



# **UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA**

*La Universidad Católica de Loja*

## **ÁREA TÉCNICA**

**TÍTULO DE INGENIERO EN SISTEMAS INFORMÁTICOS Y  
COMPUTACIÓN**

**Aplicación de un modelo para evaluar el rendimiento en el proceso de  
migración de una aplicación monolítica hacia una orientada a microservicios.**

**TRABAJO DE TITULACIÓN.**

**AUTORA:** Yaguachi Pereira, Lady Elizabeth

**DIRECTOR:** Guamán, Daniel Alejandro, Mgs.

**LOJA – ECUADOR**

**2017**



*Esta versión digital, ha sido acreditada bajo la licencia Creative Commons 4.0, CC BY-NY-SA: Reconocimiento-No comercial-Compartir igual; la cual permite copiar, distribuir y comunicar públicamente la obra, mientras se reconozca la autoría original, no se utilice con fines comerciales y se permiten obras derivadas, siempre que mantenga la misma licencia al ser divulgada. <http://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>*

2017

## **APROBACIÓN DEL DIRECTOR DEL TRABAJO DE TITULACIÓN**

Magister.

Daniel Alejandro Guamán Coronel.

**DOCENTE DE LA TITULACIÓN**

De mi consideración:

El presente trabajo de titulación: Aplicación de un modelo para evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios, realizado por Lady Elizabeth Yaguachi Pereira, ha sido orientado y revisado durante su ejecución, por cuanto se aprueba la presentación del mismo.

Loja, noviembre de 2017

f) .....

## DECLARACIÓN DE AUTORÍA Y CESIÓN DE DERECHOS

“Yo, Lady Elizabeth Yaguachi Pereira, declaro ser autor (a) del presente trabajo de titulación: Aplicación de un modelo para evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios, de la Titulación de Sistemas Informáticos y Computación, siendo el Mgs. Daniel Alejandro Guamán director (a) del presente trabajo; y eximo expresamente a la Universidad Técnica Particular de Loja y a sus representantes legales de posibles reclamos o acciones legales. Además certifico que las ideas, conceptos, procedimientos y resultados vertidos en el presente trabajo investigativo, son de mi exclusiva responsabilidad.

Adicionalmente declaro conocer y aceptar la disposición del Art. 88 del Estatuto Orgánico de la Universidad Técnica Particular de Loja que en su parte pertinente textualmente dice: “Forman parte del patrimonio de la Universidad la propiedad intelectual de investigaciones, trabajos científicos o técnicos y tesis de grado o trabajos de titulación que se realicen con el apoyo financiero, académico o institucional (operativo) de la Universidad”

f).....

Autora: Lady Elizabeth Yaguachi Pereira

Cédula. 1104620792

## DEDICATORIA

Dedico este trabajo de fin de titulación a mis padres Angel y Almida quienes han sido el sustento y guía durante toda mi vida y han sabido forjar en mí los valores necesarios para salir adelante en ella.

A mis hermanos Nixon y Jonathan, pilares fundamentales en mi crecimiento personal.

A mis compañeros de titulación, por ser partícipes y co-actores de mi formación universitaria.

**Lady Elizabeth Yaguachi Pereira.**

## **AGRADECIMIENTO**

Agradezco a Dios primeramente por guardar mi caminar y darme la fortaleza para seguir adelante a pesar de las adversidades.

A mis padres por el apoyo y paciencia depositados en mí durante estos cinco años de formación universitaria.

A mi director de trabajo de fin de titulación por la guía y los consejos impartidos a mi persona para llevar a feliz término este trabajo.

A mi hermano Jonathan por su ayuda y consejo constantes mientras desarrollé mi tesis.

A mis compañeros de aula, Nicholas y Jacqueline por compartir a más del conocimiento, una parte de su vida conmigo.

A todos ellos expreso mi más sincero agradecimiento y gratitud.

**Lady Elizabeth Yaguachi Pereira.**

## INDICE DE CONTENIDOS

APROBACIÓN DEL DIRECTOR DEL TRABAJO DE TITULACIÓN.....	ii
DECLARACIÓN DE AUTORÍA Y CESIÓN DE DERECHOS.....	iii
DEDICATORIA .....	iv
AGRADECIMIENTO .....	v
INDICE DE CONTENIDOS .....	vi
ÍNDICE DE FIGURAS.....	xi
INDICE DE TABLAS.....	xiii
RESUMEN.....	1
ABSTRACT .....	2
INTRODUCCIÓN.....	3
Objetivos .....	4
General.....	4
Específicos.....	4
GLOSARIO DE TÉRMINOS .....	5
CAPITULO I: ESTADO DEL ARTE .....	7
1    Estado del Arte .....	8
1.1    Microservicios .....	8
1.1.1    Definición de Microservicios. ....	8
1.1.2    Características de los Microservicios.....	9
1.1.3    Ventajas y Desventajas de los Microservicios.....	10
1.1.4    Funcionamiento de los Microservicios. ....	13
1.1.5    Rendimiento de los Microservicios.....	16
1.2    Servicios REST .....	17
1.2.1    Definición de Servicios REST. ....	18
1.2.2    Características de los Servicios REST.....	18

1.2.3	Ventajas y Desventajas de los Servicios REST. ....	21
1.2.4	Funcionamiento de los Servicios REST. ....	21
1.2.5	Rendimiento de los Servicios REST. ....	23
1.3	Monolitos .....	25
1.3.1	Definición de Monolitos.....	25
1.3.2	Características de los Monolitos. ....	26
1.3.3	Ventajas y Desventajas de los Monolitos. ....	26
1.3.4	Funcionamiento de los Monolitos.....	29
1.3.5	Rendimiento de los Monolitos.....	30
1.4	Microservicios VS Servicios REST VS Monolitos .....	31
1.4.1	Semejanzas.....	31
1.4.2	Diferencias. ....	32
1.5	Criterios de Evaluación de Arquitecturas de Software.....	34
1.5.1	Definición de Rendimiento. ....	34
1.5.2	Características de Rendimiento. ....	35
1.5.3	Factores que afectan al rendimiento de las aplicaciones. ....	36
1.5.4	Métricas de rendimiento en base a hardware. ....	38
1.5.5	Patrones de Diseño orientados al rendimiento.....	39
CAPITULO II: DISEÑO DE PROCESO DE MIGRACIÓN DE APLICACIÓN MONOLÍTICA A APLICACIÓN CON MICROSERVICIOS .....		43
2	Diseño de proceso de migración de aplicación monolítica a aplicación con microservicios	44
2.1	Consideraciones antes de la migración.....	44
2.1.1	Identificación de candidatos en el monolito.....	45
2.1.2	Asignar tamaño a los microservicios.....	46
2.1.3	Adoptar estrategias de Integración Continua y Entrega Continua. ....	47
2.1.4	Patrón Strangler. ....	47
2.1.5	Integración de DevOps con microservicios y contenedores. ....	48



2.2	Modelo de refactorización de IBM .....	50
2.2.1	Reempaquetado de la aplicación.....	50
2.2.2	Refactorización del código.....	51
2.2.3	Refactorización de los datos.....	52
2.3	Modelo de refactorización de NGINX .....	53
2.3.1	Estrategia 1: Stop Diggin. ....	54
2.3.2	Estrategia 2: Dividir Frontend de Backend.....	54
2.3.3	Estrategia 3: Extraer servicios. ....	55
2.4	Modelo de refactorización de Medium.....	55
2.4.1	Refactorización incremental.....	56
2.4.2	Monolito con API's.....	56
2.4.3	Nuevas funcionalidades.....	56
2.4.4	Contextos limitados. ....	57
2.4.5	IC/DC.....	57
2.5	Comparación de modelos de migración propuestos.....	57
2.6	Propuesta de migración de aplicación monolítica hacia una orientada a microservicios.	
	57	
2.6.1	Identificar funcionalidades en el monolito. ....	58
2.6.2	Definir API REST para funcionalidades del monolito. ....	59
2.6.3	Convertir funcionalidades en servicios autónomos. ....	60
CAPITULO III: DISEÑO E IMPLEMENTACIÓN DEL PROCESO DE MIGRACIÓN DE APLICACIÓN MONOLÍTICA HACIA APLICACIÓN ORIENTADA A MICROSERVICIOS .....		63
3	Diseño e Implementación del proceso de migración de aplicación monolítica hacia aplicación orientada a microservicios.....	64
3.1	Implementación de Monolito Versión 1.....	64
3.1.1	Diseño de Monolito Versión 1.....	64
3.1.2	Desarrollo de Monolito Versión 1.....	66
3.1.3	Despliegue de Monolito Versión 1. ....	67

3.2	Implementación de Monolito Versión 2.....	68
3.2.1	Diseño de Monolito Versión 2. ....	68
3.2.2	Desarrollo de Monolito Versión 2. ....	70
3.2.3	Despliegue de Monolito Versión 2. ....	71
3.3	Implementación de Servicios REST.....	72
3.3.1	Diseño de Servicios REST.....	72
3.3.2	Desarrollo de Servicios REST. ....	75
3.3.3	Despliegue de Servicios REST.....	78
3.4	Implementación de Aplicación orientada a Microservicios.....	79
3.4.1	Diseño de los Microservicios. ....	80
3.4.2	Desarrollo de Microservicios.....	84
3.4.3	Despliegue de MicroserviciosEn esta sección se explica el paso final que constituye a una arquitectura de microservicios la cual es el despliegue, para ello se explican los dos métodos de despliegue usados para los microservicios: Nativo (Bare Metal) y Contenedores (usando Docker).....	90
CAPITULO IV: PRUEBAS Y RESULTADOS .....		96
4	Pruebas y Resultados .....	97
4.1	Pruebas .....	97
4.1.1	Equipo de Prueba.....	97
4.1.2	Escenario de Prueba. ....	98
4.1.3	Herramienta de Medición: Apache JMeter.....	99
4.1.4	Pruebas Realizadas.....	99
4.2	Resultados.....	106
4.2.1	Por número de Clientes. ....	107
4.2.2	Por aplicación analizada.....	111
4.2.3	Por recurso analizado.....	121
CONCLUSIONES.....		128
RECOMENDACIONES.....		130

BIBLIOGRAFÍA.....	132
ANEXOS.....	136
Anexo A: Documento de Especificación de Requerimientos para SRCL .....	137
Anexo B: Modelo Entidad-Relación de SRCL.....	144
Anexo C: Despliegue de Monolito V1 .....	144
Anexo D: Despliegue de Monolito V2 .....	145
Anexo E: Despliegue de Servicios REST .....	145
Anexo F: Despliegue local de Eureka.....	146
Anexo G: Despliegue local de API REST matriculas .....	146
Anexo H: Despliegue Local de Zuul.....	146
Anexo I: Instalación de Docker en Ubuntu 16.04 .....	147
Anexo J: Instalación de Docker Compose en Ubuntu 16.04 .....	147
Anexo K: Configuración de Docker con la base de datos local.....	148
Anexo L: Archivo de configuración docker-compose.yml.....	149
Anexo M: Levantar contenedores con Docker Compose .....	150
Anexo N: Ver contenedores activos en Docker .....	151
Anexo O: Ver hosts de contenedores Docker.....	151
Anexo P: Despliegue de Eureka en contenedor Docker .....	152
Anexo Q: Despliegue de API REST Matrículas en contenedor Docker.....	152
Anexo R: Despliegue de Zuul en contenedor Docker .....	153
Anexo S: Instalación de JMeter .....	153
Anexo T: Definición de Términos de Métricas de Rendimiento en base a hardware...	154
Anexo U: Pruebas de rendimiento. Escenario: 10 Clientes.....	156
Anexo W: Pruebas de rendimiento. Escenario: 100 Clientes.....	158
Anexo X: Pruebas de rendimiento. Escenario: 1.000 Clientes .....	160

## ÍNDICE DE FIGURAS

Figura 1 Representación de Microservicios .....	10
Figura 2 Funcionamiento de Microservicios .....	15
Figura 3 Funcionamiento de Microservicios con API Gateway .....	16
Figura 4 Esquema REST .....	19
Figura 5 Funcionamiento de REST .....	24
Figura 6 Representación de Monolito .....	26
Figura 7 Funcionamiento de un Monolito .....	30
Figura 8 Métodos de Despliegue de Aplicaciones Web .....	37
Figura 9 Patrón Strangler.....	48
Figura 10 Esquema de Integración DevOps + Microservicios + Contenedores .....	50
Figura 11 Modelo de Refactorización de IBM .....	53
Figura 12 Modelo de refactorización de NGINX .....	56
Figura 13 Modelo de Refactorización de Medium .....	58
Figura 14 Diagrama de ruta de migración propuesta .....	61
Figura 15 Modelo de proceso de Ruta de Migración propuesta .....	62
Figura 16 Arquitectura de Monolito: Versión 1 .....	66
Figura 17 Organización de Monolito Versión 1.....	67
Figura 18 Diagrama de Componentes de Monolito Versión 2 .....	69
Figura 19 Diagrama de Despliegue de Monolito Versión 2 .....	69
Figura 20 Procedimientos Almacenados en base de datos SRCL .....	70
Figura 21 Organización del Monolito Versión 2.....	71
Figura 22 Ejemplo de solicitud y respuesta de un servicio RESTful.....	73
Figura 23 Arquitectura de Servicios REST en SRCL.....	75
Figura 24 Implementación de servicio: Matriculas por Curso .....	77
Figura 25 Implementación de Singleton mediante codificación .....	78
Figura 26 Organización de Servicios REST .....	79
Figura 27 Propuesta de implementación de Arquitectura de Microservicios.....	83
Figura 28 Gestión de solicitudes HTTP en arquitectura de Microservicios .....	85
Figura 29 Configuración de Eureka en Spring .....	85
Figura 30 Configuración de API REST en Spring.....	87
Figura 31 Organización del API REST matrículas.....	88
Figura 32 Configuración de Zuul en Spring.....	89
Figura 33 Configuración de perfil "container" para API REST obtener_matriculas .....	91

Figura 34 Creación de Grupo de hilos en JMeter.....	101
Figura 35 Configuración de Listener en JMeter.....	102
Figura 36 Reporte: Gráfico de Resultados en JMeter .....	103
Figura 37 Reporte: Resumen en JMeter .....	104
Figura 38 Reporte: PerfMon Metrics Collector en JMeter.....	105
Figura 39 Reporte: Árbol de Resultados en JMeter .....	106
Figura 40 Rendimiento con 10 Clientes .....	108
Figura 41 Rendimiento con 100 Clientes .....	110
Figura 42 Rendimiento con 1.000 Clientes .....	112
Figura 43 Rendimiento en Monolito Versión 1 .....	113
Figura 44 Rendimiento en Monolito Versión 2 .....	114
Figura 45 Rendimiento en Servicios REST (con patrones de diseño) .....	116
Figura 46 Patrón Singleton en Servicios REST (con patrones de diseño).....	116
Figura 47 Rendimiento en Servicios REST (sin patrones de diseño) .....	118
Figura 48 Rendimiento en Microservicios (con patrones de diseño) .....	119
Figura 49 Rendimiento en Microservicios (sin patrones de diseño).....	121
Figura 50 Rendimiento del CPU vs Aplicaciones analizadas .....	122
Figura 51 Rendimiento de Memoria VS Aplicaciones analizadas.....	123
Figura 52 Rendimiento de la red VS Aplicaciones desarrolladas .....	125
Figura 53 Rendimiento de la Base de Datos VS Aplicaciones desarrolladas .....	127
Figura 54. Caso de uso para SRCL .....	143
Figura 55 Modelo E-R de SRCL .....	144
Figura 56. Funcionalidad: Consultar matriculas de un curso en Monolito V1 .....	144
Figura 57 Funcionalidad: Consultar matriculas de un curso en Monolito V2 .....	145
Figura 58 Despliegue de servicio: Obtener Matrículas .....	145
Figura 59 Despliegue local de Eureka .....	146
Figura 60 Despliegue local de API REST matrículas .....	146
Figura 61 Despliegue local de Zuul.....	146
Figura 62 Levantar contenedores con Docker Compose .....	150
Figura 63 Ver contenedores activos en Docker.....	151
Figura 64 Ver hosts de contenedores Docker .....	151
Figura 65 Despliegue de Eureka en contenedor Docker .....	152
Figura 66 Despliegue de API REST Matrículas en contenedor Docker .....	152
Figura 67 Despliegue de Zuul en contenedor Docker .....	153

## INDICE DE TABLAS

Tabla 1 Características de Microservicios .....	11
Tabla 2 Ventajas y Desventajas de los Microservicios .....	14
Tabla 3 Características de los Servicios .....	19
Tabla 4 Ventajas y Desventajas de los servicios REST .....	22
Tabla 5 Características de los Monolitos.....	27
Tabla 6 Ventajas y Desventajas de los Monolitos .....	28
Tabla 7 Diferencias entre Monolitos, Servicios REST y Microservicios .....	32
Tabla 8 Métricas de rendimiento en base a Hardware .....	40
Tabla 9 Patrones de diseño relacionados con el Rendimiento .....	41
Tabla 10 Comparación de modelos de refactorización.....	59
Tabla 11 Funcionalidades identificadas en SRCL .....	65
Tabla 12 Características del Monolito Versión 1 .....	67
Tabla 13 Características del Monolito Versión 2 .....	72
Tabla 14 Requisitos de Servicios Web de SRCL .....	73
Tabla 15 Identificadores RESTful para SRCL .....	74
Tabla 16 Características de Servicios REST implementados.....	80
Tabla 17 Características de la arquitectura de MS implementada.....	95
Tabla 18 Características del equipo de pruebas .....	97
Tabla 19 Solicitudes HTTP por cada aplicación analizada .....	100
Tabla 20 Análisis de rendimiento con 10 Clientes.....	108
Tabla 21 Análisis del rendimiento con 100 Clientes .....	109
Tabla 22 Análisis del rendimiento con 1.000 Clientes .....	111
Tabla 23 Análisis de rendimiento para Monolito Versión 1 .....	113
Tabla 24 Análisis de rendimiento en Monolito Versión 2 .....	114
Tabla 25 Análisis de rendimiento en Servicios REST (con patrones de diseño).....	115
Tabla 26 Análisis del rendimiento en Servicios REST (sin patrones de diseño) .....	117
Tabla 27 Análisis del rendimiento en Microservicios (con patrones de diseño) .....	118
Tabla 28 Análisis de rendimiento en Microservicios (sin patrones de diseño) .....	120
Tabla 29 Análisis del rendimiento del CPU VS Aplicaciones desarrolladas.....	121
Tabla 30 Análisis de rendimiento de Memoria VS Aplicaciones desarrolladas .....	123
Tabla 31 Análisis de rendimiento de la red VS Aplicaciones desarrolladas.....	124
Tabla 32 Análisis de rendimiento de la Base de Datos VS Aplicaciones desarrolladas.....	126

Tabla 33 Definición de Términos de Métricas de Rendimiento en base a Hardware .....	154
Tabla 34 Pruebas de Rendimiento con 10 Clientes .....	156
Tabla 35 Pruebas de Rendimiento con 100 Clientes .....	158
Tabla 36 Pruebas de Rendimiento con 1000 Clientes.....	160

## RESUMEN

El presente trabajo de fin de titulación consiste en la aplicación de un modelo que permita evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios. Este trabajo se sustenta en la base investigativa que indica el uso de arquitecturas orientadas a microservicios como una alternativa a las arquitecturas tradicionales, también llamadas monolíticas. De este modo, cualquier aplicación web/ móvil en la que se desee usar una arquitectura de microservicios debe pasar por un proceso de migración que involucre la implementación de arquitecturas orientadas a servicios como REST.

El otro aspecto considerado en este trabajo es la medición del atributo de calidad de rendimiento en cada una de las aplicaciones construidas con las arquitecturas involucradas en la migración durante la implementación de cada una de ellas, por lo que también se incluye el uso de patrones de diseño orientados al rendimiento, con el fin de determinar cuál arquitectura presenta un mejor desempeño de rendimiento en cuanto a los recursos que resultan afectados durante el uso de las aplicaciones: CPU, memoria, red y base de datos.

**PALABRAS CLAVES:** arquitectura, microservicios, migración, monolitos, rendimiento, REST



## **ABSTRACT**

This undergraduate thesis consists in the application of a model that allows to evaluate the performance during the process of migration from a monolithic application towards one oriented to microservices. This work is based on the research which indicates the use of architectures oriented to microservices as an alternative to traditional architectures, also called monolithic. So, any web/mobile application which want to use a microservice architecture must go through a migration process that involves the implementation of service-oriented architectures such as REST.

The other aspect considered in this work is the measurement of the performance quality attribute in each one of the applications built with the architectures involved in the migration during the implementation of each of them so, the use of performance-oriented design patterns is also included, in order to determine which architecture has a better performance in terms of resources that are affected during the use of these applications: CPU, memory, network and database.

**KEYWORDS:** architecture, microservices, migration, monolith, performance, REST

## INTRODUCCIÓN

En los últimos años la arquitectura de Microservicios ha ganado mucha popularidad en el desarrollo de software, esta se considera como un refinamiento y simplificación de la arquitectura orientada a servicios (SOA). Es un enfoque para el desarrollo de una sola aplicación como un conjunto de servicios pequeños, donde cada uno se ejecuta en su propio proceso y se comunican con mecanismos ligeros, a menudo un recurso HTTP. Estos microservicios difieren extensamente de las arquitecturas monolíticas donde todos los servicios se desarrollan en una sola base de código, La complejidad aparece a medida que más servicios son añadidos, lo que limita la escalabilidad y el rendimiento de las aplicaciones y consecuentemente la capacidad de las empresas a innovar con nuevas versiones y características. (Lewis & Fowler, 2014).

El enfoque de microservicios responde a las necesidades de escalabilidad de las aplicaciones de negocio. Sin embargo, hay que tomar en cuenta que actualmente todas las aplicaciones aún mantienen un modelo de arquitectura monolítica (p.e 3 capas), según lo afirma (Villamizar et al., 2015) en su trabajo por lo que las empresas que deseen utilizar el enfoque de microservicios se enfrentan a la necesidad de una ruta o modelo que permita la migración exitosa desde una arquitectura monolítica hacia una orientada a microservicios y aún más importante, asegurar el rendimiento durante este proceso, ya que escalabilidad y rendimiento van de la mano, por lo general, si se mejora el rendimiento de una aplicación suele mejorar su escalabilidad.

Lo que se pretende con el presente proyecto es realizar un proceso de búsqueda, análisis y aplicación de un modelo que permita analizar y evaluar el impacto del rendimiento como atributo de calidad que se suscita durante la migración de una aplicación monolítica hacia una orientada a microservicios, para llevar a cabo este trabajo se ha organizado el trabajo de la siguiente manera:

- En el Capítulo I se realizará el proceso de búsqueda, análisis y clasificación bibliográfica referente a las arquitecturas monolíticas, orientadas a servicios y microservicios así como la identificación de modelos, patrones de diseño, métricas de hardware que permitan evaluar el rendimiento tanto en arquitecturas monolíticas como en microservicios para posteriormente construir el modelo más idóneo y aplicable al presente trabajo de titulación.

- Con la información recolectada del Capítulo I se procederá a diseñar el proceso o ruta para migrar desde un monolito hacia microservicios y evaluar el rendimiento de cada arquitectura implicada durante este proceso.
- El Capítulo III contiene la implementación de la ruta de migración propuesta lo que implica, la adaptación del modelo para evaluar el rendimiento mientras se migra desde un monolito hacia una arquitectura de microservicios.
- Finalmente, en el Capítulo IV se procederá a la emisión de resultados correspondiente a las pruebas de rendimiento aplicadas a cada arquitectura en base al modelo establecido para este fin, lo que permitirá concluir qué arquitectura presenta un mejor desempeño respecto al atributo de calidad de rendimiento.

## **Objetivos**

### **General**

- Aplicar un modelo que permita analizar y evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios.

### **Específicos**

- Comparar las ventajas y desventajas entre arquitecturas monolíticas y de microservicios.
- Establecer un enfoque que permita migrar una aplicación monolítica a una con microservicios.
- Identificar un modelo que permita medir el rendimiento en los microservicios.
- Utilizar el modelo seleccionado para medir el rendimiento en una aplicación que migra desde un monolito a una arquitectura orientada a microservicios.

## GLOSARIO DE TÉRMINOS

**MS.-** Término abreviado para referirse al estilo arquitectónico de Microservicios o a un microservicio.

**HTTP.-** En inglés: Hypertext Transfer Protocol. Es el protocolo de comunicación que permite la transferencia de información en la World Wide Web.

**API.-** En inglés: Application Programming Interface. Conjunto de subrutinas, funciones y procedimientos que ofrece un software para ser utilizado por otro.

**REST.-** En inglés: Representational State Transfer. Es un estilo arquitectónico para sistemas distribuidos como la World Wide Web.

**RESTful.-** Cualquier API construida con las reglas básicas que indica REST.

**SOA.-** En inglés: Service Oriented Architecture. Es una arquitectura donde todas las funciones están definidas como servicios independientes con interfaces invocables.

**ESB.-** En inglés: Enterprise Service Bus. Es una infraestructura de software que proporciona servicios de integración entre distintas aplicaciones a través de mensajería basada en estándares y servicios de sincronización.

**Código 200 OK.-** Código de Respuesta estándar para peticiones HTTP correctas.

**Código 500.-** Código emitido cuando se dan situaciones de error ajenas a la naturaleza del servidor web.

**Código 404.-** Código emitido cuando el servidor web no encuentra la página o recurso solicitado.

**JSON.-** Es un formato de texto ligero para el intercambio de datos.

**XML.-** Es un meta-lenguaje utilizado para almacenar datos en forma legible.

**GUI.-** Interfaz Gráfica de Usuario.

**CI/CD.-** Entrega continua y Desarrollo continuo. Enfoques de ingeniería de software en el que grupos de desarrollo producen software en ciclos de vida cortos donde se asegura la liberación continua del software.

**YAML.-** Es un formato de serialización de datos legible por humanos inspirado en lenguajes como XML.

**JAR.-** En inglés: Java Archive. Es un tipo de archivo que permite ejecutar aplicaciones Java.

**WAR.-** En inglés: Web Application Archive. Es un archivo JAR utilizado para distribuir los componentes de una aplicación.

**EAR.-** Es un formato para empaquetar en un sólo archivo varios módulos.

**SRCL.-** Sistema de Registro de Cursos en Línea. Es el nombre que recibe la aplicación que se analizó.

**IDE.-** Es un entorno de desarrollo integrado, es decir, consiste en un editor de código, un compilador, un depurador y un GUI.

**SGBD.-** Conjunto de programas que permiten el almacenamiento, modificación y extracción de la información en una base de datos además de proporcionar operaciones propias de la misma.

**Datasource.-** Origen de datos en el que se configura las conexiones hacia una fuente de datos física.

## **CAPITULO I: ESTADO DEL ARTE**

## **1 Estado del Arte**

En este capítulo se presentan los conceptos básicos a cerca de arquitecturas monolíticas, orientadas a servicios y microservicios las cuales se han considerado relevantes dentro de un proceso de migración como el que se indica en los objetivos de este trabajo de fin de titulación. Se establece una comparación entre las arquitecturas mencionadas, resaltando las principales ventajas y desventajas de cada una. También se define el atributo de calidad de rendimiento, haciendo énfasis en los factores que influyen en el mismo dentro de cada arquitectura propuesta y las métricas de hardware y software que permiten evaluarlo.

### **1.1 Microservicios**

“Microservicios” (MS) es un término al que varios autores definen de manera diferente: un estilo arquitectónico (Lewis & Fowler, 2014), un enfoque (Namiot & Sneps-Sneppe, 2014), una arquitectura (Dragoni et al., 2016). Sin embargo en lo que todos estos autores coinciden es que los microservicios representan una mejora y alternativa a las aplicaciones con arquitecturas tradicionales (p.e.n-capas), en las siguientes líneas se presenta un concepto general de los Microservicios, sus características, funcionamiento, ventajas y desventajas.

#### **1.1.1 Definición de Microservicios.**

Para (Lewis & Fowler, 2014), quienes son considerados como los padres de los microservicios, este estilo arquitectónico consiste en un enfoque para desarrollar una aplicación entera como un conjunto de servicios pequeños. Cada uno de estos servicios ejecuta sus propios procesos y se comunica mediante mecanismos ligeros como HTTP. Estos servicios se construyen en base a las capacidades empresariales y se pueden implementar y gestionar de manera independiente ya que pueden construirse con diferentes lenguajes de programación y utilizar diferentes tecnologías de almacenamiento de datos.

Una apreciación similar es la de (Dragoni et al., 2016), en la que considera a los microservicios como componentes independientes conceptualmente desplegados en aislamiento y equipados con herramientas de persistencia de memoria dedicada. Así, en los componentes de una arquitectura de microservicios, su comportamiento se deriva de la composición y coordinación de sus componentes vía mensajes, para facilitarlos, el estilo de arquitectura de microservicios no favorece ni prohíbe un paradigma particular de programación. Esto provee una guía para particionar los componentes de una aplicación distribuida en entidades independientes, cada una en dirección a un área específica del negocio, lo que significa que un microservicio puede

ser internamente implementado con cualquier lenguaje de programación mientras este ofrezca funcionalidades vía paso de mensajes.

Acorde a la afirmación anterior, (Hasselbring, 2016) mantiene que los microservicios enfatizan en la coordinación con menos transacciones entre servicios con la aceptación explícita de la consistencia eventual. Dichos microservicios se construyen alrededor de las capacidades del negocio por lo tanto llevan una implementación completa de software basada en estas.

Las definiciones de (Dragoni et al., 2016; Hasselbring, 2016; Lewis & Fowler, 2014) recalcan en que los microservicios permiten diseñar una aplicación entera como un conjunto de servicios pequeños, los cuales se implementan independientemente. Dichos servicios son elegidos y construidos en base a las necesidades de negocio.

Cada microservicio es diseñado para representar una funcionalidad específica de la aplicación construida con una arquitectura tradicional como 3 capas, de esta manera, cada microservicio puede alojarse en un servidor diferente, puede o no gestionar una base de datos y por lo tanto puede ser invocado de manera independiente. A continuación, en la Figura 1 se representa, de manera general una arquitectura de microservicios.

### **1.1.2 Características de los Microservicios.**

Tal como se mencionó anteriormente, los microservicios contemplan a una aplicación como una suite de pequeños servicios independientes tanto en desarrollo como en despliegue.

Respecto al origen de los microservicios, (Dragoni et al., 2016) menciona que SOA es el antecesor de los microservicios, de hecho, en un inicio éstos eran llamados “SOA de grano fino”, por lo que tienen características en común, no obstante a día de hoy los microservicios son la nueva tendencia en arquitectura de software. Éstos manejan la complejidad creciente al descomponer funcionalmente grandes sistemas en un conjunto de servicios independientes, llevando la modularidad al siguiente nivel.

Una descripción similar a la anterior es la que mencionan (Savchenko & Radchenko, 2015):

“Cada uno de estos servicios se comunica con otros servicios usando mecanismos ligeros como HTTP; estos servicios son construidos alrededor de las capacidades del



negocio, independientemente desplegables y pueden ser escritos por diferentes equipos de desarrollo usando diferentes lenguajes de programación y frameworks.”

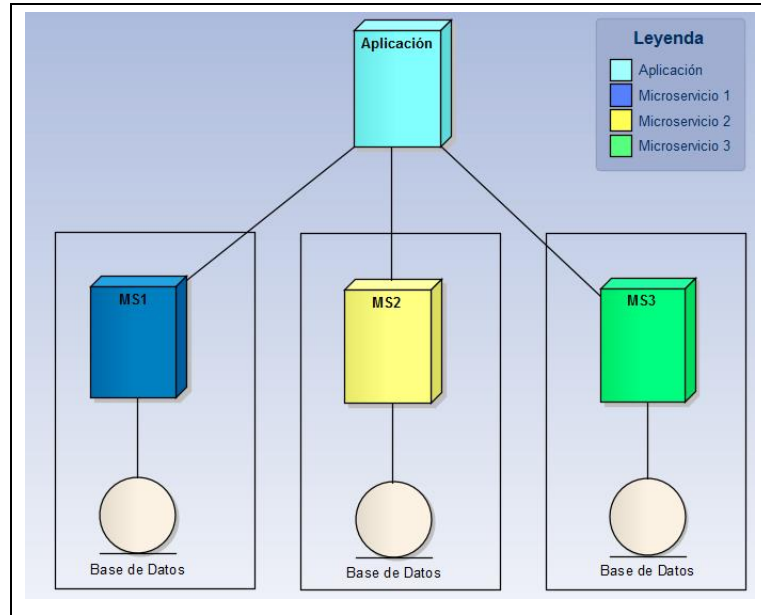


Figura 1 Representación de Microservicios

Fuente: La Autora

Elaboración: La Autora

En este contexto, (Lewis & Fowler, 2014), en su trabajo dejan por sentado las características básicas que debe tener cualquier arquitectura de software que use microservicios tales como la componentización vía servicios, la organización en base a las capacidades del negocio y la gestión de datos descentralizada. A continuación, en la Tabla 1 se mencionan las características principales de los microservicios según los autores citados anteriormente.

### 1.1.3 Ventajas y Desventajas de los Microservicios.

Dentro de los microservicios, la idea clave es lograr una gran cantidad de beneficios al crear muchos servicios independientes que trabajen juntos en armonía, por supuesto este enfoque tiene su propio conjunto de inconvenientes.

En la Tabla 2 se mencionan las ventajas y desventajas que implicarían aplicar un enfoque de microservicios, según lo mencionan (Amaral & Carrera, 2015) y (Namiot & Snep-Sneppe, 2014) en sus respectivos trabajos.

Tabla 1 Características de Microservicios

CARACTERÍSTICA	DESCRIPCIÓN
<b>Según (Dragoni et al., 2016)</b>	
<b>Tamaño</b>	Si un servicio es demasiado grande, éste se divide en dos o más servicios con el fin de preservar la granularidad y mantener el principio que un microservicio debe proporcionar una sola capacidad de negocio.
<b>Contexto limitado</b>	Las funcionalidades relacionadas se combinan en una única capacidad de negocio, que se implementa como un servicio
<b>Independencia</b>	Cada microservicio es operacionalmente independiente de otros microservicios. La única forma de comunicación entre ellos es a través de sus interfaces públicas.
<b>Flexibilidad</b>	Capaz de mantenerse al día con el entorno empresarial en constante cambio y soporta modificaciones necesarias para organizaciones que desean mantenerse competitivas en el mercado.
<b>Modularidad</b>	Un sistema está compuesto de componentes aislados donde cada uno contribuye al desenvolvimiento del sistema en lugar de tener un solo componente que ofrece la funcionalidad completa.
<b>Evolución</b>	Preserva la mantenibilidad mientras evoluciona constantemente y agrega nuevas características.
<b>Según (Savchenko &amp; Radchenko, 2015)</b>	
<b>Interfaz abierta</b>	Los microservicios proveen una descripción abierta de la interfaz y la comunicación en formato mensaje, esto también aplica para las API o la GUI.
<b>Especialización</b>	Cada microservicio provee un soporte para una parte independiente de la lógica del negocio de la aplicación.
<b>Contenerización</b>	Dentro del aislamiento de los microservicios en ambiente de ejecución, esta puede darse en un ambiente de virtualización mediante contenedores.

<b>Autonomía</b>	Los microservicios pueden ser desarrollados, probados, desplegados, destruidos, movidos o duplicados independientemente y automáticamente. La interacción continua es la única opción para lograr un acuerdo con el desarrollo y la complejidad de despliegue.
<b>Según (Lewis &amp; Fowler, 2014)</b>	
<b>Componentización vía Servicios</b>	Los servicios son desplegados de forma independiente por lo que resulta más fácil definir interfaces públicas explícitas.
<b>Organización</b>	La aplicación es dividida en servicios organizados alrededor de las capacidades de negocio, esto es: interfaz de usuario, almacenamiento persistente y cualquier colaboración externa.
<b>Enfoque de Producto</b>	En lugar de mirar al software como un conjunto de funcionalidades, se tiene la noción de que un equipo de desarrollo posee un producto durante toda su vida útil, éste los hace responsables del comportamiento del producto en producción y del soporte del mismo.
<b>Puntos finales inteligentes - Tubos mudos</b>	Los microservicios ofrecen un enfoque de comunicación alternativo: puntos finales inteligentes - tubos mudos (Smart End Points – Dumb pipes). En la práctica, esto significa que aplicación construida con microservicios recibe una solicitud, aplica la lógica según sea apropiado y produce una respuesta. Esta aplicación puede ser coreografiada usando protocolos REST simples. A parte de este, los protocolos más utilizados para la comunicación son solicitud-respuesta HTTP con recursos API y mensajes ligeros.
<b>Gobernanza no centralizada</b>	Los microservicios no estandarizan tecnologías ni normas sobre la construcción de cada microservicio, en lugar de eso, se comparten herramientas que otros desarrolladores pueden utilizar para resolver problemas similares.
<b>Gestión de datos no centralizada</b>	Además de descentralizar las decisiones sobre modelos conceptuales, los microservicios también descentralizan las decisiones de almacenamiento de datos. Los microservicios enfatizan en la coordinación sin transacciones entre servicios a sabiendas que la consistencia es sólo eventual y los problemas se tratan mediante operaciones de compensación.

<b>Infraestructura automatizada</b>	Actualmente se usan técnicas de automatización de infraestructura como la nube. Estos reducen la complejidad operacional de la construcción, despliegue y funcionamiento de microservicios.
<b>Diseñado para el error</b>	Los microservicios pueden fallar en cualquier momento. Los fallos se detectan rápidamente y se restaura automáticamente el servicio. Las aplicaciones de microservicios ponen mucho énfasis en el monitoreo en tiempo real de la aplicación, comprobando variables arquitectónicas (p.e. Nro. de solicitudes/segundo). Éste monitoreo proporciona un sistema de alerta temprana cuando algo va mal para que los equipos de desarrollo lo solucionen.
<b>Diseño evolutivo</b>	Los microservicios tienen un trasfondo evolutivo al ver la descomposición de un servicio como una herramienta que permite a los desarrolladores controlar los cambios en su aplicación sin ralentizar el cambio. Se trata más bien de conducir la modularidad a través del patrón de cambio. Con las herramientas adecuadas se pueden hacer cambios frecuentes, rápidos y bien controlados a los servicios.

Fuente: (Dragoni et al., 2016; Lewis & Fowler, 2014; Savchenko & Radchenko, 2015)

Elaboración: La Autora

#### 1.1.4 Funcionamiento de los Microservicios.

(Namiot & Sneps-Sneppe, 2014), menciona en su trabajo que existen varias maneras de habilitar la comunicación para una aplicación que use microservicios, estas pueden ser:

1. La aplicación usa los servicios directamente, pero no es factible ya que se pueden dar retrasos potenciales debido a las llamadas a procedimientos remotos.
2. Utilizar un balanceador de carga entre la aplicación y cada uno de los servicios para aminorar la carga de llamadas remotas.
3. Un tercer enfoque consiste en usar un bus de servicio, de esta manera el bus permite a la aplicación hacer solicitudes y más tarde leer la respuesta. Este enfoque es usado por SOA.

No obstante, el esquema más usado es el segundo. Una aplicación que use microservicios consistirá, por un lado, en la Aplicación de software la cual está compuesta por recursos HTML o Javascript, desde la que se hacen solicitudes usando mecanismos de comunicación ligeros,

normalmente HTTP. Esta solicitud es gestionada por un balanceador de carga que redirige la petición hacia el servicio correspondiente, éstos se mantienen aislados y pueden o no tener persistencia, es decir, manejar una base de datos.

Tabla 2 Ventajas y Desventajas de los Microservicios

<b>VENTAJAS</b>	<b>DESVENTAJAS</b>
Técnicamente, se reduce la complejidad de la aplicación al usar pequeños servicios.	En la práctica, el enfoque de microservicios significa para los desarrolladores complejidad adicional al crear un sistema distribuido.
Permiten escalar, eliminar y desplegar partes del sistema fácilmente.	Más que una desventaja, un reto de los microservicios es implementar el mecanismo de comunicación entre servicios.
Mejora la flexibilidad al darse la libertad para usar diferentes frameworks y herramientas.	Múltiples servicios requieren de una fuerte coordinación por parte del equipo(s) de desarrolladores. Realizar pruebas es más dificultoso en sistemas distribuidos.
Incrementa la escalabilidad y la resiliencia en todas direcciones.	Se requiere un mecanismo de despliegue automatizado, también llamado integración continua (IC).
Bases de código mucho más simples para servicios individuales.	Sobrecarga computacional al correr una aplicación en diferentes procesos.
Capacidad de actualizar y escalar los servicios en aislamiento	El enfoque de microservicios conlleva un alto consumo de memoria, debido a los propios espacios de dirección para cada servicio.
Habilita los servicios para ser escritos en diferentes lenguajes si así se desea.	Es un reto dividir el sistema en microservicios, así mismo decidir los límites para un servicio y determinar cuándo un servicio es demasiado grande.
Utilizar middleware variado e incluso niveles de datos para los diferentes servicios.	Costos de comunicación de red más elevados que simplemente hacer llamadas a funciones en un proceso.

Fuente: (Amaral & Carrera, 2015; Namiot & Sneys-Snepe, 2014)

Elaboración: La Autora

Cada microservicio puede estar alojado individualmente o en grupo dentro de servidores o contenedores y la comunicación se mantiene bidireccional en todo momento para contestar a las solicitudes hechas desde la aplicación. En la Figura 2 se representa el funcionamiento de una aplicación que utiliza microservicios, en base a las especificaciones dadas anteriormente.

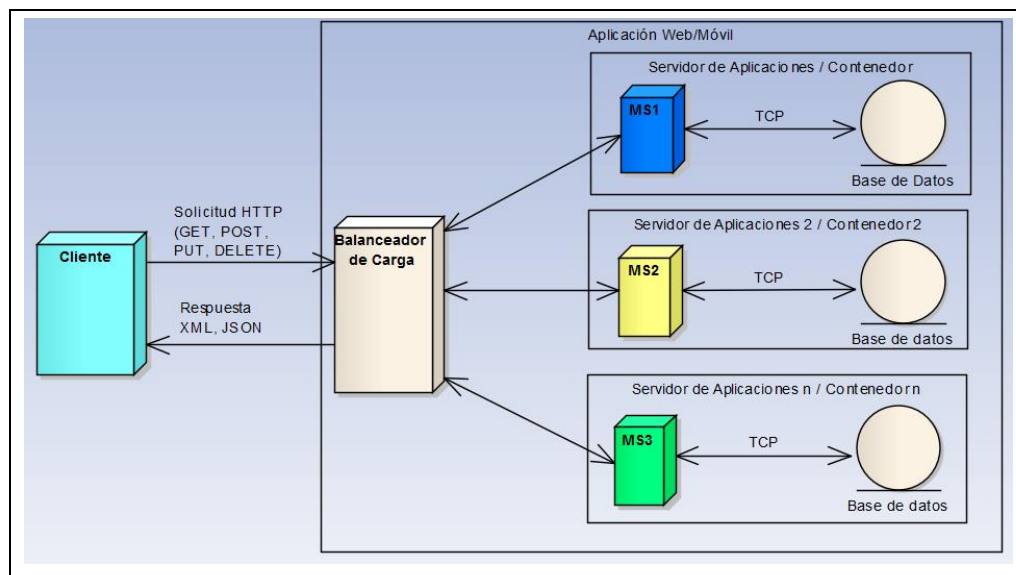


Figura 2 Funcionamiento de Microservicios

Fuente: (Namiot & Sneps-Sneppe, 2014)

Elaboración: La Autora

#### 1.1.4.1 API Gateway.

Otra forma de implementar una arquitectura de microservicios es la planteada (Santis, Florez, Nguyen, & Rosa, 2016), en este trabajo, los autores plantean el patrón API Gateway para mejorar la comunicación entre los microservicios, específicamente consiste en implementar una puerta de enlace (API Gateway) para gestionar las diferentes solicitudes que llegan al cliente y encaminarlas a los microservicios competentes a la solicitud.

A efectos prácticos, el patrón API Gateway consiste en la creación de un nivel medio para proveer interfaces adicionales que integren los microservicios. Este nivel medio se basa en un API Gateway que se ubica estratégicamente entre clientes y microservicios, esta proporciona una interfaz simplificada a los clientes, al facilitar el uso, comprensión y prueba de los servicios. En este escenario, API Gateway reduce las interacciones entre el cliente y los servicios al unificar varias solicitudes en una sola solicitud, y de vuelta, consolida la respuesta y la envía al

cliente por lo que a nivel de rendimiento, el cliente experimentará baja latencia de red. En la Figura 3 se representa el funcionamiento de API Gateway en una arquitectura de microservicios.

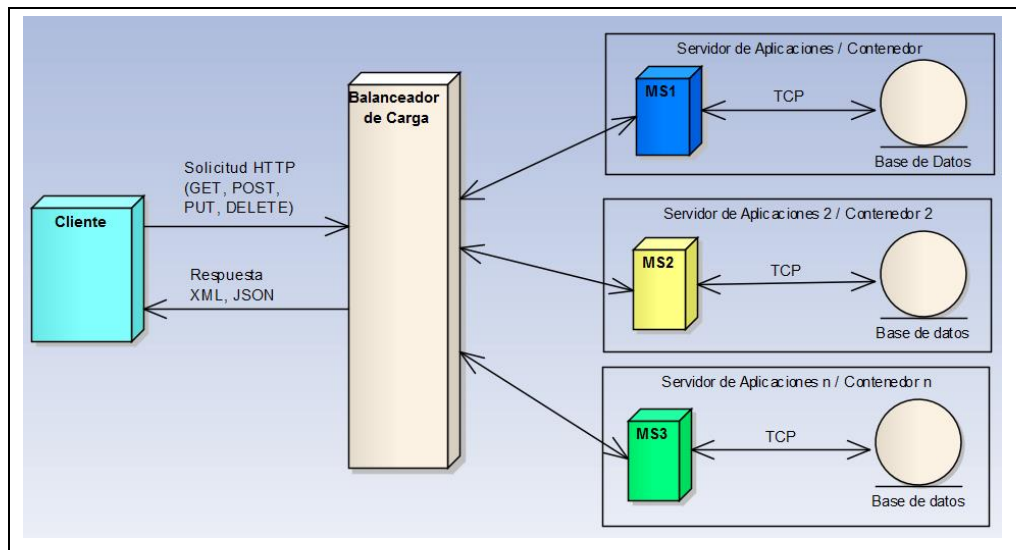


Figura 3 Funcionamiento de Microservicios con API Gateway

Fuente: (Santis et al., 2016)

Elaboración: La Autora

### 1.1.5 Rendimiento de los Microservicios.

Dado que una arquitectura de microservicios requiere de la interacción continua entre los servicios que conforman la aplicación, es necesario tener en cuenta el impacto que puede tener en el rendimiento de la misma, esto es, a nivel de transacción y red.

Tal como indica (Knoche, 2016) “Los Microservicios son una arquitectura prometedora para la modernización del software monolítico. Sin embargo, el romper un monolito en pequeños servicios puede tener un grave impacto en el rendimiento, especialmente en las transacciones.”(p.121) A efectos prácticos, “estas transacciones requieren el uso de múltiples servicios lo que puede causar cuellos de botella en cuanto al rendimiento” (Merkle & Knoche, 2015).

(Hasselbring, 2016) menciona en su trabajo que los microservicios permiten la elasticidad rápida y automatizada. Los mecanismos de tolerancia a fallos logran que los errores de los microservicios individuales no afecten a otros servicios gracias al aislamiento existente entre

ellos. Dado que los servicios pueden fallar en cualquier momento es importante ser capaz de detectar los errores rápidamente y, de la misma forma, reiniciarlos, por lo que es esencial para el éxito plantear ajustes para un monitoreo avanzado que evalúen variables de rendimiento, (por ejemplo, el número de solicitudes por segundo).

(Dragoni et al., 2016) señala que el factor prominente que impacta negativamente al rendimiento en los microservicios es la comunicación sobre la red. La latencia de la red es superior que la de memoria, esto significa que las llamadas a memoria son mucho más rápidas de completar que enviar mensajes sobre la red. Por lo tanto, en términos de comunicación, el rendimiento se degradará comparado con las aplicaciones que usan mecanismos de llamadas in-memory. Las restricciones que los microservicios ponen al tamaño asignado para un servicio también contribuyen directamente a este factor. De manera general los sistemas con contextos bien delimitados experimentan menos degradación en cuanto al rendimiento debido al acoplamiento y una menor cantidad de mensajes enviados.

Con los antecedentes mencionados, se puede concluir que el rendimiento juega un papel importante al aplicar una arquitectura orientada a microservicios, ya que el mismo estará presente en áreas específicas como las transacciones y la red, donde más se suscitan inconvenientes, por lo que es necesario prever y establecer mecanismos que permitan conocer el impacto que tendrá al rendimiento implementar microservicios en una aplicación y en el caso de implementarse, mecanismos de tolerancia a fallos que la habiliten a recuperarse de los errores de comunicación y seguir funcionando.

## **1.2 Servicios REST**

Los servicios web hacen referencia a sistemas de software diseñados para interactuar sobre la red, usando principalmente Internet (W3C, 2017). Actualmente los estilos arquitectónicos más usados para implementar servicios web son la arquitectura SOA y REST. SOA permite implementar servicios vía mensajes y REST mediante recursos con estado, emulando operaciones estándar del protocolo de HTTP, esto lo convierte en la opción más fácil y estandarizada de implementar. A continuación se detalla el concepto y las características generales de REST.



### **1.2.1 Definición de Servicios REST.**

(Costa, Pires, Delicato, & Merson, 2016) menciona que REST fue propuesto originalmente por Roy Fielding como un estilo arquitectónico de software para sistemas distribuidos, especialmente aplicaciones web, móviles y API's públicas, quien además definió las restricciones con respecto al diseño de REST: una interfaz uniforme, sin estado, y dentro de una arquitectura cliente-servidor. Hoy en día REST también es usado al incorporar Tecnologías de la Información (TI) como un mecanismo de comunicación entre las aplicaciones de una empresa.

(Zou, Mei, & Wang, 2010) considera que el estilo REST que planteó Fielding fue originalmente diseñado para apoyar los requisitos de alto rendimiento y escalabilidad del entorno hipermedia. Sin embargo, debido a su simplicidad y escalabilidad, se ha convertido en una alternativa a los Servicios Web basados en SOAP para la construcción de la Arquitectura Orientada a Servicios (SOA). REST se comporta como una máquina de estado virtual, donde la transición de un estado ocurre cuando el usuario selecciona enlaces, lo que da como resultado el siguiente estado de la aplicación que se transfiere al usuario.

(Zhang, Wen, Wu, & Zou, 2011) define a REST (Representational State Transfer) como un estilo de arquitectura de software para sistemas hipermedia distribuidos como la World Wide Web (WWW). Un servicio web REST es un servicio web simple implementado utilizando HTTP y los principios de REST. Se trata de una colección de recursos, con tres aspectos definidos:

- El URI de base para el servicio web, (p.e: <http://my.com/resources/xx>)
- El tipo MIME de los datos soportados por el servicio web. A menudo se usa JSON, XML o YAML, entre otros de tipo MIME.
- El conjunto de operaciones soportado por el servicio web utilizando métodos HTTP (por ejemplo, POST, GET, PUT o DELETE).

A continuación, en la Figura 4 se muestra el esquema inicial de los servicios REST.

### **1.2.2 Características de los Servicios REST.**

En el trabajo de (Dragoni et al., 2016) se habla del enfoque de Computación Orientada a Servicios (Service-Oriented Computing) como un paradigma emergente propio de la computación distribuida o del procesamiento de negocios electrónicos, bajo este paradigma, un

programa denominado servicio ofrece funcionalidades a otros componentes, accesibles a través de paso de mensajes.

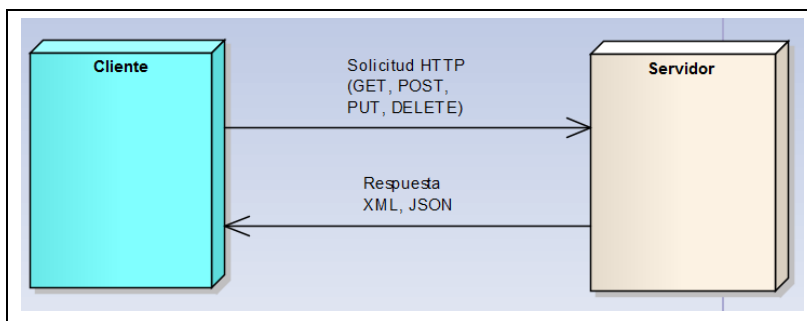


Figura 4 Esquema REST

Fuente: La Autora

Elaboración: La Autora

No obstante, tal como indica (Zhang et al., 2011), los servicios REST tienen su propio conjunto de principios y restricciones, es decir, en qué proyectos de software son aplicables. A continuación en la Tabla 3 se listan las características de los servicios REST mencionadas en los trabajos de (Zou et al., 2010) y (Zhang et al., 2011) respectivamente.

Tabla 3 Características de los Servicios

CARACTERÍSTICA	DESCRIPCIÓN
<b>Arquitectura Cliente-Servidor</b>	En REST, las solicitudes son iniciadas por usuarios agentes (Clientes) y finalmente procesadas por el Servidor el cual provee servicios a través de una jerarquía de recursos. REST intenta desacoplar la relación cliente y servidor, sin embargo, dentro de las arquitecturas basadas en REST, es necesario definir que componentes actuarán desde el cliente y cuales desde el servidor.
<b>Sin estado</b>	Toda la información que se necesita para procesar los datos de estado dentro de una solicitud se origina en el Servidor y la información de los Clientes debe ser incluida en los mensajes solicitud/ respuesta. Esto significa que el estado de los mensajes es guardado por el Servidor y mantenido, actualizado y comunicado por el Cliente.
<b>Caché</b>	Un elemento de cache actúa como un mediador entre Cliente y Servidor, con el objetivo de evitar interacciones solicitud-respuesta cuando la información aún está en la cache, para evitar el tráfico de red. Obviamente, en soluciones de arquitecturas en capas, la cache también puede estar ubicada en los niveles

	intermedios.
<b>Interfaz Uniforme</b>	La característica central que distingue a REST de otros estilos arquitectónicos es una interfaz uniforme entre componentes. La interfaz está directamente relacionada con los recursos, identificadores y representaciones. Un recurso es cualquier concepto importante en el dominio del sistema que se desea hacer accesible a través de una interfaz uniforme. Un cliente busca recursos a través de un URI y una representación de recurso, la cual puede ser un documento hipermedia (HTTP), este trae cualquier información importante acerca del estado del recurso y lo relaciona a otros recursos asociados. Finalmente la representación de un recurso deberá ser accedida desde una interfaz simple que defina una identificación para el recurso y métodos comunes para acceder.
<b>Sistema en capas</b>	En arquitecturas n-capas, los servicios REST a menudo se ubican en niveles de lado del Servidor.
<b>Visión de Recursos</b>	<p>La característica clave del estilo arquitectónico REST es que maneja una "visión de recursos" del mundo. Los principios REST relacionados a esta visión son:</p> <ul style="list-style-type: none"> <li>• El recurso puede ser identificado por un URI (Identificador de recurso único).</li> <li>• Separación del recurso abstracto y sus representaciones concretas.</li> <li>• La interacción es sin estado, cada interacción contiene toda la información de contexto y metadatos necesarios.</li> <li>• Solo se necesita un pequeño conjunto de operaciones, con semántica distinta basada en métodos HTTP: Operaciones seguras (GET, HEAD, OPTIONS, TRACE); Operaciones no seguras, idempotentes (PUT, DELETE); y operaciones no seguras, no idempotentes (POST).</li> <li>• Definir operaciones de idempotencia y metadatos de representación.</li> <li>• Promover la presencia de intermediarios como proxies, gateways o filtros para alterar o restringir las solicitudes y respuestas basada en metadatos.</li> </ul> <p>Las restricciones de REST pueden ser aplicadas usando el protocolo HTTP y tecnologías relacionadas para implementar Servicios Web que son llamados Servicios RESTful.</p>

Fuente: (Zhang et al., 2011; Zou et al., 2010)

Elaboración: La Autora

### 1.2.3 Ventajas y Desventajas de los Servicios REST.

Independientemente de la estrategia utilizada para implementar servicios, un enfoque orientado a servicios ofrece varias ventajas (Dragoni et al., 2016):

- **Dinamismo:** Se pueden lanzar nuevas instancias de un mismo servicio para dividir la carga en el sistema.
- **Modularidad y reutilización:** Los servicios complejos se componen de servicios más sencillos y los mismos servicios pueden ser utilizados por diferentes sistemas.
- **Desarrollo distribuido:** Al acordar las interfaces del sistema distribuido, los distintos equipos de desarrollo pueden particionar el sistema en paralelo.
- **Integración de sistemas heterogéneos y legados:** Los servicios solo deben implementar protocolos estándares para comunicarse.

Como menciona (Zou et al., 2010), los servicios REST son aplicables a arquitecturas en capas donde los servicios representan una capa más de la arquitectura. A diferencia de otras arquitecturas como SOA que operan vía mensajes u operaciones, en REST las peticiones son procesadas vía recursos que no tienen estado con el protocolo HTTP, lo que hace a REST muy ligero, sin sobrecargas a los recursos. Sin embargo hay algunos aspectos que aún deben ser gestionados por REST como la seguridad y la recuperación ante errores de comunicación. En la Tabla 4 se resumen las principales ventajas y desventajas identificadas en el estilo arquitectónico REST.

### 1.2.4 Funcionamiento de los Servicios REST.

En las características mencionadas dentro de REST se mencionaba que este posee una visión de recursos del mundo, además las interacciones entre cliente y servidor se consideran “sin estado”, es decir las interacciones extraerán los metadatos necesarios de cada recurso, esto habilita al sistema para una rápida recuperación ante fallos.

Dentro del entorno de REST existen tres conceptos importantes (Feng, Shen, & Fan, 2009) :

- **Recurso:** Un recurso es cualquier objeto físico o concepto abstracto, siempre y cuando ese objeto o concepto sea lo suficientemente importante como para referirse como una cosa en sí, entonces puede ser expuesto como un recurso. Dentro de un sistema se considera recurso a los datos y la funcionalidad del mismo.
- **Representación:** Es cualquier información útil sobre el estado de un recurso. Un recurso puede tener múltiples representaciones diferentes.

- **Estado:** En REST hay dos tipos de estado. El estado de recurso es la información sobre un recurso y estado de aplicación es la información sobre la ruta que el cliente ha tomado a través de la aplicación. El estado del recurso permanece en el servidor y el estado de la aplicación permanece solo en el cliente.

Tabla 4 Ventajas y Desventajas de los servicios REST

VENTAJAS	DESVENTAJAS
Al ser un estilo arquitectónico sin estado, cada petición es tratada de manera independiente, la información de estado no se mantiene en el servidor lo que significa que no es utilizada en peticiones anteriores ni posteriores, lo que aumenta las posibilidades de escalar la aplicación.	La seguridad en los servicios REST se gestionan a través del Firewall de HTTP, pero al no estar debidamente especificada, es un problema y puede llegar a ser una tarea muy difícil de implementar.
Los servicios REST proporcionan una buena infraestructura de almacenamiento en caché debido al uso del método GET de HTTP con el fin de evitar interacciones de estado innecesarias	Pese a que REST usa el estándar MIME para representar los datos entendibles para HTTP, no se definen tipos de datos para las operaciones de solicitud y respuesta por lo que, de darse un problema de comunicación resulta difícil lidiar con el problema.
REST emplea un método sencillo de intercambio de datos al utilizar el protocolo HTTP en el intercambio de datos, sin usar estándares adicionales.	
REST no depende de ninguna plataforma y disminuye la carga útil del mensaje, por consiguiente mejora el uso de recursos del sistema.	

Fuente: (Hamad, Saad, & Abed, 2010)

Elaboración: La Autora

En REST el servidor es visto como un conjunto de recursos. El cliente interactúa con el servidor a través de una interfaz uniforme y durante la interacción “sin estado” el servidor y el cliente intercambian representaciones de recursos, por lo tanto el cliente consume los servicios que se encuentran en el servidor siguiendo los enlaces proporcionados por las URI’s (Uniform

Resource Identifiers) para obtener estados representacionales de los recursos que solicita (Costa et al., 2016).

REST se implementa con el protocolo HTTP, este localiza un recurso de manera exclusiva y permite operar sobre el recurso. Las solicitudes y respuestas se realizan a través de cuatro operaciones HTTP (Hamad et al., 2010):

- **GET:** Recupera el estado actual de un recurso en alguna representación.
- **PUT:** Crea un nuevo recurso.
- **POST:** Transfiere un nuevo estado a un recurso.
- **DELETE:** Elimina un recurso existente.

Cada uno de estos métodos tiene una semántica diferente para decidir qué método es adecuado para cada recurso. Una solicitud GET no cambiará el estado del servidor. La interfaz uniforme actúa de intermediario entre cliente y servidor y permite que ambos evolucionen independientemente, mientras la interfaz permanezca sin cambios el cliente y el servidor pueden interactuar normalmente. Las interacciones, al ser sin estado limitan la demanda de que la solicitud de cada cliente deba contener todos los estados de la aplicación necesarios para comprender esa solicitud. Ninguna información de estado se mantiene en el servidor y ninguna de ella está implícita en solicitudes anteriores. A continuación, en la Figura 5 se representa el funcionamiento de REST.

### **1.2.5 Rendimiento de los Servicios REST.**

La idea base de los servicios REST es intercambiar representaciones de recursos utilizando una interfaz unificada y un protocolo de comunicación sin estado: HTTP. A priori, estos principios alientan a las aplicaciones REST a ser simples, ligeras y de alto rendimiento.

REST ha sido ampliamente utilizado para integrar servicios y aplicaciones. Su adopción como variante de SOA trae varios beneficios, pero también plantea nuevos retos y riesgos. Según (Costa et al., 2016), los principales atributos de calidad afectados al implementar REST son la seguridad, fiabilidad y rendimiento.

Tal como indica (Costa et al., 2016), REST se construye principalmente sobre una arquitectura Cliente-Servidor, la cual dentro de sus implementaciones puede tener n-capas. En este caso, una capa actúa como servidor del nivel previo y como cliente del nivel subsiguiente. De ser así,

los elementos en el nivel intermedio deben ser gestionados para mejorar la escalabilidad y la tasa de rendimiento del sistema. Sin embargo, desplegar componentes a través de algunas capas puede incrementar el tiempo de respuesta para una solicitud, debido a los multisaltos que deben hacerse entre las capas mientras se procesa una solicitud HTTP.

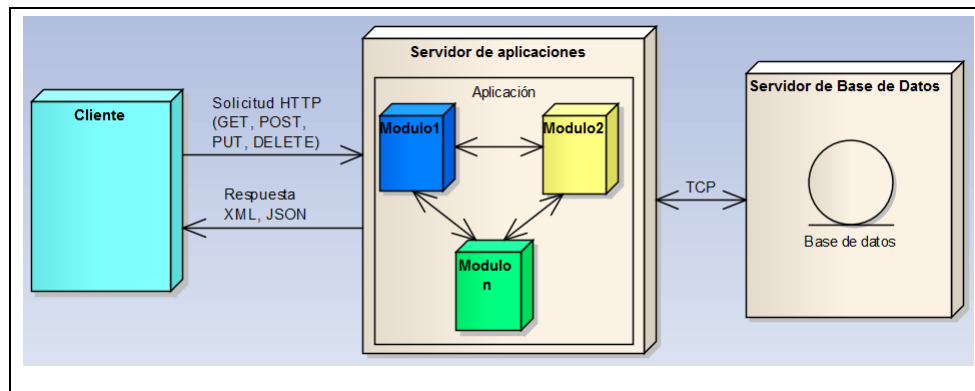


Figura 5 Funcionamiento de REST

Fuente: La Autora

Elaboración: La Autora

A más del percance mencionado, (Hamad et al., 2010) menciona que la restricción de sin estado promueve la disponibilidad, escalabilidad y confiabilidad mediante la replicación de servidores de carga equilibrada pero esta misma característica disminuye el rendimiento debido a la necesidad de enviar datos de estado conversacional embebidos en los mensajes de solicitud y respuesta. Para paliar con este problema se añade la restricción de caché que mejora el rendimiento. En este caso un elemento de caché actúa como mediador entre Cliente y Servidor. El objetivo es evitar una interacción solicitud-respuesta cuando la información está presente en la caché, para evitar el tráfico en la red. En una solución n-capas, la caché puede ubicarse en los niveles intermedios. El grado en que la caché incremente la eficiencia de la red y su rendimiento depende de la estrategia de cache usada.

(Feng et al., 2009), en cambio, considera al rendimiento como una ventaja de REST. Desde el punto de vista de infraestructura e implementación, el rendimiento en REST es mejor debido a su simplicidad inherente. REST se basa en estándares existentes ampliamente utilizados en la Web y no requiere estándares adicionales, lo que evita la dependencia de alguna plataforma significativa especial y por lo tanto disminuye la ocupación de recursos del sistema. Un ejemplo básico de esto podría ser la comparación entre REST y SOAP. REST utiliza directamente el

protocolo HTTP en el intercambio de datos lo que reduce la carga útil del mensaje, mientras SOAP debe analizar y empaquetar sus paquetes de lado del cliente y el servidor. REST también mejora el rendimiento al promover que tanto el cliente como las capas intermediarias analicen las respuestas que el servidor marcan como almacenables en caché y así eliminar cualquier interacción innecesaria para un mejor rendimiento.

### 1.3 Monolitos

El enfoque contrario a los microservicios son los monolitos, a priori, los monolitos son los antecesores de los microservicios, de hecho es el enfoque más usado actualmente ya que permite desplegar una aplicación como un todo, es decir ofrece una solución general para un problema lo que puede resultar un beneficio y una desventaja a la vez. A continuación, se presenta la definición de Monolitos, sus características, funcionamiento, ventajas y desventajas.

#### 1.3.1 Definición de Monolitos.

Para (Lewis & Fowler, 2014) un monolito es una aplicación construida como una sola unidad, normalmente dividida en 3 partes: Una interfaz de usuario del lado del cliente, una base de datos y una aplicación del lado del servidor, en este último se concentra toda la lógica de dominio en un solo ejecutable. La tarea más difícil de un monolito es la actualización de una o varias funcionalidades del servidor ya que hacerlo significa actualizar toda la aplicación.

Una apreciación similar es la de (Namiot & Sneps-Sneppe, 2014), en su trabajo definen a un monolito como una solución unificada que al ser invocada desde un cliente ejecuta todos los módulos que forman parte de su construcción, por ejemplo, una aplicación monolítica común consiste en un solo archivo ejecutable que por lo general, tiene una base de datos común para todos los módulos de la aplicación que posee internamente.

Para (Richardson & Smith, 2016), una aplicación monolítica típica consiste en al menos tres capas:

- **Presentación:** Son componentes que gestionan solicitudes HTTP e implementan una interfaz de usuario sofisticada, el nivel de presentación es a menudo un cuerpo sustancial de código.
- **Lógica de negocio:** Componentes que son el núcleo de la aplicación e implementan las reglas de negocio.



- **Acceso a datos:** Componentes que tienen acceso a componentes de infraestructura, como bases de datos e intermediarios de mensajes.

En base a los conceptos mencionados, en la Figura 6 se representa una arquitectura monolítica.

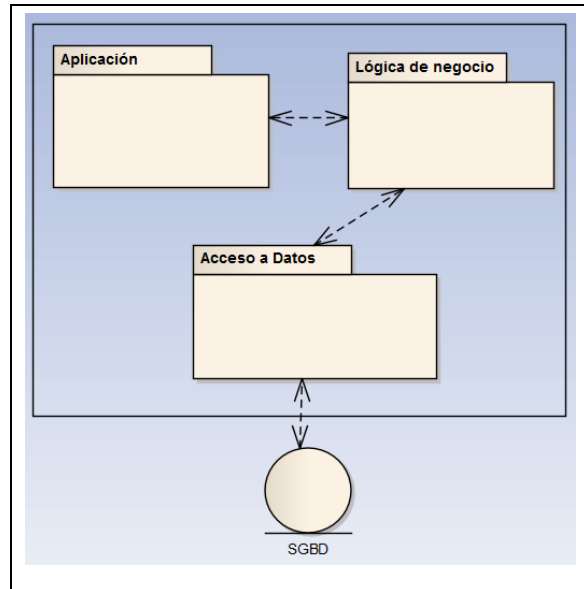


Figura 6 Representación de Monolito

Fuente: La Autora

Elaboración: La Autora

### 1.3.2 Características de los Monolitos.

Existen algunas premisas bajo las cuales la solución monolítica es la más adecuada, sin embargo la mayoría de los beneficios que ofrece esta solución tarde o temprano se ven afectados por lo que inicialmente parece ser su principal ventaja. En la Tabla 5 se muestran las características principales de los monolitos, consideradas en los trabajos de (Namiot & Sneps-Snepe, 2014) y (Dragoni et al., 2016).

### 1.3.3 Ventajas y Desventajas de los Monolitos.

En base a las características de monolitos mencionadas la principal característica de un monolito es la unificación de los módulos que lo conforman en una sola aplicación que se ejecuta como un solo programa, esta característica puede representar algunas limitantes.

Tabla 5 Características de los Monolitos

CARACTERÍSTICA	DESCRIPCIÓN
<b>Según (Namiot &amp; Sneps-Sneppe, 2014)</b>	
<b>Desarrollo fácil, actualización difícil</b>	Se sigue un enfoque arquitectónico básico desarrollado en capas. Para escalar, se corren múltiples copias de la aplicación. Sin embargo, un monolito convierte las actualizaciones frecuentes en obstáculos ya que para actualizar un pequeño servicio, es necesario redespregar toda la aplicación.
<b>Escalabilidad limitada</b>	Las aplicaciones desarrolladas con una arquitectura monolítica solo pueden escalar en una sola dimensión, para escalar se corren copias de la aplicación sin embargo resulta imposible escalar un solo servicio.
<b>Según (Dragoni et al., 2016)</b>	
<b>Comportamiento inesperado</b>	La adición o actualización de librerías en un monolito provoca un comportamiento inconsistente en el sistema: no compila, no se ejecuta o, peor aún, se comporta de manera inexplicable. En monolitos de gran tamaño, reiniciar toda la aplicación significa tiempos de caída del sistema notables, se dificulta el desarrollo, las pruebas y el mantenimiento del proyecto.
<b>Despliegue sub-óptimo</b>	El conflicto entre los recursos que necesita cada módulo provoca que el despliegue esté por debajo de lo esperado, cada módulo puede necesitar de un recurso en particular: acceso intensivo a memoria u otros recursos de computación intensiva. Para elegir un entorno de despliegue, el desarrollador está limitado a elegir uno que vaya acorde a una configuración de tamaño único que sea costosa o subestima con respecto a los módulos individuales.

Fuente: (Dragoni et al., 2016; Namiot & Sneps-Sneppe, 2014)

Elaboración: La Autora,

En la Tabla 6, se mencionan las ventajas y desventajas de mantener una aplicación con arquitectura monolítica, según lo mencionan (Villamizar et al., 2015) y (Dragoni et al., 2016) en sus respectivos trabajos.

Tabla 6 Ventajas y Desventajas de los Monolitos

VENTAJAS	DESVENTAJAS
<p>Posee un enfoque típico de despliegue en capas, lo común: Presentación – Lógica de Negocio – Persistencia.</p>	<p>El desarrollo de aplicaciones de monolitos es menos óptimo debido a los requerimientos conflictivos de los recursos de conforman los módulos, (p.e memoria intensiva y otros recursos computacionales intensivos).</p>
<p>Los monolitos se basan en la modularización. Una aplicación entera es dividida en módulos, estos a su vez coordinan y dependen el uno del otro y conforman una solución que se ejecuta de forma unificada.</p>	<p>Los monolitos son difíciles de mantener y evolucionar debido a su complejidad. Seguir la pista de los errores que se dan, requieren de grandes búsquedas en la base de código.</p>
<p>Todo el equipo debe coordinar el desarrollo y los esfuerzos de re-despliegue de la aplicación.</p>	<p>Al hacer adiciones o actualizaciones de código o librerías, éstas resultan en sistemas inconsistentes que no compilan, no corren o peor aún, se comportan anormalmente. Con la aplicación en crecimiento es difícil añadir nuevos desarrolladores, o reemplazar miembros que dejan el equipo.</p>
<p>Si el tamaño de la aplicación no es demasiado extenso, es más fácil desarrollar, ya que se mantiene una sola base de código con una sola tecnología a usar.</p>	<p>Los monolitos también representan un bloqueo en cuanto al uso de tecnologías para los desarrolladores, los cuales están limitados a usar el mismo lenguaje y frameworks de la aplicación original.</p>
<p>La ruta para la escalabilidad es clara, consiste en correr múltiples copias de la aplicación tras un balanceador de carga.</p>	<p>Los monolitos limitan la escalabilidad, la estrategia normal para gestionar el incremento en las solicitudes al servidor consiste en crear nuevas instancias de la misma aplicación y dividir la carga entre estas instancias.</p>
	<p>Cualquier cambio en un módulo de un monolito requiere reiniciar toda la aplicación. Para proyectos a gran escala, reiniciar normalmente implica tiempos ociosos, desarrollo engorroso, testeo y el mantenimiento del proyecto.</p>

Fuente: (Dragoni et al., 2016; Villamizar et al., 2015)

Elaboración: La Autora

Según lo expuesto, se puede determinar que la principal ventaja de un monolito es la facilidad para desarrollar una aplicación ya que se cuenta con un enfoque típico de construcción y en un inicio, un solo equipo de desarrollo que trabaja sobre una misma tecnología resulta beneficioso, no obstante, esta misma ventaja puede perjudicar la escalabilidad y consecuentemente al rendimiento, ya que mientras más escala el monolito más difícil se vuelve procesar las múltiples solicitudes entrantes, esto sin duda retarda el tiempo de respuesta del sistema, el cual es característica esencial del rendimiento.

#### 1.3.4 Funcionamiento de los Monolitos.

En el trabajo de (Santis et al., 2016) se menciona que las arquitecturas y patrones en capas son el resultado de una evolución en el diseño de sistemas, dentro de una aplicación con arquitectura en capas, cada capa puede evolucionar separadamente no obstante, esta evolución estará habilitada solo en una dimensión: el aspecto técnico de la arquitectura, implementar cambios al dominio de negocio resulta difícil ya que estos dependerán de los requerimientos que por lo general afectan a diferentes módulos o capas del sistema.

En este contexto, (Villamizar et al., 2015) respalda la idea que las aplicaciones monolíticas son construidas bajo arquitecturas basadas en capas con patrones relacionados, por ejemplo MVC (Modelo – Vista – Controlador).

En el trabajo de (Lewis & Fowler, 2014) se plantea el esquema de una típica arquitectura dividida en capas y se define el funcionamiento de cada capa:

- **Presentación:** Representa la interfaz de usuario del lado del cliente, en una aplicación web consistirá en varias páginas HTML y Javascript que se ejecutan desde un navegador en la máquina del usuario.
- **Persistencia:** Se refiere a la capa de persistencia, es decir, la gestión de acceso a la base de datos, ésta consta de muchas tablas insertadas en una base de datos común para todo el sistema a la cual se accede desde la capa de negocio.
- **Negocio:** Es una aplicación que está del lado del servidor, se encarga de manejar las solicitudes HTTP, ejecutar la lógica de dominio empresarial, recuperar y actualizar los datos de la base de datos, seleccionar. Ésta aplicación en sí representa un monolito ya que es un solo ejecutable lógico y cualquier cambio en el sistema implica crear e implementar una nueva versión de esta aplicación del lado del servidor.

(Lewis & Fowler, 2014) también menciona que la característica general de un monolito es que toda la lógica para gestionar una solicitud se ejecuta dentro de un solo proceso, para hacerlo, se utilizan funciones básicas del lenguaje de programación que se esté usando para dividir la aplicación en clases, funciones y espacios de nombres (namespaces). La escalabilidad se da solo horizontalmente, es decir el monolito ejecuta muchas instancias detrás de un balanceador de carga, esta solución inicialmente tiene éxito pero con el tiempo se vuelve insostenible ya que los ciclos de cambio a un módulo del monolito requieren que este sea reconstruido y desplegado. A menudo es difícil mantener una buena estructura modular, lo que hace más difícil mantener los cambios que sólo deberían afectar a un módulo. Escalar un módulo requiere escalar toda la aplicación en lugar de solo aquellas partes que requieren mayores recursos. En la Figura 7 se muestra el funcionamiento de un monolito con las características dadas previamente.

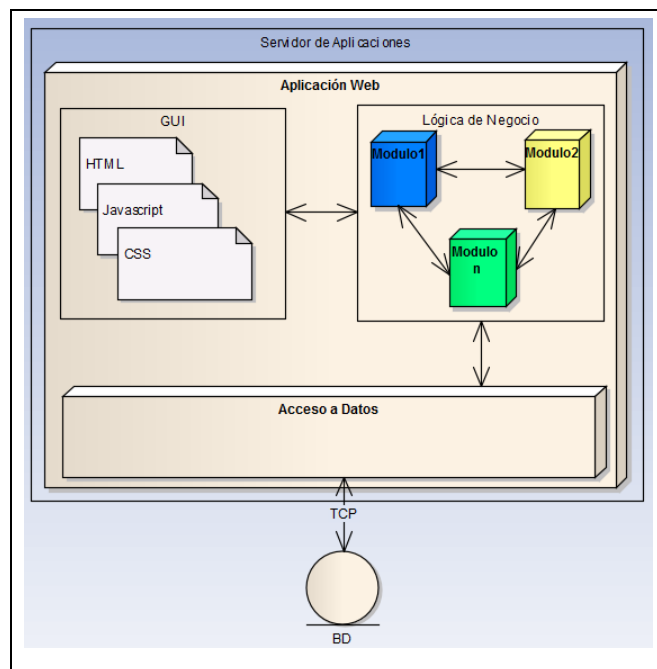


Figura 7 Funcionamiento de un Monolito

Fuente: (Lewis & Fowler, 2014)

Elaboración: La Autora

### 1.3.5 Rendimiento de los Monolitos.

El principal problema en cuanto al rendimiento de un monolito va de la mano con la escalabilidad, como indica (Lewis & Fowler, 2014), los monolitos solo permiten escalamiento

horizontal, es decir creando tantas instancias de la aplicación como sea necesario, esto sin duda deteriora el rendimiento de la misma.

(Knoche, 2016) menciona en su trabajo que al pretender dividir un monolito en servicios, el equipo de desarrollo puede enfrentarse a un impacto severo en el rendimiento, este suceso afectará principalmente a las transacciones ya que dividir un monolito implica separar también la persistencia de la aplicación (bases de datos).

En cuanto al rendimiento de la red, esta representa una ventaja frente los microservicios, según lo indican (Lewis & Fowler, 2014), ya que en un monolito la comunicación entre los módulos requiere solo de llamadas a funciones dentro del mismo proceso, no así los microservicios que deben hacer llamadas a procedimientos remotos las cuales acaban deteriorando la red.

Tal como se ha indicado, las aplicaciones desarrolladas con una arquitectura monolítica, con el pasar del tiempo presentan inconsistencias que conducen a la pérdida de rendimiento, como la necesidad de redespargar copias enteras de la aplicación en cuestión, cuando las copias actuales no alcanzan a satisfacer la carga de trabajo, esto prácticamente imposibilita que la aplicación pueda escalar cuando el volumen de datos incrementa exponencialmente.

## **1.4 Microservicios VS Servicios REST VS Monolitos**

Puesto que se han revisado los conceptos principales tanto de servicios, microservicios y monolitos, se pueden establecer claras semejanzas y diferencias entre estos tres estilos arquitectónicos que a día de hoy son los más usados en el medio empresarial.

### **1.4.1 Semejanzas.**

Aunque son estilos arquitectónicos diferentes, existen algunas características similares entre servicios, microservicios y monolitos, a saber:

- Son las arquitecturas de software más usadas actualmente para grandes aplicaciones de negocio, pese a que los microservicios están posicionándose cada vez en el mercado mientras decae el uso arquitecturas monolíticas.
- De manera conceptual, dividen la complejidad que conlleva construir una aplicación entera al trocearla en unidades funcionales llamadas “módulos” en una arquitectura monolítica, “servicios” al usar REST y “microservicios” en una arquitectura orientada a microservicios.

- Aunque existen patrones para decidir los límites de un módulo, servicio o microservicio, la mayoría de veces se realiza tomando como base las capacidades del negocio. Esta responsabilidad recae sobre el o los equipo(s) desarrollador(es).
- Los tres estilos arquitectónicos implementan mecanismos de tolerancia a fallos, cuando una funcionalidad tiene un comportamiento inesperado.

#### 1.4.2 Diferencias.

Para representar las diferencias marcadas entre una arquitectura monolítica, el enfoque orientado a servicios y microservicios se ha representado en la Tabla 7.

Tabla 7 Diferencias entre Monolitos, Servicios REST y Microservicios

<b>MONOLITOS</b>	<b>SERVICIOS</b>	<b>MICROSERVICIOS</b>
Existe un único ejecutable llamado “monolito” que contiene toda la lógica de la aplicación. Los módulos dentro de éste dependen entre sí.	Se mantiene el punto de vista de los monolitos.	Cada microservicio es operacionalmente independiente y aislado de otros microservicios.
La construcción de un monolito es visto como un proyecto.	Se mantiene el punto de vista de los monolitos.	En una arquitectura de microservicios, cada servicio es visto como un producto por el que el equipo que lo desarrolló es responsable.
Un monolito no soporta cambios o actualizaciones, al hacerlo es necesario redespigar toda la aplicación, la mayoría de veces la adición o actualización de librerías provoca un comportamiento inesperado.	Al estar implementados en arquitecturas monolíticas, los servicios REST involucrados necesitaran redespigarse al igual que en los monolitos.	Cada microservicio puede ser modificado individualmente por lo que cualquier cambio requiere redespigar solo el servicio en cuestión.

<p>El mecanismo de comunicación dentro de un monolito puede ser tan complejo como WS-Choreography o BPEL o el uso de ESB (SOA).</p>	<p>Se utiliza operaciones propias del protocolo HTTP (GET, POST, PUT, DELETE) y un formato de datos común (XML o JSON)</p>	<p>Utiliza un mecanismo de comunicación ligero HTTP usando protocolos como REST.</p>
<p>Un monolito maneja una sola base de datos compartida para toda la aplicación.</p>	<p>En cuanto al almacenamiento de datos, los servicios REST adoptan las prácticas de los monolitos.</p>	<p>Los microservicios descentralizan las decisiones de almacenamiento de datos. Se puede tener una base de datos por cada servicio que lo requiera.</p>
<p>La adición de un nuevo módulo obliga a usar la tecnología con la que se construyó originalmente el monolito.</p>	<p>La implementación de servicios REST es independiente de la tecnología a usarse, debido a que la comunicación entre servicios es vía HTTP, un protocolo sin estado.</p>	<p>Los microservicios no estandarizan tecnologías de uso ni normas de construcción para los mismos.</p>
<p>Al encontrarse toda la lógica de la aplicación en un solo ejecutable, para acceder a los módulos se utilizan llamados a funciones.</p>	<p>La comunicación consiste en intercambiar representaciones de recursos mediante una interfaz única y un protocolo estandarizado, por lo general HTTP.</p>	<p>Cada servicio funciona de manera independiente por lo que para comunicarse se necesitan llamadas a procedimientos remotos (RPC) lo que aumenta el nivel de complejidad.</p>
<p>Para la construcción de un monolito se establece un único equipo de desarrolladores.</p>	<p>Al construir la aplicación con servicios REST, por lo general se adoptan las prácticas de una aplicación monolítica</p>	<p>El ideal de los microservicios es establecer varios equipos de desarrolladores, uno por cada servicio.</p>
<p>La asignación de recursos nunca resulta ser equitativa o</p>	<p>El uso de los recursos está ligado a la estrategia de caché a</p>	<p>Al trabajar como unidades independientes, cada</p>



suficiente debido a que cada módulo necesita recursos diferentes.	usarse entre las capas de la aplicación.	microservicio gestiona sus propios recursos.
Sólo se permite escalamiento horizontal mediante instancias de la aplicación entera.	El escalamiento al usar servicios REST dependerá de la arquitectura en uso, al ser una arquitectura Cliente-Servidor (Monolítica) el escalamiento es horizontal.	Los microservicios permiten escalamiento horizontal (datos) y vertical (servicios).

Fuente: (Dragoni et al., 2016; Hamad et al., 2010; Lewis & Fowler, 2014)

Elaboración: La Autora

## 1.5 Criterios de Evaluación de Arquitecturas de Software

Para evaluar las arquitecturas utilizadas en el desarrollo de las aplicaciones generalmente se realiza a través de atributos de calidad de software (p.e seguridad, escalabilidad, rendimiento, etc.), para este trabajo se ha propuesto evaluar el rendimiento por cada arquitectura que participe en el proceso de migración de un monolito hasta microservicios por lo que se ha considerado relevante incluir las características más importantes de este atributo de calidad.

El rendimiento es un concepto importante dentro de cualquier arquitectura de software ya que permite evaluar aspectos de la misma como los tiempos de respuesta y la tasa de rendimiento, necesarios para asegurar su éxito o fracaso. Si el rendimiento de una aplicación es pobre (tiempos de respuesta elevados) los usuarios podrían perder interés, aun cuando las funcionalidades de la aplicación estén correctas. A continuación se ofrece una definición de rendimiento, sus características y el ámbito de aplicación dentro de este trabajo de titulación.

### 1.5.1 Definición de Rendimiento.

La definición más general, según (IEE, 1990) es que “el rendimiento hace referencia al grado en el que un sistema o componente realiza las funciones que le han sido designadas con las restricciones básicas como velocidad, precisión o uso de memoria”.

Un concepto más cercano es el de (Smith & Williams, 1993) que definen al rendimiento como un atributo de calidad de software, el cual abarca los aspectos que involucran el

comportamiento de sistemas de software. Básicamente se refiere a la capacidad de respuesta, es decir al tiempo requerido para responder a eventos específicos o el número de eventos procesados en un intervalo de tiempo dado.

Para (Microsoft, 2009a), el rendimiento es un indicador de la capacidad de respuesta de un sistema para ejecutar acciones específicas en un intervalo de tiempo dado. Se mide en términos de latencia o rendimiento. La latencia es el tiempo necesario para responder a un evento. El rendimiento es el número de eventos que tienen lugar en una determinada cantidad de tiempo. El rendimiento de una aplicación afecta directamente a su escalabilidad, y la falta de escalabilidad puede afectar al rendimiento. Si se mejora el rendimiento de una aplicación suele mejorar su escalabilidad mediante la reducción de la probabilidad de contención de recursos compartidos.

### **1.5.2 Características de Rendimiento.**

Las características relacionadas al rendimiento de un sistema incluyen la demanda de una acción específica y la respuesta del sistema a dicha demanda. Para (Microsoft, 2009) las restricciones técnicas que afectan al rendimiento son:

#### **1.5.2.1 *Tiempo de respuesta al cliente.***

Un aumento en el tiempo de respuesta a una solicitud específica repercute mucho en la reducción del rendimiento, esto se debe hacia una mayor demanda de recursos al servidor que se esté usando por lo que es necesario asegurar que la estructura de la aplicación se despliegue en un sistema o sistemas que proporcionen suficientes recursos. Cuando la comunicación debe cruzar los límites del proceso o nivel, se pueden usar interfaces de grano grueso que requieran el número mínimo de llamadas para ejecutar una tarea específica, así mismo se usa comunicación asíncrona.

#### **1.5.2.2 *Consumo de memoria.***

Si aumenta el consumo de memoria, esto también repercute en un rendimiento reducido a efectos prácticos, errores de caché excesivos y repetidos accesos a la base de datos.

### **1.5.2.3 *Procesamiento del servidor de base de datos.***

Consiste en la reducción del rendimiento debido a transacciones redundantes, bloqueos, subprocesos y enfoques de colas. Es necesario definir consultas eficientes para minimizar el impacto en el rendimiento y evitar buscar en toda la base de datos cuando solo se debe presentar un recurso específico. La falta de diseño para el procesamiento eficiente de la base de datos puede implicar en una carga innecesaria en el servidor de base de datos.

### **1.5.2.4 *Ancho de banda de la red.***

El aumento del consumo de ancho de banda resulta en tiempos de respuesta altos y un aumento en la carga de los sistemas. Es necesario establecer mecanismos de comunicación remota apropiados. Uno de los mayores retos del rendimiento es tratar de reducir el número de transiciones a través de los diferentes niveles de la aplicación y minimizar la cantidad de datos enviados a través de la red. Se puede trabajar por lotes para reducir las llamadas a través de la red.

### **1.5.3 *Factores que afectan al rendimiento de las aplicaciones.***

En su trabajo, (Happe, Koziolk, & Reussner, 2011) plantea algunas preguntas que los arquitectos de software suelen hacerse a menudo antes y durante el desarrollo y despliegue de una arquitectura de software para entender las interrelaciones de rendimiento en su diseño:

- ¿Cómo afecta la arquitectura candidata a los tiempos de respuesta y rendimiento para la carga de trabajo esperada?
- ¿Cómo influye la ejecución de componentes específicos en el rendimiento?
- ¿Cómo influye el rendimiento en la asignación de componentes a los recursos?
- ¿Cómo funcionará el sistema si la carga de trabajo aumenta inesperadamente?

Además de las métricas de rendimiento comunes (CPU, memoria, etc) de cualquier aplicación independientemente de su tipo y arquitectura, existen otras que cobran especial interés dentro de una aplicación construida con una arquitectura orientada a servicios ya que impactan directamente al desenvolvimiento de la aplicación. (Knoche, 2016) menciona que dentro de los microservicios existen varias restricciones principales referente al rendimiento las cuales se mencionan a continuación.

### 1.5.3.1 Red.

Durante la migración, los accesos inmediatos como llamadas de función nativas o las operaciones de combinación (JOIN) en la base de datos, tienen que ser sustituidos por invocaciones a servicios. Al hacer esto reducen potencialmente el rendimiento, debido a factores hardware como la serialización y la comunicación en red.

### 1.5.3.2 Transacciones en la base de datos.

La introducción de invocaciones de servicios aumenta el flujo dentro de una transacción y por tanto la duración de bloqueo mientras se ejecuta lo que conduce potencialmente a una reducción del rendimiento en las transacciones.

### 1.5.3.3 Método de despliegue.

De igual manera, el método utilizado para desplegar una aplicación impacta directamente en el desempeño del rendimiento, según menciona (Amaral & Carrera, 2015). En el mismo trabajo se mencionan los 3 métodos de despliegue actuales, también representados en la Figura 8.

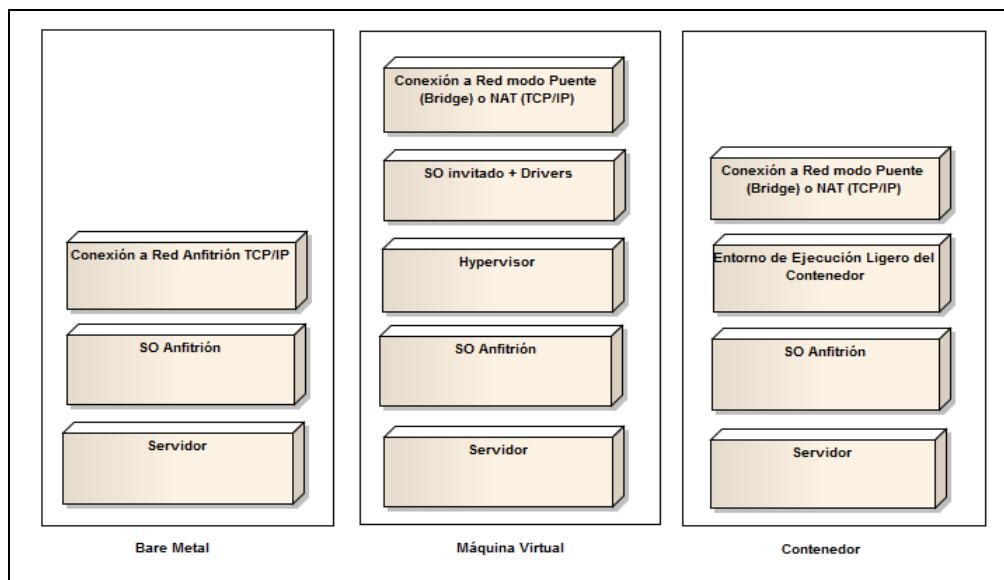


Figura 8 Métodos de Despliegue de Aplicaciones Web

Fuente: (Amaral & Carrera, 2015)

Elaboración: La Autora

#### *1.5.3.3.1 Despliegue Nativo (Bare Metal).*

La aplicación web y sus componentes son instalados directamente en el Sistema Operativo anfitrión.

#### *1.5.3.3.2 Máquina Virtual.*

El hardware que contendrá la aplicación web es emulado sobre un sistema operativo host, entonces la aplicación web será desplegada sobre un sistema operativo instalado en el hardware emulado.

#### *1.5.3.3.3 Contenedor.*

Es un mecanismo que provee virtualización a nivel de sistema operativo, esto significa que un contenedor crea un ambiente aislado para la aplicación utilizando el sistema operativo host.

(Lewis & Fowler, 2014) plantea el escenario en el que aun cuando se ha completado la migración de un monolito a microservicios, las operaciones de rendimiento no se detienen sino que cobran un papel mucho más importante que al inicio, donde recalca que las aplicaciones deben diseñarse de manera que sean tolerantes a los fallos de los servicios. Dado que los servicios pueden fallar en cualquier momento, es importante detectar éstas fallas rápidamente y, de ser posible, restaurar automáticamente el servicio.

Debido a esto, las aplicaciones que usan microservicios ponen mucho énfasis en el monitoreo en tiempo real de la aplicación, este monitoreo recoge métricas de rendimiento en tiempo real, por ejemplo, número de solicitudes por segundo a la base de datos y métricas relevantes para el negocio, como el número de órdenes por minuto que se reciben. Este monitoreo proporciona un sistema de alerta temprana que los equipos de desarrollo deben seguir e investigar. Para ello, desde un inicio se debe establecer configuraciones sofisticadas de monitoreo y registro para cada servicio individual, p.e. tableros de control que recojan el estado de cada servicio, su tasa de rendimiento actual y la latencia generada.

### **1.5.4 Métricas de rendimiento en base a hardware.**

El presente trabajo de titulación consiste en la evaluación del rendimiento al migrar desde una aplicación monolítica hacia una orientada a microservicios, por lo que es necesario tener claro qué métricas se evaluarán a cerca de este atributo de calidad.

Todas las métricas de rendimiento pueden ser analizadas desde dos perspectivas: Cliente (Usuario Final) y Servidor, siendo éste quien llevará la mayor carga, encargado de recibir, resolver y responder a las solicitudes del cliente. Tomando en cuenta lo mencionado anteriormente, las métricas de hardware a tomar en cuenta dentro de la evaluación del rendimiento para las arquitecturas mencionadas son las indicadas en la Tabla 8. En base a esta tabla y tomando en cuenta los aspectos de rendimiento en los microservicios mencionados en la sección 1.1.5 las métricas a evaluarse por cada una de las arquitecturas propuestas son las del lado del servidor, ya que en definitiva, este es el que asume toda la carga al procesar peticiones.

### **1.5.5 Patrones de Diseño orientados al rendimiento.**

Dentro de la programación orientada a objetos, un patrón de diseño permite reutilizar la solución a un problema que ya se ha suscitado antes, según The Gang of Four (GOF):

“Describen soluciones simples y elegantes a problemas específicos y recurrentes en el diseño de software orientado a objetos” (Gamma, Helm, Johnson, & Vlissides, 1997).

En el trabajo mencionado anteriormente y en (Guerrero, Suárez, & Gutiérrez, 2013) se mencionan 23 patrones de diseño que hacen frente a problemas recurrentes en el desarrollo de software, estos se encuentran clasificados y agrupados a partir de dos criterios que son propósito y alcance, las categorías son:

- **Creacionales:** Se ocupan del proceso de creación de clases y objetos y se encargan de abstraer el proceso de instanciación o creación de objetos, así mismo ayudan a que el sistema sea independiente de la creación, integración y representación de objetos.
- **Estructurales:** Se ocupan de cómo se agrupan las clases y objetos para formar estructuras más grandes, es un objetivo de este tipo de patrón que los cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos.
- **De comportamiento:** Representan la forma en que las clases y/o los objetos interactúan y distribuyen responsabilidades. Se ocupan de las opciones de comportamiento de la aplicación, por lo que el comportamiento varía en tiempo de ejecución.

Tabla 8 Métricas de rendimiento en base a Hardware

	<b>Categoría</b>	<b>Métrica</b>	<b>Descripción</b>
<b>Lado de Cliente</b>	<b>Transacciones</b>	Usuarios activos	Número de usuarios activos durante la ejecución de la prueba.
		Transacciones	Número de transacciones por segundo.
		Transacciones exitosas	El tiempo de respuesta en segundos para las transacciones que han finalizado con éxito.
		Transacciones Fallidas	El tiempo de respuesta en segundos para las transacciones que han fallado.
<b>Lado de Servidor</b>	<b>Método de Despliegue</b>	Despliegue Nativo	Aplicación web instalada nativamente en el SO.
		Máquina Virtual (MV)	Aplicación web sobre hardware emulado
		Contenedor	Aplicación web aislada y virtualizada en el SO local.
	<b>CPU</b>	Consumo de CPU	Cantidad que indica consumo de CPU durante la ejecución de la prueba.
		Uso de CPU (%)	Porcentaje de uso de CPU durante la ejecución de la prueba.
	<b>Memoria</b>	Memoria usada (mb)	Memoria consumida por la aplicación durante la ejecución de la prueba.
		Memoria Libre (mb)	Memoria libre en la aplicación después de la ejecución de la prueba.
		Memoria Total (mb)	Memoria total recopilada de la ejecución de la prueba.
		Max Memoria (mb)	Memoria máxima expresada en MB consumida durante la ejecución de la prueba.
		Min Memoria (mb)	Memoria mínima expresada en MB consumida durante la ejecución de la prueba.
	<b>Red</b>	Velocidad de conexión	Velocidad de conexión media durante la ejecución de la prueba.
		Bytes enviados/sec	Bytes enviados por segundo al cliente.
		Bytes recibidos/sec	Bytes recibidos por segundo desde el cliente.
		Máximo de conexiones	Número máximo de conexiones aceptadas por el servidor durante la ejecución de la prueba.
		Transacciones/sec	Número de transacciones por segundo procesadas durante la ejecución de la prueba.

	<b>Base de Datos</b>	Sentencia SQL	Sentencia SQL que se ejecuta.
		Ejecuciones	Número total de ejecuciones.
		Exec Avg (ms)	Tiempo medio en milisegundos empleados en la ejecución de la sentencia SQL.
		Exec Min (ms)	Tiempo mínimo en milisegundos empleados en la ejecución de la sentencia SQL.
		Exec Max (ms)	Tiempo máximo en milisegundos empleados en la ejecución de la sentencia SQL.
		Tiempo de respuesta (ms)	Tiempo de respuesta en milisegundos.
		Ejecuciones fallidas (%)	Porcentaje de fallo en las ejecuciones de sentencias SQL.

Fuente: (Amaral & Carrera, 2015; Villamizar et al., 2015)

Elaboración: La Autora

(Kumar & Prabhakar, 2010) indica que los patrones de diseño están orientados a problemas específicos de las aplicaciones y pueden ser evaluados mediante los atributos de calidad de software, en su trabajo también menciona qué patrones de diseño están relacionados al atributo de calidad de rendimiento, como se muestra en la Tabla 9. En el presente trabajo de fin de titulación se considerarán estos patrones de diseño en el proceso de construcción de las aplicaciones involucradas en la ruta de migración.

Tabla 9 Patrones de diseño relacionados con el Rendimiento

<b>Nombre</b>	<b>Propósito</b>	<b>Descripción</b>
<b>Singleton</b>	Creacional	Asegura que una clase tiene una sola instancia y proporciona un punto de acceso global a ella. Se usa cuando hay clases que deben gestionar de manera centralizada un recurso.  Además de evitar que el espacio de nombres se contamine de variables globales con instancias únicas, la clase Singleton puede ser subclassificada para configurar una aplicación con una instancia de esta clase extendida o con una instancia de la clase que se necesita en tiempo de ejecución.



<b>Prototype</b>	Creacional	Especifica los tipos de objetos a crear mediante la clonación de un prototipo que es una instancia ya creada, así mismo especifica nuevos objetos copiando este prototipo. Referente al rendimiento, es recomendable usarlo cuando el costo de crear objetos es muy alto o no se tienen los detalles para hacerlo, por lo que es mejor solicitar una copia o un clon del mismo.
<b>Flyweight</b>	Estructural	También llamado objeto ligero. Elimina o reduce el número de objetos a crear por lo que se reduce la redundancia cuando se tiene una gran cantidad de objetos que contienen información idéntica. Como consecuencia se decrementa el uso de memoria y recursos.
<b>Proxy</b>	Estructural	Proporciona un sustituto para otro objeto y así controlar su acceso al objeto principal, lo que introduce un nivel de indirección al acceder a un objeto, esta indirección tiene muchos usos, por ejemplo, un proxy virtual permite crear objetos bajo demanda. Se usa para dar respuestas inmediatas al cliente, el proxy espera la respuesta del objeto real y al llegar la entrega al cliente.

Fuente: (Gamma et al., 1997; Kumar & Prabhakar, 2010)

Elaboración: La Autora

**CAPITULO II: DISEÑO DE PROCESO DE MIGRACIÓN DE APLICACIÓN MONOLÍTICA  
A APLICACIÓN CON MICROSERVICIOS**

## **2 Diseño de proceso de migración de aplicación monolítica a aplicación con microservicios**

La migración de una aplicación con determinada arquitectura hacia otra requiere una serie de pasos ordenados dentro de una ruta de migración que garantice el éxito de este proceso, esto con el fin de adaptar la aplicación web actual a los procesos de negocio actuales de una empresa, en el caso de este trabajo de fin de titulación se ha propuesto medir el rendimiento en el proceso de migración de una aplicación web con una arquitectura monolítica hacia una orientada a microservicios.

Esta transición no puede realizarse de manera automática, sino que es necesario especificar modelos que involucren pasar primeramente por un enfoque orientado a servicios, como bien menciona en su trabajo, (Knoche, 2016):

“La mayoría de los monolitos son demasiado grandes para ser migrados hacia otra arquitectura en un solo paso, por lo que esta migración debe llevarse a cabo gradualmente, en varias etapas, también llamados incrementos donde cada incremento es liberado en producción.”

Para realizar el proceso de migración se ha tomado en cuenta 3 modelos de refactorización propuestos por 3 diferentes autores, estos son:

1. Modelo de refactorización de IBM, (Santis et al., 2016)
2. Modelo de refactorización de NGINX, (Richardson & Smith, 2016)
3. Modelo de refactorización de Medium, (Medium, 2017)

A continuación se puntualizan las consideraciones antes de realizar la migración, nombradas en (Santis et al., 2016) y (Richardson & Smith, 2016), posteriormente se explica el funcionamiento de cada modelo de migración para finalmente, en base a estos modelos, establecer la ruta de migración de una aplicación web con arquitectura monolítica hacia una orientada a microservicios.

### **2.1 Consideraciones antes de la migración**

“Refactorización” es un término reciente en el contexto de las aplicaciones que hoy por hoy desean empatar sus capacidades a las necesidades de negocio emergentes, en palabras de (Brown Kyle, 2016):

“Refactorización es una práctica que moderniza la aplicación y gana recursos proporcionados por nuevas estructuras y plataformas, es la transformación de código que preserva el comportamiento”.

En este contexto, la migración de una aplicación monolítica a microservicios sigue la misma trayectoria. La refactorización agrega microservicios a una aplicación sin cambiar el propósito de la misma, sin embargo, antes de realizar un proceso de migración es necesario situarse en el contexto actual de la aplicación para evaluar si cumple con ciertos criterios (Brown, 2016):

### **2.1.1 Identificación de candidatos en el monolito.**

Los candidatos a la evolución de una arquitectura de microservicios son aplicaciones monolíticas con componentes que cumplen estas condiciones:

- i. No se puede implementar la aplicación lo suficientemente rápido como para cumplir con los requisitos debido a la dificultad de mantener, modificar y liberar en producción rápidamente, lo que resulta en largos ciclos de tiempo de salida al mercado para implementar nuevos servicios.
- ii. No se puede aplicar una sola implementación. Por lo general, es necesario involucrar otros módulos o componentes para crear, probar e implementar un pequeño cambio o porque no hay separación de módulos y componentes dentro de un mismo paquete.
- iii. Sólo se utiliza una plataforma tecnológica y no se puede aprovechar tecnologías, bibliotecas o técnicas que están siendo adoptadas por otras empresas.
- iv. El sistema actual tiene las siguientes características:
  - Gran cantidad de datos en memoria.
  - Operaciones que requieren de alto uso de CPU.
  - No se puede escalar una parte de la aplicación; generalmente toda la aplicación debe ser escalada.
  - No se puede actualizar y mantener fácilmente.
  - Existen dependencias de código difíciles de gestionar.
  - Resulta difícil la integración de nuevos desarrolladores debido a la complejidad y el gran tamaño del código.

### 2.1.2 Asignar tamaño a los microservicios.

Una de las tareas más imprecisas al diseñar un sistema de microservicios es decidir el número y el tamaño de los microservicios individuales. No existe una regla estricta con respecto al tamaño óptimo, pero pueden adoptarse algunas buenas prácticas probadas en sistemas reales. Las siguientes técnicas pueden usarse solas o combinadas:

- i. **Funcionalidades:** Como norma básica, se recomienda no añadir nuevas funcionalidades a la aplicación monolítica a migrar, para todas las nuevas características, utilizar microservicios independientes.
- ii. **Número de archivos:** Se puede medir el tamaño de un microservicio en un sistema por el número de archivos que compone. Esta técnica es imprecisa pero puede ser útil si un microservicio es demasiado grande, físicamente. Los servicios grandes son difíciles de trabajar, difíciles de implementar y tardan más en arrancar y detenerse pero no es factible llegar al otro extremo con microservicios demasiado pequeños: nanoservicios, en este caso el costo de los recursos para desplegar y operar un nanoservicio eclipsa su utilidad.
- iii. **Demasiadas responsabilidades:** Un servicio que es responsable simultáneamente de diferentes operaciones tiene la probabilidad de dividirse porque normalmente se vuelve difícil probar, mantener y desplegar.
- iv. **Tipo de servicio:** Esta regla se refiere a que un microservicio realiza una sola cosa (p.e: Gestionar la autenticación, servir varios puntos finales REST, servir varias páginas web). Normalmente, estas responsabilidades heterogéneas no deben mezclarse. A diferencia de la regla anterior, esta se trata de la calidad de las responsabilidades.
- v. **Separación de contexto limitada:** Esta técnica es importante cuando un sistema existente está siendo dividido en partes que son relativamente autosuficientes, por lo que hay pocos enlaces a un servidor que pueden convertirse en microservicios. p.e, si un microservicio necesita hablar con otros 10 microservicios para completar su tarea, esto indica que la división se hizo en un lugar incorrecto del monolito.
- vi. **Organización del equipo:** Muchos sistemas de microservicios están organizados alrededor de equipos que son responsables de escribir el código. La expectativa es que el número de microservicios y el tamaño están dictados por principios organizativos y técnicos.

### **2.1.3 Adoptar estrategias de Integración Continua y Entrega Continua.**

La integración continua (CI) es una práctica de desarrollo de software en la que los desarrolladores integran su trabajo a menudo, diariamente, lo que resulta en una integración múltiple por día. La idea es aislar los posibles problemas de integración que puedan ocurrir. CI está respaldado por un proceso automatizado que incluye actividades de construcción y pruebas para verificar cada compilación y detectar errores de integración rápidamente. Debido a que los problemas potenciales pueden aislarse y detectarse con anterioridad, CI ayuda a reducir significativamente los problemas de integración durante los ciclos de desarrollo y, finalmente, ayuda a entregar los productos rápidamente.

Normalmente CI suele ir acompañada de Entrega Continua (CD), ésta en cambio, se refiere a la liberación en producción continua del código que pasa el periodo de pruebas. Mediante la adopción de CI y CD, los desarrolladores pueden reducir el tiempo dedicado a corregir los errores y aumentar el esfuerzo dedicado a desarrollar nuevas características o mejorar las existentes.

La adopción de CI y CD genera un mayor impacto en el valor del negocio, es decir ambas desempeñan un papel crucial en la evolución de una arquitectura. Dentro de los microservicios, uno de los objetivos de implementación es lograr la comunicación entre los servicios, para ello estos deben estar contruidos apropiadamente, este proceso puede vigilarse a través de una funcionalidad de CI y CD por cada servicio.

### **2.1.4 Patrón Strangler.**

Este patrón de Martin Fowler (Fowler, 2004) se refiere a una estrategia de modernización de las aplicaciones que consiste en la refactorización incremental de una aplicación. En el contexto de los microservicios se trata de construir una nueva aplicación con microservicios alrededor de una aplicación monolítica existente.

Al pasar de una arquitectura monolítica a microservicios, se puede refactorizar de forma incremental, gradualmente se construye una nueva aplicación que consiste en microservicios. A continuación, se ejecuta esta aplicación junto con la aplicación monolítica, con el tiempo, la funcionalidad manejada por la aplicación monolítica se reduce gradualmente hasta que desaparece completamente o se convierte en otro microservicio. El patrón Strangler tiene dos componentes principales (Ver Figura 9):

- i. **Despachador:** Este componente funciona como un enrutador de peticiones que maneja las solicitudes entrantes de la aplicación monolítica, redirigiéndolas a los nuevos servicios. Dentro de una arquitectura de microservicios, el patrón API Gateway (sección 1.1.4.1) cumple las funciones de Despachador.
- ii. **IC:** Estos componentes residen en los nuevos servicios, en la aplicación monolítica o en ambos. Son responsables de integrar la aplicación monolítica con los servicios recién creados.

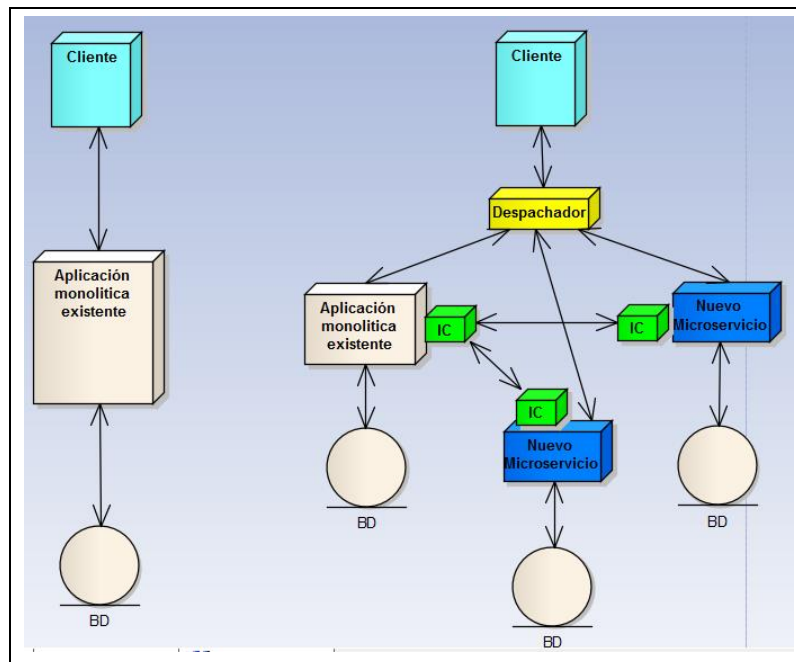


Figura 9 Patrón Strangler

Fuente: (Santis et al., 2016)

Elaboración: (Santis et al., 2016)

### 2.1.5 Integración de DevOps con microservicios y contenedores.

Dentro del entorno empresarial el término DevOps ha cobrado relevancia en los últimos años, como define (Davis & Daniels, 2016), este término hace referencia a una cultura o movimiento basado en la colaboración, integración y comunicación entre desarrolladores y profesionales de Tecnologías de la Información (IT). Generalmente DevOps está asociado a metodologías ágiles, Integración, Entrega y Despliegue continuos. (de Kort, 2016) define las 6 fases especificadas para DevOps:

- **Planificación:** En esta fase se recolectan los requerimientos, se crean las tareas y las actividades del proyecto.
- **Codificación:** Esta fase se da durante el desarrollo o actualización del código de acuerdo a la tarea planificada, por lo que es importante definir herramientas de versionamiento de código.
- **Construcción:** En esta fase se realiza la construcción del código junto a su compilación, así mismo se ejecutan las primeras pruebas unitarias.
- **Pruebas:** Se realiza una revisión al código antes de su despliegue en producción mediante herramientas que no requieran de intervención manual.
- **Despliegue:** Se configuran herramientas que permitan realizar el despliegue de las aplicaciones de forma automatizada.
- **Monitoreo:** Se supervisa el funcionamiento de las aplicaciones desplegadas sobre todo para identificar posibles fallos que no se hayan controlado en fases anteriores.

En el trabajo de (Farías, 2017) se describe un esquema de integración entre la cultura DevOps, Microservicios y la tecnología de contenedores el cual se recomienda utilizarse para una mayor efectividad de las tareas a realizarse mientras se construye una aplicación con arquitectura de microservicios.

Adoptar DevOps junto el uso de metodologías de desarrollo ágil como Scrum incluso desde la planificación del sistema permite tener un ciclo de retroalimentación rápido debido a la relación cercana con prácticas continuas (CI/CD). DevOps se convierte en un soporte para las organizaciones cuyo propósito es reducir costos, evitar riesgos, mejorar la calidad y aumentar la velocidad en la entrega de aplicaciones, reduciendo también los tiempos de retroalimentación del producto. En la Figura 10 se presenta el esquema de integración propuesto por (Farías, 2017).

En un ambiente real, al dividir una aplicación monolítica entera en microservicios no es factible esperar a desarrollar cada uno de los microservicios para luego desplegarlos, el escenario ideal consiste en que cada microservicio que se codifique sea automáticamente añadido al código base y desplegado en contenedores (p.e Docker) para operar, dicho escenario se relaciona con el esquema de integración propuesto por (Farías, 2017) y el patrón de construcción de microservicios Strangler, por lo que se ha considerado en este trabajo de fin de titulación.



Un sistema de microservicios bien diseñado utiliza una combinación de estas técnicas, las mismas requieren un grado de buen juicio que se adquiere con el tiempo y la experiencia con el sistema. Evaluadas estas consideraciones, se procede a explicar el funcionamiento de cada modelo de migración.

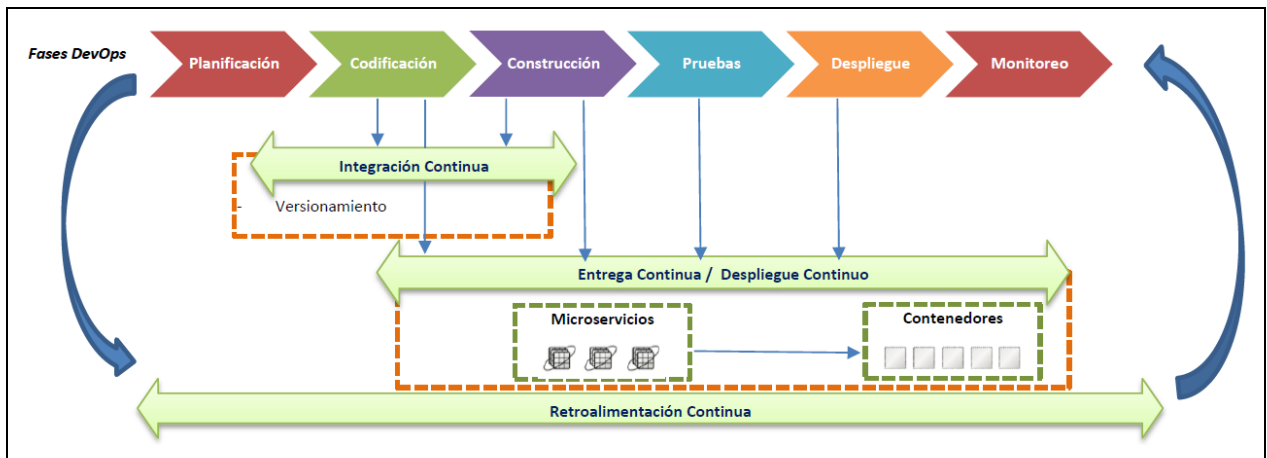


Figura 10 Esquema de Integración DevOps + Microservicios + Contenedores

Fuente: (Farías, 2017)

Elaboración: (Farías, 2017)

## 2.2 Modelo de refactorización de IBM

A breves rasgos, el modelo de refactorización ofrecido por IBM (Santis et al., 2016) considera como aplicación monolítica candidata a una aplicación con una arquitectura n-capas desarrollada con tecnología Java, consta de tres fases bien diferenciadas donde cada fase contiene tareas adicionales, a continuación se explican cada una de estas.

### 2.2.1 Reempaquetado de la aplicación.

Por lo general, las aplicaciones monolíticas se almacenan en un solo archivo EAR que se implementa en un servidor físico, donde cualquier cambio significa volver a desplegar todo, lo que resulta costoso. Para reempaquetar la aplicación se debe realizar las siguientes tareas:

#### 2.2.1.1 Dividir los EAR

Dividir los archivos WAR relacionados a los EAR en WAR's independientes.

### **2.2.1.2 Contenedor por servicio**

Aplicar el patrón "Singler Service per Host" y desplegar cada WAR en un servidor, si se desea, cada WAR en un contenedor propio (p.e. Docker).

### **2.2.1.3 Construir, implementar y administrar de forma independiente**

Una vez divididos los WAR, se pueden administrar de forma independiente mediante estrategias de entrega continua (IC/DC).

## **2.2.2 Refactorización del código.**

Teniendo WAR independientes, se puede aplicar las técnicas de desagregación de servicios para refactorizar los WAR a niveles aún más granulares. Llegado a este punto pueden darse 3 casos:

### **2.2.2.1 Caso 1. Servicios REST o JMS existentes**

Este es el caso más común y sencillo de refactorización, donde los servicios existentes ya son compatibles con la arquitectura de microservicios. La tarea consiste en separar cada servicio REST del resto de WAR's, y desplegarlo como un servicio aparte. A este nivel se permite la duplicación de archivos JAR de apoyo.

### **2.2.2.2 Caso 2. Servicios SOAP o EJB existentes**

Los servicios existentes fueron construidos siguiendo un enfoque funcional como operaciones CRUD en un solo objeto. De ser así se puede aplicar una interfaz RESTful y convertir las representaciones de objetos en JSON.

### **2.2.2.3 Caso 3. Interfaces simples Servlet/JSP**

En el caso de tener una aplicación desarrollada en Java como Servlet con front-end JSP lo más probable es que no tenga capa de objeto de dominio. De ser así, la creación de la capa de dominio puede representarse como un servicio RESTful. La identificación de los objetos de dominio mediante la aplicación del diseño controlado por el dominio ayuda a identificar la capa de servicios de dominio que falta. Una vez se tenga cada nuevo servicio en su propio WAR, se puede refactorizar la aplicación Servlet / JSP existente para usar el nuevo servicio o crear una interfaz totalmente nueva, con JavaScript, HTML y CSS.

### **2.2.3 Refactorización de los datos.**

A menudo la base de datos es el activo más importante en un sistema de TI. Cuando se hace una refactorización de la aplicación, este activo también debe cambiarse para alinearse con los cambios del código de la aplicación. El reto más importante al adoptar una arquitectura de microservicios es refactorizar las estructuras de datos en las que se basan las aplicaciones. Estas son las reglas que hay que tener en cuenta:

#### **2.2.3.1 Islas de datos**

Se trata de examinar las tablas de base de datos que usa la aplicación. Debido a que la lógica de los productos se traslada completamente al nuevo microservicio, los datos también deben migrarse gradualmente, esto elimina cualquier latencia potencial cuando el nuevo servicio necesita hacer llamadas a la base de datos. Si las tablas utilizadas son independientes de todas las demás u están aisladas de unas cuantas tablas unidas por relaciones, entonces se dividen del resto del modelo de datos.

#### **2.2.3.2 Tipo de base de datos**

Ante preguntas como ¿Usar Oracle o MySQL?, ¿Usar una base de datos No SQL o seguir usando SQL? La respuesta depende del tipo de consultas que se realicen a los datos de la aplicación. Si la mayoría de consultas son simples, a claves primarias, se puede optar por el enfoque No SQL mientras que si se tienen combinaciones realmente complejas que varían ampliamente (consultas impredecibles), lo recomendable es permanecer con SQL. Las tecnologías para la implementación de un nuevo catálogo son numerosas. Sin embargo, al considerar un enfoque evolutivo, también se debe considerar mantener la misma pila de tecnológica que se utiliza en la aplicación monolítica, para evitar llevar grandes cambios al mismo tiempo.

#### **2.2.3.3 Desnormalización de tablas**

Si el modelo de datos presenta muchas relaciones entre tablas, este se puede desnormalizar. Para lograrlo es necesario mirar atrás y pensar por qué los datos fueron normalizados. A menudo, la razón de los esquemas altamente normalizados era reducir la duplicación, para ahorrar espacio, porque el espacio en disco es costoso, pero con microservicios lo importante es optimizar el tiempo de consulta y la desnormalización es una manera sencilla de lograrlo. En la Figura 11 se representa el proceso de modelo de refactorización propuesto por IBM.

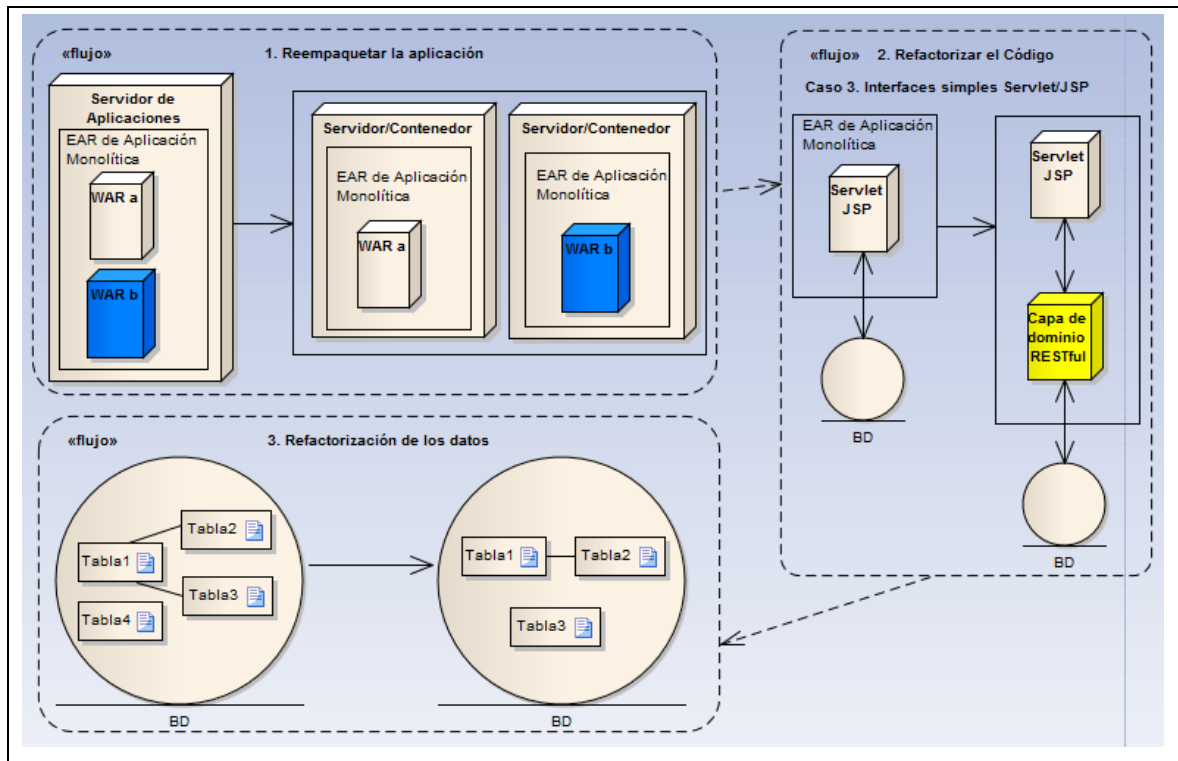


Figura 11 Modelo de Refactorización de IBM

Fuente: (Santis et al., 2016)

Elaboración: La Autora

### 2.3 Modelo de refactorización de NGINX

El modelo de refactorización de NGINX (Richardson & Smith, 2016) considera aplicación monolítica a una diseñada con una arquitectura 3 capas, independientemente de la tecnología que se use para implementarla. Este modelo ofrece 3 estrategias de refactorización de un monolito a microservicios, tomando a la refactorización como una forma de “modernización de la aplicación”.

En el mundo real, las aplicaciones monolíticas que desean migrar a microservicios no reescriben su funcionalidad desde cero, en su lugar, refactorizan su aplicación de forma incremental en un conjunto de microservicios. Las estrategias de NGINX son:

1. Implementar una nueva funcionalidad como microservicios.
2. Dividir los componentes de presentación de los componentes de negocio y de acceso a datos.
3. Convertir los módulos existentes del monolito en microservicios.

Estas estrategias se alinean a los principios del patrón Strangler, indicado anteriormente. A continuación se explica cada una de las estrategias propuestas.

### **2.3.1 Estrategia 1: Stop Diggin.**

Se trata de dejar de hacer el monolito más grande, en la práctica esto significa que si se desea implementar una nueva funcionalidad, no se debería agregar más código al monolito, en su lugar, la nueva funcionalidad se agregaría a un microservicio independiente. Para comunicar el nuevo microservicio con el monolito legado deben existir otros dos componentes. Un enrutador de solicitudes que envía peticiones (HTTP) correspondientes a la funcionalidad al nuevo servicio, es decir, enruta peticiones que antes eran procesadas por el monolito. El otro componente es el código de integración, este código enlaza e integra el microservicio con el monolito. Un servicio rara vez existe de forma aislada, a menudo necesita acceder a los datos pertenecientes al monolito. El servicio utiliza el código de integración para leer y escribir datos pertenecientes al monolito. Para acceder a los datos del monolito desde un microservicio existen las siguientes opciones:

- a) Invocar una API remota proporcionada por el monolito.
- b) Acceso directo a la base de datos del monolito.
- c) El microservicio tiene su propia copia de los datos, la cual está sincronizada con la base de datos del monolito.

### **2.3.2 Estrategia 2: Dividir Frontend de Backend.**

Una aplicación empresarial típica consiste en al menos tres tipos diferentes de componentes: Capa de presentación, Lógica de negocio y Capa de acceso de datos. Normalmente existe una separación clara entre la lógica de presentación y la lógica de negocio y de acceso a datos ya que la capa de lógica de negocio tiene una API de grano grueso que a su vez consta de una o más fachadas, que encapsulan los componentes de lógica de negocio. Esta API está construida de tal manera que se puede dividir el monolito en dos aplicaciones más pequeñas. Una aplicación contiene la capa de presentación. La otra aplicación contiene la lógica de negocio y la capa de acceso a datos. Después de la división, la aplicación de lógica de presentación hace llamadas remotas a la aplicación de lógica de negocio. Dividir el monolito de esta manera tiene dos beneficios principales: Permite desarrollar, implementar y escalar las dos aplicaciones independientemente entre sí. Otra ventaja de este enfoque es que se expone una API remota que puede ser llamada por los microservicios que se desarrolle.

### **2.3.3 Estrategia 3: Extraer servicios.**

La tercera estrategia de refactorización es convertir los módulos existentes dentro del monolito en microservicios independientes. Una vez que se hayan convertido suficientes módulos a microservicios, el monolito desaparece por completo o se vuelve lo suficientemente pequeño como para ser solamente otro servicio. Este proceso requiere de las siguientes fases:

#### **2.3.3.1 Fase 1: Definir interfaces entre los módulos y el monolito.**

La interfaz puede ser una API bidireccional, ya que el monolito necesitará datos pertenecientes al servicio y viceversa. A menudo es difícil implementar una API de este tipo debido a las dependencias cruzadas y los patrones de interacción de grano fino entre el módulo y el resto de la aplicación. La lógica de negocio implementada utilizando el patrón del modelo de dominio es especialmente difícil de refactorizar debido a numerosas asociaciones entre clases de modelos de dominio. A menudo se necesita realizar cambios de código significativos para romper estas dependencias.

#### **2.3.3.2 Fase 2: Convertir el módulo a un servicio autónomo.**

Una vez que se implemente una interfaz de grano grueso se puede convertir el módulo en un servicio independiente. Para ello, se debe escribir código que permita que el monolito y el servicio se comuniquen a través de la API, la cual utiliza un mecanismo de comunicación entre procesos, como REST, con esto se consiguen servicios que se pueden desarrollar, desplegar y escalar independientemente del monolito y de cualquier otro servicio. En este caso, el código API que integra el servicio con el monolito se convierte en una capa de prevención que traduce entre los dos modelos de dominio. Con el tiempo, el monolito se reduce y se tendrá un número alto de microservicios.

Debido a la familiaridad de la estrategia 3 con la propuesta del presente trabajo de titulación se ha designado a esta como un modelo de migración de una aplicación monolítica hacia una orientada a microservicios. En la Figura 12 se representa la tercera estrategia de migración de NGINX.

## **2.4 Modelo de refactorización de Medium**

Refactorizar un monolito en microservicios resulta un reto para las empresas actuales ya que por lo general resulta costoso y más aún, las empresas se sienten temerosas de la ruptura de su sistema actual porque la refactorización representa, romper la empresa. Tomando en cuenta

este precedente, en (Medium, 2017) se mencionan las directrices para refactorizar el monolito a microservicios.

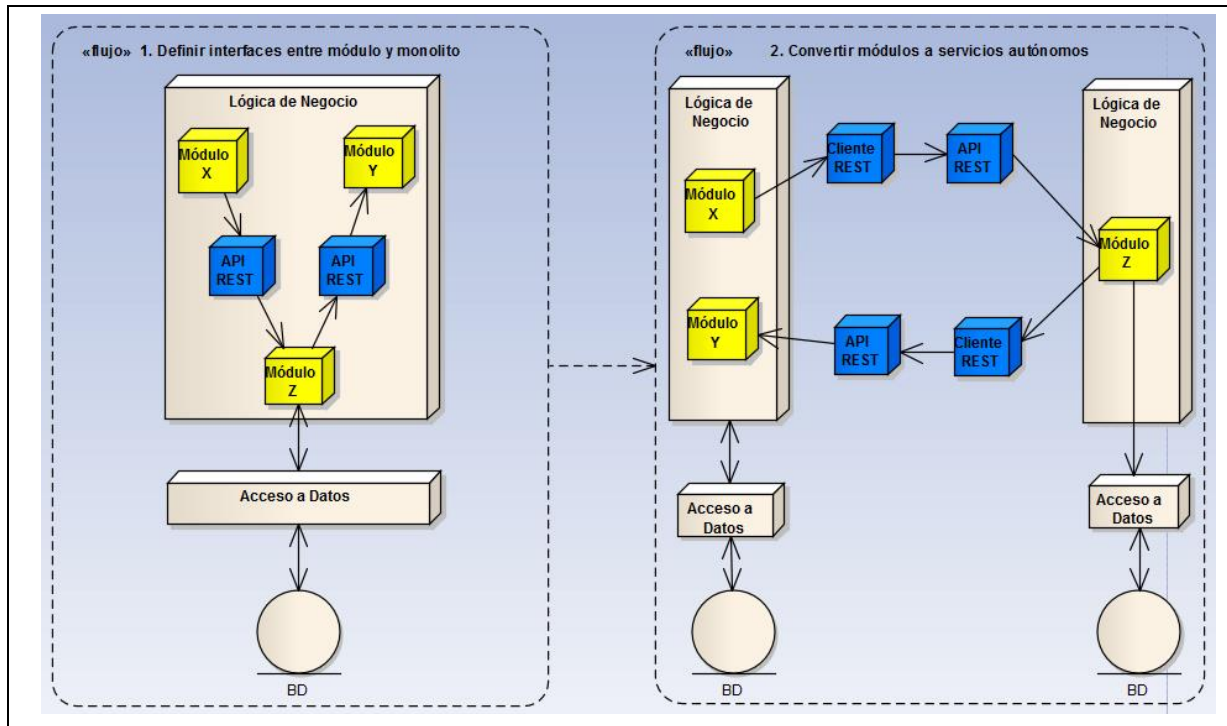


Figura 12 Modelo de refactorización de NGINX

Fuente: (Richardson & Smith, 2016)

Elaboración: La Autora

### 2.4.1 Refactorización incremental.

No se refactoriza todo el monolito a la vez.

### 2.4.2 Monolito con API's.

Se debe considerar al monolito como una unidad con algunas API que promuevan la comunicación entre procesos, como REST.

### 2.4.3 Nuevas funcionalidades.

Si se desea adicionar nuevas características al sistema actual, se recomienda crear microservicios separados con un API para cada uno de ellos y hacer que interactúen con las API de monolito. Después de algún tiempo se notará que las piezas del monolito se están

accediendo sólo a través de las nuevas API, a pesar de que siguen siendo una parte de la base de código del monolito.

#### **2.4.4 Contextos limitados.**

Una vez que se identifiquen contextos limitados en los módulos del monolito, se considera conveniente separarlos y convertirlos a microservicios separados.

#### **2.4.5 IC/DC.**

Los microservicios están ligados a la automatización. Para hacer posible esta característica, se deben establecer prácticas y herramientas de desarrollo continuo (DC) e integración continua (IC), repositorios, contenedores y monitoreo.

Dentro de las recomendaciones que indica (Medium, 2017), enfatiza que estos cambios deben introducirse gradualmente. En cuanto al equipo de desarrollo de software se indica que estos deben encargarse del desarrollo de microservicios desde la primera línea de código, el despliegue y las operaciones de producción. Se recomienda comenzar con un equipo de trabajo en un contexto acotado, una vez que el equipo gana un poco de experiencia, una persona de este grupo puede ser líder de otro equipo de desarrollo. En la Figura 13 se representa el modelo de refactorización propuesto por Medium.

### **2.5 Comparación de modelos de migración propuestos**

Revisados los 3 modelos de refactorización propuestos por los diferentes autores, a continuación, en la Tabla 10 se compara cada uno de estos modelos en base a varios parámetros como tecnología usada, base de datos, fases, etc.

### **2.6 Propuesta de migración de aplicación monolítica hacia una orientada a microservicios.**

Para proponer la ruta de migración desde una arquitectura monolítica hacia una orientada a microservicios se ha tomado en cuenta los modelos de refactorización revisados anteriormente. Sin embargo, debido a que estos modelos poseen enfoques diferentes e incluso están ligados a una tecnología en particular, es necesario abstraer en el grado que sea posible las fases que se ajusten a los requisitos de la aplicación que se desea migrar a microservicios.



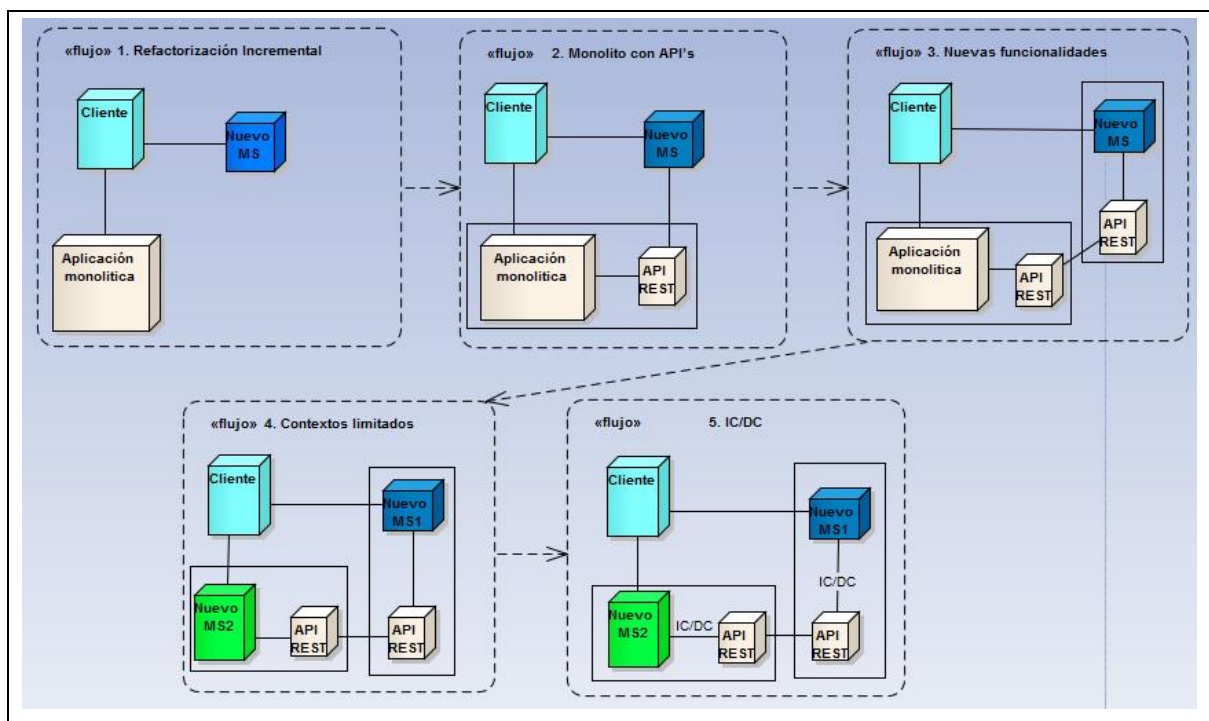


Figura 13 Modelo de Refactorización de Medium

Fuente: (Medium, 2017)

Elaboración: La Autora

Para establecer la ruta de migración se ha tomado a consideración:

- **Atributo de calidad a medir:** Rendimiento
- **Fase de medición de atributo de calidad:** Implementación
- **Patrones de diseño a usar:** Singleton, Proxy
- **Fase de aplicación de patrones de diseño:** Desarrollo
- **Enfoque a aplicar:** Esquema de integración DevOps, Microservicios y contenedores

La ruta de migración propuesta consta de las siguientes fases:

### 2.6.1 Identificar funcionalidades en el monolito.

Esta es una fase de preparación antes de iniciar el proceso de migración, donde se evalúa el estado actual de la aplicación y se identifican las funcionalidades que posee la misma, en base a las indicaciones de la sección 2.1.2 se establece cuáles son las funcionalidades más importantes y emergentes que necesitan desplegarse independientemente. Este paso es importante ya que en base a estas funcionalidades se construyen los microservicios.

Tabla 10 Comparación de modelos de refactorización

Modelo	Fases Especificadas	Arquitectura Origen	Tecnología de desarrollo empleada	Considera refactorización de base de datos	Mecanismo de comunicación entre procesos
<b>Modelo de refactorización de IBM</b>	<ul style="list-style-type: none"> <li>• Reempaquetado de la aplicación</li> <li>• Refactorización del código</li> <li>• Refactorización de los datos</li> </ul>	Arquitectura n capas	JAVA	Si	Mecanismos IPC: API REST
<b>Modelo de refactorización de NGINX</b>	<ul style="list-style-type: none"> <li>• Definir interfaz entre un módulo y el monolito</li> <li>• Convertir el módulo a un servicio autónomo</li> </ul>	Arquitectura en 3 capas	No especifica	No	Mecanismos IPC: API REST, Apache Trift
<b>Modelo de refactorización de Medium</b>	<ul style="list-style-type: none"> <li>• Refactorización incremental</li> <li>• Monolito con API's</li> <li>• Nuevas funcionalidades</li> <li>• Contextos limitados</li> <li>• IC/DC</li> </ul>	No especifica	No especifica	No	Mecanismos IPC: API REST

Fuente: (Medium, 2017; Richardson & Smith, 2016; Santis et al., 2016)

Elaboración: La Autora

### 2.6.2 Definir API REST para funcionalidades del monolito.

El objetivo de esta fase es extraer funcionalidades de la aplicación monolítica y conseguir que estas funcionalidades se sigan comunicando con la misma, esto es posible a través de la construcción de API's, las cuales utilizan un mecanismo de comunicación entre procesos, como REST.

Estas API pueden ser bidireccionales, ya que el monolito necesitará datos pertenecientes al servicio y viceversa y deben ser construidas utilizando patrones de diseño orientados al rendimiento como Singleton y Proxy.

Si se desea adicionar nuevas funcionalidades al sistema actual deben crearse API para cada una de ellas y mantener el comportamiento de la aplicación, es decir, hacer que interactúen con las API del monolito, al hacer esto se está aplicando el patrón Strangler con estrategias de CI y CD, necesarios en el proceso de migración desde el monolito hasta Microservicios.

Esta fase se basa en la Fase 1 del Modelo de refactorización de NGINX, (Richardson & Smith, 2016) y Fase 2 del Modelo de refactorización de Medium (Medium, 2017).

### **2.6.3 Convertir funcionalidades en servicios autónomos.**

En esta fase se despliega por separado cada API REST desarrollada en la fase anterior. En este caso se aplica el patrón "Contenedor por servicio", es decir, se despliega cada aplicación que contiene a la API en un servidor, esto significa asignarle a cada servicio un host y puerto propio y que cada aplicación corra en su propio contenedor (p.e.Docker).

Una vez se tenga cada servicio desplegado independientemente se puede refactorizar la interfaz del monolito para usar las API REST, en base al patrón Strangler y la estrategia de Despliegue Continuo establecido en el esquema de integración propuesto en la sección 2.1.5. Con el paso del tiempo se tendrá un alto número de microservicios.

Esta fase hace referencia a la Fase 2 del Modelo de refactorización de IBM, Fase 1 de Modelo de refactorización de IBM, Fase 1 y 3 de Modelo de refactorización de Medium. En la Figura 14 se muestra el diagrama de la ruta de migración propuesta.

Al término de todas las fases se medirá el rendimiento en la Arquitectura Monolítica, los servicios REST y la Arquitectura con Microservicios resultante en base a las métricas de rendimiento establecidas en la sección 1.5.4. Cabe recalcar que esta propuesta de migración no considera refactorización de base de datos.

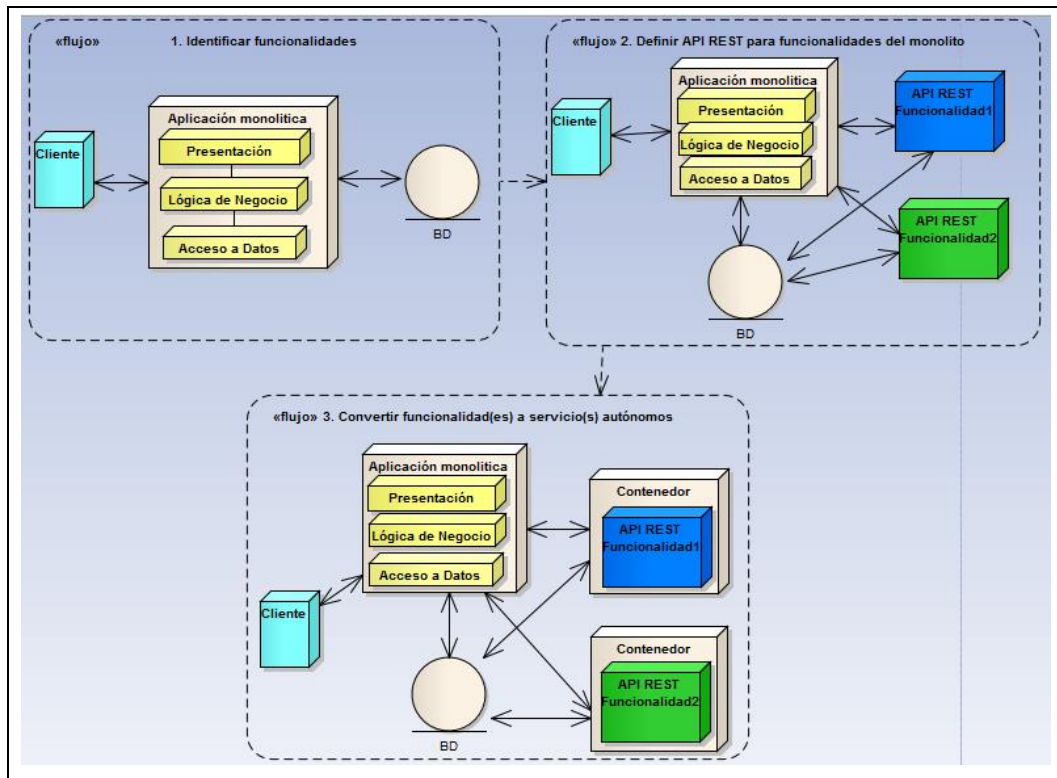


Figura 14 Diagrama de ruta de migración propuesta

Fuente: La Autora

Elaboración: La Autora

En la Figura 15 se muestra el modelo de proceso para la ruta de migración propuesta, este modelo relaciona las fases a implementarse con cada arquitectura a utilizarse durante la migración, tal como se indica a continuación:

- Fase 1: Identificar funcionalidades del monolito se implementa en la aplicación monolítica.
- Fase 2: Definir API REST para funcionalidades del monolito se implementa en la aplicación con Servicios REST. Dentro de la construcción de las API REST se incluye la subtarea que estas sean construidas con patrones de diseño: Singleton y Proxy.
- Fase 3: Convertir módulos a servicios autónomos se implementa en las aplicaciones con Microservicios

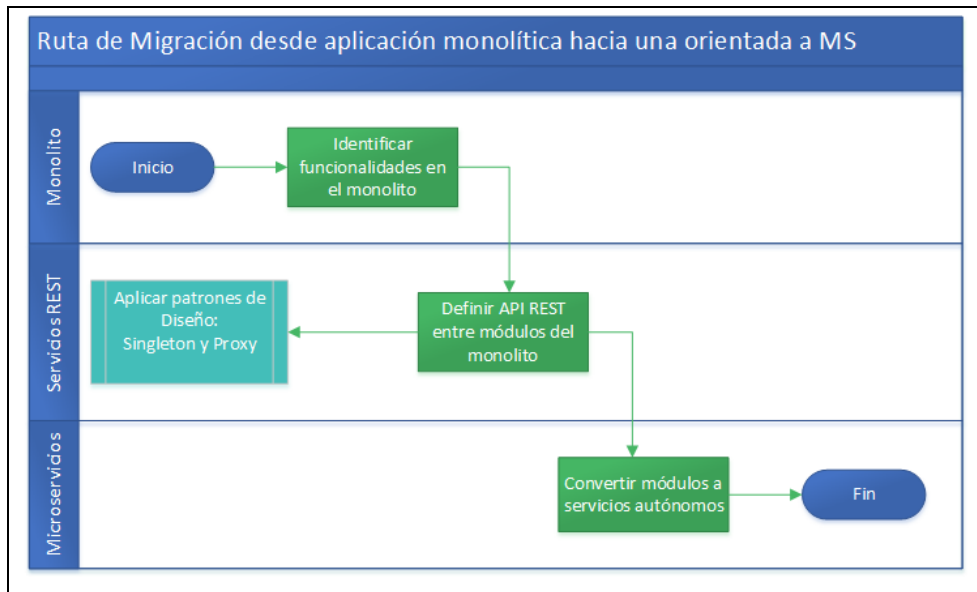


Figura 15 Modelo de proceso de Ruta de Migración propuesta

Elaboración: La Autora

Autora: La Autora

**CAPITULO III: DISEÑO E IMPLEMENTACIÓN DEL PROCESO DE MIGRACIÓN DE  
APLICACIÓN MONOLÍTICA HACIA APLICACIÓN ORIENTADA A MICROSERVICIOS**

### **3 Diseño e Implementación del proceso de migración de aplicación monolítica hacia aplicación orientada a microservicios**

Acorde a la ruta de migración diseñada en el capítulo anterior, en el presente capítulo se muestra la implementación de dicha migración la misma que parte desde una aplicación monolítica hasta alcanzar la arquitectura de microservicios. Ésta migración se realiza durante el proceso iterativo propuesto en el esquema de integración de Devops con microservicios y contenedores, también incluye el uso de estilos arquitectónicos y patrones de diseño con la finalidad de analizar el rendimiento en cada fase de migración, a su vez fue implementada con diversas tecnologías las cuales también se mencionan por cada fase.

#### **3.1 Implementación de Monolito Versión 1**

La **Fase 1** de la ruta de migración propuesta consiste en Identificar las funcionalidades del monolito por lo que acorde a este requerimiento se utilizó la aplicación monolítica: Sistema de Registro de Cursos en Línea, denominado “SRCL” a partir de ahora, a continuación se describe brevemente los requerimientos de la aplicación los cuales también se detallan en el Anexo A: Documento de Especificación de Requerimientos para SRCL.

SRCL es una aplicación web que permite la matrícula de estudiantes en cursos los cuales son ofertados por tutores independientes. Cada curso pertenece a una categoría determinada por lo que la asignación de los cursos dependerá de la edad del estudiante como se describe a continuación:

- **Junior:** de 12 a 16 años
- **Jóvenes:** de 17 a 22 años
- **Adultos:** de 23 a 50 años
- **Tercera Edad:** de 51 años en adelante

##### **3.1.1 Diseño de Monolito Versión 1.**

Como parte del diseño de la aplicación monolítica se han tomado en cuenta las funcionalidades especificadas en los requerimientos de SRCL y la representación arquitectónica de la aplicación a construir.

### 3.1.1.1 *Funcionalidades Identificadas.*

A partir de las especificaciones mencionadas, SRCL implementa las siguientes funcionalidades (Ver Tabla 11). Para las funcionalidades expuestas se han identificado dos roles principales que interactúan con la plataforma de SRCL:

- **Tutor:** Usuario que imparte un curso.
- **Estudiante:** Usuario que se matricula y recibe un curso.

Las funcionalidades están sujetas a un rol de acceso, es decir, un usuario con rol Estudiante podrá o bien registrarse, loguearse, matricularse en un curso o revisar la lista de cursos en los que se ha matriculado mientras que un Tutor tendrá acceso a revisar la lista de estudiantes de cursos asignados a su persona y registrar las notas de los mismos.

Tabla 11 Funcionalidades identificadas en SRCL

<b>Funcionalidad</b>	<b>Rol de Acceso</b>
Registro de Estudiantes	Estudiante
Login de Estudiantes	Estudiante
Matricula en Cursos	Estudiante
Consulta de Cursos tomados por un Estudiante	Estudiante
Consulta de Estudiantes de un Curso	Tutor
Registro de Notas de un curso	Tutor

Fuente: La Autora

Elaboración: La Autora

### 3.1.1.2 *Diagrama arquitectónico de la solución.*

En la primera versión de SRCL, la aplicación fue construida sin ninguna arquitectura de referencia, tal como indica la Figura 16, lo que significa que la estructura típica: GUI, Lógica de Negocio y Acceso a la base de datos están combinados en la aplicación, ésta se conecta externamente solamente con la base de datos. En el Anexo B: Modelo Entidad-Relación de SRCL se muestra el modelo de base de datos usado en SRCL.



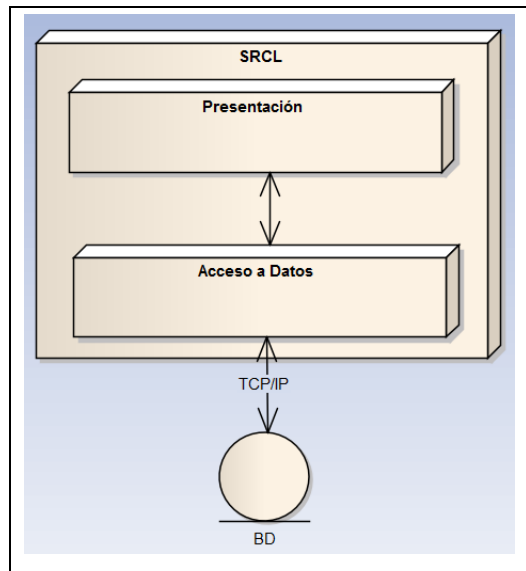


Figura 16 Arquitectura de Monolito: Versión 1

Fuente: La Autora

Elaboración: La Autora

### 3.1.2 Desarrollo de Monolito Versión 1.

Como se mencionó anteriormente, la primera versión de SRCL fue desarrollada en el lenguaje de programación PHP sin tomar en cuenta arquitectura alguna, esto quiere decir que los componentes de GUI y Lógica de Negocio están combinados en archivos PHP, existe un archivo de configuración de acceso a la base de datos el cual es llamado mediante la sentencia `<include>` según se necesita en cada uno de los archivos mencionados antes.

En estos mismos archivos se hacen las consultas que requieren las funcionalidades de la aplicación, en esta versión de la aplicación el método de acceso a la base de datos se realiza mediante consultas (queries) gestionadas directamente por la aplicación web, tal como se indica a continuación:

```
$SQL = "SELECT m.idMatricula, e.nombres, e.apellidos, m.estado, m.fecha_mat,
m.id_estudiante, m.id_curso
FROM cursos c, matriculas m, estudiantes e
WHERE c.id_curso = '$idcurso.'" AND
c.id_curso = m.id_curso AND
m.id_estudiante = e.id_estudiante
ORDER BY e.apellidos ASC";
```

En la Figura 17 se muestra la organización de la primera versión de SRCL.

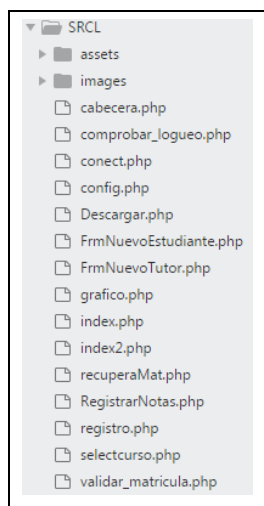


Figura 17 Organización de Monolito Versión 1

Fuente: La Autora

Elaboración: La Autora

### 3.1.3 Despliegue de Monolito Versión 1.

Al tratarse de una aplicación web con una estructura básica, se utilizó el paquete XAMP que incluye el servidor Web Apache, el SGBD MySQL y el lenguaje de programación PHP, por lo que la aplicación fue montada sobre el servidor Apache y se hizo la conexión a la base de datos de nombre **SRCL** en el puerto 3306. En el Anexo C: Despliegue de Monolito V1 se muestra el despliegue de la aplicación monolítica en su primera versión. A continuación, en la Tabla 12 se resumen las características del Monolito Versión 1:

Tabla 12 Características del Monolito Versión 1

<b>Tecnología</b>	PHP 5.6
<b>Servidor Web</b>	Apache 2.4
<b>SGBD</b>	MySQL 5.6
<b>Método de acceso a la base de datos</b>	Consultas (Queries)
<b>Patrones de Diseño</b>	Ninguno
<b>Estilo Arquitectónico</b>	Ninguno
<b>Método de despliegue</b>	Nativo (Bare Metal)
<b>Plataforma</b>	Windows

Fuente: La Autora

Elaboración: La Autora

## 3.2 Implementación de Monolito Versión 2

Para la segunda versión del Monolito se han conservado los mismos requerimientos del Monolito Versión 1, así mismo se mantienen las mismas funcionalidades establecidas en el paso anterior, no obstante se introdujeron algunos cambios como el uso de un estilo arquitectónico y otra tecnología base en cuanto a la codificación, explicados a continuación.

### 3.2.1 Diseño de Monolito Versión 2.

Para la versión 2 de SRCL se utilizó el estilo arquitectónico 3 capas, la razón de esta elección radica principalmente en que es el estilo más fácil y típico de desacoplar una aplicación de las características del Monolito Versión 1, además de familiarizarse con el modelo de aplicación monolítica mencionado por (Richardson & Smith, 2016), desde el cual se puede migrar a microservicios, por lo que esta aplicación significa el verdadero punto de partida de la ruta de migración especificada anteriormente.

#### 3.2.1.1 *Diagrama Arquitectónico de la solución.*

Entonces, para la segunda versión de SRCL la aplicación tiene 3 capas claramente definidas:

- **Capa de Datos:** Esta capa contiene componentes que acceden a la infraestructura, en este caso, a la base de datos de SRCL, esta a su vez es requerida por la capa de Lógica de negocio.
- **Capa de Lógica de Negocio (Business Logic):** Esta capa tiene componentes que son el núcleo de la aplicación e implementa la lógica de dominio, es decir las funcionalidades especificadas para SRCL las cuales son enviadas a la GUI por lo que esta capa actúa como intermediaria entre la GUI y la capa de Datos.
- **Capa de GUI (Interfaz Gráfica de Usuario):** También es llamada capa de presentación, es la capa encargada de hacer frente a las solicitudes HTTP de los usuarios mediante una interfaz web visible a través de un navegador. Esta capa se comunica únicamente con la capa de negocio.

También se añadió el componente Clases que contiene todas las clases referentes a las entidades de la base de datos, las cuales son utilizadas por la capa de Datos. A continuación se muestra el Diagrama de Componentes y el Diagrama de Despliegue para el Monolito versión 2 en la Figura 18 y Figura 19.

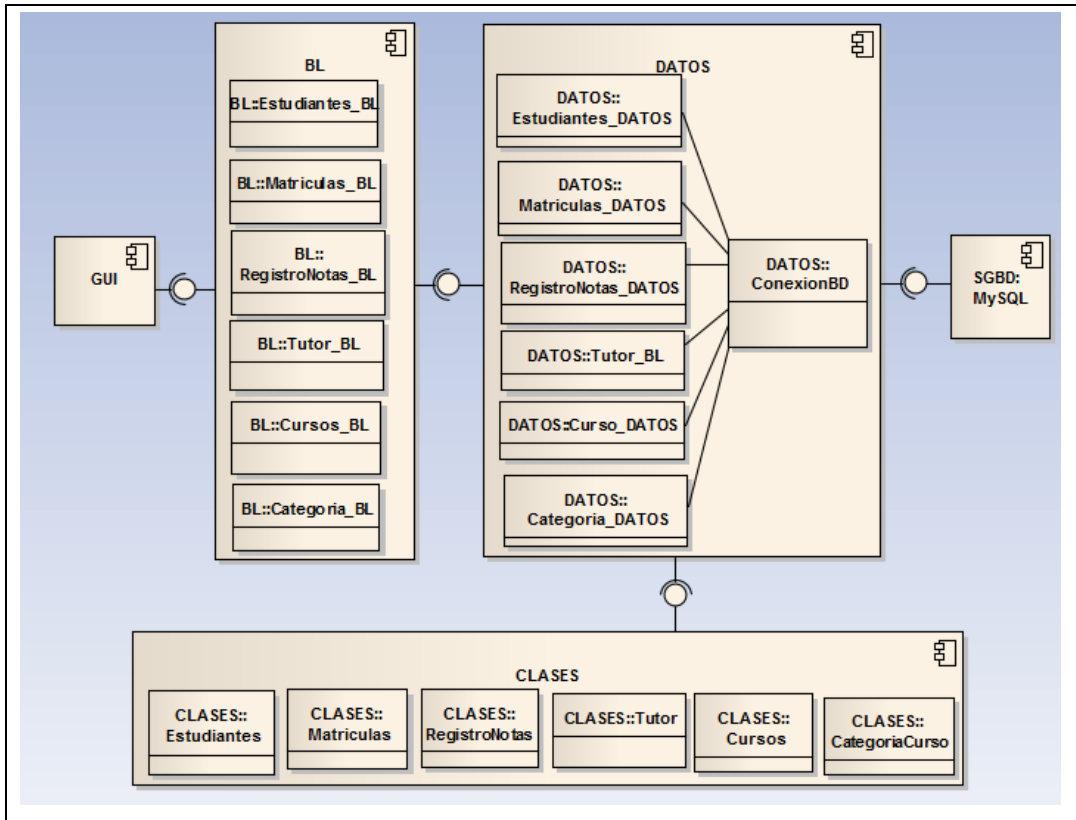


Figura 18 Diagrama de Componentes de Monolito Versión 2

Elaboración: La Autora

Autora: La Autora

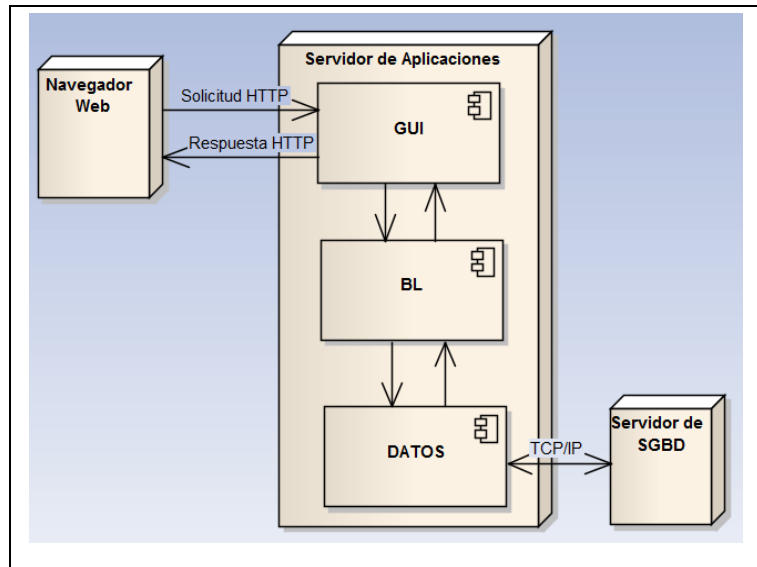


Figura 19 Diagrama de Despliegue de Monolito Versión 2

Elaboración: La Autora

Autora: La Autora

### 3.2.2 Desarrollo de Monolito Versión 2.

Acorde a la arquitectura propuesta, la aplicación fue construida bajo el estilo arquitectónico 3 capas. Como se mencionó anteriormente para la versión 2 del Monolito se modificó la tecnología base de desarrollo. Para la implementación de la aplicación se creó un proyecto Maven, para gestionar las dependencias de la aplicación (Apache Software Foundation, 2017b) en el IDE Netbeans de tipo Java Web con páginas web JSP (Polo & Villafranca, 2002) para la creación del contenido dinámico de la capa de GUI. Para las capas restantes se crearon paquetes que contienen las clases con los métodos necesarios que implementan la lógica de la aplicación (Lógica de Negocio) y métodos de acceso a la base de datos (Datos) vía procedimientos almacenados (Oracle, 2017), esto con el fin de aminorar la carga que supone para el servidor de aplicaciones acceder a la base de datos y realizar la consulta (query) además de añadir cierto nivel de seguridad.

Cabe recalcar que para esta versión y las posteriores se mantiene el mismo SGBD usado en la versión anterior. En la Figura 20 se muestran los procedimientos almacenados realizados en la base de datos.

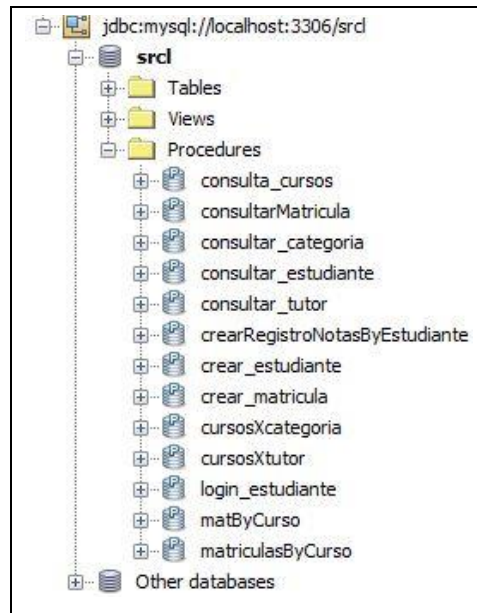


Figura 20 Procedimientos Almacenados en base de datos SRCL

Fuente: La Autora

Elaboración: La Autora

La llamada a los procedimientos almacenados se realizó como se ejemplifica a continuación

```
CallableStatement statement = connection.prepareCall("{CALL  
matByCurso(?)}");
```

Adicional a la fase anterior se incluyeron dos paquetes:

- **pkg\_CLASES:** Este paquete implementa todas las clases a partir de las entidades identificadas en la base de datos, para crear estas clases se utilizaron las herramientas: Hibernate Reverse Engineering Wizard e Hibernate Mapping Wizard del framework Hibernate (Hibernate, 2017).
- **pkg\_Servlets:** Este paquete implementa los servlets (Barrios, 2001), es decir los módulos que en conjunto con las paginas JSP reciben las solicitudes de los usuarios para luego ser procesadas en la capa de Lógica de Negocio.

En la Figura 21 se muestra la organización del Monolito Versión 2:

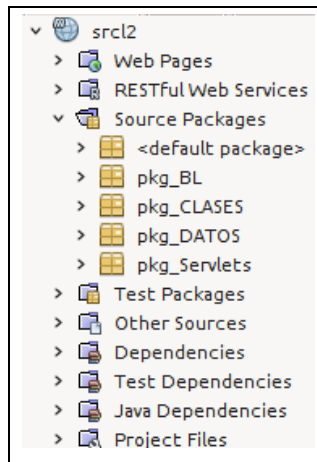


Figura 21 Organización del Monolito Versión 2

Fuente: La Autora

Elaboración: La Autora

### 3.2.3 Despliegue de Monolito Versión 2.

Para alojar la aplicación web se utilizó como servidor de aplicaciones el incluido en el IDE de desarrollo Netbeans, Glassfish 4. La aplicación corre en el host local con el puerto 8080. En el Anexo D: Despliegue de Monolito V2 se muestra la aplicación desarrollada con Java

desplegando la funcionalidad: Consultar matrículas de un curso. A continuación en la Tabla 13 se resumen las características de la segunda versión del SRCL:

Tabla 13 Características del Monolito Versión 2

<b>Tecnología</b>	Java Web 8
<b>Servidor Web</b>	Glassfish 4.0
<b>SGBD</b>	MySQL 5.6
<b>Método de Acceso a la base de datos</b>	Procedimientos Almacenados
<b>Patrones de Diseño</b>	Ninguno
<b>Estilo Arquitectónico</b>	3 capas
<b>Mapeo ORM</b>	Hibernate 4.3.1
<b>Método de Despliegue</b>	Nativo (Bare Metal)
<b>Plataforma</b>	Linux

Fuente: La Autora

Elaboración: La Autora

### 3.3 Implementación de Servicios REST

Finalizada la **Fase 1**: Identificar funcionalidades en el monolito. de la ruta de migración propuesta, y ya con una versión más estable de la aplicación monolítica (Monolito Versión 2), se puede dar inicio a la **Fase 2** el cual corresponde a: Definir API REST para funcionalidades del monolito. Para realizar esta tarea es necesario, en primer lugar diseñar los servicios que van a ser implementados en la aplicación para lo cual se han tomado en cuenta las siguientes consideraciones:

#### 3.3.1 Diseño de Servicios REST.

Según (Peña, 2016), una API REST para ser considerada como tal debe cumplir como mínimo éstas 4 reglas:

1. Usar HTTP como protocolo Cliente/Servidor sin estado, utilizando correctamente sus métodos (GET, POST, PUT DELETE, etc) y códigos de error (200, 400, 500, etc).
2. Identificación única de los recursos obtenidos desde la base de datos mediante URI's independientes del formato de respuesta (JSON, XML, etc).
3. Permitir que las respuestas sean cacheables mediante un sistema de caché.
4. El servidor debe ser un sistema capaz de ser escalado y dividido en capas.

Las reglas 1 y 2 son indiscutibles, no sólo porque son características esenciales y diferenciables de las API REST con respecto a otras arquitecturas, sino porque son las reglas que van a definir formalmente la estructura de los servicios implementados y van a influir directamente en el desarrollo de las aplicaciones clientes que se conectarán al API. Las reglas 3 y 4 afectan más a la arquitectura interna del servidor y al rendimiento que percibe el cliente pero son casi totalmente transparentes a este. Un ejemplo de un servicio web construido bajo los requisitos previamente mencionados es el indicado en la Figura 22, en la misma se especifican las partes del servicio así como la solicitud y respuesta que provee el servicio.

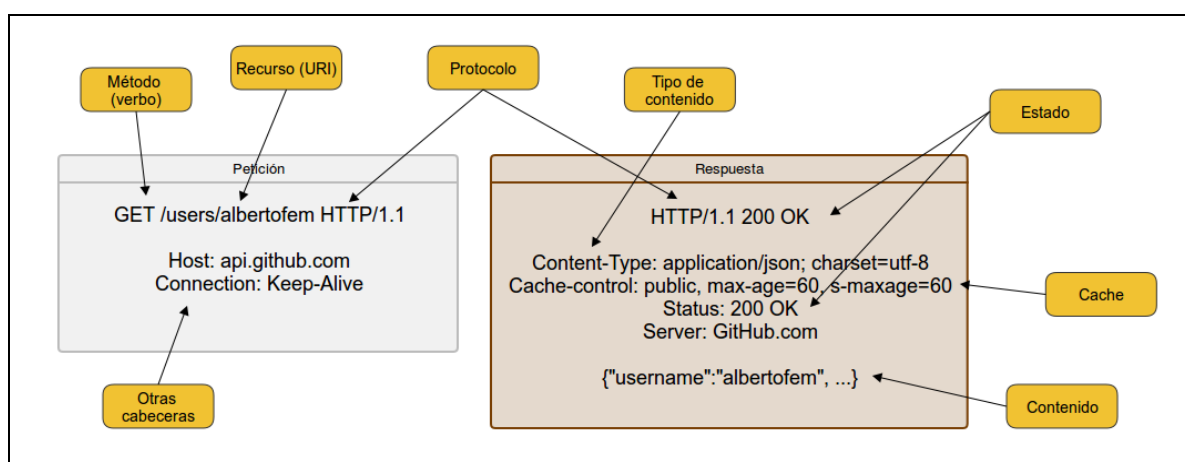


Figura 22 Ejemplo de solicitud y respuesta de un servicio RESTful

Fuente: (Fernández, 2013)

Elaboración: (Fernández, 2013)

Para los servicios web desarrollados de SRCL se ha establecido los requisitos especificados en la Tabla 14.

Tabla 14 Requisitos de Servicios Web de SRCL

<b>Protocolo de Comunicación</b>	HTTP
<b>Operación soportada por los servicios web</b>	Consulta
<b>Método de Acceso</b>	Método GET, para obtener la representación de uno o más recursos.
<b>Formato de Respuesta</b>	JSON

Fuente: La Autora

Elaboración: La Autora



### 3.3.1.1 Identificadores RESTful.

Tal como indica la regla 2 para crear API REST, es necesario establecer identificadores únicos para cada uno de los recursos a exponer, esto se hace mediante URI's (Unified Resource Identifier). Una URI es una cadena de texto que permite identificar un elemento con una estructura determinada como la que se describe a continuación (Peña, 2016):

```
{protocolo}://{dominio_o_hostname}:{puerto(opcional)}/{ruta_del_recurso}?{parametro}
```

Para asignar una URI a un recurso existen algunas reglas básicas:

- Deben ser únicas, no puede existir más de una URI para identificar un mismo recurso.
- Deben ser independientes del formato en el que se desea consultar el recurso.
- Deben mantener una jerarquía en la ruta del recurso.

No deben indicar acciones, por lo que no se usan verbos (referentes a métodos de acceso).

Adicionalmente, se aclara que los identificadores se construyen a partir de los datos que se desean exponer en el servicio, llegado a este punto en la Tabla 15 se han definido los identificadores para los recursos a los cuales se desea acceder.

Tabla 15 Identificadores RESTful para SRCL

URI	Parámetro	Recurso
http://host:puerto/srcl/rest/cursos	No se requiere parámetros	Lista todos los cursos ofertados en SRCL.
http://host:puerto/srcl/rest/cursos/estudiante/{id_estudiante}	id_estudiante	Lista todos los cursos filtrado por el id del estudiante
http://host:puerto/srcl/rest/matriculas	No se requiere parámetros	Lista todas las matriculas registradas en SRCL.
http://host:puerto/srcl/rest/matriculas/curso/{id_curso}	id_curso	Lista todas las matriculas filtradas por el id del curso.

Fuente: La Autora

Elaboración: La Autora

### 3.3.1.2 Diagrama Arquitectónico de la solución.

Acorde a lo identificadores RESTful especificados anteriormente para la aplicación SRCL, en la Figura 23 se representan los mismos como servicios, para los cuales se ha tomado en cuenta las funcionalidades a cubrir por cada uno de ellos, adicionalmente se especifican los patrones de diseño que se utilizaron durante la construcción de los servicios, estos son Singleton y Facade, en base a lo especificado en la sección 1.5.5.

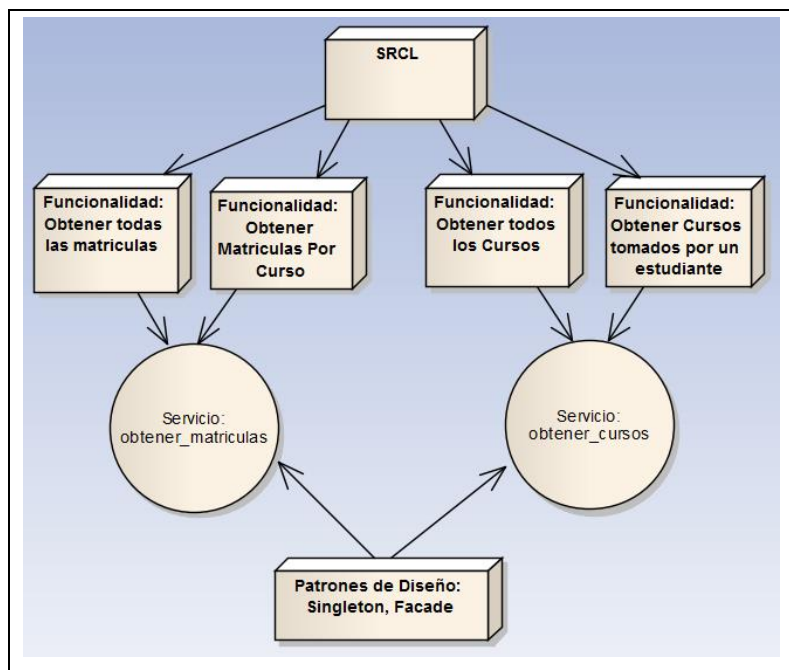


Figura 23 Arquitectura de Servicios REST en SRCL

Fuente: La Autora

Elaboración: La Autora

### 3.3.2 Desarrollo de Servicios REST.

En base a la **Fase 2** de la ruta de migración, los servicios REST fueron desarrollados independientemente del Monolito Versión 2 ya que ésta última seguirá existiendo y los nuevos servicios desarrollados deberán comunicarse con esta. Los componentes desarrollados fueron los siguientes:

- **modelo:** Este paquete implementa al igual que la versión anterior del monolito, todas las clases a partir de las entidades identificadas en la base de datos junto a los métodos de acceso para cada una de atributos mapeados con Hibernate.
- **controladores:** Este paquete contiene las clases con los métodos necesarios para extraer la información requerida por cada servicio desde la base de datos, a la vez se

implementaron métodos propios para acceder a datos específicos de la base de datos mediante procedimientos almacenados, al igual que en la versión anterior del monolito. Todas las clases han sido creadas a partir de las entidades de la base de datos y son subclases de la clase principal `AbstractFacade` de la cual heredan los métodos mencionados para acceder a la base de datos.

- **servicio:** Este paquete contiene netamente la implementación de los servicios, todas las tareas involucradas en este proceso se llevan a cabo mediante la herramienta Jersey, la cual fue incluida en las dependencias del proyecto.

Dentro del paquete **servicio** se define en primer lugar la clase `ApplicationConfig` donde se establece la ruta principal de acceso al servicio:

```
@ApplicationPath("rest")
```

Luego de esto se crean los servicios, por cada servicio se define:

- La ruta para acceder al servicio, esta ruta debe ser única según las reglas para implementar servicios RESTful.
- Llamado al método que extrae los datos desde la base de datos vía procedimientos almacenados.
- Métodos de acceso permitido, en este caso GET
- Parámetros definidos para el servicio.
- El formato de respuesta a devolver, en este caso JSON

En la Figura 24 se muestra la implementación del servicio de matrículas implementado con las características definidas previamente.

### **3.3.2.1 Patrones de Diseño implementados.**

Acorde a lo establecido en la sección **Patrones de Diseño orientados al rendimiento**, existen patrones de diseño que promueven el rendimiento de las aplicaciones como es el caso de Singleton, adicionalmente se implementó el patrón Facade, ambos descritos a continuación.

```

@Path("matriculas")
public class MatriculasCurso_REST {

    private MatriculasFacade mf = MatriculasFacade.getMatricula();

    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public List<Matriculas> allMatriculas() {
        //return mf.findAll();
        return mf.findAll();
    }

    @GET
    @Produces({MediaType.APPLICATION_JSON})
    @Path("curso/{id_curso}")
    public ArrayList<Matriculas> matriculasPorCurso(@PathParam("id_curso") Integer curso){
        return mf.matriculasByCurso(curso);
    }
}

```

Figura 24 Implementación de servicio: Matriculas por Curso

Fuente: La Autora

Elaboración: La Autora

### 3.3.2.1.1 Facade.

Conceptualmente Facade proporciona una interfaz simple para un sistema complejo al reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre estos. Fue implementado ya que era necesario al momento de construir el servicio. Como parte del proceso de construcción de los servicios se creó la clase AbstractFacade, la cual es una superclase que implementa todos los métodos de acceso a los datos del servicio, consecuentemente se crearon las clases a partir de las entidades identificadas a la base de datos, todas estas clases son hijas de AbstractFacade, la interfaz simple que describe a este patrón.

### 3.3.2.1.2 Singleton.

Este patrón permite crear una sola instancia para un recurso sin sobreabundar la memoria con objetos por cada llamada a los servicios. Dentro de los serviciosREST, existen dos maneras de implementar el patrón Singleton:

1. **Mediante la anotación @Singleton:** Esta anotación es provista por EJB, la cual viene incluida al crear cada una de las subclases de AbstractFacade. Adicionalmente se implementó el patrón. Para un singleton EJB, siempre hay una instancia por aplicación, no importa cuántos clientes accedan a ella. A menudo, los servicios REST ocupan objetos singletons, estos se crean como resultado de una solicitud, se procesa y luego descarta tan pronto como la solicitud se ha completado (Bateman, 2012).

**2. Codificar directamente el patrón Singleton:** Este patrón de diseño se encarga de que una clase determinada únicamente pueda tener un único objeto (Moya, 2015). Para conseguir que una clase sea de tipo Singleton se necesita:

- El constructor de la clase debe ser privado. De esa forma ningún programa será capaz de construir objetos de este tipo.
- Se necesita una variable estática privada que almacene una referencia al objeto que se va a crear a través del constructor.
- Por último un método estático público que se encarga de instanciar el objeto la primera vez y almacenarlo en la variable estática.

La implementación de Singleton se llevó a cabo de las dos formas, no obstante, al codificar directamente el patrón de diseño se consigue entender e implementar directamente la semántica y sintaxis del mismo, a continuación se muestra un ejemplo de la segunda alternativa en la Figura 25.

```
40 //Singleton
41 //1. Variable estatica privada
42 private static MatriculasFacade mf ;
43
44 //2. Constructor privado
45 private MatriculasFacade() {
46     super(Matriculas.class);
47 }
48
49 //3.metodo de acceso publico estatico
50 public static MatriculasFacade getMatricula() {
51     if (mf == null){
52         mf = new MatriculasFacade();
53     }
54     else{
55         System.out.println("Ya existe un objeto de tipo MatriculaFacade");
56     }
57     return mf;
58 }
```

Figura 25 Implementación de Singleton mediante codificación

Fuente: La Autora

Elaboración: La Autora

En la Figura 26 se muestra la organización final de la implementación para los servicios REST.

### 3.3.3 Despliegue de Servicios REST.

Para desplegar los servicios REST se mantuvo el uso del servidor Glassfish. Las API REST se construyeron en un solo proyecto de software por lo que todos los servicios corren en el host

local, con el puerto 8080. En el Anexo E: Despliegue de Servicios REST se muestra el despliegue del servicio obtener Matrículas. En la Tabla 16 se resumen las características de los servicios REST implementados en esta segunda fase.

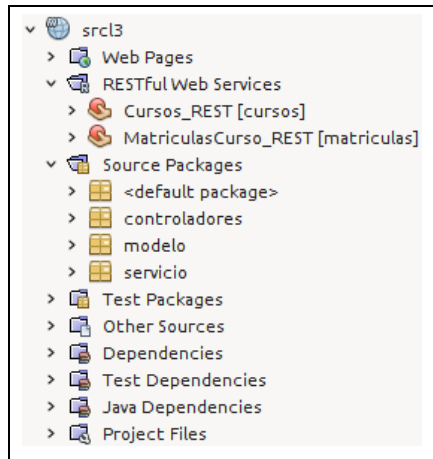


Figura 26 Organización de Servicios REST

Fuente: La Autora

Elaboración: La Autora

### 3.4 Implementación de Aplicación orientada a Microservicios

Completadas las Fases 1 y 2 de la ruta de migración, a continuación se presenta la implementación de la Fase 3: Convertir funcionalidades en servicios autónomos

En esta fase, cada servicio implementado y desarrollado en la fase anterior se despliega por separado mediante contenedores que a su vez se relacionan con la aplicación monolítica aun existente a través de API REST. En esta fase se reusó el patrón de diseño Singleton aplicado a los servicios desarrollados en la fase anterior y se implementó uno más: el patrón de diseño Proxy. Adicional a esto, se utilizaron patrones de diseño propiamente de Microservicios, explicados más adelante:

- Strangler
- Single Service per Host
- Service Registry
- API Gateway

Tabla 16 Características de Servicios REST implementados

<b>Tecnología</b>	Java RESTful
<b>Servidor Web</b>	GlassFish 4.0
<b>SGBD</b>	MySQL 5.6
<b>Método de acceso a la base de datos</b>	Procedimientos Almacenados
<b>Patrones de Diseño</b>	Facade, Singleton
<b>Estilo Arquitectónico</b>	REST
<b>Mapeo ORM</b>	Hibernate 4.3.1
<b>Método de Despliegue</b>	Nativo (Bare Metal)
<b>Plataforma</b>	Linux

Fuente: La Autora

Elaboración: La Autora

### 3.4.1 Diseño de los Microservicios.

En el diseño de la arquitectura de microservicios para la aplicación web SRCL se tomaron en cuenta factores importantes como los servicios REST desarrollados previamente, patrones de diseño orientados al rendimiento y patrones de diseño orientados a la construcción de una arquitectura de microservicios y el despliegue de los mismos, según se describe a continuación.

#### 3.4.1.1 Patrones de Diseño.

Para construir la arquitectura de microservicios se reusaron los servicios construidos en la fase anterior, por lo que se mantiene el uso del patrón Singleton, se prescindió de Facade ya que cada servicio engloba un funcionalidad específica por lo que ya no es necesario una interfaz principal de acceso. Adicional a Singleton, en la arquitectura de microservicios se implementó en patrón Proxy, estos dos patrones apoyan al rendimiento en cualquier aplicación independientemente del estilo arquitectónico usado. En cuanto a los patrones de diseño para construir arquitecturas de microservicios, en (Richardson, 2017) se mencionan varios de los cuales se usaron los siguientes:

##### 3.4.1.1.1 Strangler.

Este patrón mencionado también en la sección 2.1.3 es más bien conceptual y aplicable durante un proceso de refactorización o migración como el que se realiza en este trabajo donde la migración se realiza gradualmente y la aplicación monolítica no desaparece sino que se van

extrayendo de a poco cada una de las funcionalidades para convertirlas en API REST que se comunican con la aplicación inicial.

#### *3.4.1.1.2 Single Service per Host.*

Este patrón de diseño indica que cada instancia de servicio debe estar alojado en su propio host con el fin de controlar y restringir el uso de los recursos de procesamiento por cada servicio, entre otras ventajas. Para la arquitectura de microservicios propuesta, se han usado dos hosts, uno por cada servicio implementado.

#### *3.4.1.1.3 Service Discovery.*

Este patrón de diseño de microservicios consiste en una base de datos de los servicios, es decir cada instancia de servicio se registra para que el cliente sepa a donde dirigir sus peticiones, para implementar este patrón se utilizó el componente de software de Netflix OSS (Open Source Software): Eureka, explicado en las secciones posteriores.

#### *3.4.1.1.4 API Gateway.*

Este patrón de diseño mencionado anteriormente en la sección 1.1.4 implementa un punto de acceso único a las solicitudes entrantes de diferentes clientes, las mismas que son enrutadas y dirigidas al o los servicios correspondientes, para implementar este patrón se utilizó otro componente de Netflix OSS: Zuul.

### **3.4.1.2 Herramientas de Despliegue.**

Para respaldar los requerimientos previamente especificados de la arquitectura orientada a microservicios se han considerado las siguientes herramientas para el despliegue:

#### *3.4.1.2.1 Eureka.*

Es un software que forma parte del proyecto de Netflix OSS (Qiangdavidliu, 2014). Permite a los servicios registrarse en tiempo de ejecución, facilitando su localización a los clientes de la arquitectura de microservicios. Similar a un servicio de DNS con características adicionales como equilibrio de carga del lado del cliente. Dadas estas características, Eureka implementa el patrón Service Discovery.



#### 3.4.1.2.2 *Zuul.*

Como define (Netflix, 2014), Un desafío común al construir una arquitectura de microservicios es proporcionar una interfaz unificada a los clientes de un sistema. Aun cuando la aplicación esté dividida en servicios, este hecho no debe ser perceptible a los clientes. Para dar respuesta a este problema existe Zuul, otro proyecto de Netflix OSS, Zuul actúa como un proxy inverso que proporciona un punto de acceso unificado al sistema que permite que cualquier cliente consuma servicios alojados en diferentes hosts además de proporcionar capacidades de enrutamiento dinámico, seguridad y monitorización de las solicitudes, balanceador de carga a través de otro componente de Netflix OSS integrado en Zuul por lo que es integrable con Eureka. Dadas estas características, Zuul implementa el patrón Proxy y a su vez el patrón de microservicios API Gateway.

#### 3.4.1.2.3 *Spring Cloud.*

Es una librería open source perteneciente al framework Spring que proporciona las herramientas necesarias para implementar los patrones más comunes de sistemas distribuidos como el Descubrimiento de servicios y enrutamiento dinámico, a través de su integración con componentes de Netflix OSS (Pivotal Software, 2017).

#### 3.4.1.2.4 *Docker.*

Es un proyecto de código abierto que permite la creación de contenedores o máquinas virtuales ligeras que permitan desplegar aplicaciones de manera portable y ligera. Los contenedores Docker son livianos por diseño e ideales para permitir el desarrollo de aplicaciones de microservicios (Docker Inc., 2017b).

#### **3.4.1.3 *Propuesta de Implementación.***

Con el panorama claro acerca de los patrones de diseño a utilizar dentro de la arquitectura de microservicios, los componentes a reusar como son los servicios REST desarrollados en la fase anterior y revisadas las herramientas que permitan implementar los patrones deseados y reusar los servicios existentes, a continuación, se explica la propuesta de implementación de la aplicación con una arquitectura de microservicios.

### 3.4.1.3.1 Diagrama arquitectónico de la solución.

Tal como se observa en la Figura 27, la propuesta de implementación de la arquitectura de microservicios refleja todos los componentes de la aplicación, la cual, al estar inmersa en un proceso de migración contempla los servicios desplegados independientemente lo cuales deben coexistir con la aplicación monolítica hasta que toda la aplicación sea gestionada mediante microservicios, de igual manera, se mantiene la conexión con la base de datos MySQL local.

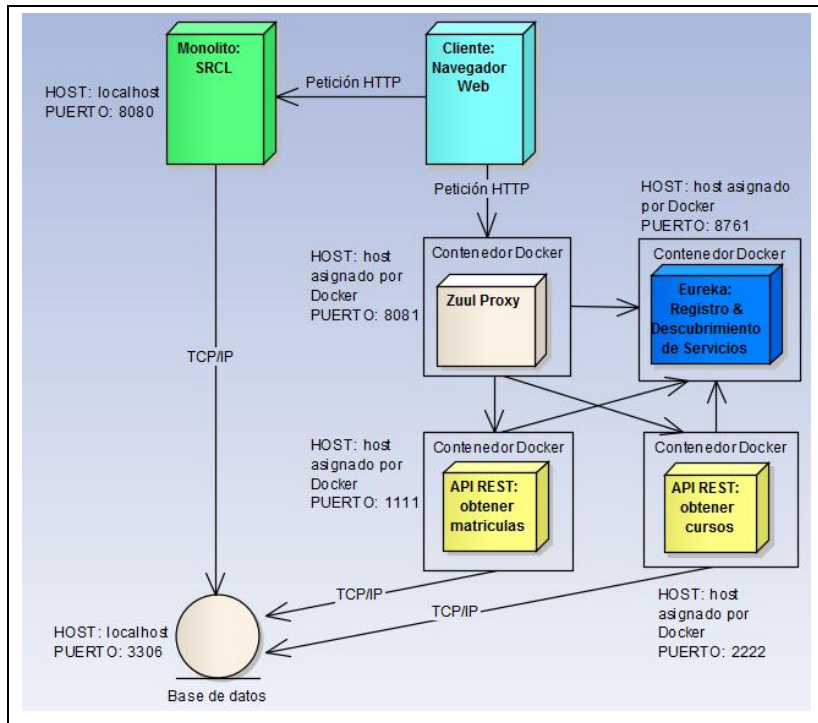


Figura 27 Propuesta de implementación de Arquitectura de Microservicios

Fuente: La Autora

Elaboración: La Autora

En cuanto a los servicios REST, según el patrón Service Discovery, todos los servicios deben registrarse en un servicio que actúe como una base de datos de los servicios, este patrón es implementado mediante Eureka. Las peticiones entrantes son gestionadas por un proxy que redirige las peticiones al servicio correspondiente, aplicar un proxy supone la implementación de una puerta de acceso principal a los servicios como la que describe el patrón API Gateway, este patrón es implementado con el software Zuul.

Para que los servicios REST desarrollados puedan considerarse microservicios deben ser implementados y desplegados independientemente unos de otros, por lo que para la

arquitectura propuesta cada servicio se ha desplegado en contenedores de Docker, al hacerlo, se establece un host y puerto diferente por cada servicio REST, en la práctica, esto significa implementar el patrón Single Service per host. Adicionalmente los servicios de Eureka y Zuul también se desplegaron en contenedores.

#### *3.4.1.3.2 Gestión de Solicitudes con Arquitectura de Microservicios.*

Con la arquitectura propuesta, el flujo normal de gestión para una solicitud entrante sería como el que se describe en la Figura 28.

1. En primer lugar, los servicios REST desplegados deben registrarse en Eureka para ser identificados durante el procesamiento de la petición.
2. Cuando se realiza una petición desde un cliente, en este caso un navegador web, la solicitud se dirige primeramente hacia Zuul.
3. Zuul consulta a Eureka si el servicio que se está solicitando se encuentra registrado.
4. En caso afirmativo Eureka responde a Zuul con la identificación del servicio. De no ser así, responderá con valor nulo por lo que la solicitud no se procesará y devolverá el código de error HTTP 404.
5. Una vez que Zuul recibe confirmación de la existencia del servicio, redirige la petición hacia el microservicio correspondiente.
6. El microservicio procesa la solicitud y retorna la respuesta en el formato especificado (en este caso JSON) a Zuul y este a su vez la entrega al cliente con código HTTP 200.

#### **3.4.2 Desarrollo de Microservicios.**

Para la implementación de la arquitectura de microservicios se utilizó el framework Spring, ya que este a su vez se implementa la librería Spring Cloud para implementar los patrones más comunes de los microservicios.

Puesto que los microservicios deben ser desarrollados y desplegados independientemente, se creó un proyecto por cada servicio desarrollado utilizando el framework Spring, incluido los servicios de Eureka y Zuul. Finalmente, cada uno de estos servicios fue desplegado en un contenedor de Docker. Se usó Spring Initializr (Spring Initializr & Pivotal Web Services, 2017), una herramienta web de Spring que ayuda a configurar y descargar las dependencias iniciales de la aplicación en un proyecto para posteriormente descargarlo.

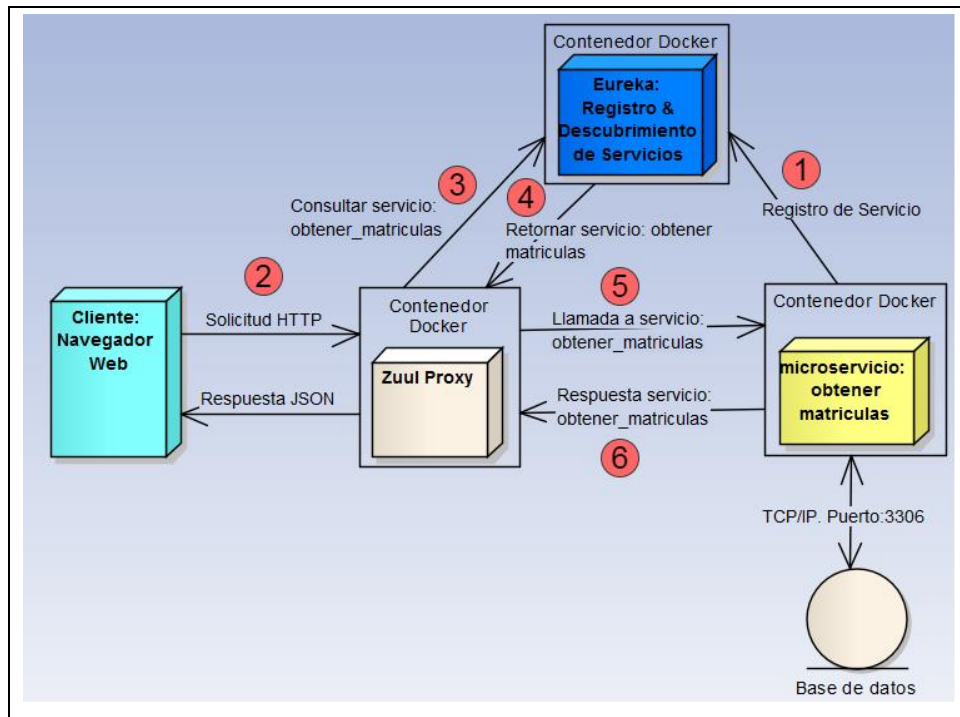


Figura 28 Gestión de solicitudes HTTP en arquitectura de Microservicios

Fuente: La Autora

Elaboración: La Autora

### 3.4.2.1.1 Configuración.

1. Se añade la dependencia del servidor de Eureka a la aplicación:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

2. En el archivo de configuración, `application.yml`, se configura el puerto en el que va a correr la aplicación (`port`) y el host con el que va a ser identificado por otros servicios (`hostname`), como muestra la Figura 29.

```
1  server:
2  port: 8761
3  eureka:
4  instance:
5  hostname: localhost
```

Figura 29 Configuración de Eureka en Spring

Fuente: La Autora

Elaboración: La Autora

3. Finalmente se añade la anotación `@EnableEurekaServer` al inicio de la clase `main`, para habilitar a la aplicación como servidor Eureka.

### 3.4.2.2 *Codificación de API REST.*

Esta sección muestra cómo se realizó la configuración de los servicios realizados en la Fase 2 de la ruta de migración, esta vez con el Framework Spring.

#### 3.4.2.2.1 *Configuración.*

Los servicios REST al registrarse en Eureka actúan como clientes de esta herramienta por lo que fue necesario añadir la dependencia de Cliente Eureka en cada aplicación de los servicios REST, así mismo se añadieron las dependencias del conector MySQL y JPA, para manejar la persistencia de la aplicación, a continuación se muestran las dependencias añadidas:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Para ambos servicios, en el archivo de configuración `application.yml`, se establece el nombre de la aplicación el cual servirá para ser identificado por Eureka (`name`), también se configura el `datasource` desde el cual se extraerán los datos (`datasource`) y la instancia de Eureka hacia al cual apuntarán los servicios para registrarse (`eureka`) y por último, el puerto sobre el cual correrá la lo servicios (`port`). Esta configuración se muestra en la Figura 30.

```

1  spring:
2    application:
3      name: matriculas-ms
4    jpa:
5      hibernate:
6        ddl-auto: none
7    datasource:
8      url: jdbc:mysql://localhost:3306/srcl?useSSL=false
9      username: root
10     password: root
11     initialize: false
12  eureka:
13    client:
14      serviceUrl:
15        defaultZone: http://localhost:8761/eureka/
16  server:
17    port: 1111

```

Figura 30 Configuración de API REST en Spring

Fuente: La Autora

Elaboración: La Autora

#### 3.4.2.2.2 Codificación.

Ambos servicios: obtener\_matriculas y obtener\_cursos se organizaron en 3 paquetes que se describen a continuación:

1. **controllers:** El paquete controllers tiene la configuración de las URI's del servicio así como la llamada a métodos o procedimientos almacenados que los datos solicitados por los clientes. Para ambos servicios se han mantenido las URI's y la llamada a los procedimientos almacenados de la fase anterior.
2. **modelo:** Este paquete contiene todas las clases de las entidades mapeadas desde la base de datos, para ello se reusaron las clases obtenidas con Hibernate de la fase anterior. Aquí también se configuran los procedimientos almacenados a ser utilizados por la aplicación como se indica a continuación:

```

@NamedStoredProcedureQuery(
    name = "matByCurso",
    procedureName = "matByCurso",
    parameters = {
        @StoredProcedureParameter(name = "_idcurso", mode =
ParameterMode.IN, type = Integer.class)
    })

```

En cuanto a la implementación del patrón de diseño Singleton, en Spring es posible configurarlo a través de la anotación Scope con valor “singleton” que se define por cada entidad mapeada como se describe a continuación:

```
@Scope(value="singleton")
```

De esta manera se asegura que solo se devolverá una instancia por cada solicitud HTTP procesada, cabe mencionar que este es el alcance por defecto para un API REST en Spring.

- 3. repositorio:** Contiene una interfaz que implementa todos los métodos comunes a una entidad (Create-Read-Update-Delete) que pueden implementarse en un servicio, estos métodos los hereda de la librería CrudRepository.

Finalmente, en el método main de cada aplicación se añade la anotación @EnableDiscoveryClient, para permitir que Eureka busque y registre el servicio implementado. Es así que las aplicaciones de las API REST quedan organizadas como indica la Figura 31.

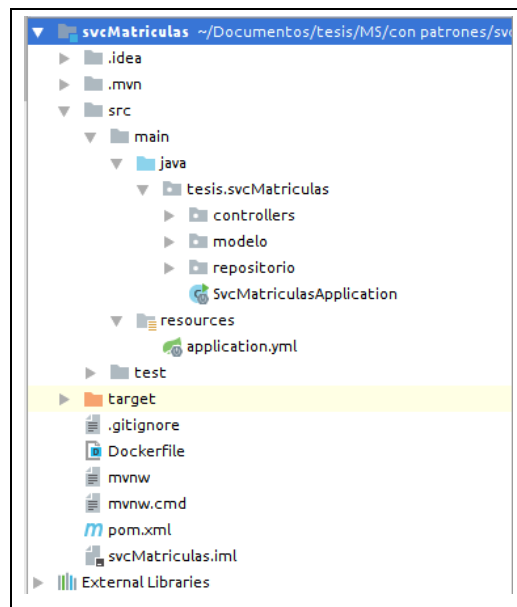


Figura 31 Organización del API REST matrículas

Fuente: La Autora

Elaboración: La Autora

### 3.4.2.3 Codificación de Zuul.

En esta sección se explica el proceso de configuración y codificación de la herramienta que implementa el patrón de diseño Proxy y el patrón de construcción de microservicios: Zuul.

#### 3.4.2.3.1 Configuración.

1. Se añaden las dependencias de Zuul y Eureka Cliente a la aplicación:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

2. En el archivo de configuración, `application.yml`, se configura el puerto en el que va a correr la aplicación (`port`), el nombre de la aplicación (`name`), y los servicios que Zuul va a ser capaz de filtrar (`routes`), especificando por cada uno de ellos: el id de la aplicación (`serviceId`), la ruta del servicio (`path`) y si se va a eliminar el prefijo de la ruta antes de reenviarla (`strip-prefix`), también se configura la url de Eureka con quien deberá consultar primero acerca de la existencia de los servicios (`defaultZone`). Esta configuración se muestra la Figura 32.

```
1  server:
2    port: 8081
3  spring:
4    application:
5      name: zuul-proxy
6  zuul:
7    routes:
8      matriculas-ms:
9        path: /srcl/rest/matriculas/**
10       serviceId: matriculas-ms
11       strip-prefix: false
12      cursos-ms:
13        path: /srcl/rest/cursos/**
14        serviceId: cursos-ms
15        strip-prefix: false
16  eureka:
17    client:
18      register-with-eureka: false
19      service-url:
20        defaultZone: http://localhost:8761/eureka/
```

Figura 32 Configuración de Zuul en Spring

Fuente: La Autora

Elaboración: La Autora



3. Finalmente se añaden las anotaciones `@EnableDiscoveryClient` y `@EnableZuulProxy` al inicio de la clase `main`, la primera para habilitar a la aplicación como un servicio que también debe ser reconocido por Eureka y la segunda para habilitarlo como el servicio de enrutamiento y proxy Zuul.

### **3.4.3 Despliegue de Microservicios** En esta sección se explica el paso final que constituye a una arquitectura de microservicios la cual es el despliegue, para ello se explican los dos métodos de despliegue usados para los microservicios: **Nativo (Bare Metal) y Contenedores (usando Docker).**

#### **3.4.3.1 Despliegue local**

Las aplicaciones desarrolladas en Spring vienen configuradas por defecto con el servidor web Tomcat Versión 8 embebido lo que elimina la necesidad de descargar y configurar un servidor web tanto localmente como en Docker. Con la configuración realizada a cada aplicación en la fase anterior, se puede explicar el procedimiento para poner en marcha la arquitectura de microservicios.

1. Como se explica en la sección 3.4.1.3.2, el primer paso es que las API REST estén activas para que puedan ser descubiertas y posteriormente registradas por Eureka. El servicio de Eureka cuenta con una interfaz disponible a la cual se accede desde el puerto que se configuró para el mismo (Ver Anexo F: Despliegue local de Eureka ). Esta interfaz muestra los servicios que se encuentran activos y disponibles así que se sobreentiende que Eureka los ha descubierto y registrado.
2. Para comprobar que el servicio esté funcionando correctamente se accede a este desde el host y puerto especificado en la configuración (Ver Anexo G: Despliegue local de API REST matriculas). Es importante comprobar la funcionalidad del servicio ya que si no funciona por sí mismo, tampoco va a funcionar con Zuul.
3. Por último se comprueba que Zuul esta redirigiendo las solicitudes al servicio correspondiente al hacer una petición en el puerto asignado a Zuul: 8081. (Ver Anexo H: Despliegue Local de Zuul).

#### **3.4.3.2 Despliegue en Docker**

Hasta ahora se ha descrito el proceso de configuración y despliegue de cada una de las aplicaciones para que funcionen de manera local, a continuación se explica el despliegue de la aplicación en contenedores de Docker.

### 3.4.3.2.1 Configuración del contenedor en las aplicaciones: Perfiles en Spring.

La configuración realizada para un despliegue local de cada aplicación es conveniente en un entorno de desarrollo, no obstante utilizar la tecnología de contenedores en entorno de despliegue y aún en entorno de desarrollo es muy beneficioso ya que permite que la aplicación sea portable y ligera.

En Spring es posible establecer varias configuraciones según el entorno en el que se esté trabajando, a través de perfiles. Por lo que se creó un perfil en cada aplicación desarrollada que funcione mientras se trabaje con contenedores de Docker. En la Figura 33 se ve la configuración del perfil para la API REST de obtener\_matriculas.

```
18 ---
19 spring:
20   profiles: container
21   application:
22     name: matriculas-ms
23   jpa:
24     hibernate:
25       ddl-auto: none
26   datasource:
27     url: jdbc:mysql://srclbd:3306/srcl?useSSL=false&noAccessToProcedureBodies=true
28     username: root
29     password: root
30     initialize: true
31   eureka:
32     client:
33       serviceUrl:
34         defaultZone: http://eureka:8761/eureka/
35   server:
36     port: 1111
```

Figura 33 Configuración de perfil "container" para API REST obtener\_matriculas

Fuente: La Autora

Elaboración: La Autora

Para añadir una nueva configuración, además de los campos ya establecidos se añade el nombre del nuevo perfil (`profiles`). En el caso de la configuración de Eureka, es necesario establecer en el campo `defaultZone` el host que asignará Docker a la aplicación de Eureka, este paso se explica más adelante.

### 3.4.3.2.2 Creación de imágenes en Docker: Dockerfile.

Luego de instalar Docker (Ver Anexo I: Instalación de Docker en Ubuntu 16.04), es posible empezar a crear imágenes de las aplicaciones realizadas y correrlas en contenedores independientemente. En Docker, una imagen es la base que se configura para a partir de ella

crear contenedores, por ello se deben especificar los pasos y comandos que se irán ejecutando para crear la imagen. Por lo general una imagen siempre se crea a partir de otras existentes.

Para crear las imágenes de las aplicaciones realizadas en Docker se crea un archivo Dockerfile que contiene los siguientes comandos:

1. Configurar la imagen base para usar. Se agrega la imagen de Java 8 (con la cual corren las aplicaciones) con el comando FROM. Así, cada vez que se cree un contenedor para las aplicaciones se usará la imagen descargada desde Docker Hub (Docker Inc., 2016).
2. Con el comando ADD se especifica la ubicación desde el cual se copiará el ejecutable de la aplicación de la maquina host al contenedor.
3. Con el comando ENTRYPOINT se establece el punto de entrada (comando) que se cargará por defecto al iniciar el contenedor. En este caso es necesario definir que se cargará la aplicación con el perfil creado en el paso anterior en modo activado.

Este procedimiento se sigue para las cuatro aplicaciones que se desplegarán en contenedores (Eureka, API REST obtener\_matriculas, API REST obtener\_cursos, Zuul). El archivo Dockerfile resultante se muestra a continuación:

```
FROM java:8
ADD /target/svcMatriculas.jar svcMatriculas.jar
ENTRYPOINT ["java", "-Dspring.profiles.active=container", "jar",
"svcMatriculas.jar"]
```

Al final de este paso se tendrán configurados 4 Dockerfile que crean imágenes de Docker, una por cada aplicación realizada, estos pueden publicarse y usarse en Docker Hub.

#### *3.4.3.2.3 Comunicación entre servicios: Docker Compose.*

Luego de crear las imágenes para cada una de las aplicaciones desarrolladas, es momento de desplegarlas en contenedores, estos contenedores se crean a partir de las imágenes existentes. Es posible correr cada contenedor por separado con el comando run de Docker pero existe una herramienta también de Docker llamada Docker Compose (Docker Inc., 2017a) que permite definir, ejecutar y enlazar múltiples contenedores Docker en un entorno aislado. Con Compose se usa un archivo YAML para configurar los servicios (a partir de los cuales se crearán los contenedores), luego, con un solo comando se crean e inician todos los servicios a

partir de la configuración realizada. Para ver el manual de instalación de esta herramienta, ver Anexo J: Instalación de Docker Compose en Ubuntu 16.04. A continuación se describen los pasos para crear el archivo de configuración `docker-compose.yml`

1. Especificar la versión del archivo `docker-compose.yml` a usar (Github, 2017). Es importante que la versión del archivo a usar sea compatible con la versión de Docker Compose instalada, de no ser así algunas configuraciones podrían fallar por comandos no reconocidos entre versiones.
2. Configurar cada uno de los servicios. Se asigna un nombre a cada servicio y se especifica la configuración para cada uno de ellos, entre los comandos que se usaron están:
  - a. **build:** Con este comando se especifica al servicio la ruta al contexto de compilación, es decir donde se encuentra el Dockerfile de la imagen a usarse.
  - b. **ports:** Permite establecer el puerto con el que el servicio estará expuesto a la máquina host.
  - c. **expose:** Permite establecer el puerto con el que el servicio será identificado por otros servicios especificados dentro del archivo de configuración YAML.
  - d. **depends\_on:** Sirve para indicar la dependencia entre servicios, en este caso los servicios de las API REST dependen de Eureka y Zuul depende de Eureka y las API REST.
  - e. **links:** Este comando permite enlazar los contenedores creados a partir de los servicios especificados. Se mantiene la misma relación que en `depends_on`.
  - f. **extra\_hosts:** Permite agregar un host al contenedor con un nombre como identificador, este host estará disponible mientras el contenedor este levantado. Este comando se usó para añadir el host de la base de datos al contenedor de las API REST ya que la base de datos se mantuvo desplegada localmente. Para ver la configuración adicional que permite que los contenedores se enlacen con la base de datos local, ver Anexo K: Configuración de Docker con la base de datos local.

Es importante que este archivo de configuración este ubicado en un directorio desde donde pueda acceder a las aplicaciones. En el Anexo L: Archivo de configuración `docker-compose.yml`, se muestra el archivo final. Cabe recalcar que al crear el perfil: `container` de configuración en las aplicaciones de Spring se utilizó el nombre de los servicios que se

especifican en el archivo `docker-compose.yml` como `hosts`. Esto es necesario ya que Docker asigna `hosts` a los contenedores aleatoriamente y con Docker Compose, al usar el nombre del servicio como `host` se estandariza y asegura que la aplicación Spring siempre va a encontrar el `host` designado a los servicios con los que tiene que enlazarse.

#### 3.4.3.2.4 *Levantar contenedores.*

Con la configuración realizada para coordinar y comunicar todos los servicios, es momento de desplegar la arquitectura de microservicios implementados en contenedores de Docker. A continuación se describen los pasos para levantar los contenedores:

1. En el directorio donde se encuentra el archivo de configuración `docker-compose.yml`, ejecutar el comando `docker-compose up`. Este comando leerá y ejecutará todos los pasos especificados en el archivo de configuración YAML creando un entorno aislado para los servicios especificados comunicándolos a través de una red que se crea por defecto. En este momento también se crearán las imágenes especificadas en cada uno de los `Dockerfile` creados y consecuentemente se levantarán los contenedores (Ver Anexo M: Levantar contenedores con Docker Compose).
2. Con los comandos `docker images` y `docker ps -a` se puede comprobar las imágenes que han sido creadas y los contenedores actualmente levantados respectivamente, este último es importante ya que permite conocer metadatos relevantes de los contenedores como el `id`, el nombre de la imagen en la cual se basa, el comando que está ejecutando, el tiempo de creación y de estar activo, los puertos que tiene asignados finalmente el nombre que tiene asignado el contenedor (Ver Anexo N: Ver contenedores activos en Docker).
3. Con el nombre o el `id` del contenedor se puede conocer el `host` que le ha asignado Docker a cada contenedor, para ello se usa el comando `docker exec id_contenedor cat /etc/hosts`. Al correr este comando para las API REST se observa que se ha añadido el `host` que se especificó en el archivo `docker.compose.yml` en el que está corriendo en la base de datos (Ver Anexo O: Ver `hosts` de contenedores Docker).
4. Conociendo cada uno de los `hosts` asignados por Docker a los contenedores, se puede probar la arquitectura de microservicios de la misma manera que se hace localmente, no obstante los `hosts` y puertos serán los que se hayan configurado y asignado en Docker (Ver Anexo P: Despliegue de Eureka en contenedor Docker, Anexo Q: Despliegue de API REST Matrículas en contenedor Docker y Anexo R: Despliegue de Zuul en contenedor Docker).

5. Para dar de baja los contenedores se usa el comando `docker-compose down`, esto parará y eliminará los contenedores activos mientras que las imágenes creadas seguirán disponibles, aunque es posible eliminarlas con el comando `docker rmi id_imagen`.

Finalmente, en la Tabla 17 se resume las características principales de la arquitectura de microservicios implementada.

Tabla 17 Características de la arquitectura de MS implementada

<b>Tecnología</b>	Java RESTful
<b>Servidor Web</b>	Tomcat 8
<b>SGBD</b>	MySQL 5.6
<b>Método de acceso a la base de datos</b>	Procedimientos Almacenados
<b>Patrones de Diseño</b>	Orientados al rendimiento (GOF): Singleton, Proxy
	Orientados a microservicios: Strangler, Single Service per Host, Service Discovery, API Gateway
<b>Estilo Arquitectónico</b>	Microservicios
<b>Mapeo ORM</b>	Hibernate 4.3.1
<b>Framework</b>	Spring
<b>Despliegue</b>	Docker Container
<b>Plataforma</b>	Linux

Fuente: La Autora

Elaboración: La Autora

## **CAPITULO IV: PRUEBAS Y RESULTADOS**

## 4 Pruebas y Resultados

El capítulo de Pruebas y Resultados consiste en describir paso a paso las pruebas realizadas por cada una de las aplicaciones desarrolladas frente a los recursos que resultan más afectados cuando se procesan múltiples peticiones HTTP. Estas pruebas arrojan resultados los cuales serán analizados para consecuentemente generar resultados que permitan definir qué arquitectura refleja un mayor desempeño en el rendimiento.

### 4.1 Pruebas

En la sección de pruebas se explican los recursos que se usaron para realizar las pruebas, empezando por las características del equipo que se usó, el escenario de pruebas planteado y la herramienta que se usó para procesar las peticiones concurrentes la cual es JMeter y finalmente las pruebas que se realizaron.

#### 4.1.1 Equipo de Prueba.

En la Tabla 18 se resumen las características del equipo en que se realizó las pruebas de rendimiento para cada una de las aplicaciones desarrolladas.

Tabla 18 Características del equipo de pruebas

<b>Software</b>	<b>S.O</b>	Windows 7		
		Ubuntu 16.04.3 LTS		
<b>Hardware</b>	<b>Fabricante</b>	Lenovo		
	<b>Procesador</b>	4x Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz		
	<b>Almacenamiento</b>	<b>General</b>	465,76 GB	
		<b>Part. Windows</b>	148,18 GB	
		<b>Part. Linux</b>	295,65 GB	
	<b>Memoria</b>	<b>General</b>	7,84 GB	
		<b>Tarjeta 1</b>	SODIMM DDR3 Síncrono 1600 MHz (0,6 ns) 4 GB	
		<b>Tarjeta 2</b>	SODIMM DDR3 Síncrono 1600 MHz (0,6 ns) 4 GB	
	<b>Red</b>	<b>Wireless</b>	Intel Corporation Centrino Advanced-N 6205 (Taylor Peak)	
		<b>Ethernet</b>	Intel Corporation 82579LM Gigabit Network Connection	

Fuente: La Autora

Elaboración: La Autora



#### **4.1.2 Escenario de Prueba.**

Se planteó el escenario de pruebas según indica la sección 1.5.4 Métricas de rendimiento en base a hardware, considerando solo las métricas referentes al servidor ya que éste es el encargado de recibir, procesar y responder a las solicitudes hechas por los clientes, por consiguiente es quien llevará la mayor carga. Así mismo, es importante mencionar los patrones de diseño orientados al rendimiento que se mencionan en la sección 1.5.5 ya que la presencia o no de estos, impacta directamente en el rendimiento de las aplicaciones independientemente del estilo arquitectónico usado. Las pruebas se organizaron considerando estos 4 aspectos:

##### **4.1.2.1 Número de clientes**

Se considera el caso que 10, 100, y 1000 clientes hacen solicitudes HTTP concurrentemente, estas cantidades se han tomado en base a la notación científica.

##### **4.1.2.2 Número de registros**

Se considera que los clientes acceden a 100, 1000, y 10000 registros a la vez, igualmente, las cantidades propuestas se tomaron en base a la notación científica.

##### **4.1.2.3 Recurso analizado**

Los recursos de hardware a analizar están organizados en las siguientes subcategorías: CPU, Memoria, Red y Base de Datos.

##### **4.1.2.4 Aplicación desarrollada**

Las métricas relacionadas en la sección anterior son referentes a hardware, no obstante para medir la efectividad del rendimiento en cuanto al software se analiza cada aplicación desarrollada, en el caso de las aplicaciones que se construyeron con patrones de diseño, se toma en cuenta el uso o no de los mismos:

- Monolito Versión 1
- Monolito Versión 2
- Servicios REST (construidos con patrones de diseño)
- Servicios REST (construidos sin patrones de diseño)
- Microservicios (construidos con patrones de diseño)
- Microservicios (construidos sin patrones de diseño)

### **4.1.3 Herramienta de Medición: Apache JMeter.**

La herramienta que se usó para hacer las pruebas de rendimiento fue Apache JMeter. Existen varias herramientas que permiten evaluar diferentes aspectos del rendimiento pero la mayoría están ligadas a una tecnología en específico, la razón de seleccionar JMeter radica en que es una herramienta genérica que permite analizar el rendimiento a solicitudes HTTP, entre otras funciones de prueba y dando varias opciones de visualización de resultados, lo que la hace apta para usarla en todas las aplicaciones realizadas. En el Anexo S: Instalación de JMeter se puede revisar el manual de instalación de JMeter.

JMeter, como describe en su página oficial es (Apache Software Foundation, 2017a):

“Un software de código abierto diseñada para probar el rendimiento tanto en recursos dinámicos como estáticos y aplicaciones web dinámicas. Se usa para simular una gran carga en el servidor de aplicaciones para probar las fortalezas de la aplicación o analizar el rendimiento general bajo diferentes tipos de carga. JMeter ofrece la capacidad para cargar y probar el rendimiento de diferentes aplicaciones/ servidores/ tipos de protocolos, p.e: Web, HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, etc)”.

### **4.1.4 Pruebas Realizadas.**

Como se indicó en el escenario de pruebas, se organizaron las solicitudes HTTP entrantes por número de clientes, número de registros, recursos analizados y las aplicaciones desarrolladas. Con esto se da por sentado que las pruebas evalúan las métricas de rendimiento de hardware frente a cada aplicación construida con y sin patrones de diseño.

#### **4.1.4.1 Solicitudes HTTP.**

El escenario general es una solicitud HTTP realizada por cada número de clientes planteados. La solicitud HTTP consiste en una llamada al recurso dependiendo de la implementación de cada aplicación explicada en el capítulo 3, en la Tabla 19 se muestran las solicitudes a probarse, tomado en cuenta la aplicación a la cual pertenecen, la consulta interna que se hace en la base de datos y el formato de respuesta esperado.

Respecto a las solicitudes se debe aclarar ciertos aspectos que se describen a continuación:

- Como se mencionó anteriormente, para las pruebas se tomó en cuenta el uso de patrones orientados al rendimiento con los que las aplicaciones fueron construidas, por

lo que para las aplicaciones de servicios REST y Microservicios se analizan 2 casos: con y sin patrones de diseño.

- Todas las aplicaciones corren en el servidor local, a excepción de las orientadas a microservicios ya que al usar Docker este les asigna un host propio.
- En el caso de la aplicación de Microservicios sin patrones de diseño, el host y puerto de la solicitud HTTP es diferente al de la aplicación con patrones de diseño, esto se dio ya que el patrón Proxy se implementa con Zuul, por lo tanto, acceder directamente al servicio representa no usar el patrón de diseño mencionado en Microservicios.

Tabla 19 Solicitudes HTTP por cada aplicación analizada

Aplicación	Método de acceso a la BD	Solicitud HTTP	Formato de Respuesta
<b>Monolito Versión 1</b>	Consulta SQL	localhost:80/RegistroCursoOnline/Proyecto/SRCL/selectcurso.php?curso={id_curso}	HTML
<b>Monolito Versión 2</b>	Procedimiento Almacenado	localhost:8080/srcl2/RecuperarMatriculas?cursosAsignados={id_curso}	HTML
<b>Servicios REST con/sin patrones de diseño</b>	Procedimiento Almacenado	localhost:8080/srcl/rest/matriculas/curso/{id_curso}	JSON
<b>Microservicios con/sin patrones de diseño</b>	Procedimiento Almacenado	172.19.0.5:8081/srcl/rest/matriculas/curso/{id_curso}	JSON
		172.19.0.4:1111/srcl/rest/matriculas/curso/{id_curso}	

Fuente: La Autora

Elaboración: La Autora

#### **4.1.4.2 Creación de escenario de pruebas en JMeter.**

En JMeter se configuran las pruebas requeridas con los siguientes pasos.

##### **4.1.4.2.1 Grupo de Hilos**

El primer paso consiste en crear un Grupo de Hilos donde se guarda toda la configuración para un escenario de prueba, en este caso, la solicitud HTTP a evaluar, el número de clientes que accederán concurrentemente a la aplicación y los resultados que se deseen visualizar. Un

grupo de hilos representa el número de clientes que van a acceder simultáneamente a la aplicación un cliente que va a realizar solicitudes HTTP.

Se configuraron 3 grupos de hilos para 10, 100 y 1000 clientes a través del campo: Número de Hilos, adicionalmente, en el campo Nombre se puede especificar un nombre para el grupo de hilos. En la Figura 34 se muestra la configuración de un grupo de 1000 hilos en JMeter para la aplicación de microservicios con patrones de diseño.

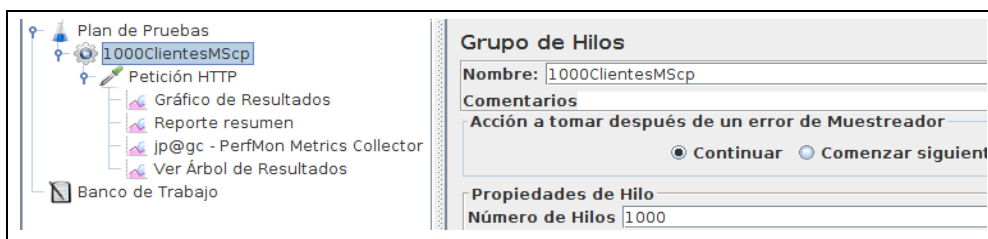


Figura 34 Creación de Grupo de hilos en JMeter

Fuente: La Autora

Elaboración: La Autora

#### 4.1.4.2.2 Configuración de Solicitud HTTP

El siguiente paso consiste en configurar la solicitud HTTP a analizar (Listener en JMeter) dentro del Grupo de Hilos establecidos en el paso anterior. El Listener permite establecer para cada solicitud HTTP:

- Host (Nombre de Servidor o IP)
- Puerto (Puerto)
- Método de acceso (GET, POST, etc.)
- Ruta de la solicitud (Ruta)
- Parámetros de la petición con la opción Enviar Parámetros Con la Petición.

En la Figura 35 se puede ver la configuración de la solicitud HTTP para la aplicación de Microservicios construida con patrones de diseño.

#### 4.1.4.2.3 Receptores

Finalmente se añaden Receptores a la solicitud HTTP que no son más que las diferentes opciones de visualización de los resultados que arroja JMeter, en este caso se añadieron:

Gráfico de Resultados, Reporte Resumen, PerfMon Metrics Collector, Árbol de Resultados. A continuación se explica las métricas que se extrae por cada uno de ellos.

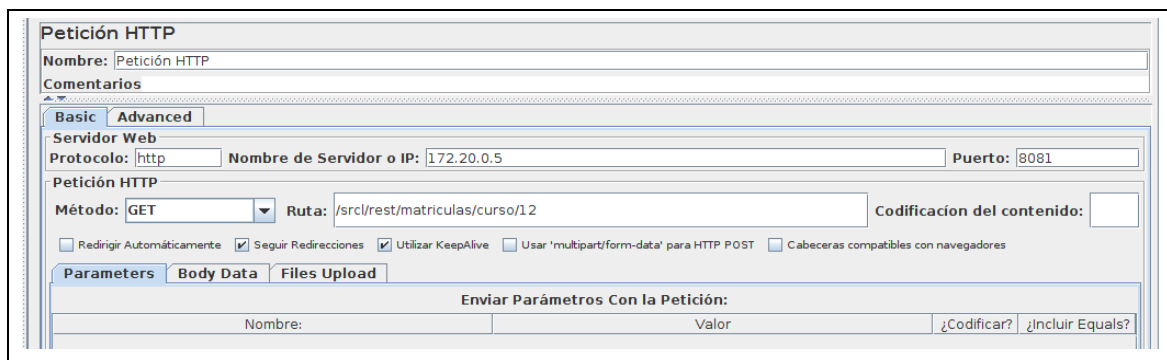


Figura 35 Configuración de Listener en JMeter

Fuente: La Autora

Elaboración: La Autora

#### 4.1.4.2.3.1 Gráfico de Resultados

Este reporte grafica variables estadísticas que se dan al procesar un escenario de prueba para lo cual asigna un color a cada variable además de presentar el tiempo que se demoró en procesar la solicitud HTTP se muestra en milisegundos:

- Numero de muestras, negro
- Promedio actual de todas las muestras procesadas, azul
- Desviación estándar actual, rojo
- Tasa de rendimiento actual, verde

Para el escenario de prueba: 1.000 clientes acceden a 10.000 registros en la aplicación Servicios REST con patrones de diseño el reporte “Grafico de Resultados” (Ver Figura 36) presenta un comportamiento que tiende al alza a través del tiempo, es decir, el tiempo de respuesta es proporcional al número de clientes que acceden a la aplicación concurrentemente.

El número de muestras (solicitudes HTTP) que se procesaron durante las prueba es 1.000, la media de todas las solicitudes procesadas es 957, la desviación estándar con valor 390 indica la dispersión de datos respecto a la muestra por lo que la desviación es muy alta, el número de solicitudes procesadas por minuto es indicado por la variable de rendimiento, el cual, considerando el elevado número de clientes que acceden concurrentemente sumado al alto

número de registros accedidos es de aproximadamente 14.000 registros por minuto, todos estos resultados se dan en un tiempo de 1,45 segundos.

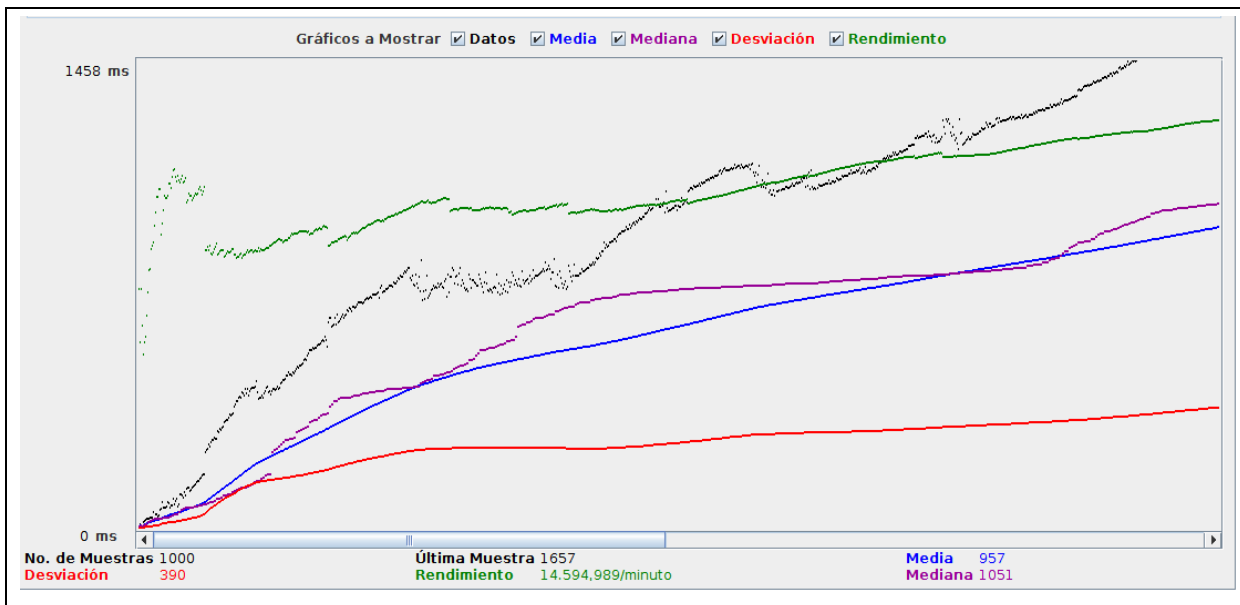


Figura 36 Reporte: Gráfico de Resultados en JMeter

Fuente: La Autora

Elaboración: La Autora

#### 4.1.4.2.3.2 Reporte Resumen

Este reporte presenta la mayoría de variables de métricas de Red y de Base de datos en formato tabla, a continuación se describen las métricas.

- Número de muestras analizadas (# Muestras).
- Tiempo promedio, más bajo, más alto y la desviación estándar transcurridos para procesar la prueba (Media, Min, Max, Desv. Estándar) expresado en milisegundos.
- Porcentaje de solicitudes con errores, es decir que no se procesaron (% Error).
- El rendimiento actual o throughput el cual se mide en solicitudes por segundo/minuto/hora (Rendimiento).
- La cantidad de kilobytes por segundo enviados y recibidos durante la ejecución de la prueba (Sent KB/sec y KB/sec).

En la Figura 37 se muestra los resultados generados por “Reporte Resumen” para el escenario de prueba: 1.000 clientes acceden a 10.000 registros en la aplicación Servicios REST sin patrones de diseño.

En la tabla que genera “Reporte Resumen” se ubica el nombre que se le asignó a la petición HTTP, el número de peticiones entrantes correspondiente a la configuración del grupo de hilos, en esta prueba el tiempo promedio de ejecución fue de aproximadamente 45 segundos, el tiempo menor registrado fue 22 segundos y el mayor, 1.81 minutos. La desviación estándar con respecto al número de muestras tiene un valor elevado lo que se refleja en el hecho que para esta aplicación, el 88.6 % de las peticiones no se procesaron correctamente, igualmente, la tasa de rendimiento es muy baja con solo 9 peticiones procesadas por segundo, la cantidad de Bytes enviados y recibidos para esta prueba es de 0.15 y 1855.4 respectivamente lo cual tiene sentido considerando que al trabajar con Servicios REST se hace énfasis en llamadas a procedimientos remotos y finalmente se muestra el promedio de bytes recibidos es de 209461.0

The screenshot shows the 'Reporte resumen' window in JMeter. It includes a form for naming the report and saving it, and a summary table with the following data:

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Estándar	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de Bytes
Petición HTTP	1000	45939	22434	108844	16577,82	88,60%	9,1/sec	1855,41	0,15	209461,0
Total	1000	45939	22434	108844	16577,82	88,60%	9,1/sec	1855,41	0,15	209461,0

Figura 37 Reporte: Resumen en JMeter

Fuente: La Autora

Elaboración: La Autora

#### 4.1.4.2.3.3 PerfMon Metrics Collector

Este tipo de reporte permite configurar las métricas que se deseen mostrar referente al comportamiento de CPU y Memoria. Para las pruebas a realizarse se escogieron las siguientes métricas.

- Uso de CPU, expresado en porcentaje (%)
- Memoria usada (used), expresada en Mb
- Memoria (free), expresada en Mb
- Memoria (total), expresada en Mb

En la Figura 38 se muestra los resultados generados por “PerfMon Metrics Collector” para la solicitud realizada a la aplicación de Microservicios sin patrones de diseño con el escenario: 1.000 clientes accediendo a 10.000 registros. Según la gráfica, el uso de CPU es casi nulo al inicio de la prueba pero aumenta significativamente en el primer segundo llegando a usarse

más del 80% de este recurso y se mantiene este comportamiento hasta casi el final de la prueba donde nuevamente el uso de CPU se reduce.

En cuanto al uso de memoria, este es alto casi la mayoría del tiempo (7.43 Gb, aprox) donde se ocupa casi el 90% para procesar solicitudes por lo que los valores de memoria total y usada van a ser elevados y similares la mayoría del tiempo no obstante hay momentos en los que se refleja picos que indican la memoria libre disponible.

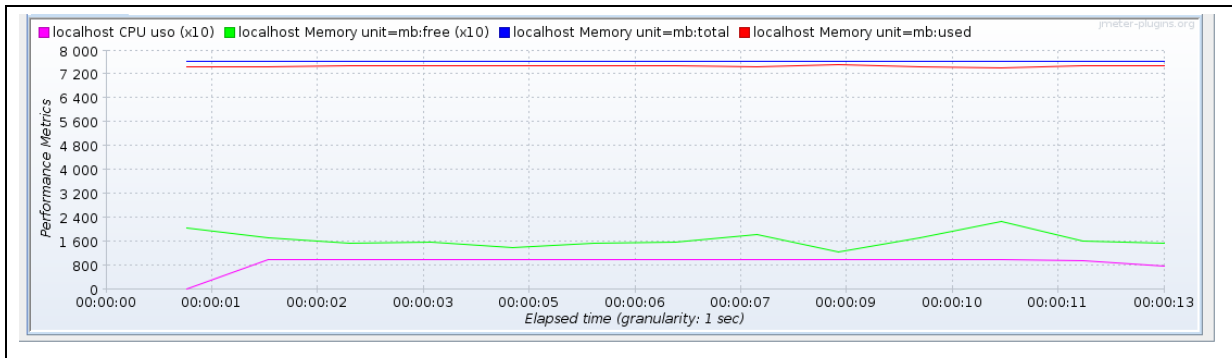


Figura 38 Reporte: PerfMon Metrics Collector en JMeter

Fuente: La Autora

Elaboración: La Autora

#### 4.1.4.2.3.4 Árbol de Resultados

El reporte de Árbol de Resultados muestra la respuesta generada de cada solicitud realizada durante la prueba, esta respuesta incluye:

- Formato de respuesta
- Tiempo que le tomo a la aplicación responder a la solicitud
- Código de respuesta
- Otras variables.

En la Figura 39 se muestra los resultados generados por “Árbol de Resultados” para el escenario de prueba: 1.000 clientes accediendo a 10.000 registros en la aplicación Servicios REST sin patrones de diseño. En la parte izquierda del reporte se presentan todas las peticiones que se han procesado con y sin error, al pulsar sobre cada una de ellas, en la parte izquierda se presenta la información de la solicitud seleccionada. Si es una petición exitosa se presenta el tiempo de carga expresado en milisegundos, 35.4 ms. El tamaño de la respuesta en bytes



general o desglosado (header, body) Así mismo se presenta el código de respuesta de la solicitud, en este caso al ser una petición correcta el servidor responde con código HTTP 200 OK además de información referente al servidor. Por otra parte si la petición no fue procesada con éxito nos muestra el código de respuesta 500 el cual indica un error general del servidor de aplicaciones.

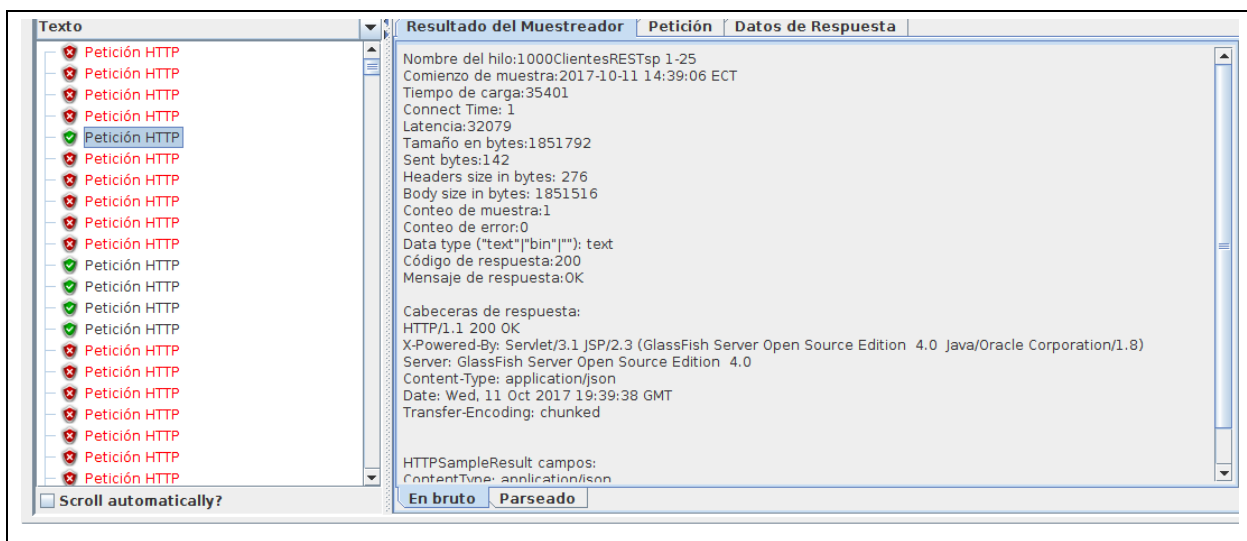


Figura 39 Reporte: Árbol de Resultados en JMeter

Fuente: La Autora

Elaboración: La Autora

## 4.2 Resultados

En la sección de Resultados se presentan y explican los resultados obtenidos con JMeter para los escenarios de pruebas planteados en la sección anterior, esto significa que las tablas que recogen los resultados arrojados por JMeter consideran todos los parámetros necesarios para la evaluación del rendimiento como son: el número de clientes que acceden concurrentemente a las aplicaciones desarrolladas, el número de registros a los que acceden los mismos, las aplicaciones que se evalúan (construidas con/sin patrones de diseño) y el recurso a analizar (p.e CPU, Memoria, etc.).

Para interpretar cada una de las métricas recogidas, en el Anexo T: Definición de Términos de Métricas de Rendimiento en base a hardware se define los términos empleados así como la unidad de medida por cada uno de ellos. Las tablas de resultados se pueden consultar en: Anexo U: Pruebas de rendimiento. Escenario: 10 Clientes, Anexo W: Pruebas de rendimiento.

Escenario: 100 Clientes y Anexo X: Pruebas de rendimiento. Escenario: 1.000 Clientes. No obstante, para una mayor comprensión se han organizado los resultados en 3 categorías principales:

- Por número de Clientes
- Por aplicación analizada
- Por recurso analizado

A continuación se desglosan en tablas comparativas los resultados según las categorías mencionadas, por cada una de ellas se especifica el escenario de prueba y una gráfica de barras de los resultados que recogen sólo las métricas más relevantes a analizar.

#### **4.2.1 Por número de Clientes.**

Los resultados que se muestran a continuación contemplan el caso donde un número determinado de clientes acceden simultáneamente a 10.000 registros de la base de datos, dependiendo del método de acceso a la base de datos de cada aplicación (consultas SQL, procedimientos almacenados o servicios REST). Los escenarios de prueba que se usaron son:

- 10 Clientes
- 100 Clientes
- 1.000 Clientes

##### **4.2.1.1 Pruebas con 10 Clientes**

**Escenario:** 10 Clientes acceden a 10.000 registros concurrentemente.

Como se observa en la Tabla 20, con 10 clientes que realizan peticiones concurrentes para acceder a 10.000 registros no se registra ejecuciones fallidas, es decir el 100 % de las peticiones fueron respondidas con código 200 OK, pero el uso de los recursos analizados si varía según la aplicación a la que se realice la petición HTTP. En la Figura 40 se observa que el menor consumo de recursos como CPU y memoria se registró en las aplicaciones que usan patrones de diseño como son Servicios REST y Microservicios. De igual manera, se registra un mayor número de transacciones/segundo en las aplicaciones mencionadas tanto si se usa red cableada como Wireless debido a las transacciones sobre red que se dan al usar servicios REST y microservicios. El menor tiempo de respuesta se dio para las aplicaciones de Microservicios con y sin patrones de diseño.

Tabla 20 Análisis de rendimiento con 10 Clientes

		Monolito Versión 1	Monolito Versión 2	Servicios REST (Con patrones de diseño)	Servicios REST (Sin patrones de diseño)	MS (Con patrones de diseño)	MS (Sin patrones de diseño)
<b>CPU</b>	<b>Uso de CPU (%)</b>	98,10%	97,30%	47,00%	33,00%	30,00%	28,00%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	4,20	7,15	4,04	4,15	6,06	5,72
<b>Red</b>	<b>Cableada: Transacciones/s (x/s)</b>	3,50	7,30	9,60	9,40	9,40	2,10
	<b>Wireless: Transacciones/s (x/s)</b>	3,10	7,00	6,10	9,00	9,50	9,70
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	2,05	12,48	0,73	0,34	0,19	0,17
	<b>Ejecuciones fallidas (%)</b>	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Fuente: La Autora

Elaboración: La Autora

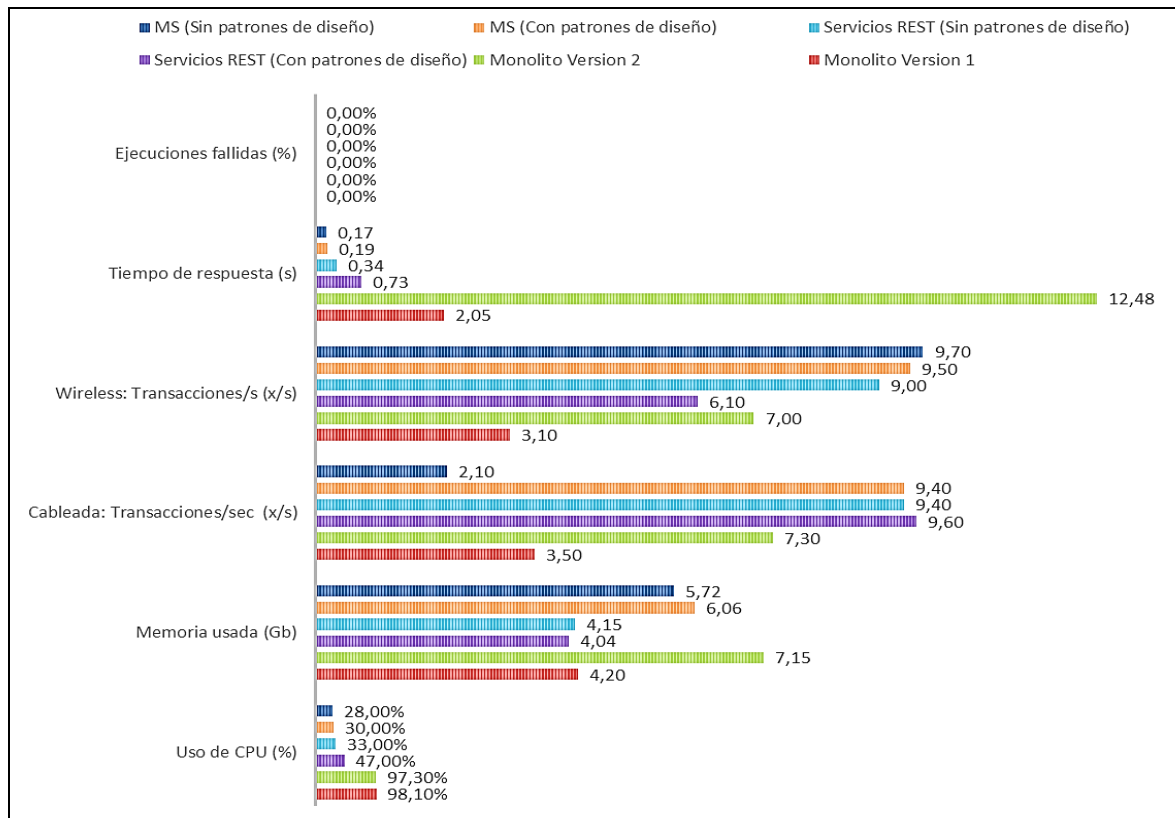


Figura 40 Rendimiento con 10 Clientes

Fuente: La Autora

Elaboración: La Autora

#### 4.2.1.2 Pruebas con 100 Clientes

**Escenario:** 100 Clientes acceden a 10.000 registros concurrentemente.

Según la Tabla 21 que muestra los resultados del escenario que considera 100 clientes que hacen peticiones concurrentemente ya se empiezan a registrar ejecuciones fallidas, sobre todo en las aplicaciones que no usan patrones de diseño (Monolito V1, Servicios REST).

En la Figura 41 queda demostrado que aumenta el uso de los recursos CPU y memoria drásticamente en las aplicaciones Monolito Versión 1, Servicios REST y Microservicios sin patrones de diseño sobre todo si se compara con el escenario de 10 clientes sin embargo se registra un mayor número de transacciones por segundo en la aplicación de Microservicios con patrones de diseño (15 y 18 trans/s) considerando que ejecuta todas las peticiones sin error, así mismo el tiempo de respuesta no se ve afectado al usar esta arquitectura,

Para el Monolito V1, una aplicación construida con una estructura básica y sin patrones de diseño todas las pruebas resultaron fallidas (100 %) debido a que el servidor abortó el escenario de pruebas incluso antes que se pueda procesar alguna petición exitosa.

Tabla 21 Análisis del rendimiento con 100 Clientes

		Monolito Versión 1	Monolito Versión 2	Servicios REST (Con patrones de diseño)	Servicios REST (Sin patrones de diseño)	MS (Con patrones de diseño)	MS (Sin patrones de diseño)
<b>CPU</b>	<b>Uso de CPU (%)</b>	98,10%	95,00%	83,40%	98,50%	98,60%	99,30%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	4,57	7,30	7,12	7,05	7,45	7,41
<b>Red</b>	<b>Cableada: Transacciones/s (x/s)</b>	102,20	7,30	2,50	7,40	15,00	2,10
	<b>Wireless: Transacciones/s (x/s)</b>	122,40	7,10	1,90	2,60	18,00	2,90
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	0,001	9,19	12,99	32,19	5,02	34,38
	<b>Ejecuciones fallidas (%)</b>	100,00%	0,00%	0,00%	26,50%	0,00%	0,00%

Fuente: La Autora

Elaboración: La Autora

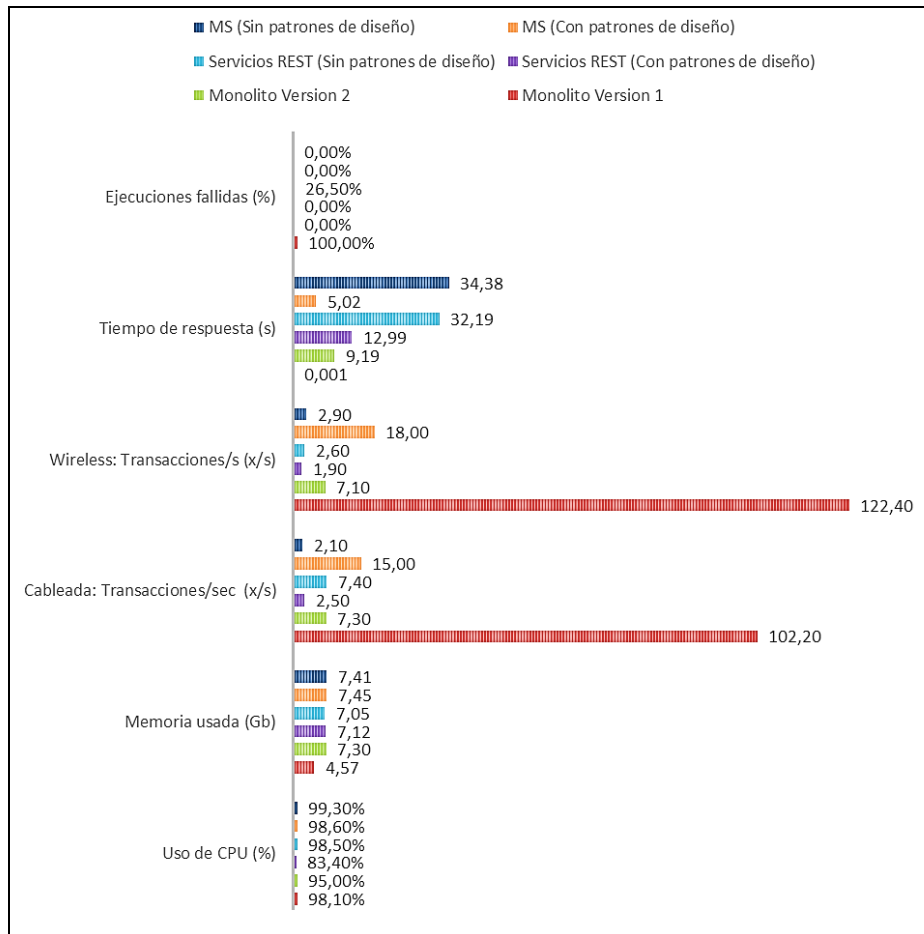


Figura 41 Rendimiento con 100 Clientes

Fuente: La Autora

Elaboración: La Autora

#### 4.2.1.3 Pruebas con 1.000 Clientes

**Escenario:** 1000 Clientes acceden a 10.000 registros concurrentemente. Ver Tabla 22.

Como indica la Tabla 22, con 1000 clientes accediendo simultáneamente el rendimiento se ve afectado en todas las aplicaciones ya que todas reportan ejecuciones fallidas, no obstante el menor porcentaje de error se da en las aplicaciones que usan patrones de diseño, lo cual es razonable debido al uso de los patrones Singleton y Proxy además de los patrones propios de microservicios.

En la Figura 42 se observa que al tener 1.000 clientes haciendo peticiones concurrentes al servidor y sin importar el número de registros accedidos el uso de CPU superó el 95 % y el de

memoria superó los 7 Gb de los 8 disponibles durante la ejecución de la prueba. En cuanto al desempeño de la red, el mayor número de transacciones por segundo se dan en las aplicaciones construidas con patrones de diseño, tanto con red cableada como Wireless (27 y 72 trans/s).

Tabla 22 Análisis del rendimiento con 1.000 Clientes

		Monolito Versión 1	Monolito Versión 2	Servicios REST (Con patrones de diseño)	Servicios REST (Sin patrones de diseño)	MS (Con patrones de diseño)	MS (Sin patrones de diseño)
CPU	Uso de CPU (%)	98,10%	96,80%	98,50%	97,30%	98,60%	97,30%
Memoria	Memoria usada (Gb)	4,57	7,19	6,27	7,26	7,53	7,63
Red	Cableada: Transacciones/s (x/s)	102,20	12,60	27,90	14,30	72,00	51,60
	Wireless: Transacciones/s (x/s)	122,40	48,00	17,80	9,10	57,70	1,20
Base de datos	Tiempo de respuesta (s)	0,001	11,143	31,007	61,436	10,749	8
	Ejecuciones fallidas (%)	100,00%	89,27%	81,40%	90,20%	79,20%	95,00%

Fuente: La Autora

Elaboración: La Autora

De igual manera, el menor tiempo de respuesta se registra para las aplicaciones Servicios REST y Microservicios (10-31 segundos) aunque en un ambiente real estos serían percibidos por los clientes. Este escenario es aceptable en comparación al caso del Monolito Versión 1 en el cual el tiempo de respuesta es de 1 milisegundo debido a que se abortó la operación incluso mucho antes que se pudiera procesar alguna petición exitosa.

#### 4.2.2 Por aplicación analizada.

En los resultados mostrados a continuación se evalúa el comportamiento de cada aplicación cuando acceden simultáneamente 10, 100 y 1.000 clientes, esto permite diferenciar el comportamiento de cada aplicación frente a la carga impuesta al servidor y establecer las causas de dicho comportamiento.

##### 4.2.2.1 Monolito Versión 1

**Escenario:** 10, 100 y 1.000 Clientes acceden a 10.000 registros concurrentemente en la aplicación Monolito Versión 1.

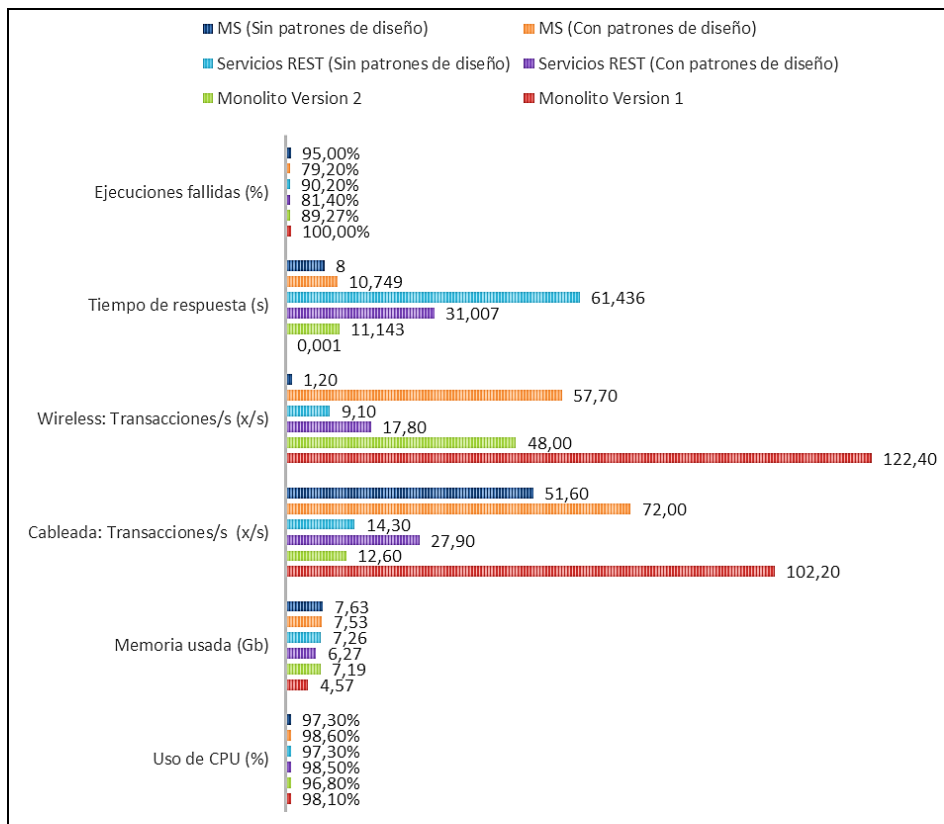


Figura 42 Rendimiento con 1.000 Clientes

Fuente: La Autora

Elaboración: La Autora

La Tabla 23 recoge los resultados de rendimiento de Monolito Versión 1, esta aplicación con una estructura básica y sin patrones de diseño presenta un desempeño escaso de rendimiento ya que se ve afectada cuando incluso para procesar peticiones concurrentes de 10 clientes el servidor ocupa casi el 100 % del CPU.

La gráfica de la Figura 43 muestra que en el Monolito V1 el número de transacciones por segundo es mucho menor que en cualquier otra aplicación ya sea con red cableada o Wireless debido a que esta aplicación trabaja con llamadas a funciones y no a procedimientos remotos. Al procesar una gran cantidad de peticiones como el escenario con 1.000 clientes, el servidor aborta la operación incluso mucho antes que se pudiera procesar alguna petición exitosa.

Tabla 23 Análisis de rendimiento para Monolito Versión 1

		Clientes		
		10	100	1000
<b>CPU</b>	<b>Uso de CPU (%)</b>	98,10%	99,10%	98,10%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	4,20	5,50	4,57
<b>Red</b>	<b>Cableada: Transacciones/s (x/s)</b>	3,50	3,00	102,20
	<b>Wireless: Transacciones/s (x/s)</b>	3,10	3,10	122,40
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	5,29	31,94	0,001
	<b>Ejecuciones fallidas (%)</b>	0,00%	0,00%	100,00%

Fuente: La Autora

Elaboración: La Autora

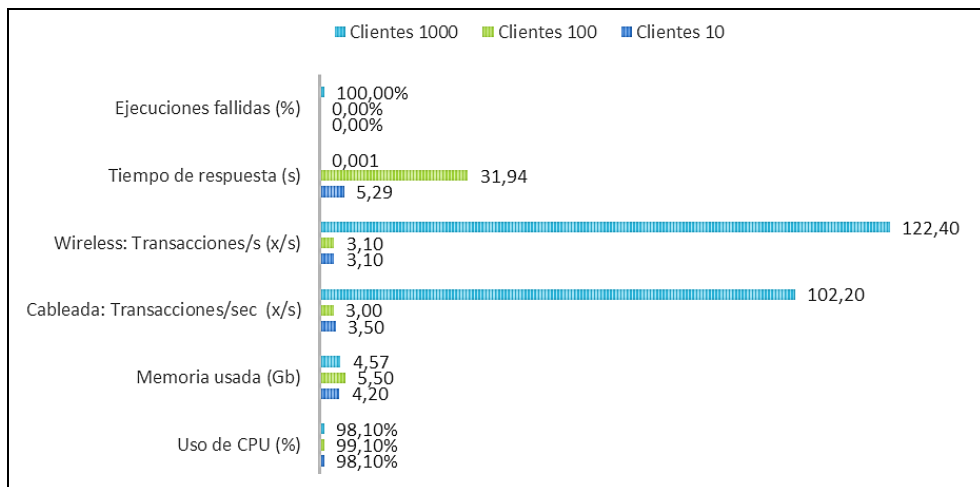


Figura 43 Rendimiento en Monolito Versión 1

Fuente: La Autora

Elaboración: La Autora

#### 4.2.2.2 Monolito Versión 2

**Escenario:** 10, 100 y 1.000 Clientes acceden a 10.000 registros concurrentemente en el Monolito Versión 2.

La Tabla 24 demuestra que al igual que el Monolito Versión 1, con el Monolito Versión 2 se registra un alto consumo de CPU y memoria, no obstante el número de transacciones por segundo aumentó respecto al Monolito Versión 1. Esta aplicación sin duda representa una mejora en comparación con la primera aplicación monolítica pero aun así, con el escenario de 1.000 solicitudes concurrentes sólo el 10 % de las peticiones fueron respondidas correctamente, es decir con código 200 OK.



Tabla 24 Análisis de rendimiento en Monolito Versión 2

		Clientes		
		10	100	1000
<b>CPU</b>	<b>Uso de CPU (%)</b>	97,30%	95,00%	96,80%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	7,15	7,30	7,19
<b>Red</b>	<b>Cableada: Transacciones/s (x/s)</b>	7,30	7,30	12,60
	<b>Wireless: Transacciones/s (x/s)</b>	7,00	7,10	48,00
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	12,48	9,19	11,143
	<b>Ejecuciones fallidas (%)</b>	0,00%	0,00%	89,27%

Fuente: La Autora

Elaboración: La Autora

Cabe destacar que el Monolito V2 fue construido bajo la arquitectura 3 capas, sin patrones de diseño por lo que esto sumado al cambio de tecnología (PHP por Java) influye en el desenvolvimiento de la aplicación, el uso de procedimientos almacenados libera al servidor de realizar múltiples consultas a la base de datos. Aun así, en todos los casos se registró un consumo de CPU y memoria rozando el límite de la cantidad disponible (97,30 % uso de CPU y 7,30 Gb de memoria), (Ver Figura 44).

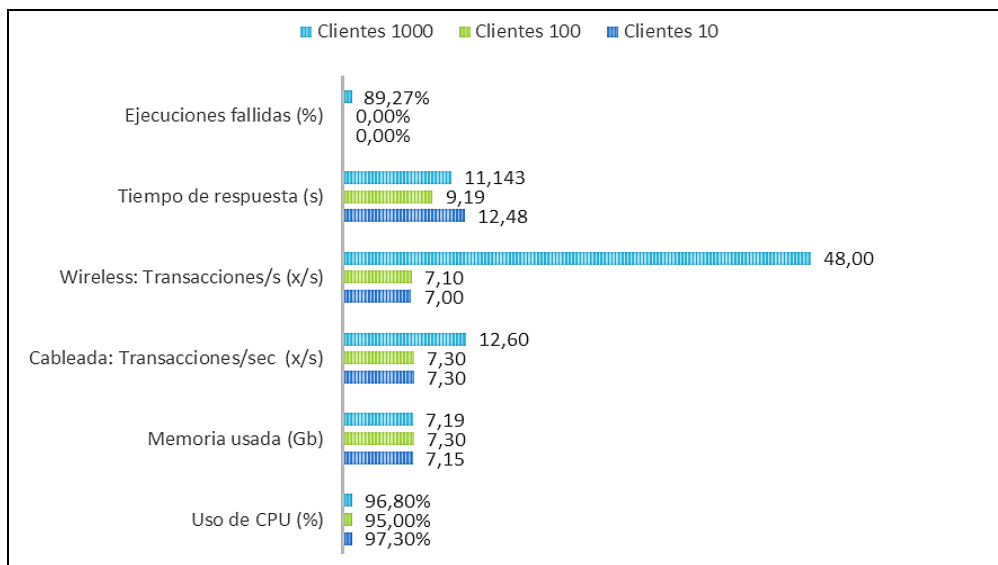


Figura 44 Rendimiento en Monolito Versión 2

Fuente: La Autora

Elaboración: La Autora

#### 4.2.2.3 Servicios REST con patrones de diseño

**Escenario:** 10, 100 y 1.000 Clientes acceden a 10.000 registros concurrentemente en la aplicación de Servicios REST desarrollada con patrones de diseño. (Ver Tabla 25).

Tabla 25 Análisis de rendimiento en Servicios REST (con patrones de diseño)

		<b>Clientes</b>		
		<b>10</b>	<b>100</b>	<b>1000</b>
<b>CPU</b>	<b>Uso de CPU (%)</b>	47,00%	83,40%	98,50%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	4,04	7,12	6,27
<b>Red</b>	<b>Cableada: Transacciones/s (x/s)</b>	9,60	2,50	27,90
	<b>Wireless: Transacciones/s (x/s)</b>	6,10	1,90	17,80
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	0,73	12,99	31,00
	<b>Ejecuciones fallidas (%)</b>	0,00%	0,00%	81,40%

Fuente: La Autora

Elaboración: La Autora

Como se puede ver en la Tabla 25, en la aplicación de Servicios REST construida con patrones de diseño el uso de recursos como CPU o memoria, transacciones ejecutadas por segundo y el tiempo de respuesta es gradual, es decir, aumenta mientras más clientes acceden a la aplicación.

En cuanto a las ejecuciones fallidas, la gráfica de la Figura 45 indica que no se presentaron errores con 10 y 100 clientes, en cambio con 1000 clientes solo el 20 % de las solicitudes fueron respondidas satisfactoriamente sin embargo este resultado es mucho mayor que las aplicaciones que la preceden, esto sin duda se debe al uso de patrones de diseño orientados al rendimiento: Singleton y Facade.

Para comprobar la eficiencia del patrón Singleton implementado se configuró un mensaje que se presentaba cada vez que se intentaba crear más de una instancia del servicio REST solicitado, tal como muestra la Figura 46.

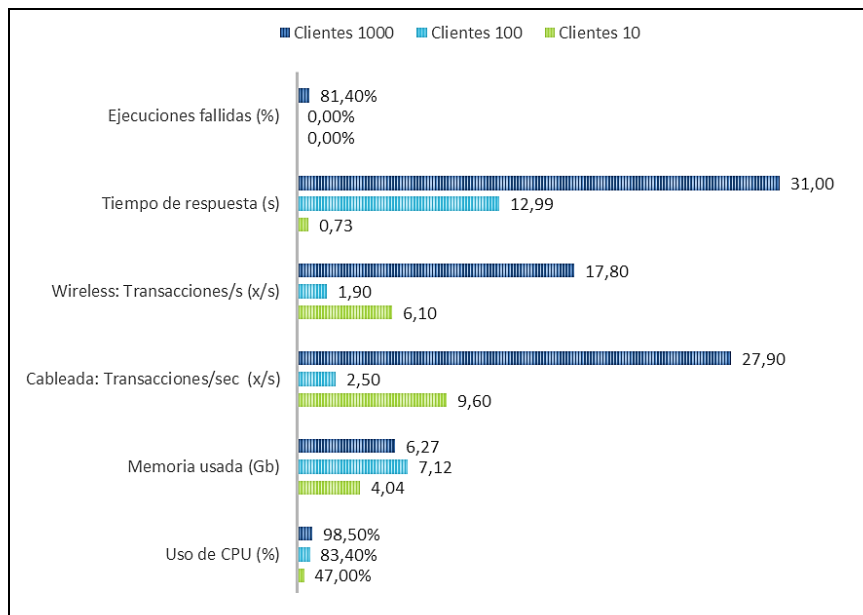


Figura 45 Rendimiento en Servicios REST (con patrones de diseño)

Fuente: La Autora

Elaboración: La Autora

```

Run (srcl3) x  Java DB Database Process x  GlassFish Server x
Información: Loading application [srcl3] at [/srcl3]
Información: srcl3 was successfully deployed in 4.189 milliseconds.
Información: Initiating Jersey application, version Jersey: 2.0 2013-05-03 14:50:15...
Información: Ya existe un objeto de tipo MatriculaFacade
Información: Ya existe un objeto de tipo MatriculaFacade
Información: Ya existe un objeto de tipo MatriculaFacade
Información: Ya existe un objeto de tipo MatriculaFacade
Información: Ya existe un objeto de tipo MatriculaFacade

```

Figura 46 Patrón Singleton en Servicios REST (con patrones de diseño)

Fuente: La Autora

Elaboración: La Autora

#### 4.2.2.4 Servicios REST sin patrones de diseño

**Escenario:** 10, 100 y 1.000 Clientes acceden a 10.000 registros concurrentemente en la aplicación de Servicios REST desarrollada sin patrones de diseño.

En la Tabla 26 se indica que al igual que la aplicación de servicios REST desarrollada con patrones de diseño el uso de los recursos CPU y memoria aumenta gradualmente mientras más clientes acceden a la aplicación, se observa el mismo comportamiento para las transacciones hechas por segundo y el tiempo de respuesta.

Tabla 26 Análisis del rendimiento en Servicios REST (sin patrones de diseño)

		<b>Clientes</b>		
		<b>10</b>	<b>100</b>	<b>1000</b>
<b>CPU</b>	<b>Uso de CPU (%)</b>	33,00%	98,50%	97,30%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	4,15	7,05	7,26
<b>Red</b>	<b>Cableada: Transacciones/ (x/s)</b>	9,40	7,40	14,30
	<b>Wireless: Transacciones/s (x/s)</b>	9,00	2,60	9,10
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	0,34	32,19	61,43
	<b>Ejecuciones fallidas (%)</b>	0,00%	26,50%	90,20%

Fuente: La Autora

Elaboración: La Autora

En cuanto a las ejecuciones fallidas, en la gráfica de la Figura 47 se indica que no hay errores al procesar las solicitudes de 10 clientes (0,0 % ejecuciones fallidas) no así para los escenarios con 100 y 1.000 clientes.

Esto se debe a que en esta aplicación no se implementa el patrón de diseño Singleton y por lo tanto se crean instancias por cada solicitud HTTP lo que causa que la memoria disponible se inunde de múltiples objetos lo que sin duda afecta al rendimiento de memoria. Para solucionar este error, se añadieron los siguientes parámetros en la configuración del IDE que ejecuta la aplicación:

```
-J-Xmx 2048m -J-XX:+UseCompressedOops
```

No obstante, el rendimiento resultó afectado para esta aplicación donde el 90 % de las peticiones se devolvieron con error de servidor 500. Estas observaciones también se reflejan en la gráfica de la Figura 47.

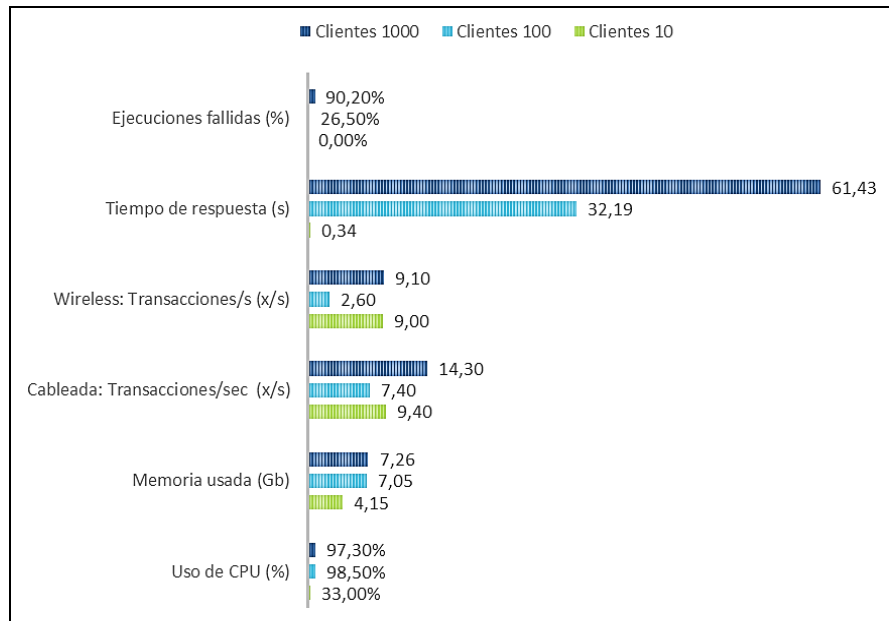


Figura 47 Rendimiento en Servicios REST (sin patrones de diseño)

Fuente: La Autora

Elaboración: La Autora

#### 4.2.2.5 Microservicios con patrones de diseño

**Escenario:** 10, 100 y 1.000 Clientes acceden a 10.000 registros concurrentemente en la aplicación de Microservicios desarrollada con patrones de diseño (Ver Tabla 27).

Tabla 27 Análisis del rendimiento en Microservicios (con patrones de diseño)

		Clientes		
		10	100	1000
<b>CPU</b>	<b>Uso de CPU (%)</b>	30,00%	98,60%	98,60%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	6,06	7,45	7,53
<b>Red</b>	<b>Cableada: Transacciones/s (x/s)</b>	9,40	15,00	72,00
	<b>Wireless: Transacciones/s (x/s)</b>	9,50	18,00	57,70
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	0,19	5,02	10,74
	<b>Ejecuciones fallidas (%)</b>	0,00%	0,00%	79,20%

Fuente: La Autora

Elaboración: La Autora

En la aplicación de Microservicios desarrollada con patrones de diseño se conserva el comportamiento de Servicios REST construida también con patrones de diseño. El uso de recursos como CPU o memoria aumenta mientras más clientes acceden a la aplicación, mientras que el número de transacciones ejecutadas por segundo con 1.000 clientes es superior respecto a 10 y 100 clientes sin embargo, el tiempo de respuesta en este escenario combinado con el número de ejecuciones fallidas da los mejores resultados de rendimiento si se compara con los resultados obtenidos para las otras aplicaciones.

Pese a que la Figura 48 indica que con 1.000 clientes accediendo a la aplicación el 79 % de las peticiones resultaron fallidas se debe considerar que en una arquitectura de microservicios además de los patrones de diseño orientados al rendimiento se utilizan patrones propios de la construcción de Microservicios que se implementan a través de otras aplicaciones como Eureka y Zuul. Esto implica que para procesar cada solicitud entrante todas las aplicaciones que forman parte de la arquitectura deben hacer llamadas a procedimientos remotos en lugar de llamadas a funciones que en la práctica son más rápidas que operaciones sobre red lo que afectará el rendimiento de cualquier arquitectura de microservicios.

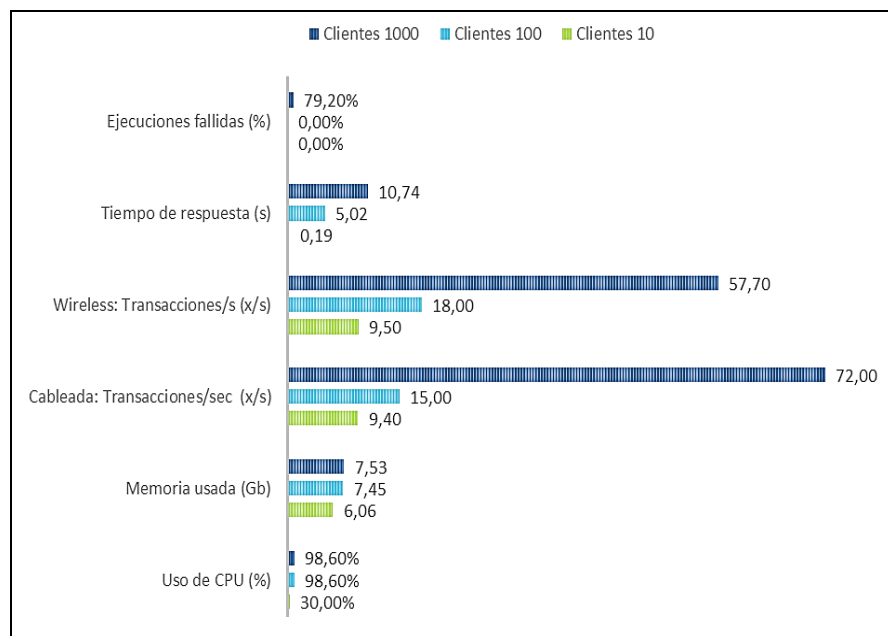


Figura 48 Rendimiento en Microservicios (con patrones de diseño)

Fuente: La Autora

Elaboración: La Autora

#### 4.2.2.6 *Microservicios sin patrones de diseño*

**Escenario:** 10, 100 y 1.000 Clientes acceden a 10.000 registros concurrentemente en la aplicación de Microservicios desarrollada sin patrones de diseño (Ver Tabla 28).

Tabla 28 Análisis de rendimiento en Microservicios (sin patrones de diseño)

		<b>Cientes</b>		
		<b>10</b>	<b>100</b>	<b>1000</b>
<b>CPU</b>	<b>Uso de CPU (%)</b>	28,00%	99,30%	97,30%
<b>Memoria</b>	<b>Memoria usada (Gb)</b>	5,72	7,41	7,63
<b>Red</b>	<b>Cableada: Transacciones/s (x/s)</b>	2,10	2,10	51,60
	<b>Wireless: Transacciones/s (x/s)</b>	9,70	2,90	1,20
<b>Base de datos</b>	<b>Tiempo de respuesta (s)</b>	0,17	34,38	8
	<b>Ejecuciones fallidas (%)</b>	0,00%	0,00%	95,00%

Fuente: La Autora

Elaboración: La Autora

En el caso de la aplicación de Microservicios que fue construida sin patrones de diseño el uso de recursos como CPU y memoria aumenta gradualmente mientras más clientes acceden a la aplicación, mientras que el número de transacciones ejecutadas por segundo es mucho menor que en la aplicación anterior, esto es debido a que tal como refleja el número de ejecuciones fallidas, en el escenario con 1000 clientes la mayoría de las solicitudes obtuvieron error 500 de servidor como respuesta. En los escenarios con 10 y 100 clientes no se presentaron errores de solicitudes procesadas con error. Cabe aclarar que en el escenario con 1.000 clientes el tiempo de respuesta se reduce a 8 segundos ya que solo 5v % resultaron correctas.

Como se explicó anteriormente, para esta aplicación el hecho de no usar patrones consistía en acceder al servicio directamente sin encaminar las peticiones primeramente por Zuul, éste incluye la aplicación Ribbon, un balanceador de carga, esto explica porque en la que Figura 49 se ve que el 95% de las peticiones se hayan procesado fallidamente, lo que comprueba que usar patrones de diseño orientados al rendimiento más los propios de microservicios apoya al rendimiento de una aplicación con arquitectura de microservicios.

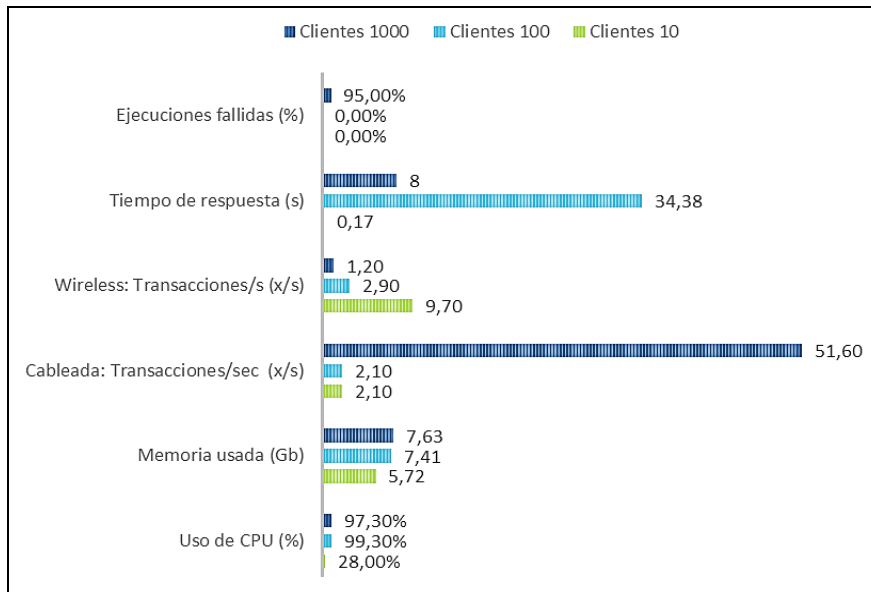


Figura 49 Rendimiento en Microservicios (sin patrones de diseño)

Fuente: La Autora

Elaboración: La Autora

### 4.2.3 Por recurso analizado.

Los siguientes resultados muestran el rendimiento de cada recurso analizado frente a cada aplicación desarrollada, estos son: CPU, memoria, red y base de datos. Para cada uno de ellos se considera el escenario donde 1.000 clientes acceden a 10.000 registros simultáneamente.

#### 4.2.3.1 CPU.

**Escenario:** Rendimiento de CPU cuando 1.000 clientes acceden a 10.000 registros concurrentemente (Ver Tabla 29).

Tabla 29 Análisis del rendimiento del CPU VS Aplicaciones desarrolladas

	Monolito Versión 1	Monolito Versión 2	Servicios REST (Con patrones de diseño)	Servicios REST (Sin patrones de diseño)	Microservicios (MS) (Con patrones de diseño)	Microservicios (MS) (Sin patrones de diseño)
Uso de CPU %	98,10%	96,80%	98,50%	97,30%	98,60%	97,30%

Fuente: La Autora

Elaboración: La Autora



Con 1.000 Clientes accediendo simultáneamente a cada una de las aplicaciones para recuperar una respuesta de 10.000 registros por cliente todas las aplicaciones usan el CPU casi en su totalidad, con valores que superan el 90 %.

Bajo este escenario, en la Figura 50 se observa un mayor uso de CPU en las aplicaciones de Servicios REST y Microservicios construidos con patrones de diseño (98.50 %, 98.60%) respecto a las que no lo usan, en este punto se debe considerar el resultado de las peticiones frente al uso del CPU, ya que aunque en todos los casos el uso del CPU es alto, el menor índice de peticiones fallidas se dió en las aplicaciones que fueron construidas con patrones de diseño lo que da a entender que usar patrones de diseño aumenta la carga del servidor de aplicaciones pero esta se ve recompensada al ofrecer un menor tiempo de respuesta a las aplicaciones.

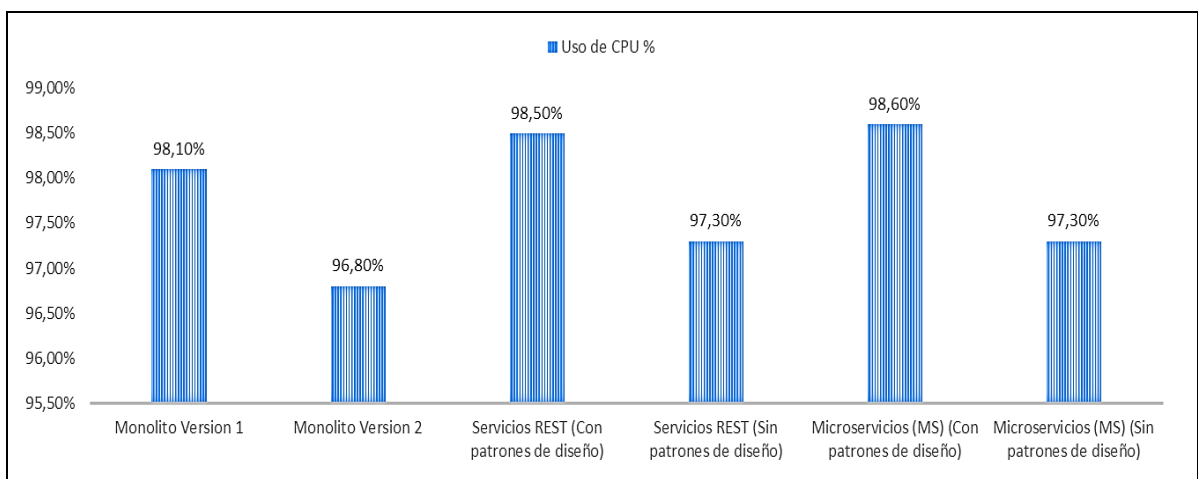


Figura 50 Rendimiento del CPU vs Aplicaciones analizadas

Fuente: La Autora

Elaboración: La Autora

#### 4.2.3.2 Memoria.

**Escenario:** Rendimiento de Memoria cuando 1.000 clientes acceden a 10.000 registros concurrentemente (Ver Tabla 30).

Tabla 30 Análisis de rendimiento de Memoria VS Aplicaciones desarrolladas

	Monolito Versión 1	Monolito Versión 2	Servicios REST (Con patrones de diseño)	Servicios REST (Sin patrones de diseño)	Microservicios (MS) (Con patrones de diseño)	Microservicios (MS) (Sin patrones de diseño)
<b>Memoria usada (gb)</b>	4,57	7,19	6,27	7,26	7,53	7,63
<b>Memoria Libre (gb)</b>	3,26	0,64	1,56	0,57	0,30	0,20
<b>Memoria Total (gb)</b>	7,83	7,83	7,83	7,83	7,83	7,83

Fuente: La Autora

Elaboración: La Autora

Al igual que con el uso de CPU, el rendimiento de memoria difiere en cada aplicación desarrollada, Según la Figura 51, el menor uso de memoria se registra para el Monolito Versión 1 (4,57 Gb) sin embargo se debe tener en cuenta que para esta aplicación el 100 % de las ejecuciones resultaron fallidas, si se aísla este caso y se compara el uso de memoria frente al número de ejecuciones fallidas por cada aplicación, el mejor desempeño de memoria se presenta en las aplicaciones que usan patrones de diseño, especialmente Microservicios (7,53 Gb).

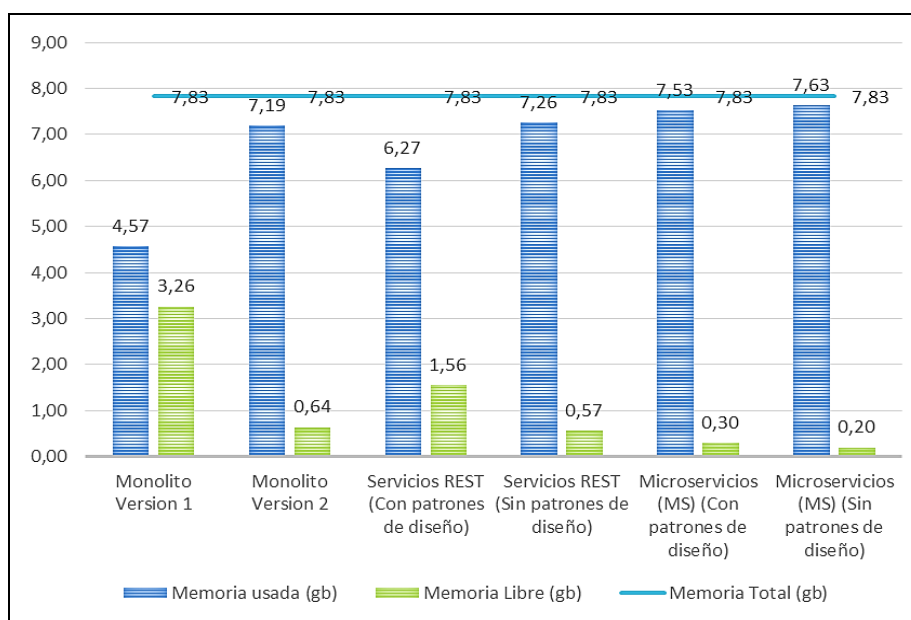


Figura 51 Rendimiento de Memoria VS Aplicaciones analizadas

Fuente: La Autora

Elaboración: La Autora

#### 4.2.3.3 Red.

**Escenario:** Rendimiento de la Red cuando 1.000 clientes acceden a 10.000 registros concurrentemente (Ver Tabla 31).

Tabla 31 Análisis de rendimiento de la red VS Aplicaciones desarrolladas

		Monolito Versión 1	Monolito Versión 2	Servicios REST (Con patrones de diseño)	Servicios REST (Sin patrones de diseño)	MS (Con patrones de diseño)	MS (Sin patrones de diseño)
<b>Cableada</b>	Velocidad de conexión (kbps)	241,00	54,48	522,00	31,30	57,40	62,90
	Bytes enviados/s (kb/s)	0,00	0,21	0,72	0,20	9,70	0,12
	Bytes recibidos/s (kb/s)	212,70	4826,27	9440,46	2569,14	5266,26	476,81
	Trans/s (x/s)	102,20	12,60	27,90	14,30	72,00	51,60
<b>Wireless</b>	Velocidad de conexión (kbps)	592,00	8,87	60,90	658,00	90,40	62,70
	Bytes enviados/s (kb/s)	0,00	0,05	0,12	0,15	7,78	0,16
	Bytes recibidos/s (kb/s)	254,74	708,35	1460,42	1855,41	1544,26	249,30
	Trans/s (x/s)	122,40	48,00	17,80	9,10	57,70	1,20

Fuente: La Autora

Elaboración: La Autora

El rendimiento de la red se refleja en la cantidad de bytes enviados y recibidos por cada aplicación considerando el escenario mencionado. De manera general, se registró una mayor cantidad de bytes enviados y recibidos en la red cableada respecto a Wireless indistintamente de la aplicación analizada. Individualmente, la aplicación que registra una mayor cantidad de

bytes enviados es la aplicación de Microservicios con patrones de diseño (9,70 kb/s) mientras que la mayor cantidad de bytes recibidos es para Servicios REST con patrones seguido de la aplicación de Microservicios igualmente con patrones de diseño.

En la Figura 52 se observa que el mayor número de transacciones por segundo se generó con el Monolito Versión 1, no obstante, este caso no se toma en cuenta ya que se sabe que todas las peticiones en esta aplicación resultaron con código de error 500 lo que deja a los Microservicios como la aplicación que procesa más transacciones por segundo (72 transacciones/s en red cableada y 57,70 con red Wireless), esto es entendible ya que los Microservicios, a diferencia de las otras aplicaciones hacen énfasis en el envío de mensajes sobre la red en lugar de llamadas a funciones.

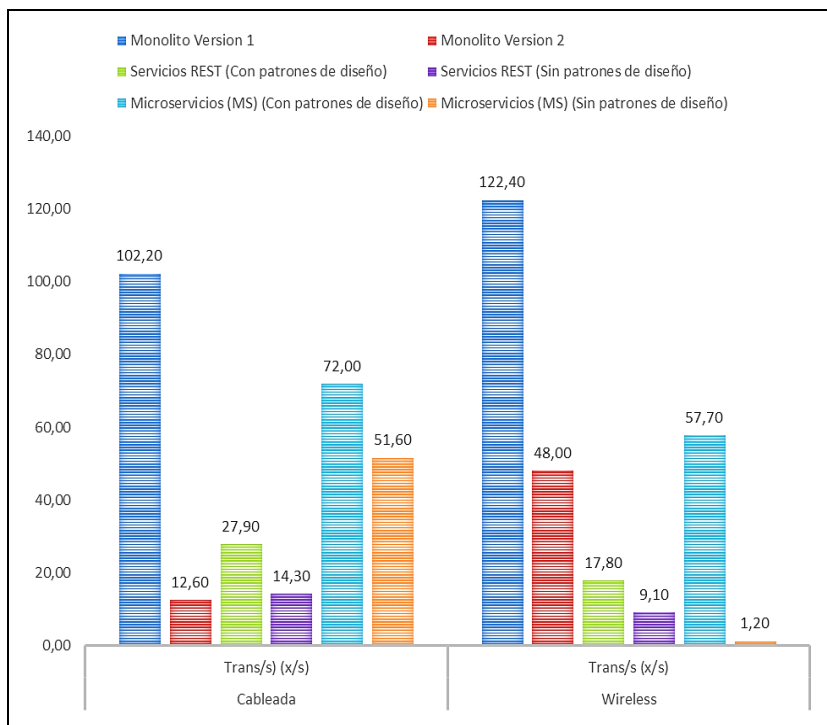


Figura 52 Rendimiento de la red VS Aplicaciones desarrolladas

Fuente: La Autora

Elaboración: La Autora

#### 4.2.3.4 Base de Datos.

**Escenario:** Rendimiento de la Base de datos cuando 1.000 clientes acceden a 10.000 registros concurrentemente (Ver Tabla 32).

Tabla 32 Análisis de rendimiento de la Base de Datos VS Aplicaciones desarrolladas

	Monolito Versión 1	Monolito Versión 2	Servicios REST (Con patrones de diseño)	Servicios REST (Sin patrones de diseño)	Microservicios (MS) (Con patrones de diseño)	Microservicios (MS) (Sin patrones de diseño)
<b>Exec Avg (s)</b>	4,17	45,51	27,95	49,93	6,23	53,72
<b>Exec Min (s)</b>	3,1	2,81	2,73	24,38	0,06	1,27
<b>Exec Max (s)</b>	5,49	73,54	33,61	68,93	13,37	59,42
<b>Tiempo de respuesta (s)</b>	0,01	11,14	31,00	61,43	10,74	8,00
<b>Ejecuciones fallidas (%)</b>	100,00%	89,27%	81,40%	90,20%	79,20%	95,00%

Fuente: La Autora

Elaboración: La Autora

El rendimiento de la base de datos se evalúa en base al tiempo de respuesta (expresado en segundos) que le toma a cada aplicación responder al escenario de prueba planteado. Como se observa en la Tabla 32, el menor tiempo de respuesta registrado se da en la aplicación Monolito Versión 1, pero tal como ya se indicó antes, este caso no se toma en cuenta ya que bajo este escenario todas las ejecuciones resultaron fallidas con error 500 del servidor.

En la gráfica de la Figura 53 se demuestra que la aplicación con menor tiempo de respuesta con relación al número de ejecuciones fallidas es la aplicación de Microservicios construida con patrones de diseño (10.74 segundos y 79,20 % de ejecuciones fallidas), seguido de la aplicación de servicios REST también construida con patrones de diseño (31 segundos y 81,40 % de ejecuciones fallidas), lo que demuestra que aplicar patrones de diseño orientados al rendimiento mejoran el desempeño de este atributo, especialmente al combinar su uso con el estilo arquitectónico REST.

Cabe mencionar que mientras se realizaron las pruebas se hizo configuraciones adicionales a JMeter, ya que al ser una herramienta desarrollada en Java a menudo presentaba errores de desbordamiento de memoria durante la ejecución de los escenarios de prueba, para solucionar este error se añadió un parámetro a la cadena que ejecuta JMeter asignándole el máximo de memoria aceptable a la JVM:

Con los resultados expuestos previamente se puede determinar que usar patrones de diseño orientados al rendimiento (combinados con los propios de Microservicios) reduce el tiempo de respuesta en las aplicaciones con arquitecturas de microservicios sin embargo aumenta el consumo de recursos como CPU y memoria, cuando no se usan patrones de diseño se produce el efecto contrario, es decir, se reduce el consumo de CPU y memoria pero aumenta el tiempo de respuesta. Adicional a las observaciones realizadas se debe aclarar que la gestión de solicitudes entrantes dependerá también de la configuración del servidor la cual no es objeto de este trabajo de fin de titulación.

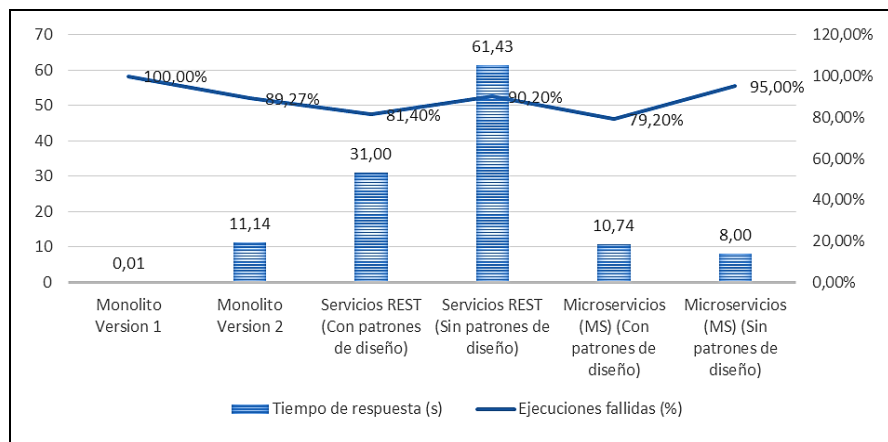


Figura 53 Rendimiento de la Base de Datos VS Aplicaciones desarrolladas

Fuente: La Autora

Elaboración: La Autora

## CONCLUSIONES

Al finalizar el presente trabajo de titulación se concluye:

- Utilizar microservicios representa una solución que se puede adoptar dentro de un entorno empresarial respecto a arquitecturas monolíticas como n-capas e incluso REST ya que cada microservicio se enfoca en una funcionalidad específica de la aplicación y puede ser desarrollado y desplegado independientemente lo que mejora la escalabilidad y por consiguiente el rendimiento de la aplicación.
- Al aplicar una arquitectura de microservicios se asume el reto de dividir una aplicación en pequeños servicios que se ponen a funcionar individualmente y se comunican entre sí, esto significa que los servicios deben integrarse en todo momento para conseguir el comportamiento esperado de la aplicación.
- Es conveniente usar un enfoque que apoye el proceso de migración de un monolito a microservicios, dicho enfoque debe considerar el uso de una metodología de desarrollo y ajustarse a las necesidades actuales de negocio, estas se refieren a tareas de integración, entrega y despliegue continuos propias de la cultura empresarial DevOps.
- La guía adecuada para migrar una aplicación con arquitectura monolítica hacia una orientada a microservicios consiste en definir y utilizar una ruta de migración gradual que especifique fases y dentro de cada fase, las tareas y objetivos a conseguir al término de cada una, las arquitecturas de software a implementarse, patrones de diseño, etc que garanticen que la migración se lleve a cabo con éxito.
- Respecto a la medición del rendimiento en una arquitectura de microservicios se debe tener en cuenta cuáles son los factores que impactan positiva y negativamente a este atributo de calidad además de considerar el uso de patrones de diseño orientados al rendimiento y a la construcción de microservicios y buenas prácticas de programación como procedimientos almacenados, estos pueden ser implementados durante el desarrollo de las aplicaciones.
- El modelo de evaluación de rendimiento que se utilizó en este trabajo es óptimo ya que evalúa y compara el rendimiento de las métricas más relevantes de los recursos hardware (CPU, memoria, red y base de datos) frente a cada aplicación desarrollada lo que permite determinar qué arquitectura presenta un mejor desempeño en cuanto al atributo de calidad de rendimiento.

- En base al modelo de evaluación de rendimiento utilizado se comprobó que la arquitectura de microservicios presenta un mejor desempeño de rendimiento ya que al procesar solicitudes de múltiples clientes el tiempo de respuesta y la tasa de error de solicitudes no procesadas correctamente fue inferior en comparación al de otras aplicaciones involucradas en la migración.
- El rendimiento óptimo de una arquitectura de microservicios dependerá del uso de patrones de diseño y de patrones propios de microservicios ya que estos requieren de llamadas a procedimientos remotos que son respaldados por dichos patrones, por esta razón se observará una mayor actividad sobre la red respecto a otras arquitecturas (n-capas, REST).



## RECOMENDACIONES

Al finalizar el presente trabajo fin de titulación se recomienda:

- Utilizar la arquitectura de microservicios en aplicaciones que no se ajustan a las necesidades de negocio actuales (p.e alto rendimiento, optimización de tiempos de respuesta, etc.) y que necesiten actualizar y gestionar sus funcionalidades de manera independiente y rápida.
- Hacer énfasis en el diseño de los microservicios, cada uno de estos debe cubrir una funcionalidad específica de la aplicación e integrarse con otros microservicios para asegurar el comportamiento esperado de la aplicación.
- Se recomienda usar la cultura empresarial DevOps dentro de un proceso de migración de aplicación monolítica a microservicios con el fin de asegurar la integración, entrega y despliegue continuos de cada uno de los microservicios desarrollados durante la migración.
- Establecer y utilizar una ruta de migración que guíe el proceso de transformar una aplicación monolítica a microservicios como el que se propone en este trabajo, con el fin de asegurar la transformación gradual de la aplicación origen hasta convertirse en microservicios escalables y de alto rendimiento.
- Para favorecer el rendimiento de una arquitectura de microservicios se recomienda implementar buenas prácticas de programación como el uso de procedimientos almacenados y patrones de diseño orientados al rendimiento (Singleton, Proxy, entre otros) y a la construcción de microservicios (Single Service per Host, API Gateway, etc.), la aplicación de estas prácticas se verá reflejado en el desenvolvimiento de los recursos de hardware.
- Utilizar el modelo de evaluación de rendimiento propuesto en este trabajo para comparar el desempeño de este atributo de calidad por cada aplicación involucrada en la migración de monolito a microservicios respecto a las métricas establecidas para los recursos hardware lo que permitirá concluir qué arquitectura posee un mejor rendimiento.
- Se recomienda configurar escenarios de prueba genéricos (número de clientes, número de registros y solicitud HTTP similares) para todas las aplicaciones que participan en la migración de monolito a microservicios, de esta manera los resultados serán objetivos y se evita que favorezcan o perjudiquen a una arquitectura en particular.

- Destinar más recursos de red a una arquitectura de microservicios ya que estas realizan llamadas a procedimientos remotos que deben ser gestionadas por cada microservicio que participa en la aplicación.

## BIBLIOGRAFÍA

- Amaral, M., & Carrera, D. (2015). Performance Evaluation of Microservices Architectures using Containers. <https://doi.org/10.1109/NCA.2015.49>
- Apache Software Foundation. (2017a). Apache JMeter - Apache JMeter™. Retrieved October 23, 2017, from <https://jmeter.apache.org/index.html>
- Apache Software Foundation. (2017b). Maven – Introduction. Retrieved August 23, 2017, from <https://maven.apache.org/what-is-maven.html>
- Barrios, J. M. (2001). Java Servlets. Retrieved August 23, 2017, from <https://users.dcc.uchile.cl/~jbarrios/servlets/general.html>
- Bateman, R. (2012). Companion Notes on RESTful Web Services with Java and Jersey. Retrieved August 24, 2017, from <http://www.javahotchocolate.com/tutorials/restful.html>
- Brown, K. (2016). Refactoring to microservices, Part 1: What to consider when migrating from a monolith. Retrieved April 28, 2017, from <https://www.ibm.com/developerworks/cloud/library/cl-refactor-microservices-bluemix-trs-1/index.html>
- Brown Kyle. (2016). Refactoring to microservices, Part 1: What to consider when migrating from a monolith. *IBM*, 1–6.
- Costa, B., Pires, P. F., Delicato, F. C., & Merson, P. (2016). The Journal of Systems and Software Evaluating REST architectures — Approach , tooling and guidelines, 112, 156–180. <https://doi.org/10.1016/j.jss.2015.09.039>
- Davis, J., & Daniels, K. (2016). *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale*. “ O’Reilly Media, Inc.”
- de Kort, W. (2016). *What Is DevOps? DevOps on the Microsoft Stack*. [https://doi.org/10.1007/978-1-4842-1446-6\\_1](https://doi.org/10.1007/978-1-4842-1446-6_1)
- Docker Inc. (2016). Docker Hub. Retrieved October 23, 2017, from <https://hub.docker.com/>
- Docker Inc. (2017a). Overview of Docker Compose | Docker Documentation. Retrieved October 23, 2017, from <https://docs.docker.com/compose/overview/>
- Docker Inc. (2017b). What is Docker? Retrieved October 23, 2017, from <https://www.docker.com/what-docker>
- Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2016). Microservices : yesterday , today , and tomorrow. <https://doi.org/10.13140/RG.2.1.3257.4961>
- Farías, K. I. (2017). *Definición de un ambiente de construcción de aplicaciones empresariales a*

- través de DevOps, Microservicios y Contenedores*. Universidad Técnica Particular de Loja.
- Feng, X., Shen, J., & Fan, Y. (2009). REST : An Alternative to RPC for Web Services Architecture. *Security*, 0–3. <https://doi.org/10.1109/ICFIN.2009.5339611>
- Fernández, A. (2013). Servicios web RESTful con HTTP. Parte I: Introducción y bases teóricas. Retrieved August 23, 2017, from <http://www.adwe.es/general/colaboraciones/servicios-web-restful-con-http-parte-i-introduccion-y-bases-teoricas>
- Fowler, M. (2004). Strangler Application. Retrieved May 10, 2017, from <https://www.martinfowler.com/bliki/StranglerApplication.html>
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1997). Design Patterns – Elements of Reusable Object-Oriented Software. *A New Perspective on Object-Oriented Design*, 334. <https://doi.org/10.1093/carcin/bgs084>
- Github. (2017). Docker Compose Releases. Retrieved October 23, 2017, from <https://github.com/docker/compose/releases>
- Guerrero, C. A., Suárez, J. M., & Gutiérrez, L. E. (2013). Patrones de diseño GOF (the gang of four) en el contexto de procesos de desarrollo de aplicaciones orientadas a la web. *Informacion Tecnologica*, 24(3), 103–114. <https://doi.org/10.4067/S0718-07642013000300012>
- Hamad, H., Saad, M., & Abed, R. (2010). Performance evaluation of restful web services for mobile devices. *International Arab Journal of E-Technology*, 1(3), 72–78. Retrieved from [http://www.iajet.org/iajet\\_files/vol.1/no.3/Performance Evaluation of RESTful Web Services for Mobile Devices.pdf](http://www.iajet.org/iajet_files/vol.1/no.3/Performance%20Evaluation%20of%20RESTful%20Web%20Services%20for%20Mobile%20Devices.pdf)  
<http://dblp.uni-trier.de/db/journals/iajet/iajet1.html#HamadSA10>
- Happe, J., Koziol, H., & Reussner, R. (2011). Facilitating Performance Predictions Using Software Components. *IEEE Software*, 27–34.
- Hasselbring, W. (2016). Microservices for Scalability. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*, 133–134. <https://doi.org/10.1145/2851553.2858659>
- Hibernate. (2017). Hibernate ORM - Hibernate ORM. Retrieved August 23, 2017, from <http://hibernate.org/orm/>
- IEE. (1990). IEEE standard glossary of software engineering terminology.
- Knoche, H. (2016). Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (pp. 121–124).
- Kumar, K., & Prabhakar, T. V. (2010). Pattern-oriented knowledge model for architecture design. *Proceedings of the 17th Conference on Pattern Languages of Programs - PLOP '10*, 1–21.

- <https://doi.org/10.1145/2493288.2493311>
- Lewis, J., & Fowler, M. (2014). Microservices. Retrieved September 3, 2016, from <http://martinfowler.com/articles/microservices.html>
- Medium. (2017). How can you refactor a monolithic application into microservices? Retrieved May 10, 2017, from <https://medium.com/@NeotericEU/how-can-you-refactor-a-monolithic-application-into-microservices-2eef8e323840>
- Microsoft. (2009). Chapter 16: Quality Attributes-Performance. Retrieved September 3, 2016, from <https://msdn.microsoft.com/en-us/library/ee658094.aspx#Performance>
- Moya, R. (2015). Patrón Singleton en Java, con ejemplos. Retrieved August 24, 2017, from <https://jarroba.com/patron-singleton-en-java-con-ejemplos/>
- Namiot, D., & Sneps-Sneppé, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9), 24–27.
- Netflix. (2014). What is Zuul? Retrieved October 23, 2017, from <https://github.com/Netflix/zuul/wiki>
- Oracle. (2017). 4.1.5 Working with Stored Procedures. Retrieved August 23, 2017, from <https://dev.mysql.com/doc/connector-net/en/connector-net-tutorials-stored-procedures.html>
- Peña, A. (2016). Qué es un servicio RESTFUL. Retrieved August 24, 2017, from <http://www.i2factory.com/es/integracion/qué-es-un-servicio-restful>
- Pivotal Software. (2017). Spring Cloud. Retrieved October 23, 2017, from <http://projects.spring.io/spring-cloud/>
- Polo, M., & Villafranca, D. (2002). *Introducción a las aplicaciones Web con JAVA. Departamento de informatic di Universidad de Valladolid*. Valladolid. Retrieved from <http://www.infor.uva.es/~jvegas/cursos/buendia/pordocente/node11.html>
- Qiangdavidliu. (2014). Eureka at a glance. Retrieved October 23, 2017, from <https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance>
- Richardson, C. (2017). A pattern language for microservices. Retrieved October 23, 2017, from <http://microservices.io/patterns/>
- Richardson, C., & Smith, F. (2016). *Microservices. From Design to Deployment*. NGINX.
- Santis, S. De, Florez, L., Nguyen, D. V, & Rosa, E. (2016). Evolve the Monolith to Microservices with Java and Node. *IBM Redbooks*.
- Savchenko, D., & Radchenko, G. (2015). Microservices validation: Methodology and implementation. *CEUR Workshop Proceedings*, 1513, 21–28.
- Spring Initializr, & Pivotal Web Services. (2017). Spring Initializr. Retrieved October 23, 2017, from <https://start.spring.io/>

- Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., & Gil, S. (2015). Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. *10th Computing Colombian Conference*, 583–590. <https://doi.org/10.1109/ColumbianCC.2015.7333476>
- W3C. (2017). Guía Breve de Servicios Web. Retrieved April 16, 2017, from <http://www.w3c.es/Divulgacion/GuiasBreves/ServiciosWeb>
- Zhang, X., Wen, Z., Wu, Y., & Zou, J. (2011). The Implementation and Application of the Internet of Things Platform based on the REST Architecture. *IEEE*.
- Zou, J., Mei, J., & Wang, Y. (2010). From Representational State Transfer to Accountable State Transfer Architecture, 299–306. <https://doi.org/10.1109/ICWS.2010.56>

## **ANEXOS**

**Anexo A: Documento de Especificación de Requerimientos para SRCL**

**Especificación de Requerimientos de Software  
SISTEMA DE REGISTRO DE CURSOS EN LÍNEA**

**Versión 1.0.0**



## Información del Documento

Título:	Especificación de Requerimientos de Software
Subtítulo:	Sistema de Registro de Cursos en línea
Versión:	1.0.0
Archivo:	SISTEMA DE REGISTROS DE CURSOS EN LÍNEA
Fuente:	Yaguachi L.
Estado:	Borrador

## Lista de Cambios

Versión	Fecha	Autor	Descripción

# ESPECIFICACIÓN DE REQUERIMIENTOS DE SOFTWARE

## 1. Introducción

### 1.1. Propósito

El presente documento provee una descripción de los requerimientos del proyecto de software Sistema de Registro de cursos en línea. Estos requerimientos son la base para el desarrollo del plan del proyecto por lo que luego de establecido éste, cualquier requerimiento adicional o modificación a los existentes deberá seguir el procedimiento de Administración de Requerimientos de Cambio, pudiendo derivar esto en modificaciones al cronograma o a los costos del proyecto. Adicionalmente, este documento define la terminología (términos técnicos, abreviaciones, acrónimos, etc.).

### 1.2. Definiciones, abreviaturas y acrónimos

**Beneficio.** Asignado por el responsable funcional del proyecto conjuntamente con los stakeholders. Representa el beneficio relativo al usuario final, y ayuda a la administración del alcance y determinación de las prioridades para el desarrollo. Sus valores posibles son:

Valor	Descripción
Crítico	Un requerimiento que implica una característica que obligatoriamente debe ser implementada. La no implementación o fallas en la implementación significan que el sistema no cumplirá las necesidades del usuario.
Importante	Requerimientos que implican características importantes para la efectividad y eficiencia de la aplicación. La liberación de una versión de la aplicación no será demorada por la falta de la implementación de una característica importante.
Útil	Requerimientos que implican características que no tienen un valor agregado importante o no incrementan significativamente la satisfacción del usuario, debido a que existen formas alternativas de resolverlo.

**Solicitado Por.** Representa la persona que solicita la implementación del requerimiento y que tiene la aprobación del responsable funcional del proyecto.

**Fecha.** Representa la fecha en que fue solicitado el requerimiento.

## 2. Requerimientos por unidad funcional

### 2.1. Funcionalidad: Registro de estudiantes

#### 2.1.1. REQ-REGE-01: Poder ingresar datos en el formulario de registro:

El ingreso de datos se llevará a cabo a través de un formulario. Dentro de estos se comprenderá:

- nombres
- apellidos
- usuario
- contraseña
- correo electrónico
- fecha de nacimiento

**Fecha:** 25 de mayo de 2017

**Beneficio:** Crítico

**Origen:** Srta. Lady Yaguachi

#### 2.1.2. **REQ-REGE-02: Validar la edad:**

Se validará la edad mediante la fecha de nacimiento ingresada en el formulario de registro.

**Fecha:** 25 de mayo de 2017

**Beneficio:** Importante

**Origen:** Srta. Lady Yaguachi

#### 2.1.3. **REQ-REGE-03: Guardar registro de estudiante:**

Se creará un nuevo registro de estudiante con los datos ingresados en el formulario de registro.

**Fecha:** 25 de mayo de 2017

**Beneficio:** Crítico

**Origen:** Srta. Lady Yaguachi

### 2.2. **Funcionalidad: Login de Estudiantes**

#### 2.2.1. **REQ-LOGE-01: Poder ingresar datos en el formulario de login:**

El ingreso de datos se llevará a cabo a través de un formulario. Dentro de estos se comprenderá:

- usuario
- contraseña

**Fecha:** 25 de mayo de 2017

**Beneficio:** Importante

**Origen:** Srta. Lady Yaguachi

### 2.3. **Funcionalidad: Matricula a Cursos**

#### 2.3.1. **REQ-MATC-01: Mostrar lista de cursos disponibles:**

Se mostrará una lista de los cursos comprendidos en la categoría permitida al estudiante según su edad.

**Fecha:** 25 de mayo de 2017

**Beneficio:** Útil

**Origen:** Srta. Lady Yaguachi

**2.3.2. REQ- MATC-02: Poder seleccionar un curso:**

El estudiante podrá seleccionar uno de los cursos disponibles según la categoría asignada al mismo.

**Fecha:** 25 de mayo de 2017

**Beneficio:** Útil

**Origen:** Srta. Lady Yaguachi

**2.3.3. REQ- MATC-03: Guardar matrícula de un estudiante:**

Se creará un nuevo registro de matrícula con la selección de curso realizada por el estudiante.

**Fecha:** 25 de mayo de 2017

**Beneficio:** Crítico

**Origen:** Srta. Lady Yaguachi

**2.4. Funcionalidad: Consulta de Cursos tomados por un Estudiante**

**2.4.1. REQ-CONC-01: Mostrar cursos de estudiante:**

Se mostrará una lista de los cursos en los que se ha matriculado un estudiante. Por cada curso se presentará la siguiente información:

- Nombre de Curso
- Categoría
- Periodo
- Nombre de tutor
- Apellido de tutor
- Estado de matrícula

**Fecha:** 25 de mayo de 2017

**Beneficio:** Importante

**Origen:** Srta. Lady Yaguachi

**2.5. Funcionalidad: Consulta de Estudiantes de un Curso**

**2.5.1. REQ-CONE-01: Mostrar estudiantes de un curso:**

Se mostrará una lista de estudiantes pertenecientes a un curso. Por cada estudiante se presentará la siguiente información:

- Nombres
- Apellidos
- Fecha de matricula
- Estado

**Fecha:** 25 de mayo de 2017  
**Beneficio:** Importante  
**Origen:** Srta. Lady Yaguachi

## 2.6. **Funcionalidad: Registro de Notas de un curso**

### 2.6.1. **REQ-REGN-01: Registrar nota de evaluación:**

Se registrará las notas de la evaluación de cada estudiante en su expediente.

**Fecha:** 25 de mayo de 2017  
**Beneficio:** Crítico  
**Origen:** Srta. Lady Yaguachi.

### 2.6.2. **REQ- REGN -02: Cálculo de nota final de curso:**

La nota final del curso por cada estudiante se calculará como la nota promedio de las notas correspondientes a las evaluaciones registradas anteriormente.

**Fecha:** 25 de mayo de 2017  
**Beneficio:** Crítico  
**Origen:** Srta. Lady Yaguachi.

### 2.6.3. **REQ- REGN -03: Cambiar estado de matrícula:**

Es estado final de la matricula se establecerá en función de la nota final del curso, si la nota final es mayor o igual a 8 el estado será "APROBADA", caso contrario el sistema registrará "REPROBADA".

**Fecha:** 25 de mayo de 2015  
**Beneficio:** Importante  
**Origen:** Srta. Lady Yaguachi.

## 3. **Requerimientos no contemplados**

## 4. **Limitaciones**

- Los cursos se ofertan a través de internet y el registro se lleva a cabo mediante un navegador web.
- Los cursos están clasificados por categorías, las mismas que son:
  - Junior: 12 a 16 años.
  - Jóvenes: 17 a 22 años.
  - Adultos: 23 a 64 años.
  - Tercera edad: De 65 años en adelante.
- Cada curso tendrá asignada una duración en base al número de horas asignado
- Los resultados finales por cada estudiantes comprenderán:
  - Nota final sobre 10.

- o El estado respectivo: aprobado o reprobado.
- Para la acreditación del estudiante en el curso es necesario un mínimo el 80% de la nota total.

## 5. Anexos

### 5.1.Caso de Uso

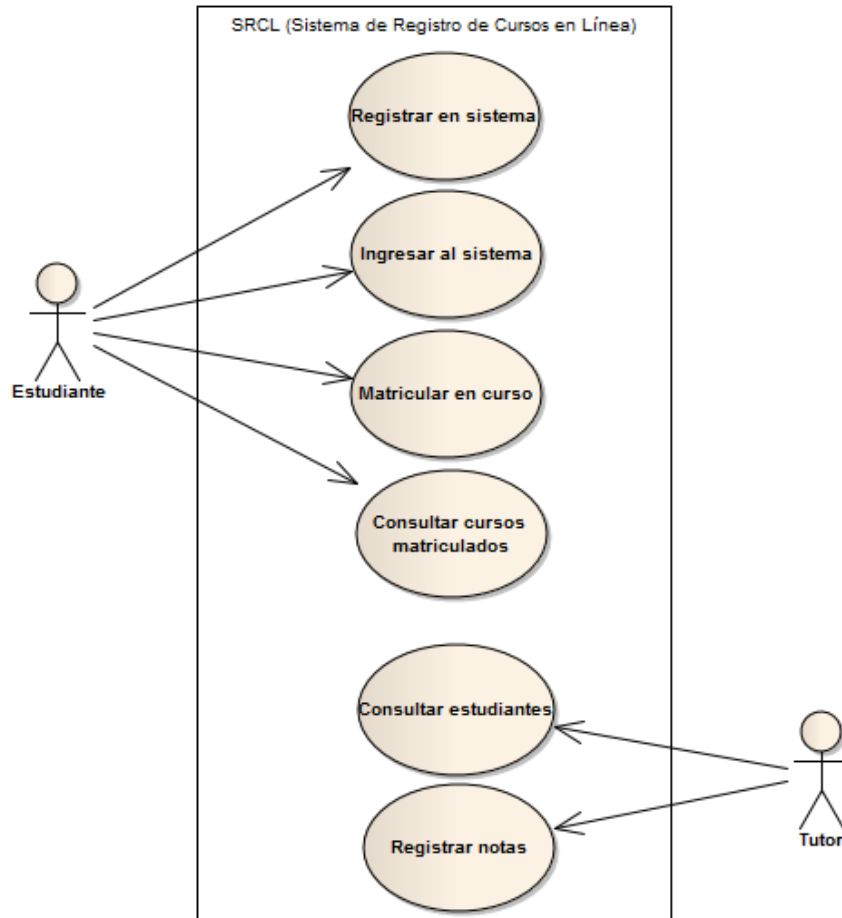


Figura 54. Caso de uso para SRCL

## Anexo B: Modelo Entidad-Relación de SRCL

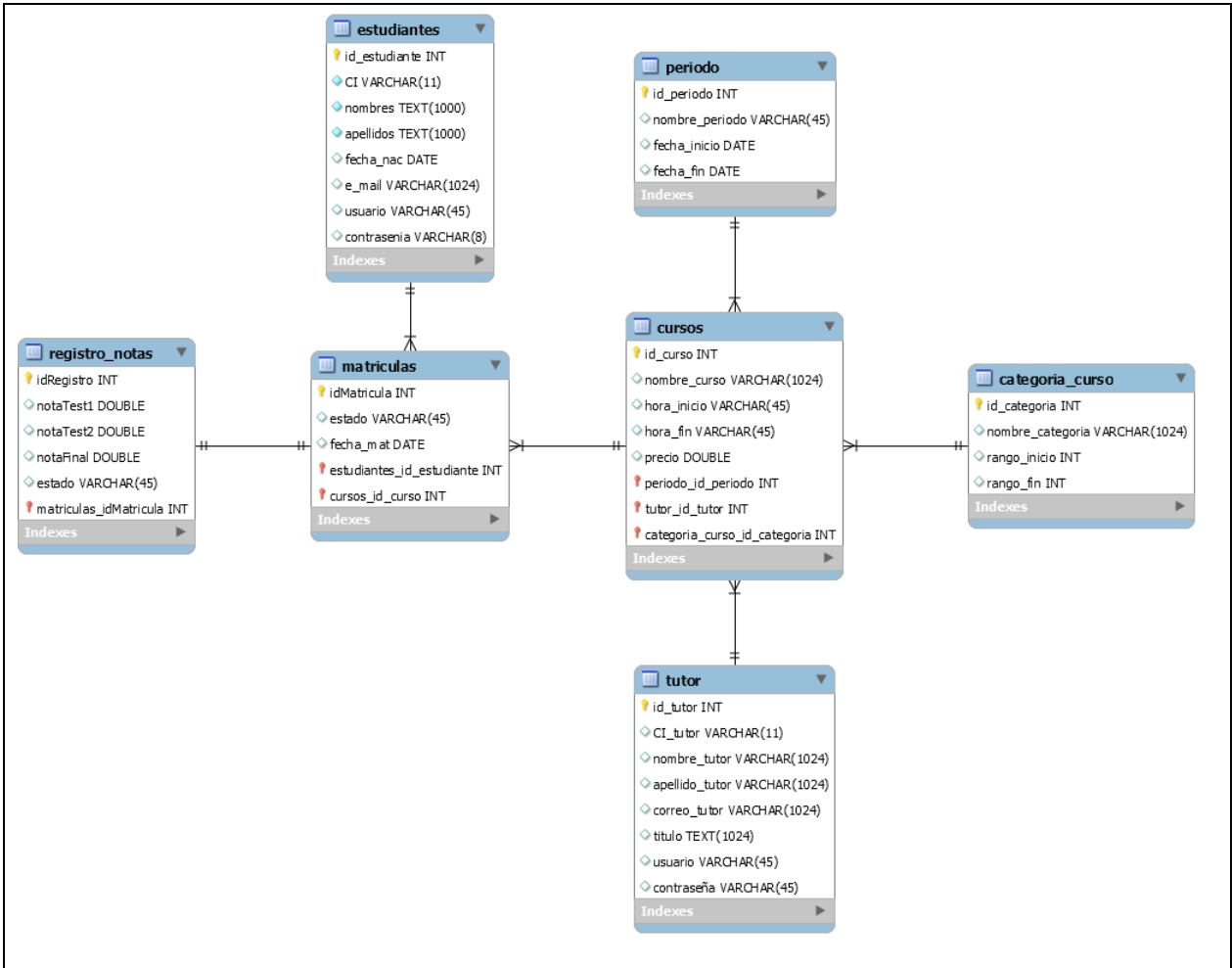


Figura 55 Modelo E-R de SRCL

Fuente: La Autora

Elaboración: La Autora

## Anexo C: Despliegue de Monolito V1

ID Matricula	Nombres	Estado
1804	Abbess Mallory	ACEPTADA
1536	Abercrombie Michaela	ACEPTADA

Figura 56. Funcionalidad: Consultar matriculas de un curso en Monolito V1

Fuente: La Autora

Elaboración: La Autora

## Anexo D: Despliegue de Monolito V2

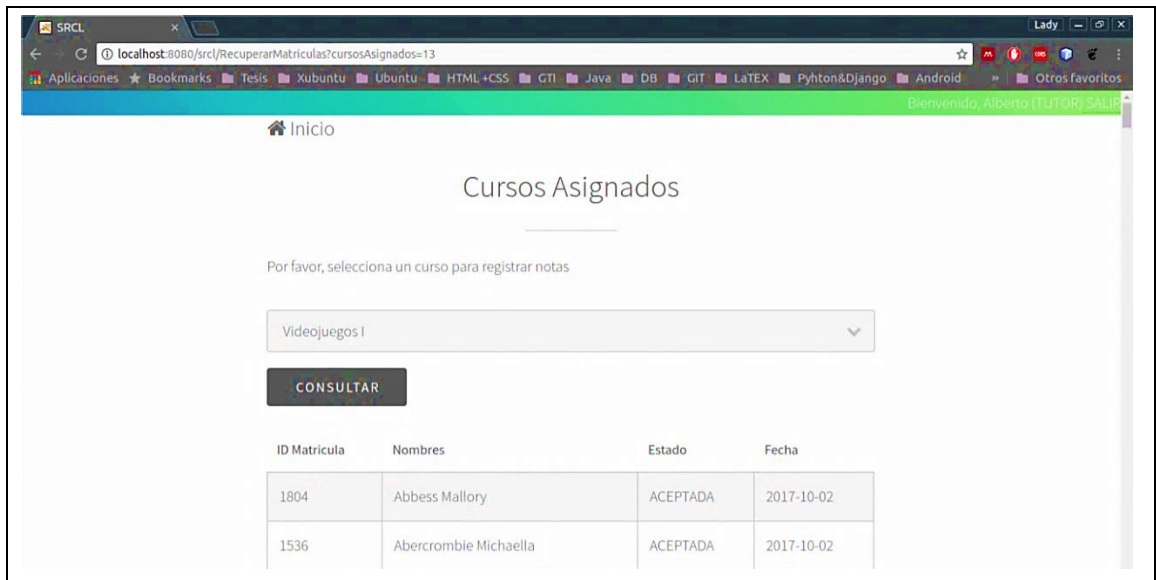


Figura 57 Funcionalidad: Consultar matriculas de un curso en Monolito V2

Fuente: La Autora

Elaboración: La Autora

## Anexo E: Despliegue de Servicios REST

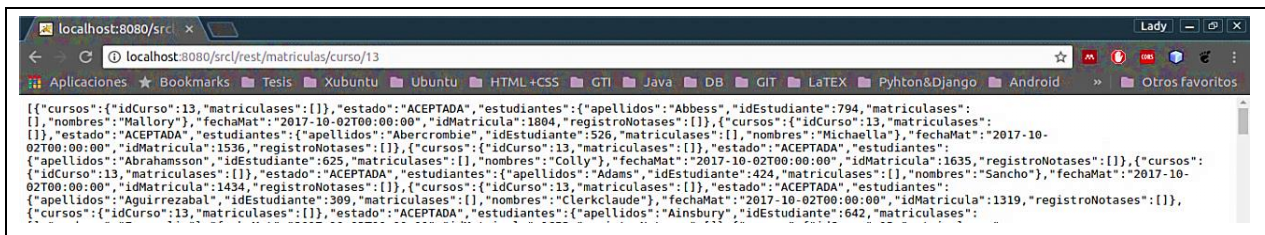


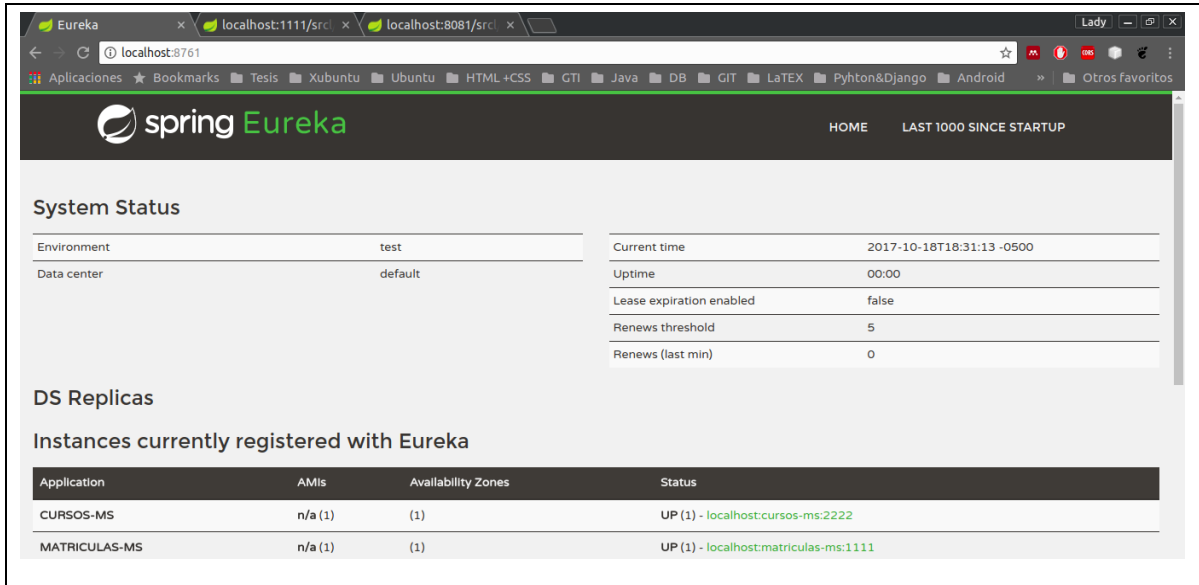
Figura 58 Despliegue de servicio: Obtener Matrículas

Fuente: La Autora

Elaboración: La Autora



## Anexo F: Despliegue local de Eureka



The screenshot shows the Eureka web interface in a browser. The page title is "spring Eureka" and the URL is "localhost:8761". The navigation bar includes "HOME" and "LAST 1000 SINCE STARTUP".

### System Status

Environment	test	Current time	2017-10-18T18:31:13 -0500
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	0

### DS Replicas

#### Instances currently registered with Eureka

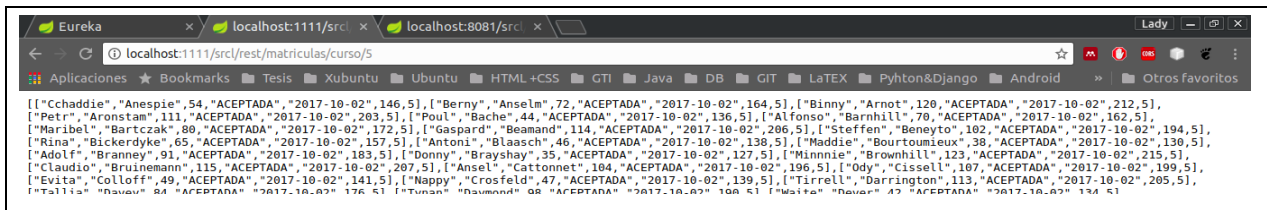
Application	AMIs	Availability Zones	Status
CURSOS-MS	n/a (1)	(1)	UP (1) - localhost:ursos-ms:2222
MATRICULAS-MS	n/a (1)	(1)	UP (1) - localhost:matriculas-ms:1111

Figura 59 Despliegue local de Eureka

Fuente: La Autora

Elaboración: La Autora

## Anexo G: Despliegue local de API REST matriculas



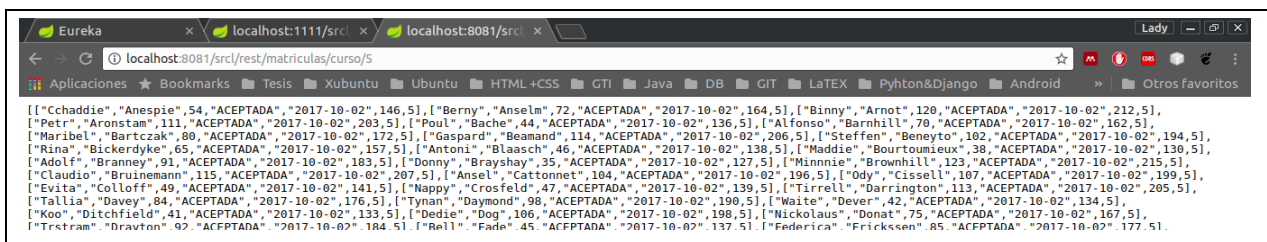
The screenshot shows the Eureka web interface displaying a list of registered instances. The URL is "localhost:1111/src/rest/matriculas/curso/5". The page contains a long list of instance names and their status, all marked as "ACCEPTADA".

Figura 60 Despliegue local de API REST matrículas

Fuente: La Autora

Elaboración: La Autora

## Anexo H: Despliegue Local de Zuul



The screenshot shows the Eureka web interface displaying a list of registered instances. The URL is "localhost:8081/src/rest/matriculas/curso/5". The page contains a long list of instance names and their status, all marked as "ACCEPTADA".

Figura 61 Despliegue local de Zuul

Fuente: La Autora

Elaboración: La Autora

## **Anexo I: Instalación de Docker en Ubuntu 16.04**

1. En un Terminal Linux, descargar la llave GPG del repositorio oficial de Docker en el equipo:  

```
sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --  
recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```
2. Agregar el repositorio de Docker:  

```
echo "deb https://apt.dockerproject.org/repo ubuntu-xenial main" |  
sudo tee /etc/apt/sources.list.d/docker.list
```
3. Actualizar la base de datos de repositorios local:  

```
sudo apt-get update
```
4. Para comprobar que se está descargando Docker del repositorio oficial y no del de Ubuntu, añadir:  

```
apt-cache policy docker-engine
```
5. Finalmente, con este comando se instala Docker  

```
sudo apt-get install -y docker-engine
```
6. Con el comando anterior se instala Docker y se configura para iniciarse al igual que el sistema, para comprobar el estado se utiliza:  

```
sudo systemctl status docker
```

## **Anexo J: Instalación de Docker Compose en Ubuntu 16.04**

Docker Compose se puede instalar a nivel de sistema operativo o dentro de un entorno virtual de Linux, esta instalación se hizo a nivel de S.O. Por lo general se recomienda instalar siempre la última versión de Docker Compose.

1. En un Terminal Linux, ejecutar:  

```
sudo curl -L  
https://github.com/docker/compose/releases/download/1.16.1/docker-  
compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

2. Se añade el siguiente comando para dar permisos de ejecución a los archivos descargados:

```
sudo chmod +x /usr/local/bin/docker-compose
```

3. Se comprueba la instalación de Docker Compose con el comando:

```
docker-compose -version
```

### **Anexo K: Configuración de Docker con la base de datos local**

Este anexo explica el proceso de configuración previo para que un contenedor de Docker pueda conectarse a una base de datos MySQL local.

1. En la base de datos, dar permisos al usuario root sobre la misma, con el parámetro: % se especifica que cualquier host tendrá acceso a la base de datos.

```
GRANT ALL PRIVILEGES ON src1.* TO 'root'@'%';
```

2. Dependiendo de la versión de MySQL, entrar a su archivo de configuración y editar:

```
sudo gedit /etc/mysql/mysql.conf.d/mysqld.cnf  
bind-address = 0.0.0.0
```

Donde 0.0.0.0 indica que se aceptarán conexiones desde cualquier host.

3. Determinar la ip local ya sea con red cableada (eth0) o Wireless (wlan0):

```
ifconfig
```

La ip resultante es la que se debe ubicar en el archivo docker-compose.yml en el parámetro `extra_hosts` junto a un nombre de host, así, cada vez que el servicio de Docker Compose necesite el host de la base de datos, utilizar el nombre establecido al host.

## Anexo L: Archivo de configuración docker-compose.yml

```
version: '3.3'
services:
  eureka:
    build: ./eureka
    ports:
      - "8761:8761"
    expose:
      - "8761"
  svc-mat:
    build: ./svcMatriculas
    ports:
      - "1111:1111"
    expose:
      - "1111"
    depends_on:
      - eureka
    links:
      - eureka
    extra_hosts:
      - "srclbd:192.168.1.6"
  svc-cursos:
    build: ./svcCursos
    ports:
      - "2222:2222"
    expose:
      - "2222"
    depends_on:
      - eureka
    links:
      - eureka
    extra_hosts:
      - "srclbd:192.168.1.6"
```

zuul:

build: ./zuul

ports:

- "8081:8081"

expose:

- "8081"

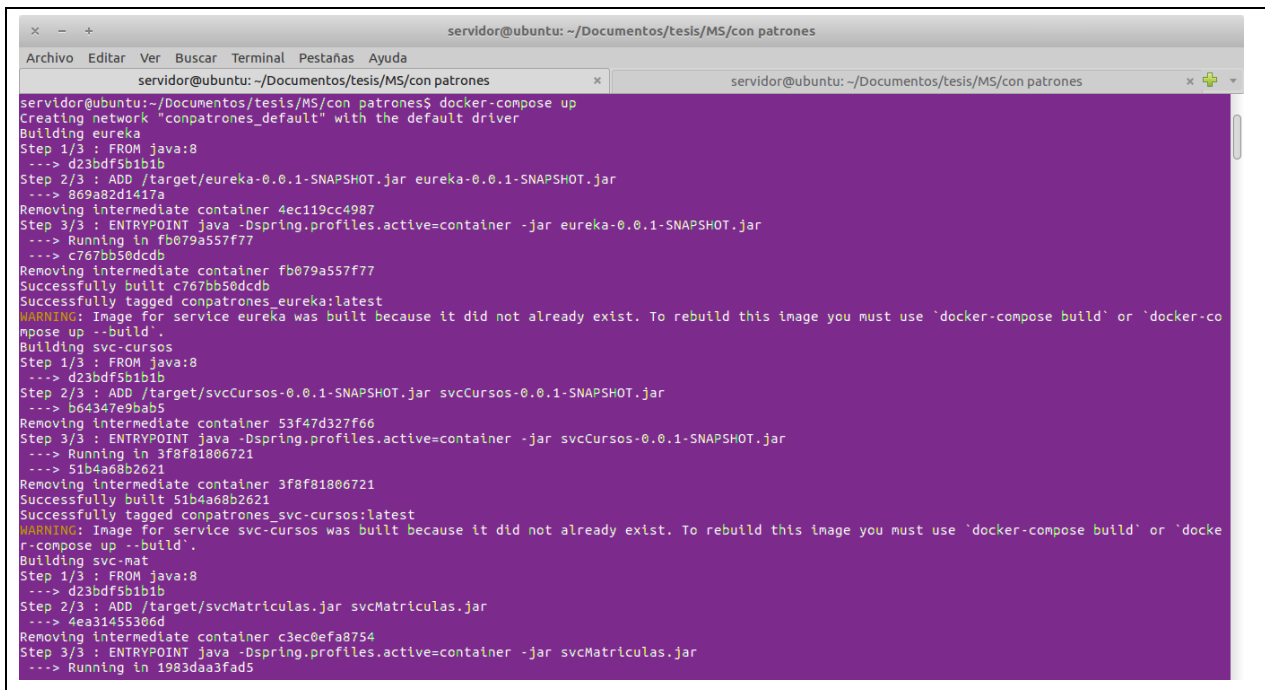
depends\_on:

- eureka
- svc-mat
- svc-cursos

links:

- eureka
- svc-mat
- svc-cursos

## Anexo M: Levantar contenedores con Docker Compose



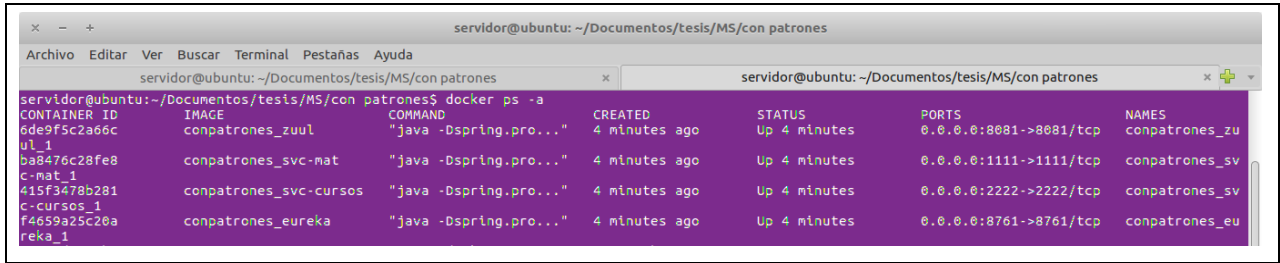
```
servidor@ubuntu: ~/Documentos/tesis/MS/con patrones
Archivo Editar Ver Buscar Terminal Pestañas Ayuda
servidor@ubuntu: ~/Documentos/tesis/MS/con patrones
servidor@ubuntu:~/Documentos/tesis/MS/con patrones$ docker-compose up
Creating network "conpatrones_default" with the default driver
Building eureka
Step 1/3 : FROM java:8
--> d23bdf5b1b1b
Step 2/3 : ADD /target/eureka-0.0.1-SNAPSHOT.jar eureka-0.0.1-SNAPSHOT.jar
--> 869a82d1417a
Removing intermediate container 4ec119cc4987
Step 3/3 : ENTRYPOINT java -Dspring.profiles.active=container -jar eureka-0.0.1-SNAPSHOT.jar
--> Running in fb079a557f77
--> c767bb50dcdb
Removing intermediate container fb079a557f77
Successfully built c767bb50dcdb
Successfully tagged conpatrones_eureka:latest
WARNING: Image for service eureka was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-co
mpose up --build`.
Building svc-cursos
Step 1/3 : FROM java:8
--> d23bdf5b1b1b
Step 2/3 : ADD /target/svcCursos-0.0.1-SNAPSHOT.jar svcCursos-0.0.1-SNAPSHOT.jar
--> b64347e9bab5
Removing intermediate container 53f47d327f66
Step 3/3 : ENTRYPOINT java -Dspring.profiles.active=container -jar svcCursos-0.0.1-SNAPSHOT.jar
--> Running in 3f8f81806721
--> 51b4a68b2621
Removing intermediate container 3f8f81806721
Successfully built 51b4a68b2621
Successfully tagged conpatrones_svc-cursos:latest
WARNING: Image for service svc-cursos was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docke
r-compose up --build`.
Building svc-mat
Step 1/3 : FROM java:8
--> d23bdf5b1b1b
Step 2/3 : ADD /target/svcMatriculas.jar svcMatriculas.jar
--> 4ea31455306d
Removing intermediate container c3ec0efa8754
Step 3/3 : ENTRYPOINT java -Dspring.profiles.active=container -jar svcMatriculas.jar
--> Running in 1983daa3fad5
```

Figura 62 Levantar contenedores con Docker Compose

Fuente: La Autora

Elaboración: La Autora

## Anexo N: Ver contenedores activos en Docker



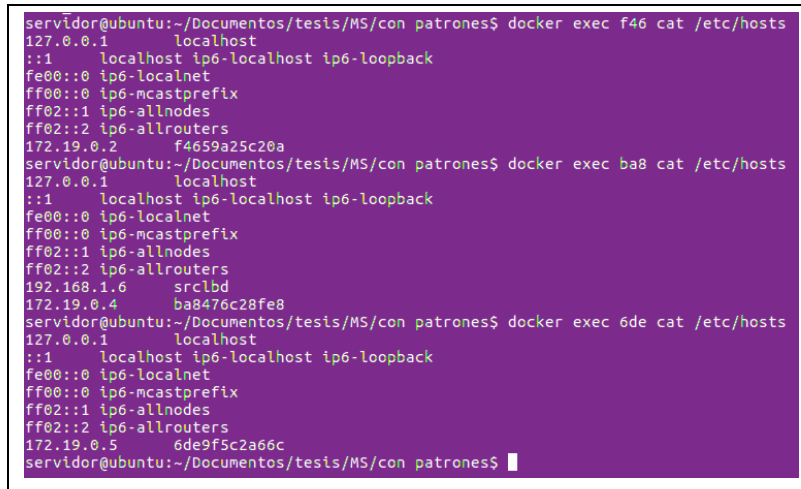
```
servidor@ubuntu: ~/Documentos/tesis/MS/con patrones$ docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                               NAMES
6de9f5c2a66c   conpatrones_zuul                    "java -Dspring.pro..." 4 minutes ago  Up 4 minutes  0.0.0.0:8081->8081/tcp              conpatrones_zu
ul_1
ba8476c28fe8   conpatrones_svc-nat                 "java -Dspring.pro..." 4 minutes ago  Up 4 minutes  0.0.0.0:1111->1111/tcp              conpatrones_sv
c-mat_1
415f3d78b281   conpatrones_svc-cursos              "java -Dspring.pro..." 4 minutes ago  Up 4 minutes  0.0.0.0:2222->2222/tcp              conpatrones_sv
c-cursos_1
f4659a25c20a   conpatrones_eureka                  "java -Dspring.pro..." 4 minutes ago  Up 4 minutes  0.0.0.0:8761->8761/tcp              conpatrones_eu
reka_1
```

Figura 63 Ver contenedores activos en Docker

Fuente: La Autora

Elaboración: La Autora

## Anexo O: Ver hosts de contenedores Docker



```
servidor@ubuntu:~/Documentos/tesis/MS/con patrones$ docker exec f46 cat /etc/hosts
127.0.0.1    localhost
::1         localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
172.19.0.2   f4659a25c20a
servidor@ubuntu:~/Documentos/tesis/MS/con patrones$ docker exec ba8 cat /etc/hosts
127.0.0.1    localhost
::1         localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
192.168.1.6  srclbd
172.19.0.4   ba8476c28fe8
servidor@ubuntu:~/Documentos/tesis/MS/con patrones$ docker exec 6de cat /etc/hosts
127.0.0.1    localhost
::1         localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
172.19.0.5   6de9f5c2a66c
servidor@ubuntu:~/Documentos/tesis/MS/con patrones$
```

Figura 64 Ver hosts de contenedores Docker

Fuente: La Autora

Elaboración: La Autora

## Anexo P: Despliegue de Eureka en contenedor Docker

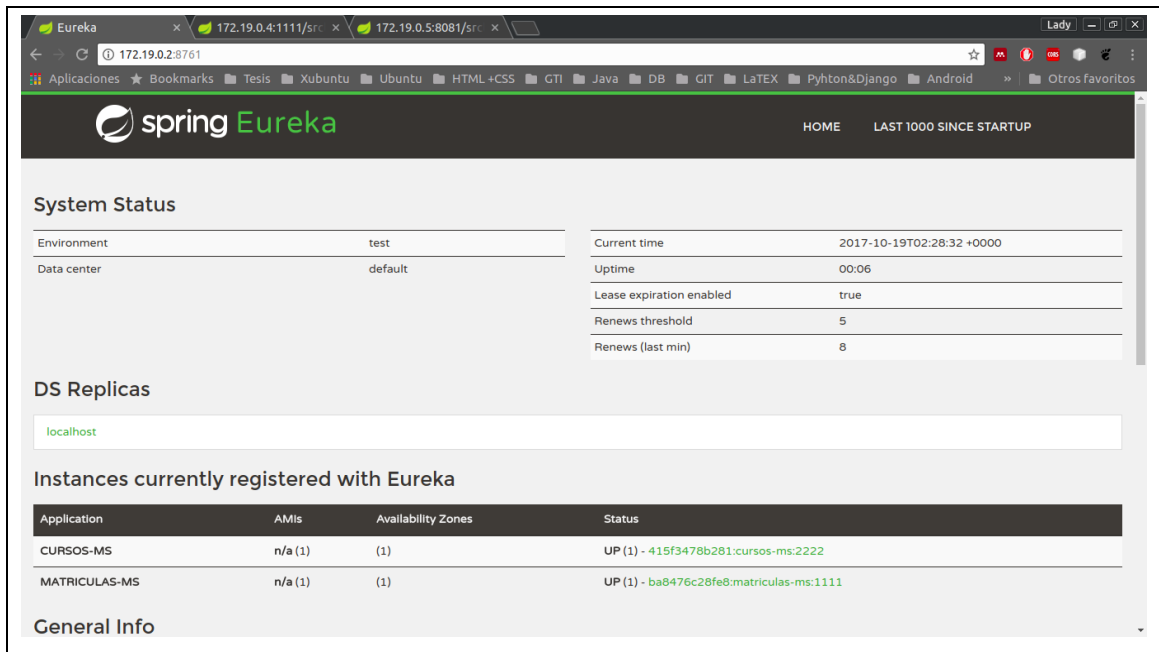


Figura 65 Despliegue de Eureka en contenedor Docker

Fuente: La Autora

Elaboración: La Autora

## Anexo Q: Despliegue de API REST Matrículas en contenedor Docker

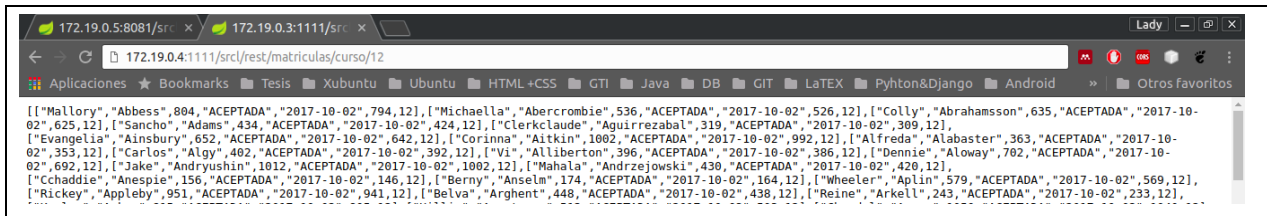


Figura 66 Despliegue de API REST Matrículas en contenedor Docker

Fuente: La Autora

Elaboración: La Autora

## Anexo R: Despliegue de Zuul en contenedor Docker

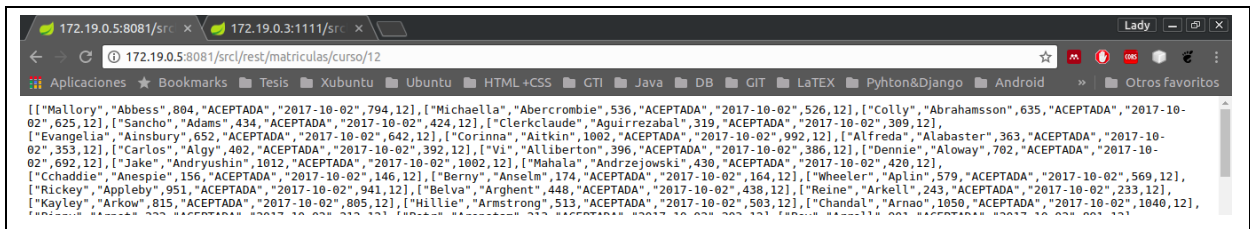


Figura 67 Despliegue de Zuul en contenedor Docker

Fuente: La Autora

Elaboración: La Autora

## Anexo S: Instalación de JMeter

1. Descargar desde la página oficial de JMeter ([https://jmeter.apache.org/download\\_jmeter.cgi](https://jmeter.apache.org/download_jmeter.cgi)) el archivo `apache-jmeter-3.3.tgz`
2. Descomprimir el archivo y en un Terminal Linux, dirigirse al directorio de JMeter: `apache-jmeter-3.3/bin`
3. Para abrir JMeter, ejecutar:  
`./ jmeter`



## Anexo T: Definición de Términos de Métricas de Rendimiento en base a hardware

Tabla 33 Definición de Términos de Métricas de Rendimiento en base a Hardware

<b>Métrica</b>		<b>Definición</b>	
<b>Plataforma</b>		S.O en el que se realizó la implementación, en este caso: Windows y Linux.	
<b>Método de Despliegue</b>	<b>Despliegue Nativo</b>	Indica el método en que fue desplegada la aplicación de prueba. En este caso existen dos valores disponibles: Despliegue Nativo o Contenedor (Docker).	
	<b>Contenedor</b>		
<b>CPU</b>	<b>Uso de CPU (%)</b>	Uso de CPU registrado durante la prueba especificado en porcentaje.	
<b>Memoria</b>	<b>Memoria usada (mb)</b>	Cantidad de memoria usada durante la ejecución de la prueba. Se expresa en megabytes.	
	<b>Memoria Libre (mb)</b>	Cantidad de memoria libre registrada durante la ejecución de la prueba. Se expresa en megabytes.	
	<b>Memoria Total (mb)</b>	Cantidad de memoria total actual. Se expresa en megabytes.	
<b>Red</b>	<b>Cableada</b>	<b>Velocidad de conexión (kbps)</b>	Velocidad de conexión registrada durante la ejecución de la prueba expresada en kilobytes por segundo.
		<b>Bytes enviados/sec (kb/s)</b>	Cantidad de bytes enviados durante la ejecución de la prueba expresada en kilobytes por segundo.
		<b>Bytes recibidos/sec (kb/s)</b>	Cantidad de bytes recibidos durante la ejecución de la prueba expresada en kilobytes por segundo.
		<b>Transacciones/s (Troughput) (x/s)</b>	Número de transacciones ejecutadas por segundo.

	<b>Wireless</b>	<b>Velocidad de conexión (kbps)</b>	Los mismos parámetros especificados para red Cableada.
		<b>Bytes enviados/sec (kb/s)</b>	
		<b>Bytes recibidos/sec (kb/s)</b>	
		<b>Transacciones/s (Troughput) (x/s)</b>	
<b>Base de Datos</b>	<b>Exec Avg (ms)</b>		Tiempo promedio en procesar la prueba establecida, se expresa en milisegundos.
	<b>Exec Min (ms)</b>		Tiempo mínimo en procesar la prueba establecida, se expresa en milisegundos.
	<b>Exec Max (ms)</b>		Tiempo máximo en procesar la prueba establecida, se expresa en milisegundos.
	<b>Tiempo de respuesta (ms)</b>		Tiempo de respuesta que le toma al servidor procesar la prueba establecida. Se expresa en milisegundos.
	<b>Ejecuciones fallidas (%)</b>		Número de peticiones fallidas durante la ejecución de la prueba, expresado en porcentaje.

Fuente: La Autora

Elaboración: La Autora

## Anexo U: Pruebas de rendimiento. Escenario: 10 Clientes

Tabla 34 Pruebas de Rendimiento con 10 Clientes

		Monolito Versión 1			Monolito Versión 2			Servicios REST (Con patrones de diseño)			Servicios REST (Sin patrones de diseño)			Microservicios (MS) (Con patrones de diseño)			Microservicios (MS) (Sin patrones de diseño)			
Plataforma		Windows			Linux			Linux			Linux			Linux			Linux			
Método de Despliegue	Despliegue Nativo	X			X			X			X			X			X			
	Contenedor													X			X			
# Registros		100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000	
CPU	Uso de CPU (%)	13,2%	17,2%	98,1%	11,5%	17,0%	97,3%	37,0%	20,0%	47,0%	31,0%	13,0%	33,0%	18,0%	18,0%	30,0%	19,0%	19,0%	28,0%	
Memoria	Memoria usada (mb)	5754,9	5836,8	4300,8	4382,7	4024,3	7321,6	3819,5	3881,0	4134,9	4167,7	4167,7	4249,6	5673,0	6133,8	6201,3	5826,6	5847,0	5857,3	
	Memoria Libre (mb)	2085,1	2003,2	3539,2	3457,3	3815,7	518,4	4020,5	3959,0	3705,1	3672,3	3672,3	3590,4	2167,0	1706,2	1638,7	2013,4	1993,0	1982,7	
	Memoria Total (mb)	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0
Red	Cableada	Velocidad de conexión (kbps)	803,3	77,5	908,5	52,2	63,7	320,5	57,3	54,1	65,2	44,1	62,0	64,1	65,3	54,9	54,1	34,1	54,1	54,1
		Bytes enviados/s (kb/s)	1,8	0,0	0,6	1,7	1,7	1,2	1,5	1,5	0,0	1,5	0,0	1,3	1,5	0,0	1,3	1,5	1,5	1,3
		Bytes recibidos/s (kb/s)	70,3	10,7	2140,1	429,3	3731,0	25855,5	197,2	1898,1	289,2	197,6	41,1	16948,4	60,0	7,5	5340,9	60,0	598,4	5609,5
		Trans/sec (x/s)	10,6	10,5	3,5	10,9	10,4	7,3	11,0	10,6	9,6	11,0	13,7	9,4	10,9	14,7	9,4	10,9	10,8	9,9
	Wireless	Velocidad de conexión (kbps)	114,0	357,4	488,1	200,0	54,5	55,1	54,5	56,8	54,4	58,9	56,4	54,1	54,9	54,1	54,7	54,9	54,1	54,1
		Bytes enviados/s (kb/s)	1,8	1,8	0,6	1,7	1,7	1,1	1,5	1,5	0,9	1,5	1,5	1,3	1,4	1,4	1,3	1,4	1,5	1,3

		<b>Bytes recibidos/s (kb/s)</b>	70,2	632,3	2052,6	429,8	3766,4	24810,2	197,4	1908,2	11060,5	196,8	1930,8	16262,5	59,3	593,8	5391,6	58,4	593,4	5506,2
		<b>Trans/s (x/s)</b>	10,6	10,4	3,4	10,9	10,5	7,0	11,0	10,6	6,1	10,9	10,8	9,0	10,8	10,7	9,5	10,7	10,7	9,7
<b>Base de Datos</b>	<b>Exec Avg (ms)</b>		45,0	64,0	1928,0	39,0	50,0	7519,0	12,0	32,0	335,0	8,0	30,0	250,0	16,0	29,0	157,0	11,0	31,0	155,0
	<b>Exec Min (ms)</b>		39,0	47,0	1395,0	26,0	32,0	687,0	10,0	28,0	180,0	4,0	19,0	158,0	13,0	22,0	129,0	8,0	19,0	123,0
	<b>Exec Max (ms)</b>		56,0	93,0	2257,0	66,0	63,0	13767,0	17,0	39,0	775,0	13,0	43,0	353,0	22,0	44,0	195,0	14,0	52,0	226,0
	<b>Tiempo de respuesta (ms)</b>		49,0	80,0	2057,0	56,0	60,0	12484,0	14,0	34,0	734,0	12,0	41,0	347,0	19,0	35,0	191,0	14,0	39,0	175,0
	<b>Ejecuciones fallidas (%)</b>		0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0

Fuente: La Autora

Elaboración: La Autora

## Anexo W: Pruebas de rendimiento. Escenario: 100 Clientes

Tabla 35 Pruebas de Rendimiento con 100 Clientes

			Monolito Versión 1			Monolito Versión 2			Servicios REST (Con patrones de diseño)			Servicios REST (Sin patrones de diseño)			Microservicios (MS) (Con patrones de diseño)			Microservicios (MS) (Sin patrones de diseño)		
Plataforma			Windows			Linux			Linux			Linux			Linux			Linux		
Método de Despliegue	Despliegue Nativo		X			X			X			X			X			X		
	Contenedor														X			X		
# Registros			100	1.000	10.000	100	1.000	10.000	100	1.000	10.000	100	1.000	10.000	100	1.000	10.000	100	1.000	10.000
CPU	Uso de CPU (%)		40,0%	98,7%	99,1%	37,1%	95,0%	95,0%	21,0%	50,0%	83,4%	30,0%	62,0%	98,5%	53,0%	57,5%	98,6%	39,0%	41,0%	99,3%
Memoria	Memoria usada (mb)		5734,4	5734,4	5632,0	4474,9	7362,6	7475,2	4198,4	4280,3	7290,9	4280,3	4331,5	7219,2	6758,4	7516,2	7628,8	5898,2	5908,5	7587,8
	Memoria Libre (mb)		2105,6	2105,6	2208,0	3365,1	477,4	364,8	3641,6	3559,7	549,1	3559,7	3508,5	620,8	1081,6	323,8	211,2	1941,8	1931,5	252,2
	Memoria Total (mb)		7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0
Red	Cableada	Velocidad de conexión (kbps)	197,2	418,3	519,0	33,7	33,6	43,7	45,7	47,6	44,1	54,1	64,7	34,1	74,1	74,1	55,4	70,1	56,0	21,3
		Bytes enviados/s (kb/s)	16,3	7,3	0,5	15,5	10,3	1,2	13,3	9,5	0,1	13,5	13,5	1,2	2,3	2,4	2,0	13,0	1,2	0,3
		Bytes recibidos/s (kb/s)	649,5	2623,3	1841,8	3859,6	23074,9	26027,7	1742,9	12358,0	1156,6	1765,1	17416,3	15703,3	89,0	950,6	90,6	530,2	510,3	643,5
		Trans/s (x/s)	97,8	43,2	3,0	98,0	64,6	7,3	99,8	68,8	2,5	97,8	97,0	7,4	17,5	17,7	15,0	97,3	9,2	2,1
	Wireless	Velocidad de conexión (kbps)	478,0	54,6	67,0	33,1	27,9	54,3	64,8	55,3	45,9	85,3	39,3	55,2	65,0	54,3	10,8	54,1	54,1	44,1
		Bytes enviados/s (kb/s)	16,2	7,0	0,5	14,8	10,8	1,1	13,3	0,2	0,3	13,3	13,4	0,1	11,8	2,3	2,4	13,2	13,0	0,4

		Bytes recibidos/s (kb/s)	646,4	2524,0	1873,9	3675,8	24216,2	25263,1	1732,8	130,5	1741,7	1742,9	17282,2	1062,7	486,2	734,5	1434,1	535,9	5317,5	2,9
		Trans/s (x/s)	97,3	41,6	3,1	93,4	67,8	7,1	96,2	1,3	1,9	96,8	96,2	2,6	88,4	16,8	18,0	98,3	96,1	2,9
Base de Datos	Exec Avg (ms)		42	855	26913	10	232	6377	5	83	3829	4	37	28348	16	1229	3364	6	42	23521
	Exec Min (ms)		18	109	7054	7	62	845	4	4	4	4	18	3339	7	33	451	5	18	357
	Exec Max (ms)		178	1809	32434	31	430	12311	16	314	18246	13	82	37618	51	5131	5072	12	142	34812
	Tiempo de respuesta (ms)		82	1500	32016	14	386	9193	7	252	12997	6	54	32194	29	5044	5024	7	60	34388
	Ejecuciones fallidas (%)		0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	26,5%	0,0%	0,0%	0,0%	0,0%	0,0%

Fuente: La Autora

Elaboración: La Autora

## Anexo X: Pruebas de rendimiento. Escenario: 1.000 Clientes

Tabla 36 Pruebas de Rendimiento con 1000 Clientes

		Monolito Versión 1			Monolito Versión 2			Servicios REST (Con patrones de diseño)			Servicios REST (Sin patrones de diseño)			Microservicios (MS) (Con patrones de diseño)			Microservicios (MS) (Sin patrones de diseño)			
Plataforma		Windows			Linux			Linux			Linux			Linux			Linux			
Método de Despliegue	Despliegue Nativo	X			X			X			X			X			X			
	Contenedor													X			X			
# Registros		100	1.000	10.000	100	1.000	10.000	100	1.000	10.000	100	1.000	10.000	100	1.000	10.000	100	1.000	10.000	
CPU	Uso de CPU (%)	98,4%	99,0%	98,1%	97,4%	95,9%	96,8%	97,7%	97,3%	98,5%	97,3%	97,3%	97,3%	97,8%	96,3%	98,6%	96,1%	96,3%	97,3%	
Memoria	Memoria usada (mb)	3553,3	3512,3	4679,7	7403,5	7516,2	7362,6	6553,6	6860,8	6420,5	6953,0	7198,7	7434,2	7639,0	7669,8	7710,7	7710,7	7710,7	7813,1	
	Memoria Libre (mb)	4286,7	4327,7	3160,3	436,5	323,8	477,4	1286,4	979,2	1419,5	887,0	641,3	405,8	201,0	170,2	129,3	129,3	129,3	26,9	
	Memoria Total (mb)	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	7840,0	
Red	Cableada	Velocidad de conexión (kbps)	574,0	826,0	241,0	52,1	54,5	54,5	53,9	52,1	522,0	24,5	28,8	31,3	13,7	45,7	57,4	352,0	368,0	62,9
		Bytes enviados/s (kb/s)	18,4	2,2	0,0	26,8	10,3	0,2	19,9	3,2	0,7	41,7	4,4	0,2	26,0	14,5	9,7	27,6	14,8	0,1
		Bytes recibidos/s (kb/s)	816,1	793,7	212,7	6673,7	23097,3	4826,3	2602,9	4091,3	9440,5	5450,8	5712,2	2569,1	847,5	1148,6	66,3	1125,4	6080,4	476,8
		Trans/s (x/s)	150,2	21,6	102,2	169,5	64,6	12,6	144,6	29,8	27,9	302,8	31,8	14,3	194,3	107,3	72,0	206,5	109,8	51,6
	Wireless	Velocidad de conexión (kbps)	827,0	156,0	592,0	29,1	93,3	8,9	24,9	91,6	60,9	180,0	183,0	658,0	79,5	107,0	90,4	69,6	52,4	62,7
		Bytes enviados/s (kb/s)	14,8	2,2	0,0	29,8	0,9	0,1	33,5	7,2	0,1	47,7	13,3	0,2	8,7	13,6	7,8	2,6	4,7	0,2
		Bytes recibidos/s (kb/s)	688,3	804,4	254,7	7425,2	2057,3	708,4	4379,5	9343,4	1460,4	6231,9	17258,9	1855,4	123,5	1225,3	544,3	105,2	1919,0	249,3

	Trans/s (x/s)	136,9	24,0	122,4	188,6	5,8	48,0	243,2	52,0	17,8	346,1	96,1	9,1	64,9	101,0	57,7	19,3	34,6	1,2
<b>Base de Datos</b>	<b>Exec Avg (ms)</b>	3252	160,3 2	4177	1107	7150	45514	2768	14968	27950	532	4596	49934	1431	3776	6230	802	4011	53727
	<b>Exec Min (ms)</b>	69	1914	3107	15	46	2814	232	30	2730	5	41	24384	7	108	61	16	65	1279
	<b>Exec Max (ms)</b>	5677	38843	5498	2037	13515	73541	3771	32474	33617	913	7898	68932	4887	7481	13374	1861	7542	59420
	<b>Tiempo de respuesta (ms)</b>	5294	31942	1	1817	12478	11143	3293	30658	31007	860	7699	61436	2621	6696	10749	1206	6745	8000
	<b>Ejecuciones fallidas (%)</b>	26,5%	40,0%	100,0%	0,0%	0,0%	89,3%	0,0%	29,8%	81,4%	0,0%	0,0%	90,2%	0,0%	22,1%	79,2%	0,0%	0,0%	95,0%

Fuente: La Autora

Elaboración: La Autora