en entornos de producción. Por otra parte, existen fuerte debates sobre definir y distinguir entre entrega y despliegue continuos, lo que les diferencia es un entorno de producción (clientes reales). Como se observa en la Fig. 26 el despliegue final en la entrega continua es un paso manual, por lo que las empresas deciden qué y cuándo desplegar, mientras que, en el despliegue continuo, se encarga de desplegar de forma automática y constante cada cambio en el entorno de producción (Shahin et al., 2017).

1.5.3. Entornos del desarrollo de software

Para cubrir el desarrollo basado en DevOps, se debe definir los entornos en los que se desarrollará diferentes tareas de verificación, para poder así pasar al siguiente entorno (Guijarro et al., 2019). En la Fig. 27 muestra una configuración de un entorno típico para el escenario en DevOps.

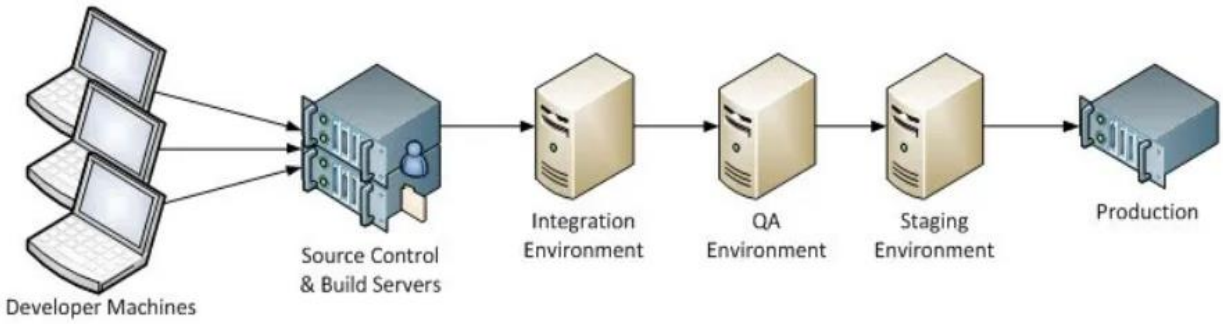


Fig. 27. Etapas de entornos de desarrollo
Fuente: https://www.infoq.com/articles/Continous-Delivery-Patterns/

A continuación, en la Tabla 8 se describe cada uno de los entornos que debe pasar el software antes de llegar a producción.

Tabla 8. Ambiente de desarrollo de software

Entorno	Definición
Desarrollo local	Se inicia en el propio computador del desarrollador, donde el código generado se compila localmente. Se emplean herramientas como Virtualbox, Vagrant o Docker para desplegar el código en un ambiente similar al de producción. Finalmente, el nuevo código se transfiere al entorno de integración.
Entornos de integración	En este entorno, los nuevos cambios se almacenan, compilan y se integran con el resto del código. Por medio de la herramienta Git se realiza el control del código fuente y se utiliza Jenkins como servidor de integración.
Entorno de test	Las pruebas funcionales y de regresión se realizan en este entorno, la cual aseguran que la funcionalidad existente en el software no se dañe y que la incorporación de nuevas funciones cumpla con los requisitos.

Entornos de preproducción	Es una réplica al entorno de producción. En este entorno, la funcionalidad del software se da por sentada y el objetivo es realizar pruebas de integración con otros componentes de software, como pruebas de rendimiento, especialmente pruebas de estrés debido a que dependen en gran medida del entorno.
Entorno de producción	Finalmente, está el entorno de producción, donde los usuarios interactúan con el software implementado.

Fuente: (Guijarro et al., 2019)

1.5.4. Arquitectura DevOps

El movimiento DevOps avanza según una cadena de herramientas, básicamente esta cadena de herramienta define cada paso del proceso de producción. La Fig. 28 muestra las fases de DevOps de una versión de software, y cada fase puede ser gestionada por un equipo diferente. (Riti, 2018).

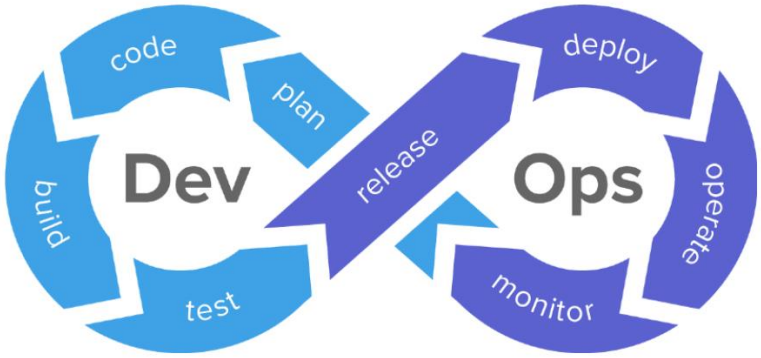


Fig. 28. Ciclo de vida DevOps

Fuente: https://wesecureapp.com/blog/what-is-devsecops-and-its-role-in-devops-architecture/

La coordinación y la comunicación son esenciales para poner en marcha una cadena completa de DevOps. Para ello, cada fase requiere una buena coordinación en cada paso (Riti, 2018). En la Tabla 9 se detalla cada una de las fases que se debe abordar para obtener un producto de software de calidad.

Tabla 9. Componentes del ciclo de vida de DevOps

Fases	Definición
Planificación	La primera fase se define la arquitectura con la que se va a trabajar.
Codificación	Durante esta fase se crea el código del software y cada desarrollador coloca su código en un repositorio común
Construcción	Esta fase está directamente relacionada con la integración continua, donde los procesos se dividen en pequeños ciclos acelerando el desarrollo y la entrega. En esta fase se realizan las primeras pruebas
Pruebas	El software construido se prueba en su totalidad mediante algún proceso automático (equipos de QA)
Liberación	En esta fase se configura el servidor, así como la infraestructura para el nuevo software
Despliegue	En esta fase se podrá ver una versión ya del sistema y cuando se realiza algún cambio esto no debe afectar la funcionalidad del software
Operación	Se realiza un seguimiento continuo, donde se verifica la funcionalidad de cada proceso en cada fase de la aplicación
Monitoreo	Esta última fase proporciona una retroalimentación continua sobre el software e infraestructura. El monitoreo es muy importante en DevOps porque permite al desarrollador obtener una retroalimentación sobre el software incluido un promedio de las fallas

Fuente: (Riti, 2018)

1.6. Herramientas tecnológicas para el desarrollo de la aplicación

1.6.1. NestJS

NestJS, es un framework basado en Node y Express, creado por Kamil Myśliwiec, que permite crear aplicaciones Backend flexibles y escalables, bajo una estructura MVC. NestJS hace uso del lenguaje Typescript en vez de un Javascript nativo. Además, proporciona una estructura base con directorios y un router listo para usarse (NestJS, 2022).

¿Por qué NestJS?

NestJS, proporciona un conjunto de buenas prácticas para crear aplicaciones del lado del servidor. Su estructura se basa en la utilización de decoradores, es decir estos componentes se emplean como cimientos que dan soporte al funcionamiento de la aplicación. NestJS se caracteriza por brindar una estructura de trabajo basada en módulos, similar a la estructura de Angular. Tiene soporte para diferentes tipos de bases de datos. Además, permite crear aplicaciones basadas en arquitecturas monolítica o de microservicios (Pham, 2020)

Estructura del Framework NestJS

A continuación, se describe los siguientes conceptos, que son la base para construir una aplicación en NestJS.

- **Nest /CLI:** Esta herramienta permite inicializar la aplicación del proyecto, con diferentes administradores de paquetes, en este caso se seleccionó npm, por lo que se debe tener instalado Node.js en el computador. Una vez generado el proyecto se crea una estructura con los siguientes archivos: app.module.ts, app.service, app.controller.ts y main.ts (Sabo, 2020).

- **Módulos:** Utiliza el decorador *@Module(),* que permite la organización de la aplicación. En esta clase da lugar a la conexión de servicios, controladores, resolver, entre otros. Cada aplicación necesita al menos un módulo (modulo raíz) para inicializar la aplicación (Pham, 2020).

- **Controladores:** Utiliza el decorador *@Controller().* Los controladores son los encargados de gestionar las solicitudes, y devolver las respuestas al cliente (NestJS, 2022). El controlador proporciona una estructura para el desarrollo de API REST, basada en los decoradores (@GET, @POST, @DELETE, @PUT) correspondientes para cada método HTTP (Sabo, 2020).

- **Proveedores:** Es una clase anotada con el decorador *@Inyectable().* El objetivo de un proveedor es inyectar dependencias, es decir que se puedan relacionarse entre componentes. (Pham, 2020).

- **Servicios:** Se encarga de la lógica empresarial de la aplicación. Los servicios pertenecen al grupo de los proveedores y son un intermediario entre el controlador y la base de datos, como se ilustra en la Fig. 29. Utiliza el decorador *@Inyectable()* (Pham, 2020).

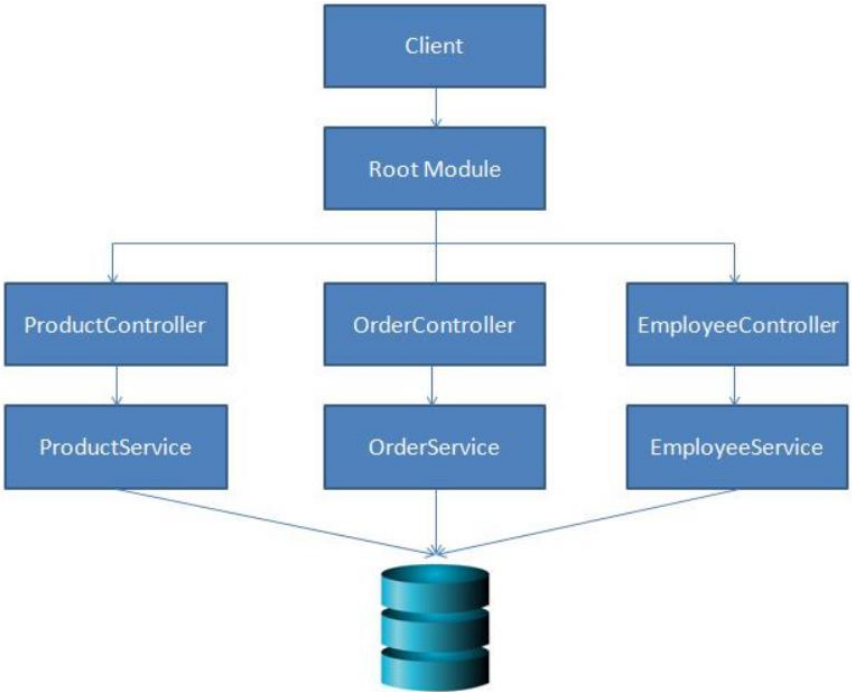


Fig. 29. Arquitectura de la API de NestJS

Fuente: https://betterprogramming.pub/getting-started-with-nestjs-a4e8b0b09db4/

Por otra parte, NestJs admite el desarrollo de servicios web GraphQL, que se basan principalmente en resolutores y esquemas.

- **Resolutores:** Esta clase usa el decorador *@Resolver.* Los resolutores son los encargados de gestionar las consultas y generar respuestas al cliente. A través de los decoradores se ejecuta las siguientes operaciones: consultas @Query, para insertar, eliminar, actualizar @Mutation y @Subscription para las suscripciones (NestJS, 2022).

Typescript

De acuerdo con las investigaciones, Typescript puede evitar que los proyectos fracasen hasta en un 15%, debido a que Typescript analiza constantemente el código en busca de posibles errores o mejores formas de codificación (Gao et al., 2017). Según las descargas de paquete, la herramienta NPM Trends indica que Typescrypt está en crecimiento y es utilizado por varias librerías, proyectos hasta la actualidad. Ver Fig. 30.

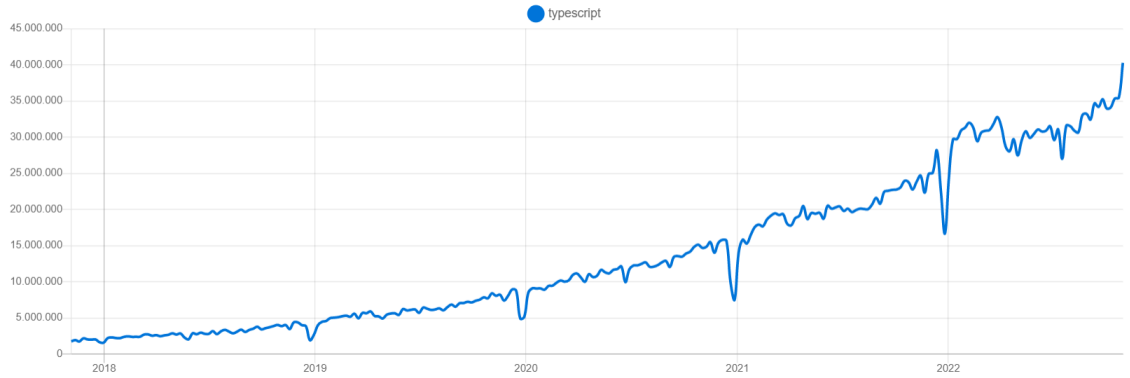


Fig. 30. Arquitectura de la API de NestJS
Fuente: (NPM Trends, 2022)

1.6.2. PostgreSQL

Para la integración de la base de dato en NestJS. Se decidió implementar PostgreSQL, debido a que ofrece un mejor rendimiento y escalabilidad al manejar grandes cantidades de datos y consultas (PostgreSQL, 2022). NestJS a través de un ORM, específicamente TypeORM, permite implementar una estructura estandarizada para cualquier tipo de base de datos. En este caso se utilizó PostgreSQL para crear las tablas y consultas. De igual manera si existe la necesidad de realizar cambios a las tablas se debe eliminar y generar la base de datos nuevamente (NestJS, 2022).

1.6.3. Docker

Docker fue elegido como la mejor opción para contenerizar aplicaciones, debido a su amplio soporte a la comunidad, su extensa documentación y en la actualidad a nivel profesional es la solución más utilizada por los desarrollares. Las estadísticas de Google Trends, muestran que Docker lidera en el mercado actual, según la comparativa realizada entre las herramientas Docker, LXD, y Podman, como se observa en la Fig. 31 (Arriba García, 2019). Por ello, se va a utilizar esta tecnología para realizar la contenerización de la aplicación.

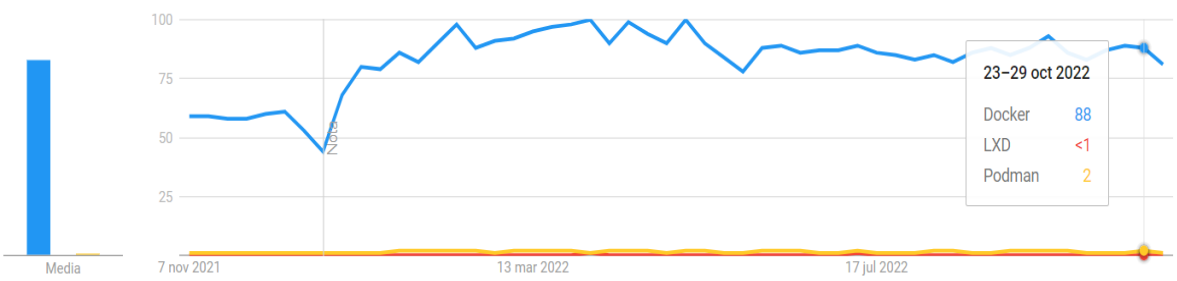


Fig. 31. Comparativa entre Docker, LXD y Podman
Fuente: (NPM Trends, 2022)

1.6.4. Bitbucket

Al desarrollar un proyecto, es importante utilizar algún tipo de control de versiones si se trabaja en proyectos en equipos o individualmente. Existe dos herramientas que destacan su popularidad a nivel mundial Bitbucket y GitHub. Bitbucket, es una tecnología de Attlasian, basado en la nube, que permite el alojamiento de código. Además, facilita la colaboración en equipos y la gestión de proyectos. También ofrece integración con herramientas como JIRA, Trello y Jenkins, por lo que se eligió este repositorio. Una de las principales diferencias es que Bitbucket se adapta mejor a la gestión de proyectos privados, mientras que GitHub se centra en repositorios públicos (Sharma, 2022).

1.6.5. Nginx

Es un servidor web *"open source",* creado por Igor Sysoev y desplegada de manera pública en el año 2004. Nginx se caracteriza por su arquitectura basada en eventos, la cual permite el manejo de varias conexiones dentro de un solo hilo. Además, actúa como un proxy inverso simple que recibe solicitudes HTTP del cliente y reenvía al servidor Backend. Según investigaciones, Nginx es considerado más eficiente que Apache 2, en cuanto al rendimiento, utilización de recursos y en capacidad (Palma, 2020).

1.7. Metodologías de Desarrollo

1.7.1. Marco de trabajo Scrum

Scrum es un marco de trabajo que permite la gestión del desarrollo de un proyecto, por tanto, los resultados se deben presentar en periodos regulares y en menor tiempo. Scrum consta de los siguientes componentes: roles, artefactos y eventos. Los roles se asignan a personas que participan en Scrum, los artefactos son registros de actividades realizadas y los eventos se refieren a las acciones que se debe realizar el equipo Scrum (Ramos et al., 2016). En la Tabla 10 se detalla cada uno de los componentes de Scrum con sus respectivos elementos. Scrum es un proceso optimizado y ágil que se vincula automáticamente a la entrega continua. Se puede aplicar en todas las fases del ciclo de vida del software: planificación, diseño, desarrollo, pruebas, despliegue y mantenimiento.

Tabla 10. Elementos SCRUM

Roles	Artefactos	Eventos
• Scrum Master • Dueño del Producto • Equipo de Desarrollo	• Épicas • Product Backlog • Sprint Backlog • Incremento	• Scrum Diario • Revisión del Sprint • Retrospectiva del Sprint

Fuente: Propia

Ciclo de vida de Scrum

Scrum se basa en un ciclo de vida iterativo e incremental que comprende tres fases: *Pre-juego, Juego* y *Post-juego,* cómo se ilustra en la Fig. 32. La fase de pre-juego inicia con la planificación mediante la generación de los requerimientos (Produt Backlog) del proyecto, es decir especifica las actividades a realizar en las iteraciones. En la fase del juego se realiza la implementación del proyecto en cada iteración mediante el Sprint Backlog y el Incremento. Una iteración puede durar entre de 2 0 4 semanas aproximadamente. Por último, en la fase de post-juego se realiza la entrega final del producto acorde a las pruebas de aceptación (Urbina et al., 2016). Por tanto, se aplicará Scrum para el desarrollo del proyecto.

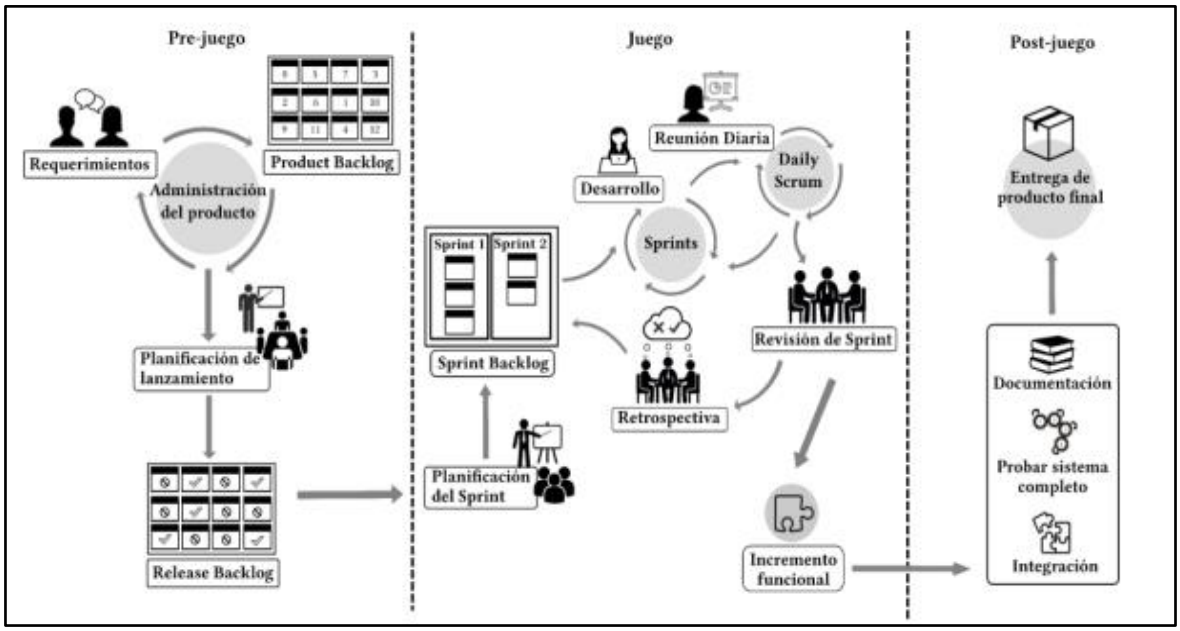


Fig. 32. Ciclo de vida de Scrum
Fuente: (Urbina Mauricio et al., 2016)

1.7.2. Experimentación en la Ingeniería de Software

La experimentación en la ingeniería de software permite determinar la causa de ciertos resultados. La experimentación no es muy sencilla, debido a que se debe preparar, realizar y analizar los experimentos. Una de las ventajas de la experimentación es el control de los sujetos, objetos y la instrumentación. La cual garantiza poder obtener conclusiones más acertadas, como también realizar un análisis estadístico a través de métodos de comprobación de hipótesis (Wholin et al., 2012).

Proceso de Experimentación

Para realizar un experimento implica seguir varios procesos como se ilustra en la Fig. 33, y en la Tabla 11 se detalla cada una de ellas.

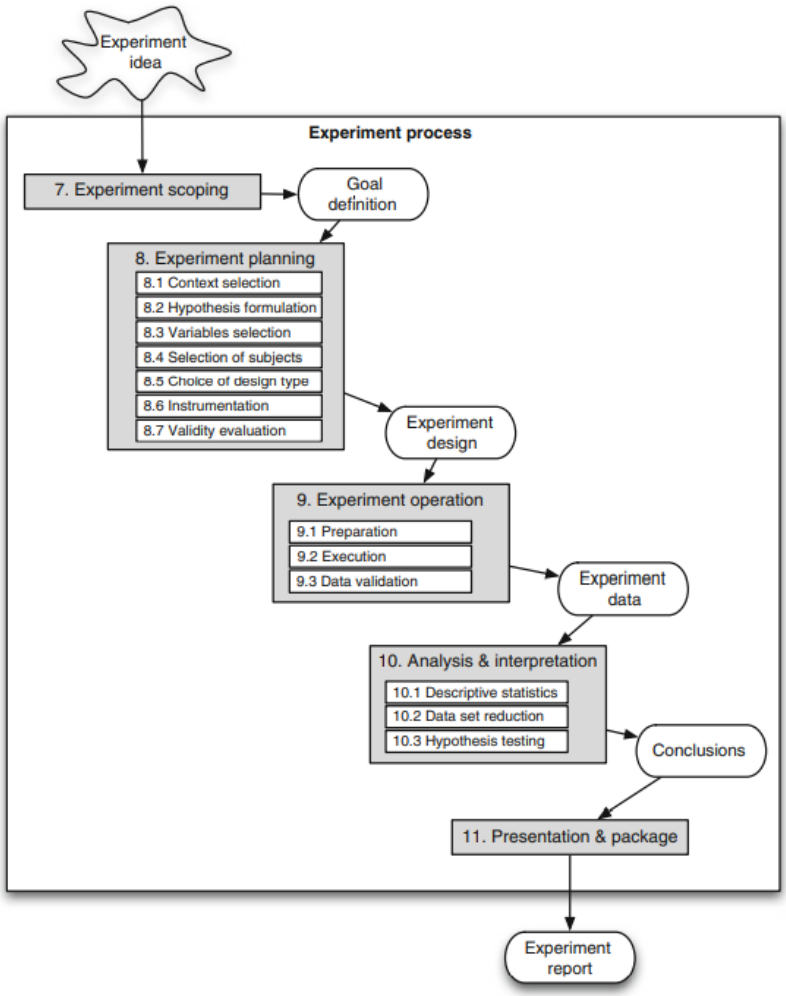


Fig. 33. Proceso experimental

Fuente: (Wholin et al., 2012)

Tabla 11. Proceso para la Experimentación Computacional

Procesos	Descripción
Determinación del alcance	Se debe plantear claramente el alcance, objetivos y las metas del experimento, donde le objeto se define a partir del problema.
Planificación	Se establece en detalle el contexto de la experimentación, esto incluye el personal y el entorno.
Operación	El proceso de operación consta de tres pasos: preparación, ejecución y validación de los datos
Análisis e interpretación	Una vez obtenidos los datos, se procede a analizar e interpretarlos. A partir del análisis se determina si es posible rechazar la hipótesis.
Presentación y paquete	En este último proceso se realiza la presentación de los resultados, la cual incluye una documentación de los resultados

Fuente: (Wholin et al., 2012)

1.7.3. ISO/IEC 25010

Esta norma internacional es parte de la división del modelo de calidad ISO/IEC 2501n de la serie SQuaRE. La cual presenta modelos de calidad para sistemas informáticos y productos de software. En base a las características y subcaracterísticas definidas en este modelo permiten medir, evaluar la calidad del producto del software El modelo de la calidad del producto determina ocho características y cada uno, está compuesto por un conjunto de subcaracterísticas como se muestra en la Fig. 34 (ISO 2510, 2015).

Fig. 34. Modelo de la calidad del producto

Fuente: (ISO 2510, 2015).

Medidas de Eficiencia de Desempeño

Las medidas de eficiencia del desempeño le permiten evaluar el rendimiento con relación a la cantidad de los recursos utilizados en determinadas condiciones. Además, se puede incluir otros productos de software como configuraciones de hardware y software del sistema (ISO 2510, 2015). Esta característica se subdivide a su vez en las siguientes subcaracterísticas:

- **Comportamiento del tiempo:** Nivel en que la respuesta, el tiempo de procesamiento y rendimiento de un sistema son satisfactorios en el desempeño de sus funciones.
- **Utilización de recursos**. Medida en que la cantidad y los tipos de recursos por el sistema son suficientes para realizar su función.
- **Capacidad**. Grado en que el límite de un parámetro de un sistema cumple con un requisito.

Para verificar la eficiencia de las APIs desplegados en la tecnología Docker y en el Entorno Localhost se empleó la medida de comportamiento del tiempo con la siguiente métrica.

- **Tiempo medio de respuesta*:*** Hace referencia al tiempo medio en el que se demora una solicitud o un proceso asíncrono al completar (ISO 2510, 2015).

CAPÍTULO 2

Desarrollo

En este capítulo muestra el proceso de la integración de la aplicación Backend en contenedores Docker. Con el fin de realizar una comparativa entre el entorno localhost con Docker, mediante los servicios GraphQL y REST. El proyecto se desarrolló en base al marco de trabajo Scrum basadas en las tres fases. La primera fase llamado pre-juego, define los requerimientos y el diseño, en la fase del juego se encarga del desarrollo del proyecto de acuerdo con los sprints (iteraciones) planteados; cada sprint no excede a más de 15 días. La tercera fase denominado post-juego establece la finalización de los sprint con la entrega del producto final. En la Fig. 35 muestra la estructura de este capítulo.

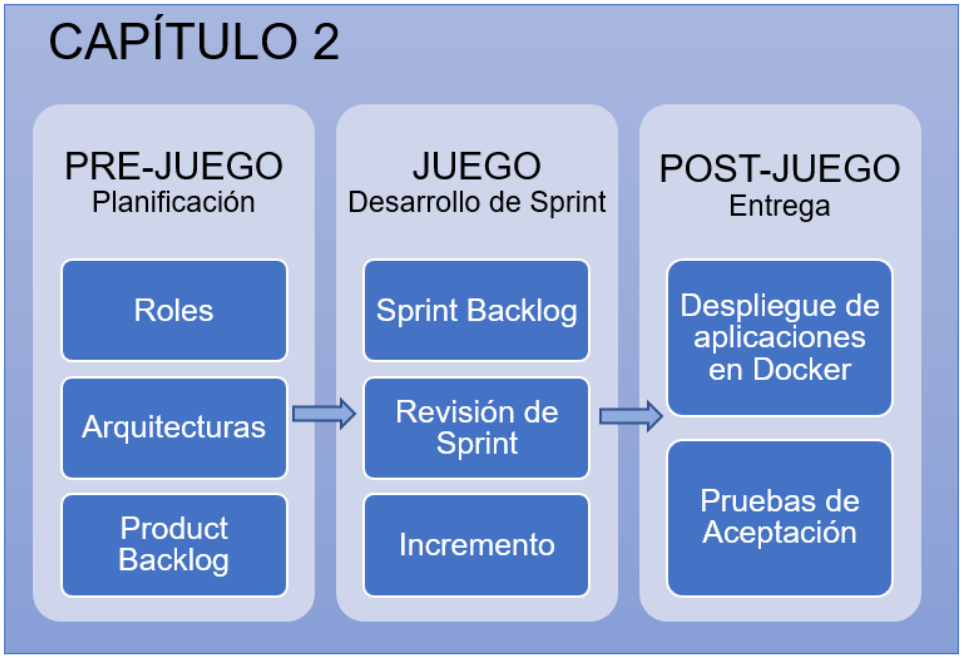


Fig. 35. Estructura del capítulo 2

Fuente: Propia

2.1. Fase I: Pre-juego

En esta fase Pre-juego llamado también Sprint 0, dio inicio al proyecto con los siguientes elementos: historias de usuarios, producto backlog, el diseño de la arquitectura tecnológica, la base de datos inicial y la configuración del entorno de desarrollo para su codificación.

2.1.1. Equipo Scrum

En la Tabla 12 define los actores encargados en el desarrollo del proyecto.

Tabla 12. Equipo Scrum

Rol	Nombre	Responsabilidad
Product Owner	Msc. Antonio Quiña	Encargado de verificar las funcionalidades del proyecto
Scrum Master	Srta. Zamia Guitarra	Encargada del seguimiento de las diferentes actividades del proyecto
Equipo de desarrollo	Srta. Zamia Guitarra	Encargada del desarrollo del proyecto

Fuente: Propia

2.1.2. Caso de Uso

Se definió los casos de uso en base a una prueba de concepto de un sistema financiero, por tanto, se estableció cuatro casos de uso con tres niveles de consulta, al que se refiere al número de relaciones que necesita para obtener los contenidos del mismo. A continuación, se enumera los distintos casos de uso:

Caso de uso 1: Insertar usuarios

- Nivel de consulta :1
- Campos: id, identification_card, name, lastname, pone, email, password, state

Caso de uso 2: Consulta usuarios

- Nivel de consulta :1
- Campos: id, identification_card, name, lastname, pone, email, password, state.

Caso de uso 3: Consulta usuarios por cuentas

- Nivel de consulta :2
- Campos: User*:* id, identification_card, name, lastname, pone, email, password, state, Account: id, account_number, account_balance, state, companies (name), typeaccount (name).

Caso de uso 4: Consulta usuarios por cuentas y movimientos

- Nivel de consulta :3
- Campos: User*:* id, identification_card, name, lastname, pone, email, password, state, Account: id, account_number, account_balance, state, companies (name), typeaccount (name), Movements: id, move_value, typemov (name).

2.1.3. Historias de usuarios

En las historias de usuario se detalla de manera general los requisitos de cada caso de uso definida en la sección 2.1.2. La adaptación de esta plantilla está estructurada en base al marco de trabajo Scrum.

HISTORIA DE USUARIO NRO. 1

Historia de Usuario			
ID:	**HU-MT-01**	**Usuario:** Desarrollador Backend	
Nombre: Insertar usuarios			
Prioridad: Alta		**Dependencia:** N/A	**Estimación:** 8
Descripción: Como desarrollador Backend quiero desarrollar un método crear usuarios en API REST y GraphQL, para poder generar varios registros de usuarios.			
Pruebas de aceptación: • Permitir crear usuarios en las arquitecturas REST y GraphQL • Validación de los campos, en caso de que exista errores mostrar un mensaje. • Generar varios registros aleatorios de usuarios 1, 100, 1000, 9000 datos y almacenarse en la BBDD. • Mostar el tiempo de ejecución			

HISTORIA DE USUARIO NRO. 2

Historia de Usuario			
ID:	**HU-MT-02**	**Usuario:** Desarrollador Backend	
Nombre: Consulta usuarios			
Prioridad: Alta		**Dependencia:** 1	**Estimación:** 5
Descripción: Como desarrollador Backend quiero realizar una consulta de la tabla usuarios en las arquitecturas REST y GraphQL, para poder visualizar usuarios registrados			
Pruebas de aceptación: • Consulta de una tabla en la arquitectura REST y GraphQL • Visualizar la consulta en las diferentes cantidades de registros 1, 100, 1000, 10000, 100000 • Mostar el tiempo de ejecución			

HISTORIA DE USUARIO NRO. 3

Historia de Usuario		
ID:	**HU-MT-03**	**Usuario:** Desarrollador Backend
Nombre: Consulta usuarios por cuentas		
Prioridad: Alta	**Dependencia:** 2	**Estimación:** 8
Descripción: Como desarrollador Backend quiero realizar una consulta de dos tablas (usuario - cuenta) en REST y GraphQL para poder visualizar sus datos		
Pruebas de aceptación: • Consulta de dos tablas en la arquitectura REST y GraphQL • Visualizar la consulta en las diferentes cantidades de registros 1, 100, 1000, 10000, 100000 • Mostar el tiempo de ejecución		

HISTORIA DE USUARIO NRO. 4

Historia de Usuario		
ID:	**HU-MT-04**	**Usuario:** Desarrollador Backend
Nombre: Consulta de usuarios por cuentas y movimientos		
Prioridad: Alta	**Dependencia:** 3	**Estimación:** 8
Descripción: Como desarrollador Backend quiero realizar una consulta de tres tablas (usuario – cuenta- movimiento) en REST y GraphQL para poder visualizar sus datos		
Pruebas de aceptación: • Consulta de tres tablas en la arquitectura REST y GraphQL • Visualizar la consulta en las diferentes cantidades de registros 1, 100, 1000, 10000, 100000 • Mostar el tiempo de ejecución		

2.1.4. Product Backlog

En la Tabla 13 se detalla las tareas a realizar en base a las historias de usuarios, mediante los parámetros: descripción, nivel de consulta y estimación en horas.

Tabla 13. Product Backlog

Orden	ID	Descripción	Nivel de consulta	Estimación (horas)
1	HU-MT-01	Insertar usuarios	1	20
2	HU-MT-02	Consulta usuarios	1	20
3	HU-MT-03	Consulta usuarios por cuentas	2	20
4	HU-MT-04	Consulta usuarios por cuentas y movimientos	3	20

Fuente: Propia

2.1.5. Arquitectura del proyecto

La arquitectura de la aplicación Backend, consta de una base de datos PostgreSQL integrada a las dos arquitecturas API GraphQL y REST. Los servicios se desarrollaron en el framework NestJS, debido a que posee una arquitectura y estructura basada en módulos. Estos servicios son consumidos tanto por el componente controller como por resolver como muestra en la Fig. 36. Las respuestas ante las solicitudes de consulta se obtuvieron en el formato JSON.

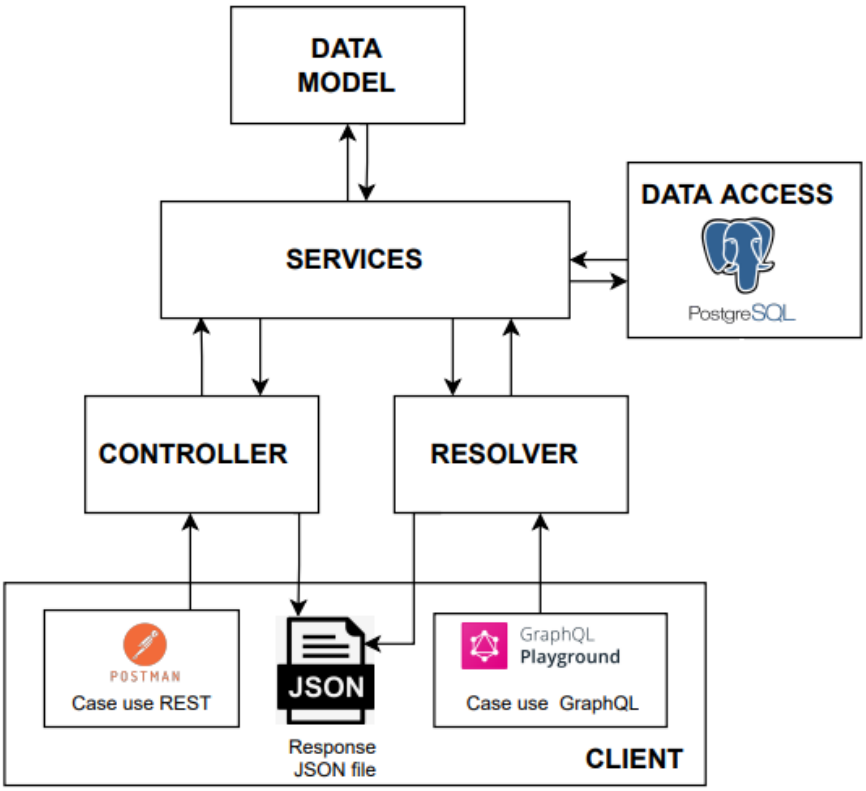


Fig. 36. Arquitectura del proyecto

Fuente: Propia

2.1.6. Diseño de la Base de Datos

En la Fig. 37, muestra el diseño de la base de datos de acuerdo con los requerimientos de una prueba de concepto de un sistema bancario, del módulo transferencias bancarias. Se definió ocho entidades relacionadas con sus respectivos atributos. Para la generación del diseño y visualización de datos se utilizó la herramienta DBeaver.

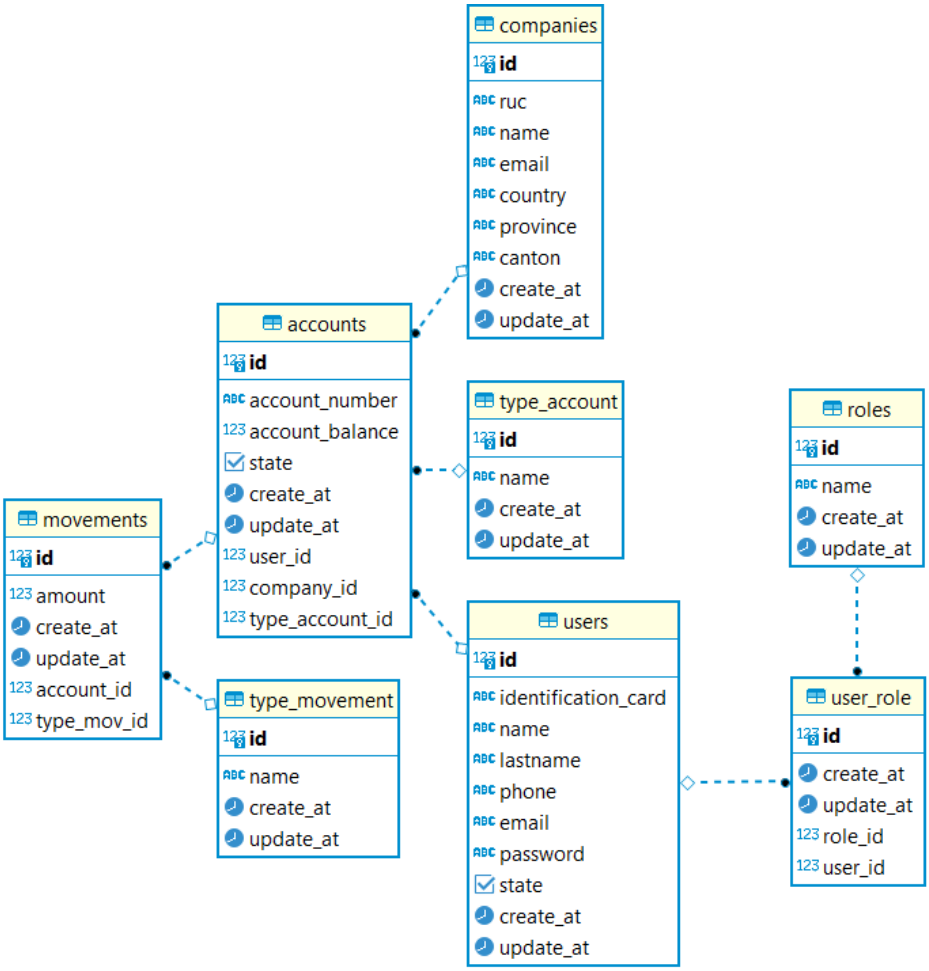


Fig. 37. Diseño de la base de datos

Fuente: Propia

2.2. Fase II: Juego

En esta sección del capítulo se lleva a cabo el desarrollo de la aplicación en base a la planificación de cada sprint. Además, se define el sprint Backlog, la cual establece un conjunto de tareas que debe ejecutar cada iteración. Para comprobar el avance del proyecto se realizar la verificación de acuerdo con los mecanismos de revisiones e incremento que detalla Scrum.

2.2.1. Planificación

Se realiza la planificación de cada sprint con una duración de dos semanas. En la Tabla 14, muestra las fechas del inicio, fin y el tiempo (duración) que se utilizó para finalizar la actividad. El sprint 0 se desarrolló en la sección 2.1 de este capítulo.

Tabla 14. Planificación de cada iteración del proyecto

Nro. Sprint	Fecha de inicio	Fecha fin	Duración (Horas)
Sprint 0	04/07/2022	15/07/2022	30
Sprint 1	18/07/2022	29/07/2022	30
Sprint 2	01/08/2022	12/09/2022	30
Sprint 3	15/08/2022	26/08/2022	30

Fuente: Propia

2.2.2. Sprint 1

En el primer sprint se realizó la funcionalidad de insertar y consultar registros de la tabla usuarios mediante la arquitectura REST y GraphQL con una complejidad de un nivel uno, debido a que emplea una tabla En este sprint se desarrolló el siguiente Sprint Backlog que se detalla en la Tabla 16, mediante la reunión de planificación realizada en la Tabla 15.

a) Reunión de planificación

Tabla 15. Reunión de planificación - Sprint 1

Sprint 1	
Fecha de la reunión:	18/07/2022
Asistentes:	Scrum Master, Product Owner y Team
Fecha de inicio del sprint	18/07/2022
Fecha fin del sprint:	29/07/2022
Objetivos del sprint: • Implementar un servicio que realice la generación de usuarios de manera aleatoria • Implementar un servicio que permita realizar la consulta de usuarios en diferentes tamaños. • Ejecutar los servicios insertar y consultar usuarios en el Entorno Localhost desarrollado en REST y GraphQL	

Fuente: Propia

- **Sprint Backlog - Sprint 1**

Tabla 16. Spring Backlog - Sprint 1

Nro. HU	Nombre	Tarea	Horas
HU-MT-01	Insertar usuarios	Construir un servicio que permita insertar usuarios	4
		Añadir un tiempo de ejecución al servicio.	1
		Crear el componente controlador y añadir el método para consumir el servicio	3
		Crear el componente resolutores y añadir el método para consumir el servicio	3
		Ejecución del servicio insertar usuario en REST y GraphQL.	2
HU-MT-02	Consulta usuarios	Construir un servicio que permita la consulta de usuarios	2
		Añadir un tiempo de ejecución al servicio	1
		Añadir un método aplicando el decorador @GET en el controlador para consumir el servicio.	2
		Añadir un método aplicando el decorador @Query en resolutores para consumir el servicio.	2
		Ejecución del servicio consultar usuario en REST y GraphQL.	2

Fuente: Propia

b) Reunión de revisión

Una vez concluido el Sprint 1, se realizó la reunión de revisión con los asistentes Scrum Master, Product Owner y el Equipo de Desarrollo. Como resultado se obtuvo la comprobación del funcionamiento de las historias de usuario de acuerdo con el incremento del producto. En la Tabla 17 muestra las tareas realizadas conforme al tiempo estimado de horas con las reales.

Tabla 17. Reunión de revisión – Sprint 1

Responsable	Tarea	Tiempo estimado (horas)	Tiempo real (horas)	Estado
Desarrollador	Construir un servicio que permita insertar usuarios	4	4	Realizado
	Añadir un tiempo de ejecución al servicio.	1	1	Realizado
	Crear el componente controlador y añadir el método para consumir el servicio	3	2	Realizado
	Crear el componente resolutores y añadir el método para consumir el servicio	3	2	Realizado
	Ejecución del servicio insertar usuario en REST y GraphQL.	2	2	Realizado
	Construir un servicio que permita la consulta de usuarios	2	2	Realizado
	Añadir un tiempo de ejecución al servicio	1	1	Realizado
	Añadir un método aplicando el decorador @GET en el controlador para consumir el servicio.	2	2	Realizado
	Añadir un método aplicando el decorador @Query en resolutores para consumir el servicio.	2	2	Realizado
	Ejecución del servicio consultar usuario en REST y GraphQL.	2	1	Realizado
Reuniones Scrum	Planificación	1	1	Realizado
	Revisión	1	1	Realizado
	Retrospectiva	1	1	Realizado
	Total	25	22	

Fuente: Propia

- **Incremento**

En esta iteración se obtuvo como resultado la implementación de los servicios insertar y consultar usuarios en las arquitecturas Rest y GraphQL. Los servicios fueron desplegados en el entorno Localhost.

Funcionalidad API – REST

Se realiza la prueba de funcionalidad de los servicios insertar y consultar usuario en Rest, mediante el cliente Postman. Ver Fig. 38 y Fig. 39

- Generar usuarios.

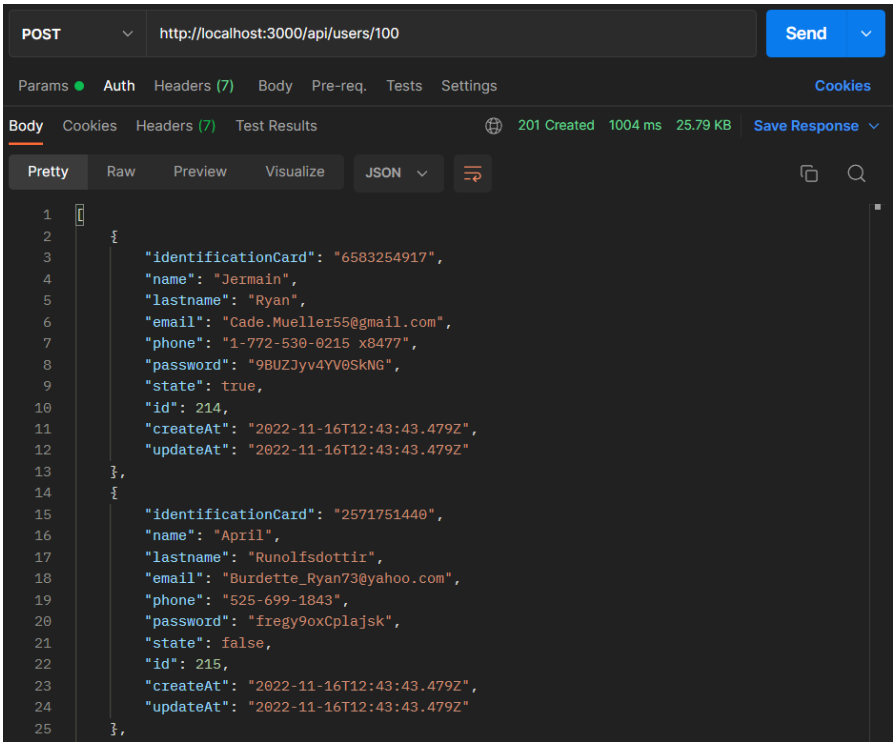


Fig. 38. Generar usuarios API - REST

Fuente: Propia

- Consultar usuarios

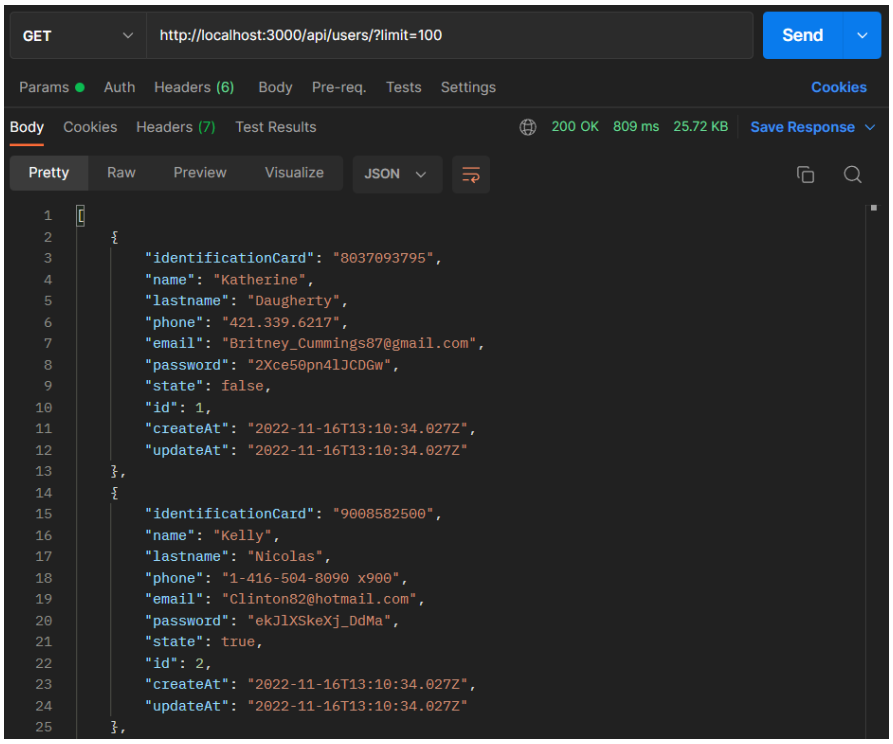


Fig. 39. Consulta usuario -API – Rest

Fuente: Propia

Funcionalidad API GraphQL

De igual manera, se realiza las pruebas de funcionalidad de los servicios insertar y consultar en GraphQL, mediante el cliente GraphQL playground. Ver Fig. 40 y Fig. 41

- Generar usuarios

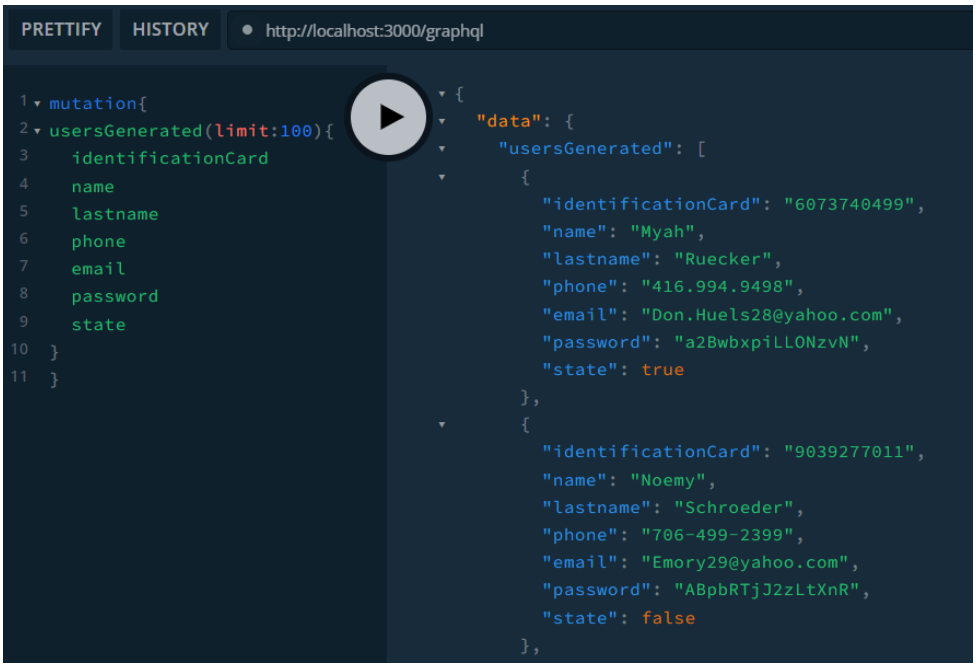


Fig. 40. Generar usuario - API GraphQL

Fuente: Propia

- Consultar usuarios

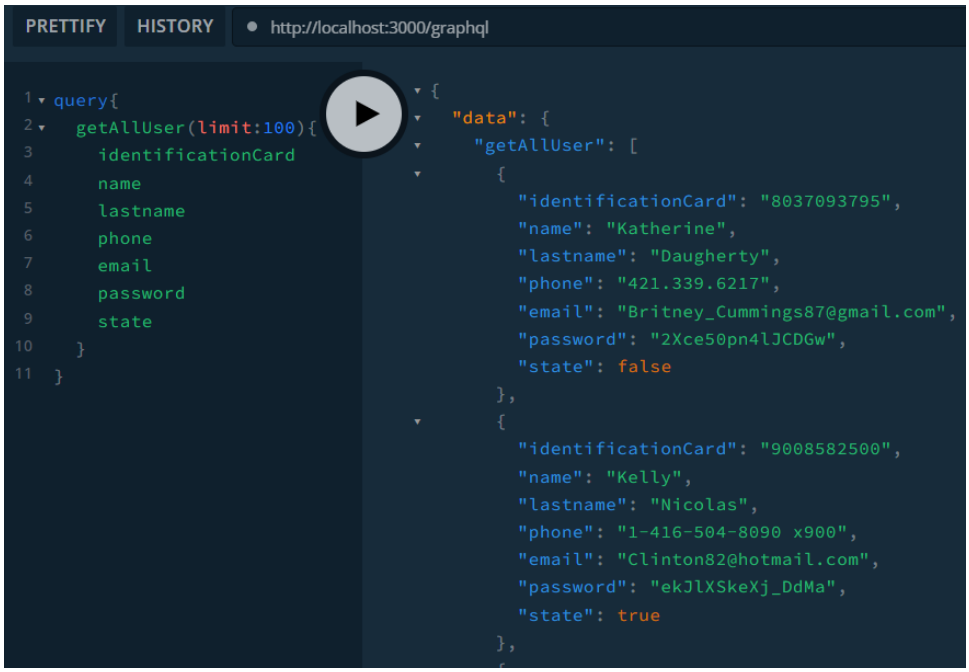


Fig. 41. Consultar usuario - API GraphQL

Fuente: Propia

2.2.3. Sprint 2

En el siguiente sprint se añade la funcionalidad de consulta usuarios por cuenta mediante las arquitecturas REST y GraphQL. Presenta una complejidad de consulta de nivel dos debido a que se emplea dos tablas relacionadas. La Tabla 19 detalla el Spring Backlog que se desarrolló en este sprint, de acuerdo con los objetivos planteados en la Tabla 18.

a) Reunión de planificación

Tabla 18. Reunión de planificación - Sprint 2

Sprint 2	
Fecha de la reunión:	01/08/2022
Asistentes:	Scrum Master, Product Owner y Team
Fecha de inicio del sprint:	01/08/2022
Fecha fin del sprint:	12/09/2022
Objetivos del sprint: • Implementar la funcionalidad de consulta usuarios por cuenta • Ejecutar el servicio consulta de usuarios por cuenta en el Entorno Localhost desarrollado en la arquitectura REST y GraphQL	

Fuente: Propia

- **Sprint Backlog - Sprint 2**

Tabla 19. Sprint Backlog - Sprint 2

Nro. HU	Nombre	Tarea	Horas
HU-MT-03	Consulta usuarios por cuenta	Construir un servicio que permita la consulta de usuarios por cuenta	2
		Añadir un tiempo de ejecución al servicio	1
		Añadir un método aplicando el decorador @GET en el controlador para consumir el servicio.	2
		Añadir un método aplicando el decorador @Query en resolutores para consumir el servicio.	2
		Ejecución del servicio consultar usuarios por cuenta en REST y GraphQL.	2

Fuente: Propia

b) Reunión de revisión

Tabla 20. Reunión de revisión – Sprint 2

Responsable	Tarea	Tiempo estimado (horas)	Tiempo real (horas)	Estado
Desarrollador	Construir un servicio que permita la consulta de usuarios por cuenta	2	2	Realizado
	Añadir un tiempo de ejecución al servicio	1	1	Realizado
	Añadir un método aplicando el decorador @GET en el controlador para consumir el servicio.	2	1	Realizado

	Añadir un método aplicando el decorador @Query en resolutores para consumir el servicio.	2	1	Realizado
	Ejecución del servicio consultar usuarios por cuenta en REST y GraphQL.	2	2	Realizado
Reuniones Scrum	Planificación	1	1	Realizado
	Revisión	1	1	Realizado
	Retrospectiva	1	1	Realizado
	Total	12	10	

Fuente: Propia

- **Incremento**

En este sprint se obtuvo como resultado la implementación del servicio consulta de usuario por cuenta en las diferentes arquitecturas REST y GraphQL

Funcionalidad API - REST

Se realiza las pruebas de funcionalidad del servicio en REST. Ver Fig. 42.

- Consultar usuarios por cuenta

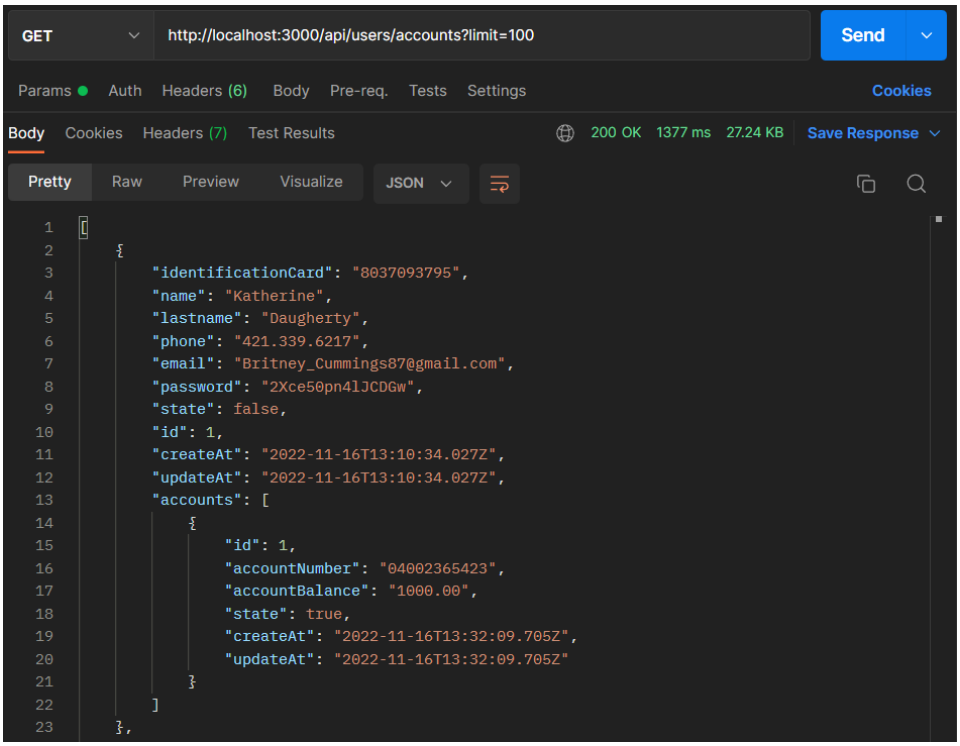


Fig. 42. Consulta de dos tablas - API REST

Fuente: Propia

Funcionalidad API - GraphQL

Consultar la funcionalidad del servicio usuario por cuenta desarrollado en la arquitectura GraphQL. Ver Fig. 43.

- Consultar usuarios por cuenta

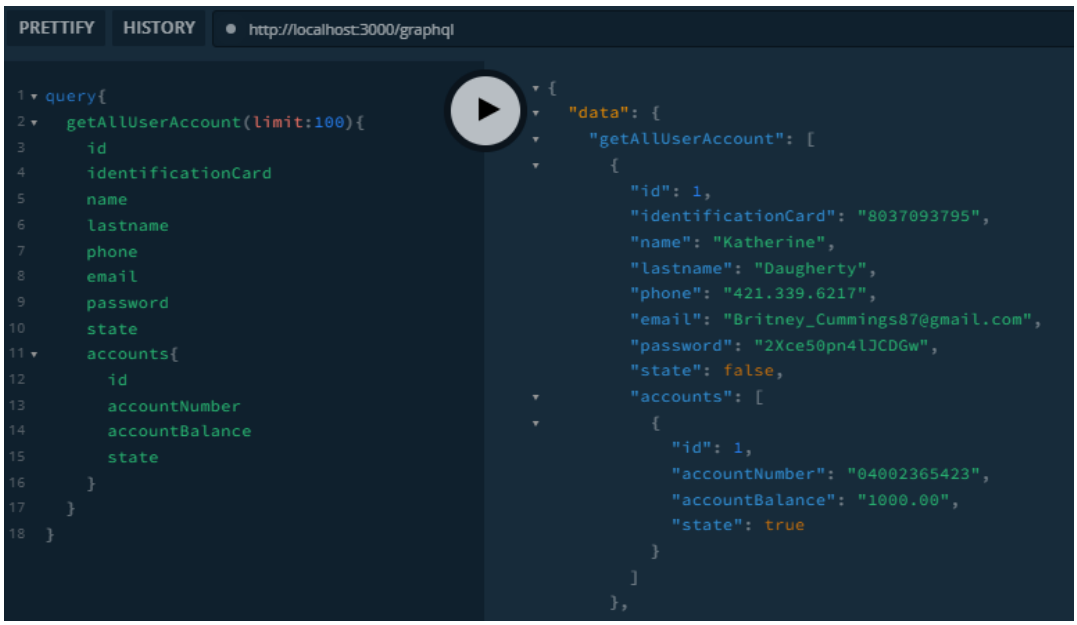


Fig. 43. Consulta de dos tablas - API GraphQL

Fuente: Propia

2.2.4. Sprint 3

En esta última iteración se implementó la funcionalidad de la consulta de tres tablas en la arquitectura GraphQL y REST. Nivel de consulta 3. En la Tabla 21 y Tabla 22 muestra el desarrollo que se realizó en esta iteración.

a) Reunión de la planificación

Tabla 21. Reunión de planificación - Sprint 3

Sprint 3	
Fecha de la reunión:	15/08/2022
Asistentes:	Scrum Master, Product Owner y Team
Fecha de inicio del sprint	15/08/2022
Fecha fin del sprint:	26/08/2022

Objetivos del sprint:

- Implementar la funcionalidad de consulta usuarios por cuenta y por movimiento
- Ejecutar el servicio consulta de usuarios por cuenta y por movimiento en el Entorno Localhost desarrollado en la arquitectura REST y GraphQL

Fuente: Propia

- **Sprint Backlog - Sprint 3**

Tabla 22. Sprint Backlog - Sprint 3

Nro. HU	**Nombre**	**Tarea**	**Horas**
HU-MT-04	Consulta usuarios por cuentas y por movimiento	Construir un servicio que permita la consulta de tres tablas	2
		Añadir un tiempo de ejecución al servicio	1
		Añadir un método aplicando el decorador @GET en el controlador para consumir el servicio.	2
		Añadir un método aplicando el decorador @Query en resolutores para consumir el servicio.	2
		Ejecución del servicio de consulta de tres tablas en REST y GraphQL.	2

Fuente: Propia

b) Reunión de revisión

Tabla 23. Reunión de revisión – Sprint 3

Responsable	Tarea	Tiempo estimado (horas)	Tiempo real (horas)	Estado
Desarrollador	Construir un servicio que permita la consulta de tres tablas	2	2	Realizado
	Añadir un tiempo de ejecución al servicio	1	1	Realizado
	Añadir un método aplicando el decorador @GET en el controlador para consumir el servicio.	2	1	Realizado
	Añadir un método aplicando el decorador @Query en resolutores para consumir el servicio.	2	1	Realizado
	Ejecución del servicio de consulta de tres tablas en REST y GraphQL.	2	2	Realizado
Reuniones Scrum	Planificación	1	1	Realizado
	Revisión	1	1	Realizado
	Retrospectiva	1	1	Realizado
	Total	12	10	

Fuente: Propia

- **Incremento**

En esta iteración se implementó de manera exitosa el servicio de consulta de tres tablas en la arquitectura REST y GraphQL.

Funcionalidad API - REST

Prueba de funcionalidad del servicio en la arquitectura API REST. Ver Fig. 44.

- Consulta usuarios por cuenta y por movimiento

Fig. 44. Consulta de tres tablas - API Rest

Fuente: Propia

Funcionalidad API – GraphQL

Prueba de funcionalidad del servicio usuario por cuenta y por movimiento en la arquitectura GraphQL. Ver Fig. 45.

- Consultar usuarios por cuenta y por movimiento

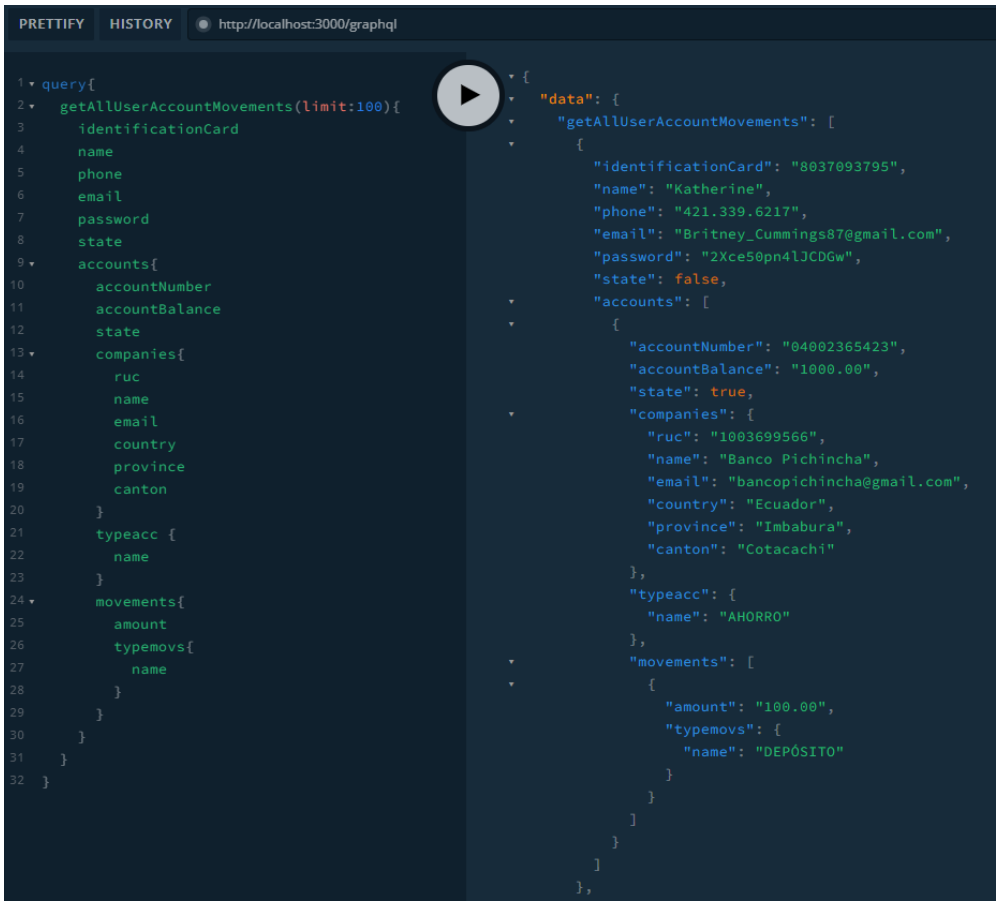


Fig. 45. Consulta de tres tablas - API GraphQL

Fuente: Propia

Una de las ventajas de GraphQL es que todas las consultas pueden se manejar por un solo endpoint a diferencia de Rest, que requiere de varios endpoint para acceder a las mismas consultas. A continuación, se detalla los diferentes endpoints utilizadas en la ejecución de la aplicación en las arquitecturas REST y GraphQL. Ver Tabla 24 y Tabla 25.

Arquitectura REST

Tabla 24. Endpoints API - REST

Endpoint	Tipo de petición	Descripción
/api/users	POST	Generar usuarios
/api/users/	GET	Consultar usuarios
/api/users/accounts	GET	Consultar usuarios por cuenta
/api/users/accounts/movements	GET	Consultar usuarios por cuenta y movimiento

Fuente: Propia

Arquitectura GraphQL

Tabla 25. Endpoints API - GRAPHQL

Endpoint	Tipo de petición	Descripción
/graphql	Mutation	Generar usuarios
	Query	Consulta de usuarios
		Consulta de usuarios por cuenta
		Consulta de usuarios por cuenta, por movimientos
		Consulta de usuarios por cuenta, por movimientos y por auditoria

Fuente: Propia

2.3. Fase 3: Post-juego

Una vez concluido con el desarrollo y el testeo de todos los sprint planificadas en la fase anterior. Se obtiene una aplicación correctamente funcional, lista para ser desplegada. Por ende, a lo largo de esta sección se define los pasos a realizar para el despliegue de la aplicación en contenedores Docker. El código de la aplicación se encuentra alojado en el repositorio Bitbucket. Ver Fig. 46.

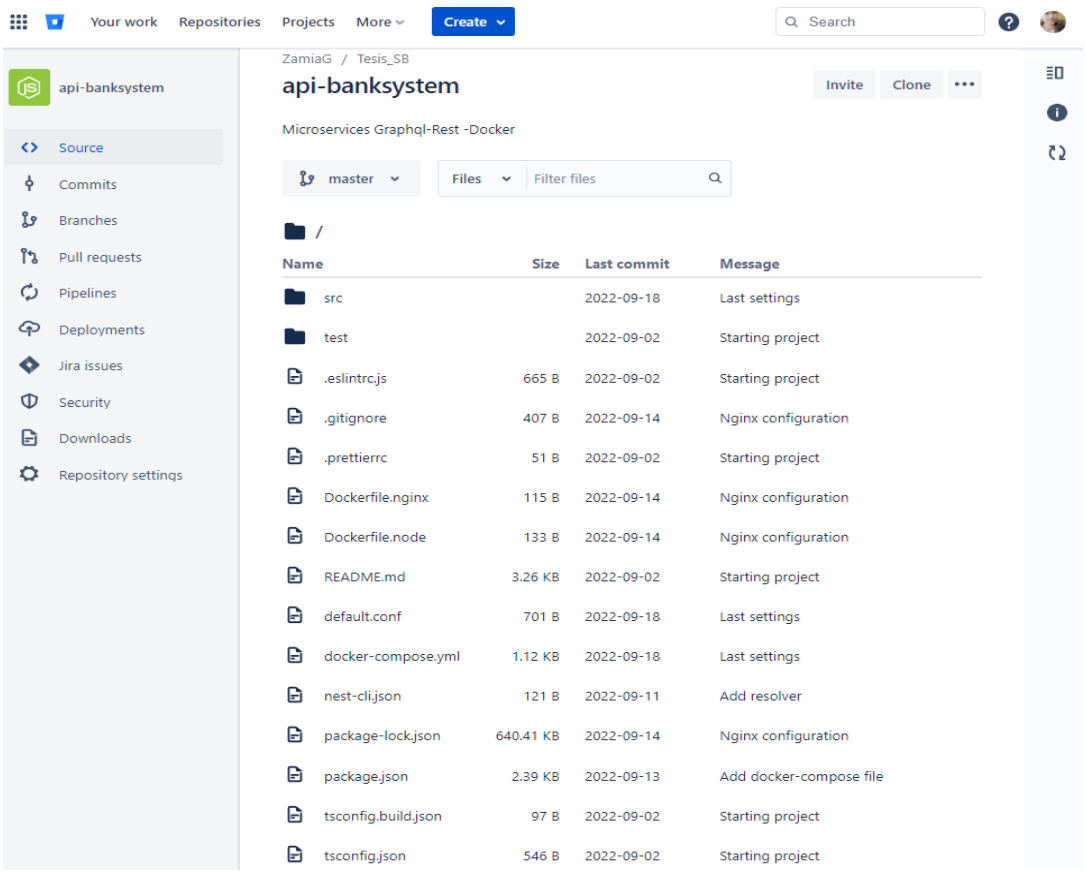


Fig. 46. Repositorio de código Bitbucket

Fuente: Propia

2.3.1. Contenerización en Docker

Primeramente, se prepara el entorno en el que se desplegará la aplicación. Por lo tanto, se instaló Docker y Docker Compose en la máquina. En este caso la instalación se realizó en el sistema operativo Linux de la distribución Ubuntu 22.04 siguiendo las instrucciones de la documentación de Docker. Para verificar su instalación se ejecuta el siguiente comando, como muestra la Fig. 47 la cual indica la versión instalada de Docker. Caso contrario si existe un error en la instalación, el comando no es reconocido por el sistema operativo.

```
zguitarra@ubuntu20:~$ docker --version
Docker version 20.10.17, build 100c701
```

Fig. 47. Versionamiento de Docker

Fuente: Propia

Al comprobar que Docker se encuentra ejecutándose correctamente, se procede a crear los archivos de configuración necesarios para que la aplicación funcione. A continuación, se describe detalladamente cada uno de ellos.

Dockerfile

Es un documento de texto que contiene instrucciones necesarias para crear imágenes automáticamente. Es necesario llamarlo con el mismo nombre *Dockerfile,* para que el cliente de Docker sea capaz de reconocer el archivo y crear la imagen (Roldán Martínez et al., 2018). En la Fig. 48 muestra el contenido de las instrucciones del archivo Dockerfile que se utilizó para crear imágenes y contenedores de la aplicación.

Dockerfile.node > FROM

```
1   FROM node:16
2
3
4   WORKDIR /usr/src/app
5
6   COPY package*.json ./
7
8   RUN npm install
9
10
11  COPY . .
12
13  RUN npm run build
14
15  CMD [ "node", "dist/main"]
```

Fig. 48. Archivo Dockerfile

Fuente: Propia

Para la ejecución del proyecto se creó dos archivos Dockerfiles, la primera para la aplicación y la segunda para el servidor Nginx. En la Tabla 26, se define la función de cada línea de comando.

Tabla 26. Instrucciones del archivo Dockerfile

INSTRUCCIONES DOCKERFILE	
FROM	La siguiente instrucción permite la descarga una imagen ya oficial que se encuentra disponible en Docker Hub, repositorio de Docker. En este caso se requiere Node de la versión: 16, que se utilizó en el desarrollo.
WORKDIR	Esta instrucción crear el directorio donde se va a trabajar y almacenar posteriores archivos de la aplicación, en este caso indica en el directorio /usr/src/app.
COPY	Copia el archivo package*.json al directorio del trabajo
RUN	Instala todas las dependencias de la aplicación por medio del comando npm install, y posterior el comando npm run build para crear una carpeta dist con la compilación de la aplicación.
CMD	Inicializa el servidor con el comando node dist/main, utilizando la compilación de la carpeta dist, debido a que se requiere realizar la ejecución en el entorno de producción.

Fuente: (Guijarro et al., 2019)

Docker Compose

Una vez implementada las imágenes, contendores de la aplicación y del servidor Ngnix, se realizó la configuración de un contenedor para la base de datos, mediante la imagen postgres obtenida directamente del repositorio Docker Hub. Para obtener la comunicación entre estos tres contenedores se utilizó la herramienta docker-compose, la cual debe encontrarse instalado y creado con el nombre *docker-compose.yml*. En algunos casos es conveniente instalar la última versión para no tener inconvenientes al momento de ejecutar el archivo. Por medio del archivo Docker Compose se logró la integración de los contenedores, volúmenes, puertos, redes en un solo archivo como se muestra en la Fig. 49. Además, Docker Compose permite con solo un comando activar o desactivar toda una aplicación (Roldán Martínez et al., 2018).

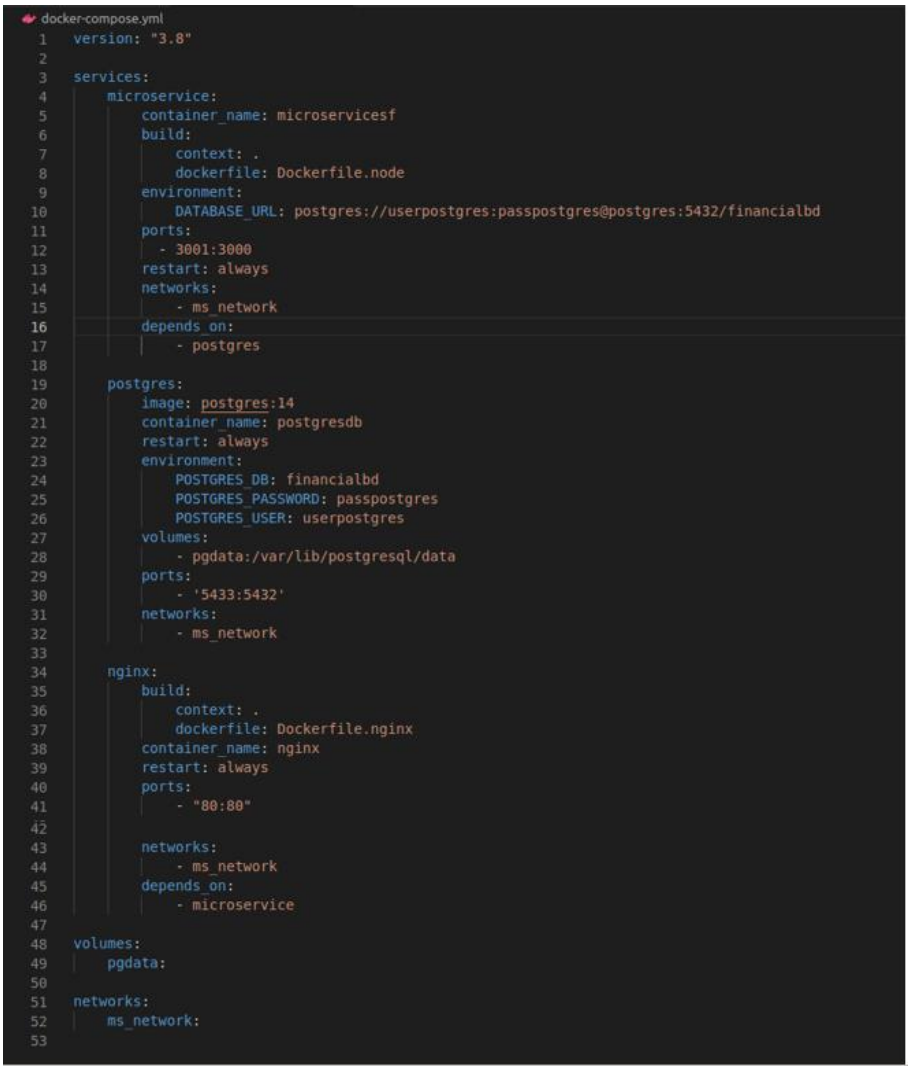


Fig. 49. Archivo Docker Compose

Fuente: Propia

En la Tabla 27 se detalla cada uno de las configuraciones e instrucciones del archivo docker-compose.yml. Además, se debe tomar en cuenta la sangría para este tipo de formato.

Tabla 27. Instrucciones del archivo Docker Compose

Instrucciones Docker-compose	
versión	Muestra la versión que va a ser utilizada. Todos los archivos de este tipo deben encontrarse versionados.

services: Indica los servicios que va a utilizar la aplicación, para este caso se aplica tres servicios: microservice, postgres y nginx, Desde la línea 4 hasta la 17 se detalla la configuración para el primer contenedor de microservicio.

container_name	Nombre del contenedor, al que se refiere el servicio.

build Se emplea para indicar donde está ubicado el Dockerfile que se necesita para crear el contenedor. Al colocar "." considera automáticamente que Dockerfile se encuentra en el actual directorio.

environment En este apartado establece las variables de entornos necesarios para la ejecución de los contenedores, como el usuario, contraseña, nombre de la base de datos.

ports Permite el mapeo de los puertos para que la aplicación sea accesible y se pueda consumir los servicios del contenedor, se asigna el puerto 3001 al contenedor.

restart La palabra *"always",* se refiere a que si existe algún error la aplicación se reinicie

networks Permite la comunicación entre contenedores.

depends_on Indica la dependencia entre servicios, es decir la ejecución se realiza siempre y cuando los demás servicios estén listos. En este caso se ejecutará el contenedor cuando ya se encuentre disponible el servicio postgres.

Desde la línea 19 hasta la 32 se detalla la configuración para el segundo contenedor postgres referente a la base de datos

image Esta instrucción permite la descarga de la imagen postgres, versión 14 la cual fue utilizada para su desarrollo. Esta imagen fue obtenida del repositorio Docker Hub.

ports Se le asignó el puerto 5433 debido a que se encuentra ocupado, pero normalmente porstgres ocupa el puerto 5432.

volumes Permite la persistencia de datos, es decir, si un contenedor se elimina y se lo vuelve a crear, la información de la base de datos no se pierde.

De la línea 34 hasta la 46 se detalla la configuración para el tercer contenedor del servidor Ngnix, la cual sigue la misma configuración del primer contenedor.

(Roldán Martínez et al., 2018)

2.3.2. Despliegue de la aplicación

Para verificar si se realizó correctamente las configuraciones en el archivo Docker Compose, se ejecuta el siguiente comando "*docker-compose up - -build -d*" como muestra en la Fig. 50. La instrucción *"-d"* permite ejecutar la aplicación en segundo plano, es decir si se cierra el terminal o el entorno de ejecución la aplicación seguirá funcionando.

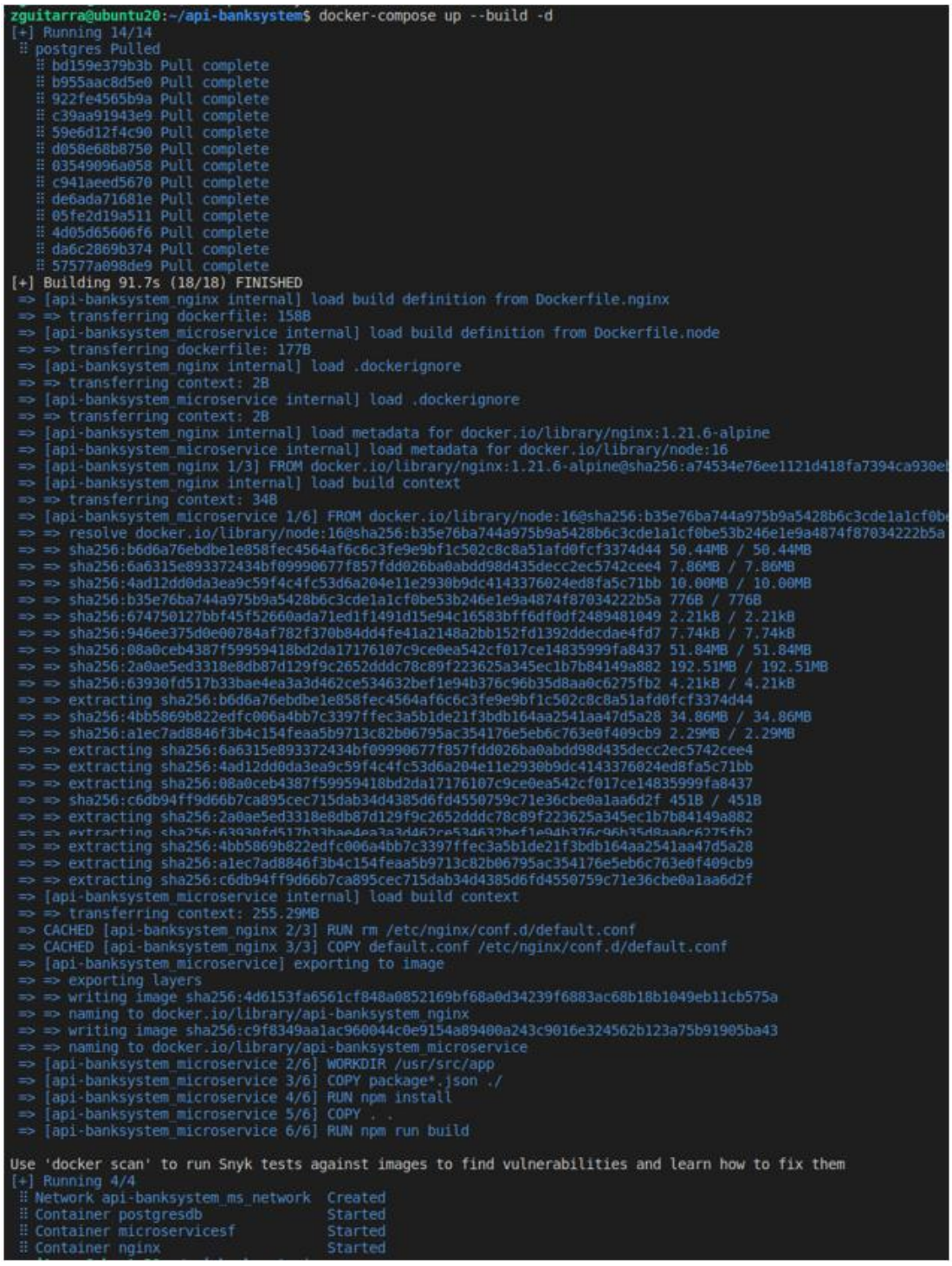


Fig. 50. Ejecución Docker Compose

Fuente: Propia

Con un solo comando se desplegó en cuestión de segundos toda la aplicación, sin necesidad de instalar ninguna herramienta de Node, Postgres o Nginx en la máquina. En la Fig. 51 muestra la ejecución de los tres contenedores creados: postgresdb, nginx y microservicessf. De esta manera se obtiene el despliegue y la integración de la aplicación Backend dentro de contenedores Docker.

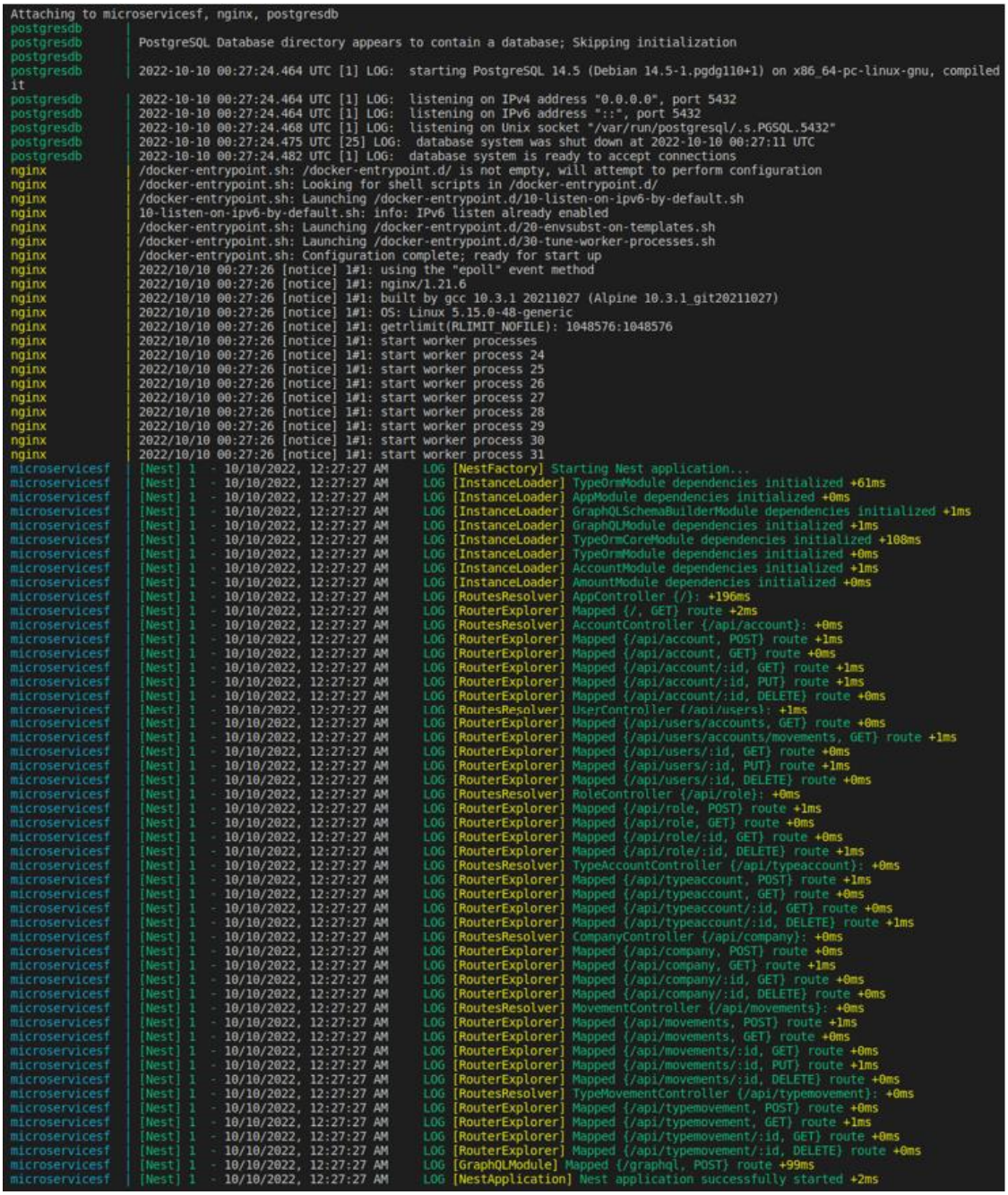


Fig. 51. Muestra de los contenedores creados

Fuente: Propia

Una vez realizada la ejecución de los contenedores se verifica el correcto funcionamiento de la aplicación. Como se observa en las Fig. 52 y Fig. 53

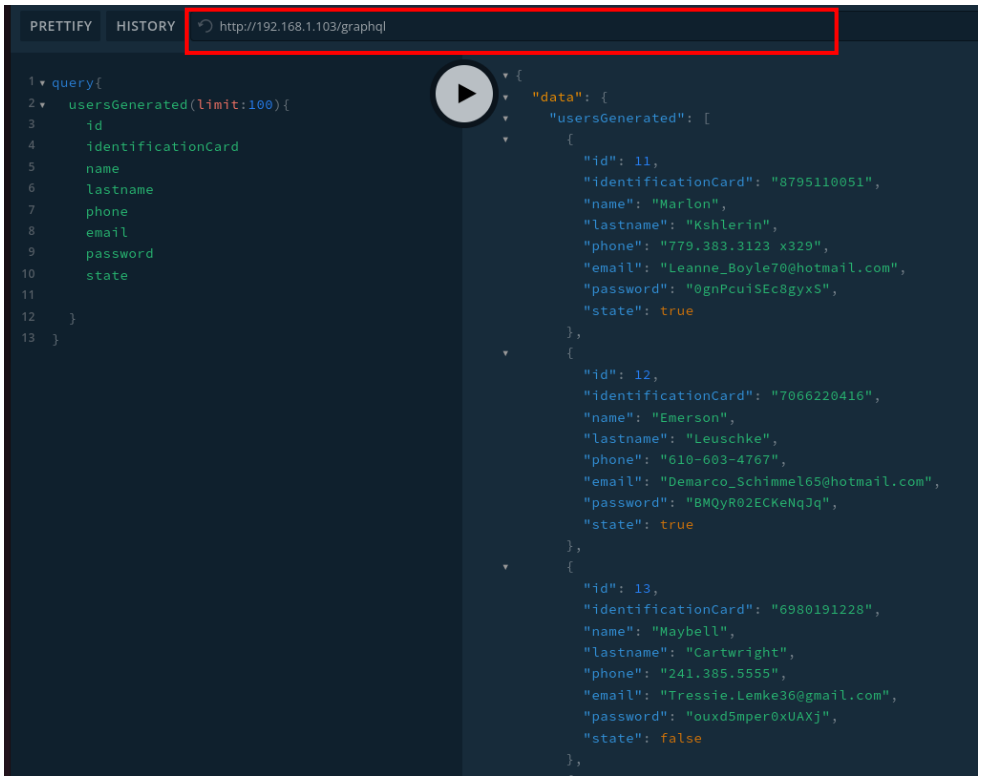


Fig. 52. Funcionamiento de API GraphQL en el contenedor Docker

Fuente: Propia

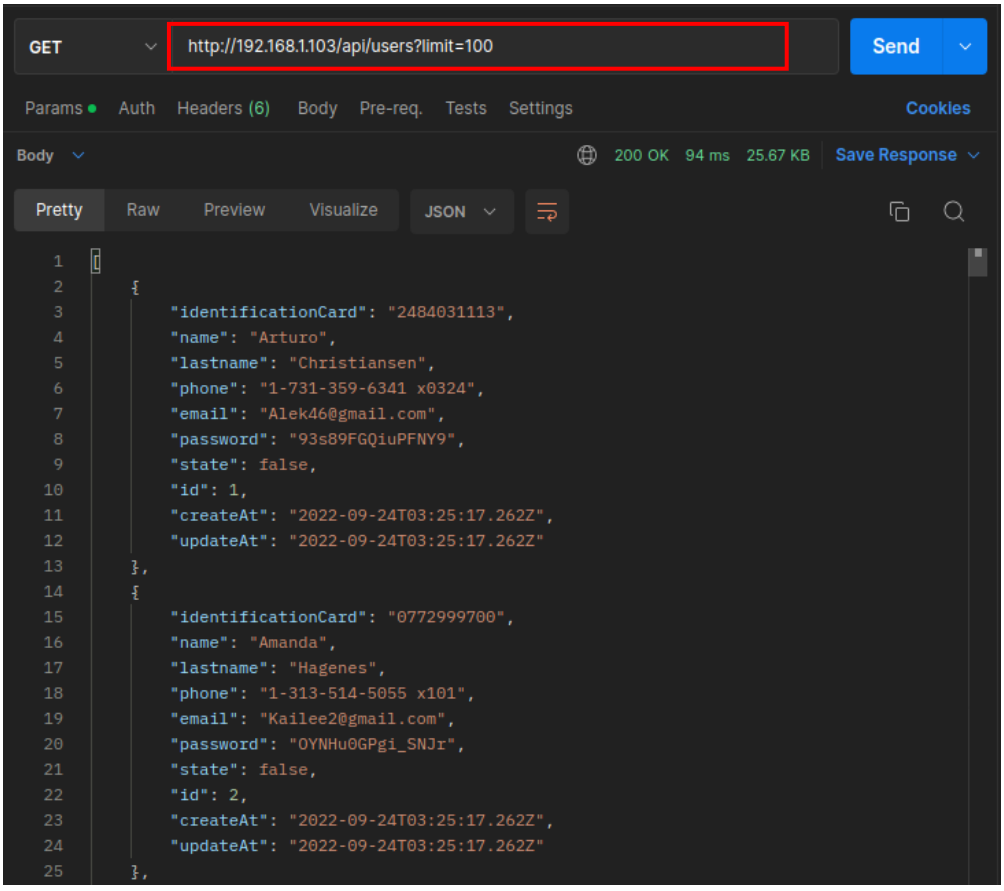


Fig. 53. Funcionamiento de API REST en el contenedor

Fuente: Propia

2.3.3. Pruebas de aceptación

Se realizó las pruebas de aceptación mediante el despliegue de la aplicación en el Entorno Localhost como en contenedores Docker, que se detalla en la Tabla 28 y Tabla 29.

Entorno Localhost (REST – GraphQL)

A través del empaquetamiento de los archivos en una carpeta dist, se ejecutó la aplicación en un ambiente de producción, debido a que se quiere obtener los tiempos de ejecución más precisos en el comportamiento de los casos de uso. De la misma manera se empleó el servidor Nginx como proxy inverso para el redireccionamiento de las APIs. Para el funcionamiento de la aplicación se instaló las herramientas: Node, Ngnix y Postgres en la maquina local.

Tabla 28. Pruebas de aceptación Localhost

Historia de Usuario	Nombre	Funcionalidad	Aceptación	
			SI	NO
HU-MT-01	Insertar Usuario	Generar usuarios aleatorios en REST y GraphQL	**X**	
HU-MT-02	Consulta usuarios	Consultar usuarios en REST y GraphQL	**X**	
HU-MT-03	Consulta usuarios por cuentas	Consultar usuarios por cuenta en REST y GraphQL	**X**	
HU-MT-04	Consulta usuarios por cuentas y movimientos	Consultar usuarios por cuentas y movimientos en REST y GraphQL	**X**	
HU-MT-05	Consulta usuarios por cuentas, por movimientos y por auditoria	Consultar usuarios por cuentas, por movimientos y por auditoria en REST y GraphQL	**X**	

Fuente: Propia

Docker (REST – GraphQL)

La aplicación fue desplegada mediante contenedores Docker, por lo que no fue necesario instalar alguna herramienta para su funcionamiento, En la sección 2.3.1 se detalla el proceso realizado.

Tabla 29. Pruebas de aceptación Docker

Historia de Usuario	Nombre	Funcionalidad	Aceptación	
			SI	NO
HU-MT-01	Insertar usuarios	Generar usuarios aleatorios en REST y GraphQL	**X**	
HU-MT-02	Consulta usuarios	Consultar usuarios en REST y GraphQL	**X**	
HU-MT-03	Consulta usuarios por cuentas	Consultar usuarios por cuenta en REST y GraphQL	**X**	
HU-MT-04	Consulta usuarios por cuentas y movimientos	Consultar usuarios por cuentas y movimientos en REST y GraphQL	**X**	
HU-MT-05	Consulta usuarios por cuentas, por movimientos y por auditoria	Consultar usuarios por cuentas, por movimientos y por auditoria en REST y GraphQL	**X**	

Fuente: Propia

CAPÍTULO 3

Validación de Resultado

Introducción

En este capítulo se realizó la validación de la investigación, mediante un experimento controlado basado en la guía de Wholin, (2012) "*Experimentación en Ingeniería de Software"*. Esta experimentación consistió en comparar la eficiencia de las arquitecturas, mediante los casos de uso presentados en el capítulo anterior. Por lo tanto, se planteó la siguiente pregunta de investigación $\boldsymbol{PI}$: ¿En qué condiciones es más adecuado utilizar un entorno virtualizado (Docker) y con qué arquitecturas, para el despliegue de microservicios? Para medir estos efectos se utilizó las normas ISO/IEC 25010 con respecto la característica *Eficiencia de rendimiento,* de la calidad de un producto del software.

3.1. Entorno Experimental

3.1.1. Objetivo del Experimento

Comparar la eficiencia de desempeño de las APIs GraphQL y REST en contenedores Docker con el Entorno Localhost, respecto a la calidad del producto del software.

3.1.2. Factores y tratamientos

El factor a investigar es la arquitectura de software, específicamente en el despliegue de los servicios APIs de los servicios. Los tratamientos que se emplearon fueron:

- Arquitectura Docker para el despliegue de servicios APIs.
- Entorno Localhost para el despliegue de servicios APIs.

En cada una de las arquitecturas se desplegaron las APIs GraphQL y REST.

3.1.3. Variable

Se definió como *variable independiente* a la "Arquitectura de software para el despliegue de servicios APIs, la Arquitectura Docker y el Entorno Localhost, la cual se encuentra detallada en la Tabla 30. De igual manera, se determinó como *variable dependiente* a la "Calidad del producto de software" en función a la característica "Eficiencia de Rendimiento".

Tabla 30. Variables para la experimentación

Variable Independiente	Arquitectura Docker para el despliegue de servicios APIs
	Entorno Localhost para el despliegue de servicios APIs.
Variable Dependiente	Calidad del producto del software" en función a la característica "Eficiencia de Rendimiento

Fuente: Propia

A continuación, se detalla la métrica referida a la característica "Eficiencia de Rendimiento" de la norma ISO/IEC 25023.

Tiempo medio de respuesta

Hace referencia al tiempo en el que se demora una petición al completar, es decir el tiempo medio que emplea para completar un proceso asíncrono. La función de medición al aplicar fue:

$$X = \sum_{i=1\ an} (B_i = A_i)/n$$

Donde, $\boldsymbol{A_i}$ es el tiempo de inicio de un trabajo i; $\boldsymbol{B_i}$ tiempo culminado del trabajo i; y $\boldsymbol{n}$ para el número de mediciones.

3.1.4. Hipótesis

En esta sección se definió las preguntas de la investigación que se derivan de la $\boldsymbol{PI}$.

- $\boldsymbol{PI_1}$**:** ¿Qué efectos produce en la calidad del producto del software al desplegar servicios APIs en la Arquitectura Docker?

Una vez detallada las preguntas de la investigación, se determinó las hipótesis para el experimento de acuerdo con $\boldsymbol{PI_1}$.

- $\boldsymbol{H_0}$: *(Hipótesis nula):* No existe diferencia significativa en los efectos de la calidad del producto del software, al desplegar servicios APIs en la Arquitectura Docker.

- $\boldsymbol{H_1}$: *(Hipótesis alternativa 1):* Existe diferencia en la calidad del producto del software al aplicar servicios GraphQL y REST en la Arquitectura Docker. La calidad del software de servicios GraphQL y REST en la Arquitectura Docker es superior a la calidad de software que produce el Entorno Localhost.

- H_2: *(Hipótesis alternativa 2):* Existe diferencia en la calidad del producto del software al aplicar servicios GraphQL y REST en el Entorno Localhost. La calidad del software de servicios GraphQL y REST en el Entorno Localhost es superior a la calidad de software que produce la Arquitectura Docker.

3.1.5. Diseño

El propósito del diseño experimental es crear las condiciones o recurso necesarios para comparar la eficiencia de los APIs en la arquitectura Docker como en el Entorno Localhost. Por lo cual, se definió cuatro tareas experimentales con los mismos escenarios de consulta. La primera tarea desplegó servicios del API REST en el Entorno Localhost, la segunda tarea desplegó servicios del API REST en contenedores Docker, la tercera tarea desplegó servicios del API GraphQL en el Entorno Localhost y la cuarta tarea desplegó servicios del API GraphQL en contenedores Docker. En la Tabla 31 muestra el diseño del experimento, donde se determina los cuatro casos de uso. Cada caso de uso se ejecutó cinco veces en distintos tamaños de registros. Para poder verificar la eficiencia de las APIs se utilizó la métrica definida en la sección en 3.1.3, la cual permitió obtener el tiempo de respuesta en cada caso.

Tabla 31. Diseño del experimento

Caso de Uso	Repeticiones	REST (Localhost - Docker)	GraphQL (Localhost - Docker)
CU-01	5	Registros: 1,100,1000,9000	Registros:1,100,1000,9000
CU-02	5	Registros: 1,100,1000,10000,100000	Registros: 1,100,1000,10000,100000
CU-03	5	Registros: 1,100,1000,10000,100000	Registros: 1,100,1000,10000,100000
CU-04	5	Registros: 1,100,1000,10000,100000	Registros: 1,100,1000,10000,100000

Fuente: Propia

3.1.6. Tareas experimentales

En esta sección se define las tareas experimentales las cuales fueron desarrollados en el CAPÍTULO 2. A través de las tareas experimentales se logró la construcción de un laboratorio computacional, como se muestra en la Fig. 54.

- Consultar e insertar datos al API REST en el Entorno Localhost
- Consultar e insertar datos API REST en contenedores Docker
- Consultar e insertar datos API GraphQL en el Entorno Localhost
- Consultar e insertar datos API GraphQL en contenedores Docker

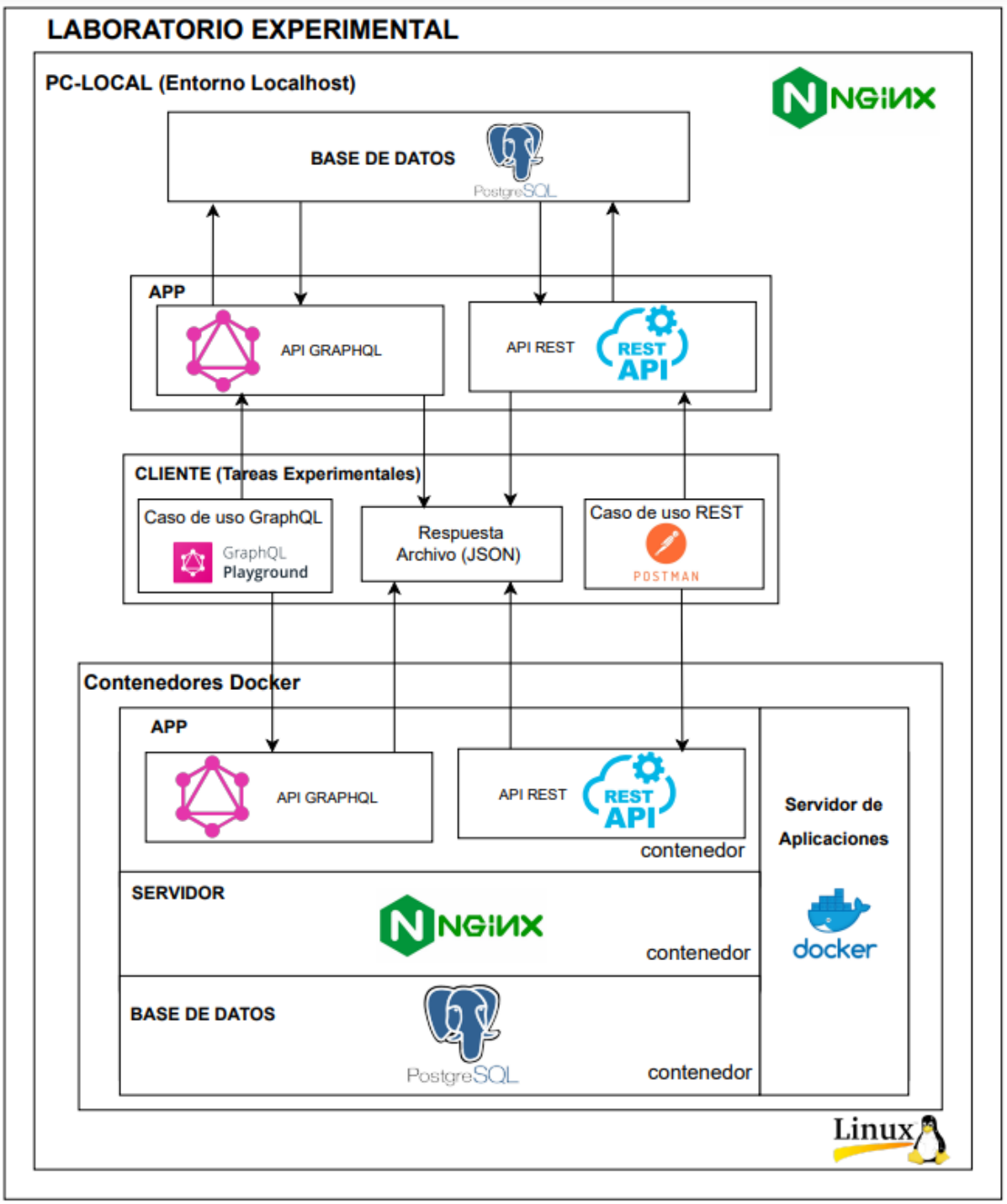


Fig. 54. Arquitectura del laboratorio experimental

Fuente: Propia

3.1.7. Instrumentación

En esta sección se especificó la infraestructura, librerías y tecnologías que contiene un laboratorio computacional.

Descripción de PC-local: La PC cuenta con las siguientes características:

- Sistema Operativo: Linux Ubuntu 22.04 64-bit
- Procesador: AMD Ryzen 7 4700U with Radeon Graphics 2.00 GHz
- Memoria: RAM 16,0 GB

Ambiente de Desarrollo: conformado por las siguientes tecnologías y aplicaciones.

- Ambiente de desarrollo integrado IDE: Visual Studio Code v1.67.2
- Lenguaje de programación: Typescript v4.7.4
- Entorno de tiempo de ejecución de Javascript: NodeJS v16.17
- Librería npm para el API GRAPHQL: @nestjs/graphql v10.1.1, @nestjs/apollo v10.1.0, @nestjs/core v9.0.0, apollo-server-express v3.10.2.
- Mapeo de estructura de base de datos: @nestjs/typeorm v9.0.1, typeorm v0.3.9
- Controlador de base de datos (PostgreSQL): pg v8.8.0.
- Aplicación cliente para el consumo del API REST: Postman v9.18.3
- Aplicación cliente para el consumo de API GRAPHQL: GraphQL Playground

Recolección y análisis de datos: consta las siguientes aplicaciones.

- Microsoft Excel 365
- IBM SPSS v27.0

3.1.8. Recolección de datos

La Tabla 32 muestra la estructura de datos que tuvo el archivo Excel para el registro de los resultados, la cual se obtuvo por medio de la ejecución del experimento que se determinó en la Tabla 31.

Tabla 32. Estructura de recolección de datos

Variable	Descripción
Muestras	Numero de muestras (autonumérico)
Arquitectura	REST o GraphQL en diferentes escenarios (Localhost / Docker)
Repeticiones	Numero de repeticiones (entre 1 a 5 veces)
Caso de Uso	El caso de uso que se ejecutó
Nivel	El nivel de complejidad que tiene la consulta en cada caso de uso
Nro. Registro	Cantidad de datos consultados e insertados
Tiempo	Tiempo de respuesta medido en milisegundos, mediante la ejecución del caso de uso

Fuente: Propia

3.1.9. Análisis

El análisis estadístico de la experimentación y la normalización de los datos se realizó en la herramienta IBM SPSS Statistic, y se utilizó el modelo lineal mixto para la comprobación de la hipótesis.

3.2. Ejecución del experimento

Este experimento se lo realizó el mes de septiembre del 2022, en base al entorno experimental que se propuso en la sección 3.1.

3.2.1. Muestra

Para la muestra se ejecutó los cuatro casos uso que fue desarrollado en el CAPÍTULO 2, a fin de obtener los datos necesarios para realizar la experimentación. El experimento se llevó a cabo de acuerdo con el diseño planteado en la sección 3.1.5.

3.2.2. Preparación

Una vez verificado el funcionamiento de las tareas experimentales especificado en la sección 3.1.6, se llevó a cabo la ejecución del experimento. La ejecución del experimento se efectuó de modo iterativo por cada caso de uso; es decir en cada corrida de la aplicación se ejecutó un caso de uso cinco veces por cada registro y arquitectura. Por ejemplo, para insertar datos se empleó 1, 100, 1000 y 9000 registros, de igual forma para los casos de consultas se aplicó 1, 100, 1000, 10000 y 100000 registros, la cual se encuentra detallado en la Tabla 31. Como resultado se obtuvo un total 380 muestras por toda la ejecución del programa. En las Fig. 55 y Fig. 56 se detalla el proceso de la experimentación.

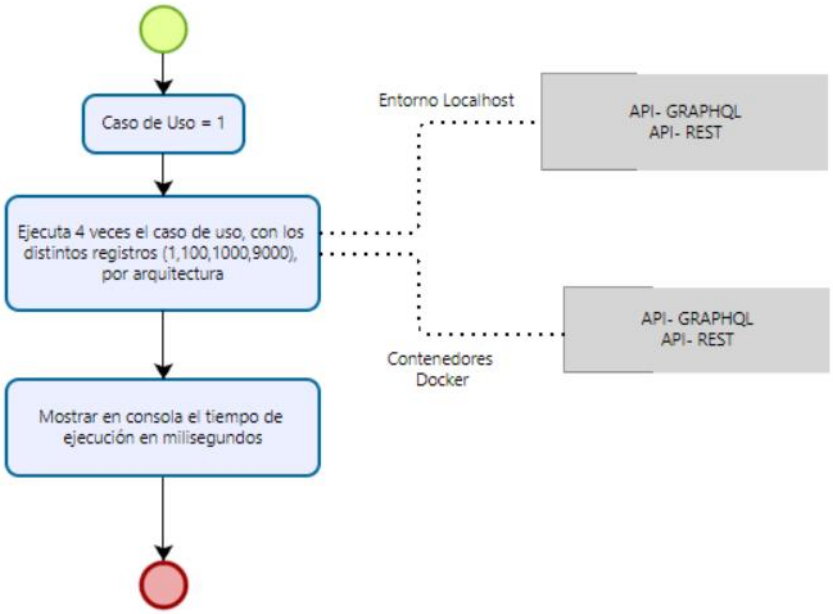


Fig. 55. Proceso de ejecución experimental – Insertar datos

Fuente: Propia

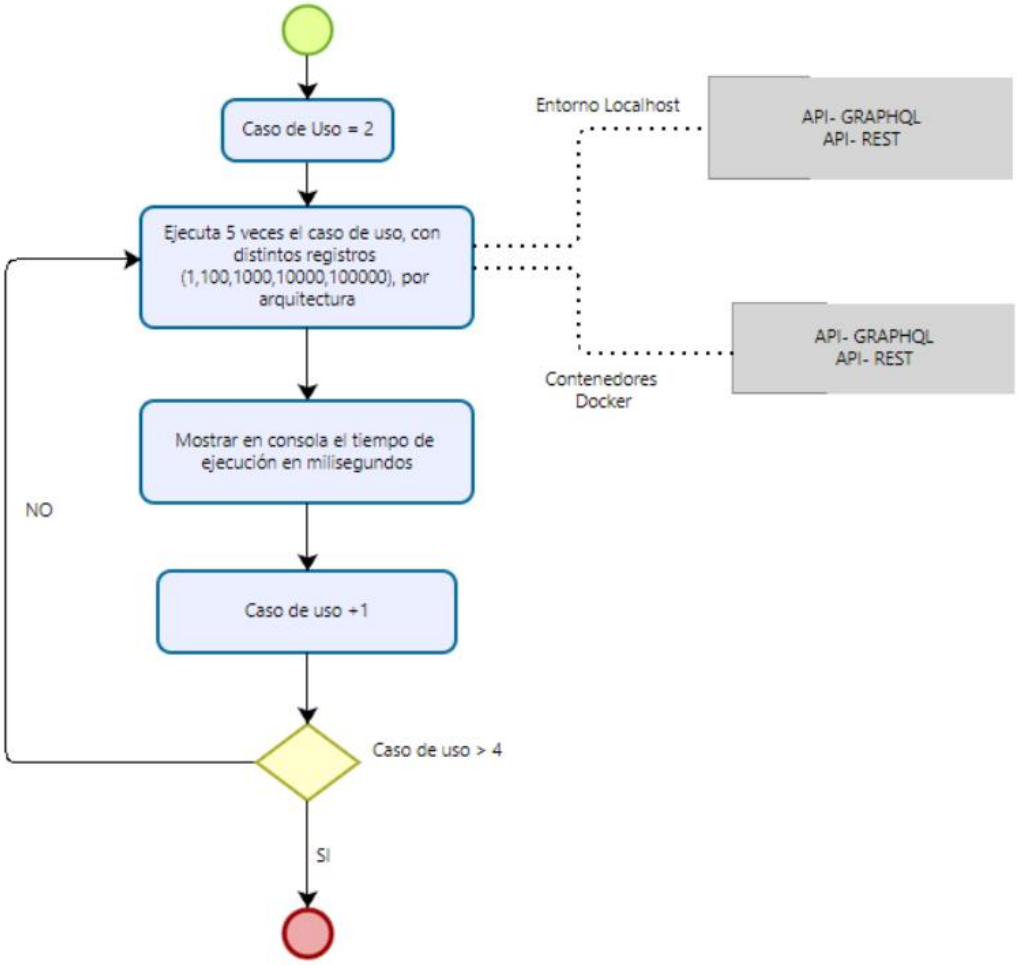


Fig. 56. Proceso de ejecución experimental – Consultar datos

Fuente: Propia

3.2.3. Recolección de datos

Una vez finalizada la ejecución de los casos de usos de acuerdo con el proceso establecido en la Fig. 55 y Fig. 56. Se realizó la recolección de los resultados de tiempo de respuesta y fueron registrados en el archivo de Microsoft Excel 365 para su tabulación. La Fig. 57, Fig. 58, Fig. 59 y Fig. 60 muestran un ejemplo de los tiempos de respuesta (milisegundos) obtenidos por medio de la consola de Visual Studio Code. De igual manera la Tabla 33, Tabla 34, Tabla 35 y Tabla 36 muestran un ejemplo de los registros de datos. Para cada caso de uso se siguió el mismo proceso en las diferentes arquitecturas.

Fig. 57. Resultados por la corrida del Cliente – Localhost/REST

Fuente: Propia

Tabla 33. Recolección de datos Localhost/REST

Muestra	Arquitectura	Repeticiones	Caso de Uso	Nivel	Registros	Tiempo(ms)
41	Localhost/REST	1	CU-02	1	100000	2130
42	Localhost/REST	2	CU-02	1	100000	2230
43	Localhost/REST	3	CU-02	1	100000	2152
44	Localhost/REST	4	CU-02	1	100000	2304
45	Localhost/REST	5	CU-02	1	100000	2137

Fuente: Propia

Fig. 58. Resultados por la corrida del Cliente – Localhost/GraphQL

Fuente: Propia

Tabla 34. Recolección de datos Localhost/GraphQL

Muestra	Arquitectura	Repeticiones	Caso de Uso	Nivel	Registros	Tiempo(ms)
136	Localhost/GraphQL	1	CU-02	1	100000	2144
137	Localhost/GraphQL	2	CU-02	1	100000	2166
138	Localhost/GraphQL	3	CU-02	1	100000	2057
139	Localhost/GraphQL	4	CU-02	1	100000	1988
140	Localhost/GraphQL	5	CU-02	1	100000	2243

Fuente: Propia

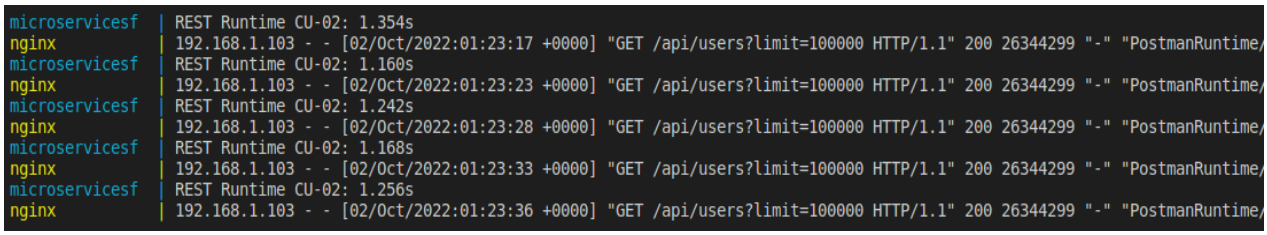


Fig. 59. Resultados por la corrida del Cliente – Docker/REST

Fuente: Propia

Tabla 35. Recolección de datos Docker/REST

Muestra	Arquitectura	Repeticiones	Caso de Uso	Nivel	Registros	Tiempo(ms)
231	Docker/REST	1	CU-02	1	100000	1354
232	Docker/REST	2	CU-02	1	100000	1160
233	Docker/REST	3	CU-02	1	100000	1242
234	Docker/REST	4	CU-02	1	100000	1168
235	Docker/REST	5	CU-02	1	100000	1256

Fuente: Propia

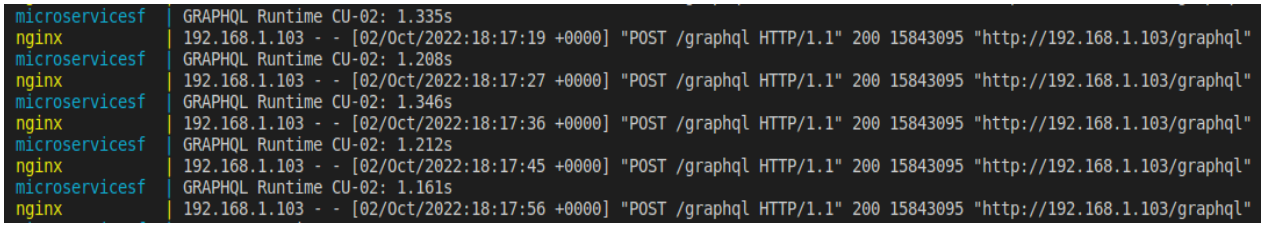


Fig. 60. Resultados por la corrida del Cliente – Docker/GraphQL

Fuente: Propia

Tabla 36. Recolección de datos Docker/GraphQL

Muestra	Arquitectura	Repeticiones	Caso de Uso	Nivel	Registros	Tiempo(ms)
326	Docker/GraphQL	1	CU-02	1	100000	1335
327	Docker/GraphQL	2	CU-02	1	100000	1208
328	Docker/GraphQL	3	CU-02	1	100000	1346
329	Docker/GraphQL	4	CU-02	1	100000	1212
330	Docker/GraphQL	5	CU-02	1	100000	1161

Fuente: Propia

3.3. Análisis de Resultados

En esta sección se analizó el impacto que presenta las arquitecturas en el despliegue de los servicios GraphQL y REST en la calidad del software. Para lo cual, se basó en la característica eficiencia de rendimiento con la métrica tiempo medio de respuesta de la norma ISO/IECC 25010. Los resultados se obtuvieron mediante la ejecución del experimento especificado en la sección 3.2.

3.3.1. Análisis estadísticos de la Eficiencia

A continuación, se realiza un análisis de la eficiencia de rendimiento de cada caso de uso en las diferentes arquitecturas.

Caso de Uso 1

Para el primer caso de uso (Insertar usuario). La Fig. 61 muestra los valores medio de respuesta que se obtuvo por cada arquitectura. Se puede observar que el tiempo medio de respuesta que presenta la Arquitectura Docker fue de 224,33 en GraphQL y 222,75 en REST, la cual es mucho menor al que presenta el Entono Localhost con un 318,83 en GraphQL y 363,91 en REST. De igual manera la Tabla 37 muestra la eficiencia en porcentaje que presenta las APIs en comparación entre estas arquitecturas, donde, GraphQL logra una eficiencia de 29,64% y REST una eficiencia de 38,79% con la Arquitectura Docker. Por lo cual se puede determinar que la Arquitectura Docker es más eficiente que el Entono Localhost.

Tabla 37. Porcentaje de eficiencia – CU-01

Arquitectura	**Media**	**Arquitectura**	**Media**
Docker/GraphQL	224,33	Docker/REST	222,75
Localhost/GraphQL	318,83	Localhost/REST	363,91
Eficiencia Docker/GraphQL	**29,64%**	**Eficiencia Docker/REST**	**38,79%**

Fuente: Propia

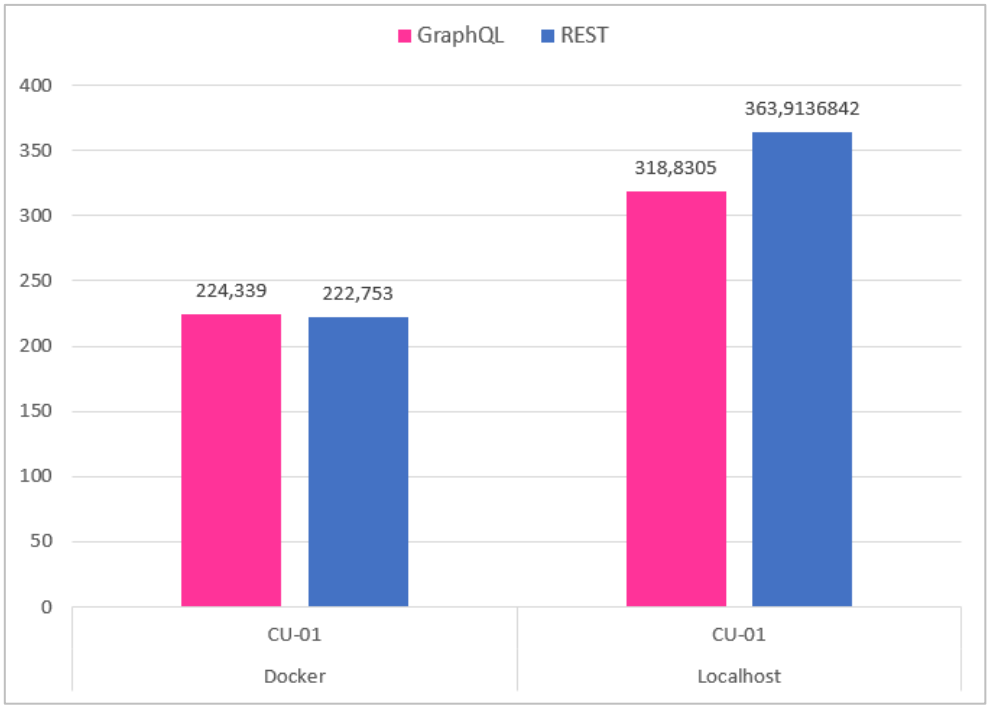


Fig. 61. Valor medio de eficiencia – CU-01

Fuente: Propia

Caso de Uso 2

Para el segundo caso de uso (Consulta usuarios) con nivel de consulta uno. La Fig. 62 muestra los valores medio de respuesta que se obtuvo por cada arquitectura. Se puede

observar que el tiempo medio de respuesta que presenta la Arquitectura Docker fue de 289,60 en GraphQL y 281,54 en REST, la cual es mucho menor al que presenta el Entorno Localhost con un 481,02 en GraphQL y 496,79 en REST. De igual manera la Tabla 38 muestra la eficiencia que presenta las APIs en comparación entre estas arquitecturas, donde, GraphQL logra una eficiencia de 39,79% y REST una eficiencia de 43,33% con la Arquitectura Docker. Por lo cual se puede determinar que la Arquitectura Docker es más eficiente que el Entono Localhost.

Tabla 38. Porcentaje de eficiencia – CU-02

Arquitectura	**Media**	**Arquitectura**	**Media**
Docker/GraphQL	289,60	Docker/REST	281,54
Localhost/GraphQL	481,02	Localhost/REST	496,79
Eficiencia Docker/GraphQL	**39,79%**	**Eficiencia Docker/REST**	**43,33%**

Fuente: Propia

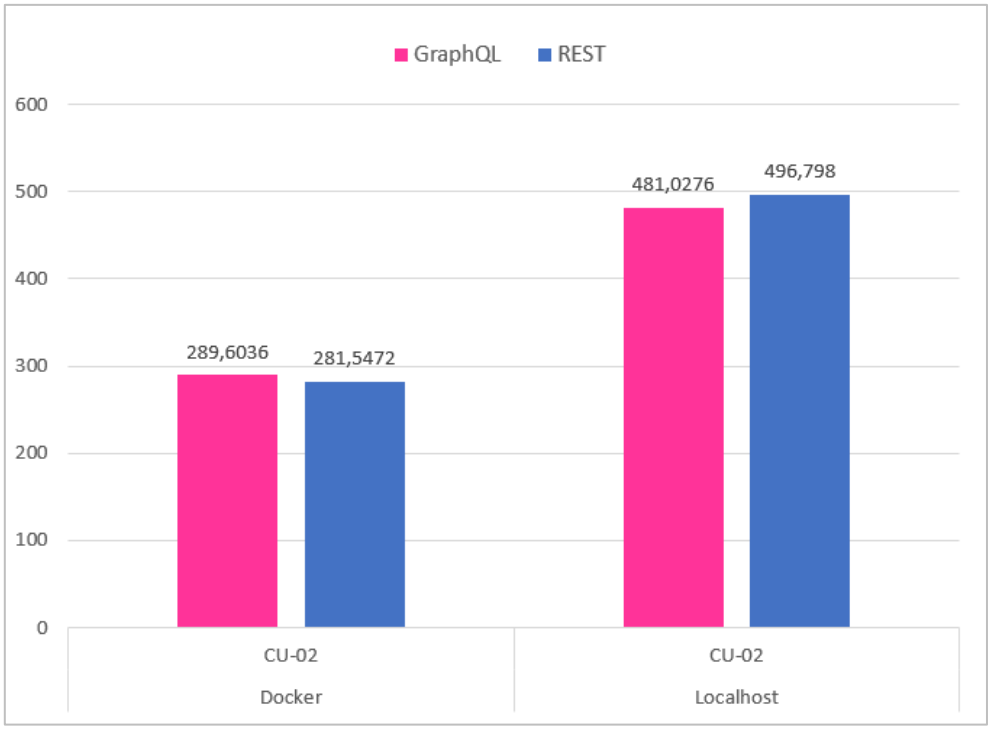


Fig. 62. Valor medio de eficiencia – CU-02

Fuente: Propia

Caso de Uso 3

Para el tercer caso de uso (Consulta usuarios por cuenta) con nivel de consulta dos. La Fig. 63 muestra los valores medio de respuesta que se obtuvo por cada arquitectura. Se puede observar que el tiempo medio de respuesta que presenta la Arquitectura Docker fue de 682,20 en GraphQL y 711,95 en REST, la cual es mucho menor al que presenta el Entono

Localhost con un 857,44 en GraphQL y 898,11 en REST. De igual manera la Tabla 39 muestra la eficiencia que presenta las APIs en comparación entre estas arquitecturas, donde, GraphQL logra una eficiencia de 20,44% y REST una eficiencia de 20,73% con la Arquitectura Docker. Por lo cual se puede determinar que la Arquitectura Docker es más eficiente que el Entono Localhost.

Tabla 39. Porcentaje de eficiencia – CU-03

Arquitectura	**Media**	**Arquitectura**	**Media**
Docker/GraphQL	682,20	Docker/REST	711,95
Localhost/GraphQL	857,44	Localhost/REST	898,11
Eficiencia Docker/GraphQL	**20,44%**	**Eficiencia Docker/REST**	**20,73%**

Fuente: Propia

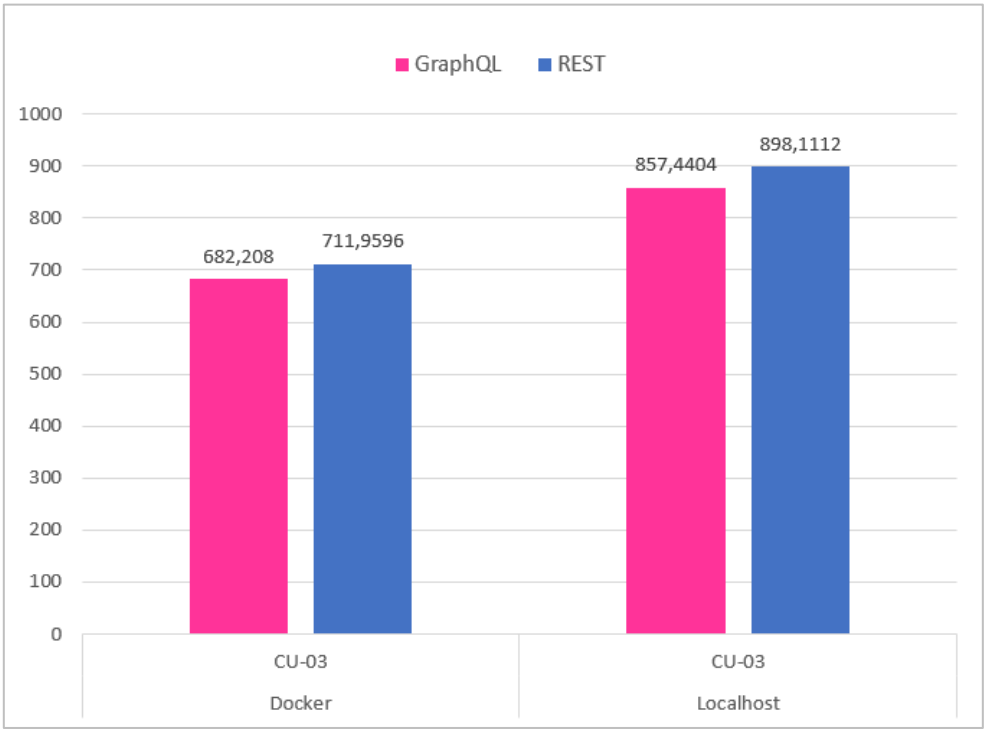


Fig. 63. Valor medio de eficiencia – CU-03

Fuente: Propia

Caso de Uso 4

Para el cuarto caso de uso (Consulta usuarios por cuenta y por movimiento) con nivel de consulta tres. La Fig. 64 muestra los valores medio de respuesta que se obtuvo por cada arquitectura. Se puede observar que el tiempo medio de respuesta que presenta la Arquitectura Docker fue de 2140,51 en GraphQL y 2158,57 en REST, la cual es mucho menor al que presenta el Entono Localhost con un 2302,71 en GraphQL y 2697,58 en REST. De igual manera la Tabla 40 muestra la eficiencia que presenta las APIs en comparación entre

estas arquitecturas, donde, GraphQL logra una eficiencia de 7,04% y REST una eficiencia de 19,98% con la Arquitectura Docker. Por lo cual se puede determinar que la Arquitectura Docker es más eficiente que el Entono Localhost.

Tabla 40. Porcentaje de eficiencia – CU-04

Arquitectura	**Media**	**Arquitectura**	**Media**
Docker/GraphQL	2140,51	Docker/REST	2158,57
Localhost/GraphQL	2302,71	Localhost/REST	2697,58
Eficiencia Docker/GraphQL	**7,04%**	**Eficiencia Docker/REST**	**19,98%**

Fuente: Propia

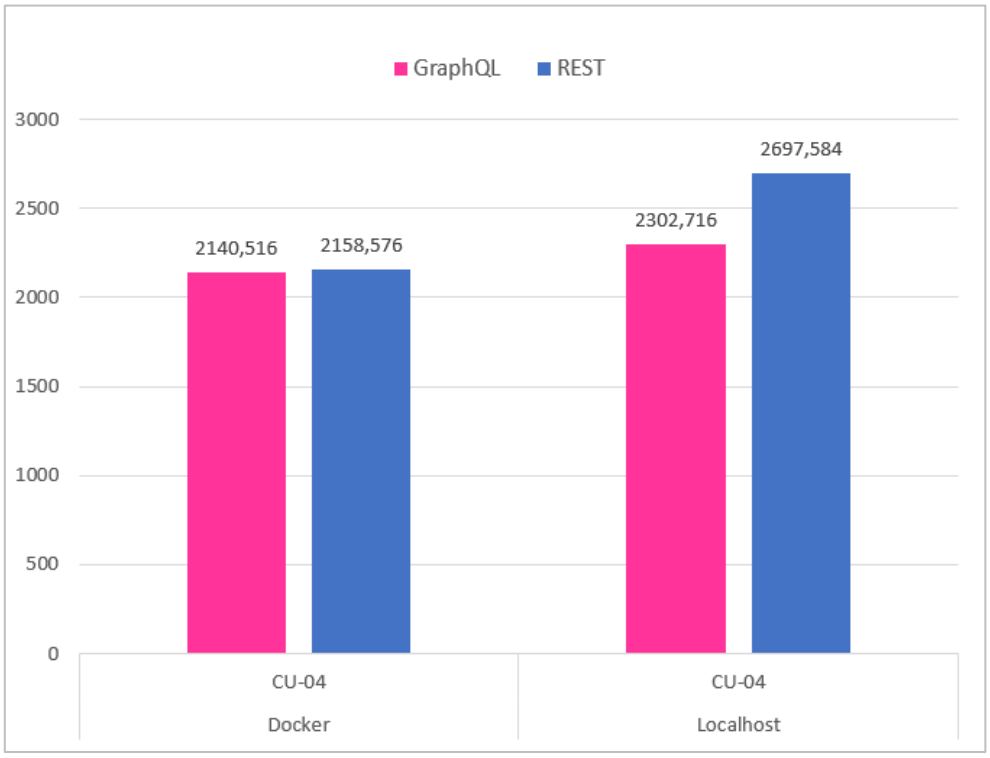


Fig. 64. Valor medio de eficiencia – CU-04

Fuente: Propia

En la Fig. 65 muestra el comportamiento de los cuatro casos de uso en las diferentes arquitecturas. En el caso uno y dos realizó la consulta en relación con una tabla, el tercer caso realizó la consulta en relación con dos tablas y el cuarto caso realizó la consulta en relación con tres tablas. Por lo cual, se puede decir que los tiempos medio de respuesta aumentan al ejecutar consultas en base a las relaciones de las tablas. También se pude observar que GraphQL presenta un tiempo de respuesta menor en la mayoría de los casos de usos a excepción del caso uno y dos en Docker que muestra un valor mayor.

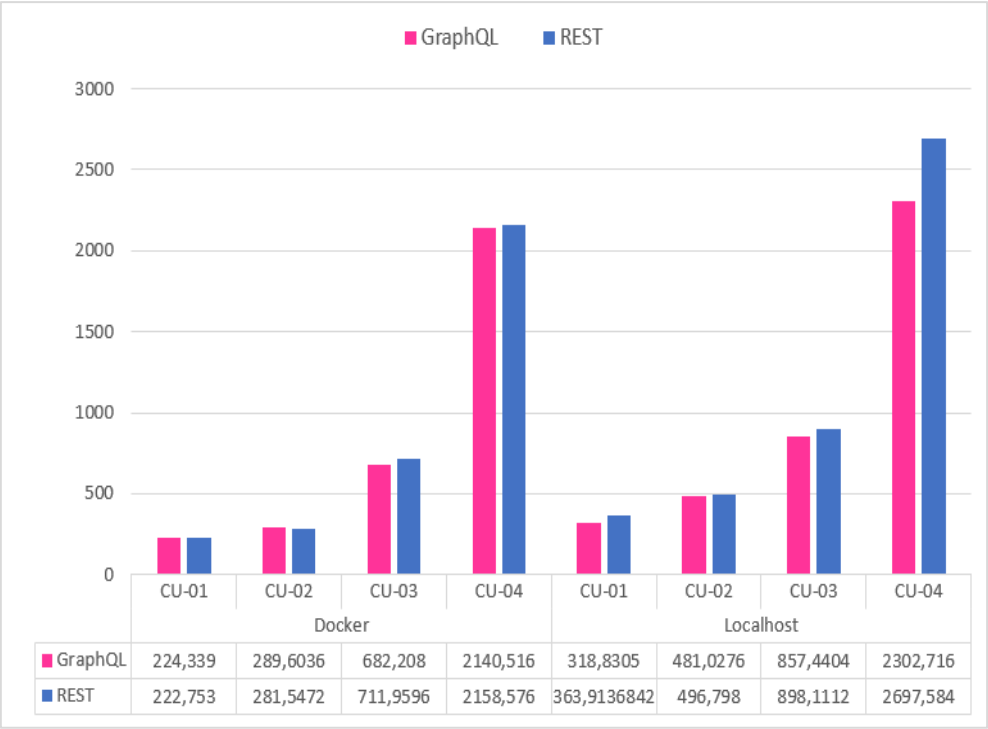


Fig. 65. Valor medio de eficiencia entre GraphQL y REST

Fuente: Propia

En la Fig. 66 y Fig. 67 se puede apreciar los resultados estadísticos globales, donde el valor medio de eficiencia para GraphQL en el Entorno Localhost fue de 1025,328; 2319,840 (Media; Desviación estándar), con un intervalo de confianza del 95% [552,7528–1497,9043]. En cambio, REST tuvo un valor medio de 1149,820; 2709,509 (Media; Desviación estándar), con un intervalo de confianza del 95% de [597,865–1701,775], que fueron mayores a comparación de GraphQL. Por lo cual se determinó que el tiempo medio de respuesta que presenta GraphQL es más eficiente que REST en el Entorno Localhost.

De la misma manera el análisis estadístico en la arquitectura Docker muestra que el valor medio de la eficiencia en GraphQL fue de 866,262; 2108,299 (Media; Desviación estándar), con el 95% de intervalo de confianza [436,780-1295,745]; mientras que REST tuvo 876,39; 2132,421 con un intervalo del 95% de confianza [441,994-1310,787]. Por lo cual se determinó que el tiempo de respuesta que presenta GraphQL es más eficiente que REST en la Arquitectura Docker.

	Arquitectura			Estadístico	Error estándar
GraphQL	Localhost	Media		1025,3285	238,01065
		95% de intervalo de confianza para la media	Límite inferior	552,7528	
			Límite superior	1497,9043	
		Mediana		142,3600	
		Varianza		5381661,743	
		Desviación estándar		2319,84089	
		Mínimo		1,87	
		Máximo		10397,00	
		Rango		10395,13	
		Asimetría		3,266	,247
		Curtosis		10,207	,490
	Docker	Media		866,2628	216,30698
		95% de intervalo de confianza para la media	Límite inferior	436,7803	
			Límite superior	1295,7454	
		Mediana		121,6400	
		Varianza		4444927,403	
		Desviación estándar		2108,29965	
		Mínimo		,79	
		Máximo		9368,00	
		Rango		9367,21	
		Asimetría		3,468	,247
		Curtosis		11,390	,490
REST	Localhost	Media		1149,8202	277,98979
		95% de intervalo de confianza para la media	Límite inferior	597,8650	
			Límite superior	1701,7755	
		Mediana		157,0700	
		Varianza		7341440,687	
		Desviación estándar		2709,50931	
		Mínimo		1,75	
		Máximo		12544,00	
		Rango		12542,25	
		Asimetría		3,449	,247
		Curtosis		11,495	,490
	Docker	Media		876,3908	218,78182
		95% de intervalo de confianza para la media	Límite inferior	441,9944	
			Límite superior	1310,7873	
		Mediana		121,1200	
		Varianza		4547221,118	
		Desviación estándar		2132,42142	
		Mínimo		,88	
		Máximo		9441,00	
		Rango		9440,12	
		Asimetría		3,431	,247
		Curtosis		11,154	,490

Fig. 66. Estadística descriptiva de eficiencia entre las arquitecturas

Fuente: Propia

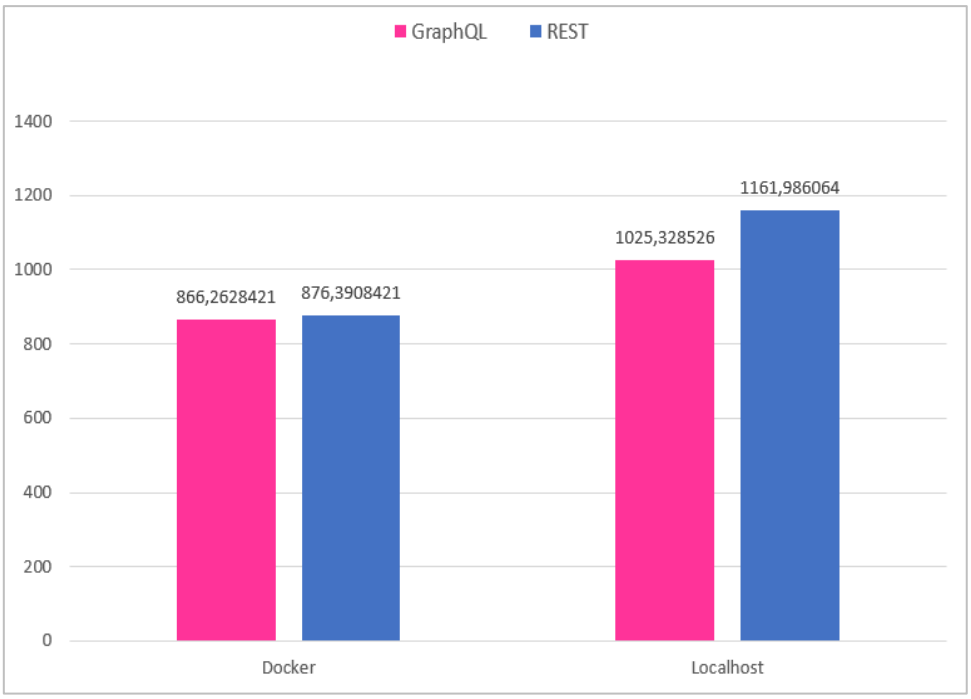


Fig. 67. Comparativa de eficiencia entre arquitecturas

Fuente: Propia

3.3.2. Análisis de Impactos

Una vez realizado el análisis estadístico de eficiencia, se determinó que la Arquitectura Docker fue más eficiente en el despliegue de los servicios APIs, con una eficiencia del 15,51% en GraphQL y un 24,58% en REST como se observa en la Tabla 41.

Tabla 41. Eficiencia de la Arquitectura Docker

Arquitectura	**Media**	**Arquitectura**	**Media**
Docker/GraphQL	866,262	Docker/REST	876,390
Localhost/GraphQL	1025,328	Localhost/REST	1161,986
Eficiencia Docker/GraphQL	**15,51%**	**Eficiencia Docker/REST**	**24,58%**

Fuente: Propia

De la misma forma, según el análisis realizado se determinó que GraphQL fue más eficiente que REST. Como se puede observar en la Tabla 42, GraphQL muestra una eficiencia del 1,16% en la Arquitectura Docker y una eficiencia de 11,76% en el Entorno Localhost. Para lo cual se seleccionó la eficiencia de Docker/GraphQL debido a que Docker es más eficiente en el despliegue de los servicios como se mencionó en la Tabla 41. Sin embargo, no presenta mucha diferencia, esto puede ocurrir por el CACHE que tiene REST y que GraphQL no dispone.

Tabla 42. Eficiencia de GraphQL entre las arquitecturas

Arquitectura	**Media**	**Arquitectura**	**Media**
Docker/GraphQL	866,262	Localhost/GraphQL	1025,328
Docker/REST	8766,390	Localhost/REST	1161,986
Eficiencia Docker/GraphQL	**1,16%**	**Eficiencia Localhost/GraphQL**	**11,76%**

Fuente: Propia

Finalmente, los resultados fueron validados mediante pruebas estadísticas, estableciendo así la base para la selección y aceptación de una de las hipótesis propuestas en este experimento. Este trabajo de investigación promueve la utilización de nuevas tecnologías como Docker para el desarrollo y despliegue de aplicaciones dentro de contenedores.

CONCLUSIONES

- Se estableció el estado del arte mediante una revisión bibliográfica y conceptualización de las arquitecturas REST y GraphQL orientadas a microservicios, de igual manera la virtualización de entornos de desarrollo en tecnología Docker, como base teórica para desarrollar la investigación.

- En base a los conceptos establecidos se desarrolló una prueba de concepto (aplicación Backend) con la estructura de transferencias bancarias, a través de un ambiente virtualizado (Docker). Por medio de la herramienta Docker Compose se realizó la integración de los servicios, logrando el despliegue de la aplicación Backend con APIs GraphQL y REST dentro de contenedores.

- Para la validación del uso de contenedores Docker, se planteó un experimento computacional definida en la sección 3.1, donde se comparó la eficiencia de rendimiento entre la Arquitectura Docker y el Entorno Localhost, en base al entorno de la calidad del software para el despliegue de los servicios APIs. Para lo cual, se respondió a la pregunta de la investigación planteada $\boldsymbol{PI}$:¿En qué condiciones es más adecuado utilizar un entorno virtualizado (Docker) y con qué arquitecturas, para el despliegue de microservicios? Para ello, se implementó cuatro casos de uso en las diferentes arquitecturas y se midió la eficiencia del rendimiento mediante la métrica *"tiempo medio de respuesta"* establecida por la ISO/IEC 25010 e ISO/IEC 25023.

- Una vez realizado el experimento computacional se logró responder a la pregunta derivada de la investigación. $\boldsymbol{PI_1}$**:** ¿Qué efectos produce en la calidad del producto del software al desplegar servicios APIs en la Arquitectura Docker?, De acuerdo con la ejecución de las tareas experimentales definidas en la sección 3.1.6. y por medio del análisis estadístico de la eficiencia realizado en la sección 3.3.1, se comprobó que el tiempo medio de respuesta de la arquitectura Docker es menor al que presenta el Entorno Localhost en el despliegue del microservicio. De la misma manera se comprobó que GraphQL tuvo un valor medio de respuesta menor al API REST en las diferentes arquitecturas. Por consiguiente, se concluye que la arquitectura Docker es más eficiente en el despliegue de los servicios especialmente en GraphQL. Por lo cual se acepta la hipótesis alternativa definida en 3.1.4.

RECOMENDACIONES

- Para obtener resultados precisos se recomienda realizar el despliegue de la aplicación en un entorno de producción, puesto que ahí se encuentra todo el proyecto empaquetado logrando obtener los tiempos reales en la ejecución de los servicios. Por otra parte, los tiempos de ejecución puede variar dependiendo el recurso del computador.

- Para realizar las consultas de los servicios en GraphQL se puede agregar CACHÉ y comparar la eficiencia con los servicios de REST y observar que comportamientos ocurren, si cambian o no con la investigación presentada.

- Se recomienda el uso del NestJS, que es framework basado en Node para el desarrollo de aplicaciones Backend. Debido a que dispone de una estructura modular que permite fácilmente la implementación de los servicios como GraphQL y REST.

- Para futuros trabajos se puede implementar la herramienta Jenkins para realizar la integración continua, ya que es el siguiente proceso en el ciclo de vida de DevOps.

REFERENCIAS

Álvaro, I., & Martínez, I. (2021). *Docker para DevOps de noob a experto*. http://dockerparadevops.com/downloads/docker_para_devops_r1.pdf

Arcidiácono, J., Bazán, P., & Lliteras, A. B. (2021). *Una arquitectura de Microservicios para dar soporte a la creación y ejecución de actividades de recolección de datos con intervención humana*.

Barrios, D. (2018). *Market Cart App: Aplicación móvil para la gestión de compra de víveres en línea*. http://revistas.udistrital.edu.co/ojs/index.php/tia/issue/archive

Bernardi, T., Zanatta, A., Beux, J., Biduski, D., & Bellei, É. (2017). *Learning by Doing em Fábrica de Software: Relato de uma Experiência no Mestrado Profissional em Computação Aplicada*.

Brito, G., Mombach, T., & Valente, M. (2019). *Migrating to GraphQL: A Practical Assessment*.

Brito, G., & Valente, M. T. (2020). REST vs GraphQL: A controlled experiment. *Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020*, 81–91. https://doi.org/10.1109/ICSA47634.2020.00016

Carballo, L., & Barrientos, I. (2018). *Propuesta para evaluar arquitecturas de software*. https://www.researchgate.net/publication/352843642_PROPUESTA_PARA_EVALUAR_ARQUITECTURAS_DE_SOFTWARE_PROPOSAL_TO_EVALUATE_SOFTWARE_ARCHITECTURES

Cervantes, H., Velazco, P., & Castro, L. (2016). *Arquitectura de Software*. https://www.researchgate.net/publication/281137715

De Paz, J. M. (2017). *Diseño e implementación de una Arquitectura escalable basada en Microservicios para un Sistema de Gestión de Aprendizaje con características de red social* [Universidad de San Carlos de Guatemala]. http://www.repositorio.usac.edu.gt/7023/1/JOS%C3%89%20MANUEL%20DE%20PAZ%20ESTRADA.pdf

Docker. (2022). *Docker Documentation*. https://docs.docker.com/

Eizinger, T. (2017). *API Design in Distributed Systems: A Comparison between GraphQL and REST*.

Gao, Z., Bird, C., & Barr, E. T. (2017). *To Type or Not to Type:Quantifying Detectable Bugs in JavaScript.*

Garcia, R. (2020). *Gestión de una arquitectura de Microservicios con Istio*.

Gouigoux, J. P., & Tamzalit, D. (2017). From monolith to microservices. *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, 62–65. https://doi.org/10.1109/ICSAW.2017.35

Guijarro, J., Caparros, J., & Cubero, L. (2019). *DevOps y seguridad cloud*. Editorial UOC. https://elibro.net/es/lc/utnorte/titulos/128889

Guimarey, A. (2020). *Beneficios y riesgos de migrar una arquitectura monolítica a microservicios*. https://www.researchgate.net/publication/348309479_Beneficios_y_riesgos_de_migrar_una_arquitectura_monolitica_a_microservicios

ISO 2510. (2015). *Sistemas e Ingeniería de Software-requisitos y evaluación de sistemas y calidad de software (SQuare)-Modelos de calidad del sistema y software (ISO/IEC 25010:2011, IDT)*.

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, *35*(3), 24–35. https://doi.org/10.1109/MS.2018.2141039

Kang, H., Le, M., & Tao, S. (2016). Container and Microservice Driven Design for Cloud Infrastructure DevOps. *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-Located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, 202–211. https://doi.org/10.1109/IC2E.2016.26

Khan, M., Khan, A., Khan, F., Khan, M., & Whangbo, T. (2022). *Critical Challenges to Adopt DevOps Culture in Software Organizations*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9690862

Martin, R. C. (2018). *Clean Architecture A Craftsman's Guide to Software Structure and Design*.

Mas, G. S., Budiwitjaksono, G. S., Marufi, H. A., & Sampurno Ilham Ade. (2021, April 27). *Comparative Analysis of Rest and GraphQL Technology on Nodejs-Based Api Development*. https://doi.org/10.11594/nstp.2021.0908

Naciones Unidas. (2022). *La Agenda para el Desarrollo Sostenible - Desarrollo Sostenible*. https://www.un.org/sustainabledevelopment/es/development-agenda/

NestJS. (2022). *Documentation | NestJS - A progressive Node.js framework*. https://docs.nestjs.com/

Neumann, A., Laranjeiro, N., & Bernardino, J. (2018). An Analysis of public REST Web Service APIs. *IEEE Transactions on Services Computing*, *14*(4), 957–970. https://doi.org/10.1109/TSC.2018.2847344

Pacheco, J. L. (2018). *Estudio comparativo entre una arquitectura con microservicios y contenedores Dockers y una arquitectura tradicional (Monolítica) con comprobación aplicativa*.

Palma, N. (2020). *Eficiencia de los servidores web Apache 2 y Nginx: un estudio de caso* (Vol. 13, Issue 9). http://publicaciones.uci.cu

Pham, A. D. (2020). *Developing BACK-END of a Web Application with NestJS Framework*.

PostgreSQL. (2022). *PostgreSQL 13.7 Documentation The PostgreSQL Global Development Group*.

Qian, L., Chen, H., Zhu, G., Zhun, J., Ren, C., Pang, H., Xu, M., & Wang, L. (2020). *Research on Micro Service Architecture of Power Information System Based on Docker Container*. https://doi.org/10.1088/1755-1315/440/3/032147

Raj, A., & Jasmine, K. S. (2021). *Building Microservices with Docker Compose*.

Ramos, D. B., Ramos, I. M. M., Viana, W. D. S., Silva, G. R. E., & Oliveira, E. H. T. (2016). *On the use of Scrum for the management of research-oriented projects*. http://www.tise.cl/volumen12/TISE2016/589-594.pdf

Riti, P. (2018). Introduction to DevOps. In *Pro DevOps with Google Cloud Platform* (pp. 1–18). Apress. https://doi.org/10.1007/978-1-4842-3897-4_1

Rodríguez, Á. I., Padilla, J. I., & Parra, H. A. (2019). *Arquitectura basada en microservicios para aplicaciones web*.

Rodríguez, Z., Rodríguez, L., & Gonzales, J. C. (2020). Arquitectura basada en Microservicios y DevOps para una ingeniería de software continua. *Industrial Data*, *23*(2), 141–149. https://doi.org/10.15381/idata.v23i2.17278

Roldán Martínez, David., Valderas Aranda, P. J., & Torres Bosch, Victoria. (2018). *Microservicios: un enfoque integrado*. 184. https://doi.org/10.0/CSS/ALL.MIN.D74D1A5D029B.CSS

Rosado, A. A., & Jaimes, J. C. (2018). Revisión de la incorporación de la Arquitectura Orientada a Servicios en las organizaciones. *REVISTA COLOMBIANA DE TECNOLOGIAS DE AVANZADA (RCTA)*, *1*(31). https://doi.org/10.24054/16927257.v31.n31.2018.2769

Sabo, M. (2020). *NestJS*. https://urn.nsk.hr/urn:nbn:hr:126:058660

Sánchez, J. D., Martínez, A., Quesada Christian, & Jenkins, M. (2020). *Caracterización de las prácticas de DevOps en organizaciones que desarrollan software*. https://www.proquest.com/openview/9bbd7b1a28d247d2f1722d3364e57f54/1?pq-origsite=gscholar&cbl=1006393

Sayago, J., & Flores, E. (2019). *Análisis Comparativo entre los Estándares Orientados a Servicios Web SOAP, REST y GraphQL*. https://doi.org/10.5281/zenodo.3592004

Shahin, M., Ali, M., & Zhu, L. (2017). *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*.

Sharma, L. (2022, August 23). *Bitbucket vs GitHub: qué repositorio usar en 2022 - Geekflare*. https://geekflare.com/es/bitbucket-vs-github/

Singh, C., Gaba, N., & Kaur Manjot. (2019). *Comparison of Different CI/CD Tools Integrated with Cloud Platform*.

Sollfrank, M., Loch, F., Denteneer, S., & Vogel-Heuser, B. (2021). Evaluating Docker for lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation. *IEEE Transactions on Industrial Informatics*, *17*(5), 3566–3576. https://doi.org/10.1109/TII.2020.3022843

Universidad Técnica del Norte. (2022). *Perfil de Egreso Carrera de Software*. https://software.utn.edu.ec/gestion-academica/perfil-de-egreso/

Urbina, M., Abud, M., Camarena, G., Hernández, G. A., & Sánchez, A. (2016, September 28). *Propuesta de un modelo de integración de PSP y Scrum para mejorar la calidad del proceso de desarrollo en una MiPyME*. https://rcs.cic.ipn.mx/2016_120/Propuesta%20de%20un%20modelo%20de%20integracion%20de%20PSP%20y%20Scrum%20para%20mejorar%20la%20calidad%20del%20proceso.pdf

Wholin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*.

Woods, E. (2016). *Software Architecture in a Changing World*. https://www.eoinwoods.info/media/writing/Woods-SoftwareArchitectureChangingWorld.pdf

Yarygina, T., & Bagge, A. H. (2018). Overcoming Security Challenges in Microservice Architectures. *Proceedings - 12th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2018 and 9th International Workshop on Joint Cloud Computing, JCC 2018*, 11–20. https://doi.org/10.1109/SOSE.2018.00011

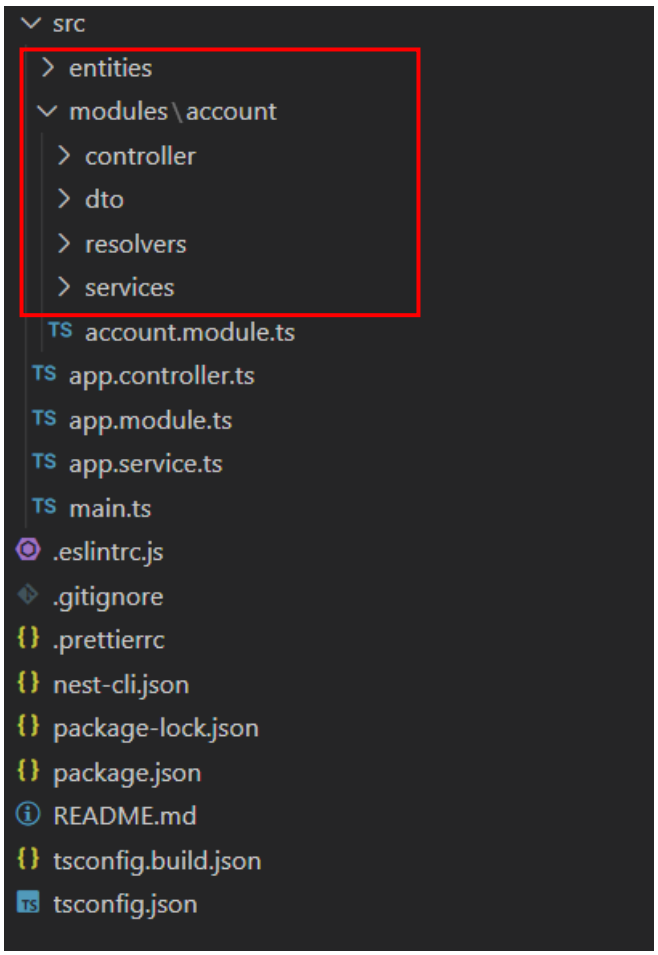
You, L., & Sun, H. (2022). *Design of Docker Technology Based Authority Management System*. https://doi.org/10.1155/2022/5325694

ANEXOS

ANEXO A: Estructura de la aplicación - prueba de concepto.

- Desarrollada con el Framework NestJS.

  Creación de los componentes: entities, services, controller para la arquitectura Rest y resolver para la arquitectura GraphQL.



- Conexión a la base de dato.

  Para la obtener las entidades es necesario que se encuentre sincronizado, sin embargo, para el despliegue en producción se recomienda desactivar esta opción.

src > config > TS typeOrmConfig.ts > ...

```ts
import { TypeOrmModuleOptions } from '@nestjs/typeorm';

export const typeOrmConfig: TypeOrmModuleOptions = {
  type: 'postgres',
  //url: process.env.DATABASE_URL,
  host: 'localhost',
  port: 5432,
  username: 'postgres',
  password: 'passpostgres',
  database: 'financial',
  entities: [__dirname + '/../**/*.entity.{js,ts}'],
  synchronize: true,
};
```

- Servicio de consulta.

```ts
//GET ALL USER by ACCOUNT -------------CASE USE THREE
async findAllUserAccount({ limit, offset }: FilterUserDto): Promise<User[]> {
  console.time('Runtime CU-03');
  const result = await this.userRepository.find({
    order: { id: 'ASC' },
    relations: ['accounts'],
    skip: offset,
    take: limit,
  });
  console.timeEnd('Runtime CU-03');
  return result;
}
```

- Componente resolver del servicio de consulta en la arquitectura API- GraphQL.

```ts
//----------------GET ALL USER by ACCOUNT -----------------CASE THREE
@Query(() => [User], { name: 'getAllUserAccount' })
findAllUserAccount(@Args() args: FilterUserDto): Promise<User[]> {
  return this.userService.findAllUserAccount(args);
}
```

- Componente controller del servicio de consulta en la arquitectura API- REST.

```ts
@Get('/accounts')
@ApiOperation({
  summary: 'List all users by account -------- CASE USE THREE',
})
findAllUserAccount(@Query() size: FilterUserDto): Promise<User[]> {
  return this.userService.findAllUserAccount(size);
}
```

- Instalar y configurar el servidor Nginx en la maquina local.

```
server {
        listen 80 default_server;
        listen [::]:80 default_server;

        # SSL configuration
        #
        # listen 443 ssl default_server;
        # listen [::]:443 ssl default_server;
        #
        # Note: You should disable gzip for SSL traffic.
        # See: https://bugs.debian.org/773332
        #
        # Read up on ssl_ciphers to ensure a secure configuration.
        # See: https://bugs.debian.org/765782
        #
        # Self signed certs generated by the ssl-cert package
        # Don't use them in a production server!
        #
        # include snippets/snakeoil.conf;

        root /var/www/html;

        # Add index.php to the list if you are using PHP
        index index.html index.htm index.nginx-debian.html;

        server_name _;

        location / {
                # First attempt to serve request as file, then
                # as directory, then fall back to displaying a 404.
                try_files $uri $uri/ =404;
        }

        location /api/docs {
                proxy_pass http://localhost:3000;
        }

        location /api/users {
                proxy_pass http://localhost:3000;
        }

        location /graphql {
                proxy_pass http://localhost:3000;
        }
```

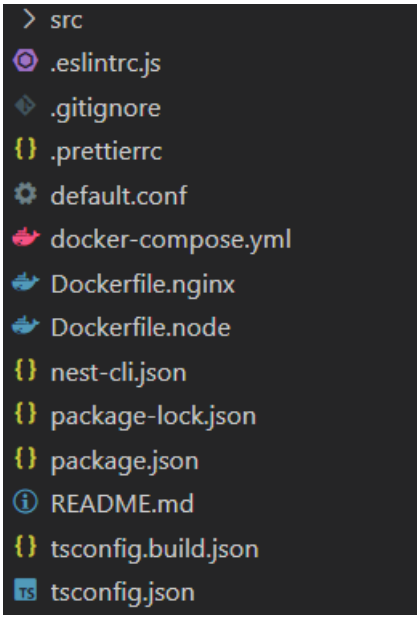
- Verificar el estado del servidor Nginx.

  sudo service nginx status.

```
zguitarra@ubuntu20:~$ sudo service nginx status
● nginx.service - A high performance web server and a reverse proxy server
     Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2022-09-29 01:57:43 -05; 2s ago
       Docs: man:nginx(8)
    Process: 52118 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
    Process: 52119 ExecStart=/usr/sbin/nginx -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
   Main PID: 52120 (nginx)
      Tasks: 9 (limit: 18344)
     Memory: 7.3M
        CPU: 19ms
     CGroup: /system.slice/nginx.service
             ├─52120 "nginx: master process /usr/sbin/nginx -g daemon on; master_process on;"
             ├─52121 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
             ├─52122 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
             ├─52123 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
             ├─52124 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
             ├─52125 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
             ├─52126 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
             ├─52127 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
             └─52128 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""

sep 29 01:57:43 ubuntu20 systemd[1]: Starting A high performance web server and a reverse proxy server...
sep 29 01:57:43 ubuntu20 systemd[1]: Started A high performance web server and a reverse proxy server.
```

- Creación y configuración de los archivos Docker en el proyecto.



ANEXO B: Comandos Docker

- docker-compose ps:

  Muestra contenedores creados en el archivo docker-compose.

```
zguitarra@ubuntu20:~/api-banksystem$ docker-compose ps
NAME                COMMAND                  SERVICE             STATUS
microservicesf      "docker-entrypoint.s…"   microservice        running
nginx               "/docker-entrypoint.…"   nginx               running
postgresdb          "docker-entrypoint.s…"   postgres            running
```

- docker ps:

  Muestra contenedores activos, al añadir la letra **-a** muestra contenedores ejecutados y detenidos

```
zguitarra@ubuntu20:~$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED
132f8af7e67f   api-banksystem_nginx           "/docker-entrypoint.…"   18 minutes ago
0ff0ef27e420   api-banksystem_microservice    "docker-entrypoint.s…"   18 minutes ago
a7781d61307a   postgres:14                    "docker-entrypoint.s…"   18 minutes ago
```

- docker stop <id_container>: detener un contenedor
- docker rm <id_container>: eliminar un contenedor
- docker start <id_container>: iniciar un contenedor

```
zguitarra@ubuntu20:~$ docker start 132f8af7e67f
132f8af7e67f
zguitarra@ubuntu20:~$ docker stop 132f8af7e67f
132f8af7e67f
zguitarra@ubuntu20:~$ docker rm 132f8af7e67f
132f8af7e67f
```

- docker volume ls:

  Muestra los volúmenes configurados.

```
zguitarra@ubuntu20:~$ docker volume ls
DRIVER    VOLUME NAME
local     api-banksystem_pgdata
```

- docker network ls:

  Muestra las redes

```
zguitarra@ubuntu20:~$ docker network ls
NETWORK ID     NAME                         DRIVER    SCOPE
6de2767cc569   api-banksystem_ms_network    bridge    local
5745a9eb7435   bridge                       bridge    local
dc33322a2abc   host                         host      local
082065d5b817   none                         null      local
```

- docker images:

  Muestra todas las imágenes creadas

```
zguitarra@ubuntu20:~$ docker images
REPOSITORY                  TAG       IMAGE ID       CREATED          SIZE
api-banksystem_microservice   latest    c9f8349aa1ac   28 minutes ago   1.46GB
postgres                    14        e270a11b9c8a   4 days ago       376MB
api-banksystem_nginx          latest    4d6153fa6561   6 days ago       23.4MB
```