UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

La Universidad Católica de Loja

ÁREA TÉCNICA

TÍTULO DE INGENIERO EN SISTEMAS INFORMÁTICOS Y

COMPUTACIÓN

Identificación y aplicación de un modelo para evaluar la interoperabilidad en Microservicios.

TRABAJO DE TITULACIÓN

AUTOR: Cueva Tacuri, Jose Luis

DIRECTOR: Guamán Coronel, Daniel Alejandro, Ing.

LOJA – ECUADOR

2017

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: Identificación y aplicación de un modelo para evaluar la interoperabilidad en Microservicios, realizado por José Luis Cueva Tacuri, ha sido orientado y revisado durante su ejecución, por cuanto se aprueba la presentación del mismo.

Loja, septiembre de 2017

f). ……

DECLARACIÓN DE AUTORÍA Y CESIÓN DE DERECHOS

“Yo, Jose Luis Cueva Tacuri, declaro ser autor (a) del presente trabajo de titulación: “Identificación y aplicación de un modelo para evaluar la interoperabilidad en Microservicios”, de la Titulación de Sistemas Informáticos y Computación, siendo el Ing. 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).……...……..

Autor: Jose Luis Cueva Tacuri

Cédula. 1106085697

DEDICATORIA

Dedico este trabajo a mis amados padres Cumar y Josefa por haberme brindado su apoyo incondicional en todo momento, ya que gracias a ellos he podido culminar mis estudios profesionales, demostrándome que siempre debemos ser fuertes ante las dificultades de la vida, y que cada problema es una nueva oportunidad para superarnos.

A mis hermanos, que son una de mis motivaciones para mi superación personal y profesional.

A mis compañeros de estudio y amigos que han sido parte de este constante aprendizaje.

A mis maestros que me guiaron y me impartieron sus valiosos conocimientos.

Jose Luis Cueva Tacuri.

AGRADECIMIENTO

Primeramente a Dios por darme la vida, la salud y la sabiduría.

A mis padres por el apoyo y sus consejos que me han permitido crecer espiritual y humanamente.

A la Universidad Técnica Particular de Loja por darme la oportunidad de prepararme tanto profesionalmente como humanísticamente.

A mi director de tesis por la paciencia y el apoyo que durante todo este tiempo tuvo en el desarrollo de tesis.

A mi comité de tribunal por sus correcciones y aportaciones en el presente trabajo.

A todos ellos expreso mi más sincero agradecimiento y estima.

Jose Luis Cueva Tacuri.

INDICE DE CONTENIDOS

INDICE DE TABLAS

INDICE DE FIGURAS

RESUMEN

El presente trabajo de fin de titulación presenta un estudio orientado a la identificación implementación de un modelo que permita evaluar la interoperabilidad en una arquitectura de Microservicios. Se realiza este estudio desde el punto de vista investigativo en consecuencia de que es una alternativa a la arquitectura tradicional Monolítica, y desde el punto de vista de aplicación con el fin de comparar los resultados de ambas en cuanto al cumplimiento del atributo de calidad Interoperabilidad a través de una implementación básica.

Para justificar este estudio se realiza el diseño e implementación de un prototipo de aplicaciones web basada en estas dos arquitecturas, donde se busca exponer información y al mismo tiempo identificar la interoperabilidad de los aplicativos. La evaluación de la interoperabilidad como factor primordial de este análisis se enfoca en el análisis de una combinación de modelos que permitan evaluar adecuadamente la interoperabilidad existente en los aplicativos, desde la definición arquitectónica, escritura del código fuente, configuraciones a nivel del servidor e implementación de las aplicaciones web.

PALABRAS CLAVE: Aplicaciones web, Arquitectura, contenedores, Docker, Interoperabilidad, Microservicios, RESTful.

ABSTRACT

The present work of end of degree presents a study focused on the implementation of a model of evaluation of interoperability in Microservices architecture. This study is conducted as a result that is an alternative to the traditional monolithic architecture, in order to compare results both in terms of the interoperability quality attribute.

To justify this study presents the design and implementation of a prototype of web applications based on these two architectures, which seeks to expose information and at the same time identify the interoperability of applications. Assessment of interoperability as primary factor in this analysis focuses on the identification of a combination of models that adequately identify interoperability in applications, from the architectural definition, writing of the code source, server configuration and deployment of web applications.

KEYWORDS: Web application, Microservices, Interoperability, architecture, RESTful, Docker, containers.

INTRODUCCION

Hoy en día, existen diferentes enfoques en cuanto al uso de arquitecturas para la construcción y evolución de los sistemas con la finalidad de que éstos sean escalables, disponibles y puedan ejecutarse en diferentes entornos tales como web, móvil o en modo cliente, permitiendo a las organizaciones implementar frecuentemente actualizaciones logrando así la evolución de los sistemas de software. La mayoría de las aplicaciones de software con la que cuentan las organizaciones, sean estas cliente, web o móviles, pueden ser desarrolladas siguiendo una arquitectura monolítica como: cliente servidor, 2 capas, n capas u otras que combinadas con patrones de diseño y utilizando diferentes tecnologías de implementación hace que la comunicación resulte compleja; además en algunas ocasiones estas las aplicaciones no poseen características importantes como tolerancia a fallos, acoplamiento, independencia, o atributos requeridos por la arquitectura de software como patrones arquitectónicos y de diseño que son necesarios utilizarlos acorde al uso e implementación de la solución de software.

Como parte de la evolución arquitectónica, los Microservicios (MS) que según definición de Gadea & Ionescu (2016) son un estilo arquitectónico que está ganando cada vez más popularidad, en el ámbito académico tienen como objetivo desarrollar una sola aplicación como un conjunto de servicios pequeños, cada uno ejecutándose en su propio procesador y con mecanismos de comunicación de peso ligero, es decir de rápida implementación como REST, JSON u otros; este conjunto de servicios están construidos alrededor de las capacidades de negocio y con independencia de despliegue debido a una implementación totalmente automatizada.

La característica principal detrás de MS según Killalea (2016) es la separación clara de funciones, la cual centra la atención de cada servicio en algunos aspectos bien definidos de la aplicación en general, además pueden estar compuestos con la articulación flexible entre los servicios, y pueden ser desplegados de forma independiente; brindando una clara separación de las capas, que necesitan mínimo acoplamiento, con potencial para una mayor tasa de intercambio lo que conduce a una mayor agilidad en los negocios y la velocidad de ingeniería.

Entre los atributos de calidad que forman parte de una arquitectura de MS constan: la modularidad, cuya ventaja es permitir realizar correcciones, mejoras, mantenimientos y actualizaciones de forma más eficiente que la arquitectura general; otro de los atributos es la interoperabilidad, el cual permite que el intercambio y la reutilización de la información sea más fácil, tanto interna como externamente, adicionalmente los

protocolos de comunicación, interfaces y formatos de datos son las consideraciones clave para la interoperabilidad.

El presente trabajo de titulación tiene como finalidad investigar, analizar y reportar información relacionada a MS y su relación con Interoperabilidad, para ello se diseñará un prototipo, que permita utilizando los modelos resultantes de la investigación validar la interoperabilidad en microservicios. Como objetivos específicos propuestos para el presente trabajo constan: analizar las características, ventajas y desventajas en arquitecturas de microservicios (atraves de la clasificación previa de la Bibliografía ver **Anexo A**: Clasificación de Bibliografía); identificar modelos de referencia para evaluar interoperabilidad en microservicios; valorar distintas tecnologías que se pueden utilizar para implantar una arquitectura basada en microservicios; y finalmente analizar la interoperabilidad a nivel de microservicios.

GLOSARIO DE TÉRMINOS

API.- Application Programming Interface, es un conjunto de reglas, procedimientos y especificaciones que cumplen una o muchas funciones, con el fin de ser utilizadas por otro software

DevOps.- es una relación más colaborativa y productiva entre los equipos de desarrollo y los equipos de operaciones, al momento del desarrollo de una aplicación.

Polyglott.- Múltiples tecnologías de almacenamiento de datos.

Atributos de Calidad: es un conjunto de métricas que permiten evaluar la calidad de los sistemas de software.

IDE.- Ambiente de desarrollo integrado que facilita la programación de software.

JSON.- Formato de texto ligero para intercambio de datos.

XML.- Leguaje de marcado para intercambio de datos.

RESTful: corresponde a la denominación de aplicaciones de software que adoptan el estilo arquitectónico REST.

Servidor.- Lugar donde se alojan las aplicaciones de software para ser consumidas por el usuario.

Glassfish.- Tipo de servidor de aplicaciones Java, permite ejecutar aplicaciones web.

Framework.- Marco de trabajo que permite construir software de forma estructurada.

PHP.- Lenguaje de programación que permite la creación de aplicaciones web de contenido dinámico.

Node.- Es un entorno de ejecución que permite la creación de aplicaciones web.

Base de datos: es un repositorio de información donde se reúne información categorizada y ordenada.

HTTP: Protocolo de transferencia de hipertexto, es un protocolo aplicado para realizar transacciones de comunicación dentro de la WWW.

SQL: Lenguaje de consulta estructurado, es un método de acceso a base de datos que permite ejecutar operaciones de consulta, inserción, actualización y eliminación de datos.

API: Interfaz de programación de aplicaciones, es un conjunto de reglas, métodos y funciones que es utilizado por un software.

Docker.- Aplicación que permite la implementación de aplicaciones sin importar el hardware.

Contenedor.- Implementación de una imagen que sirve para ejecutar aplicaciones en Docker.

CAPITULO I
MARCO TEÓRICO

En este capítulo se exponen los conceptos básicos relacionados con una arquitectura de Microservicios, así como los métodos que se utilizan para la implementación y los modelos que permiten evaluar la interoperabilidad en las aplicaciones.

Por lo tanto, el primer paso para trabajar con Microservicios es partir de una arquitectura monolítica, la misma que se puede representar como una aplicación que contiene todos sus componentes relacionados, cada uno realizando funcionalidades diferentes lo que implica que si uno deja de funcionar, la aplicación deja de funcionar por completo, (ver **Figura 1**) es decir, que posee todas las funcionalidades en la misma aplicación y al momento de realizar una réplica, se la debe hacer de forma completa.

El término Microservicios se puede explicar tomando como referencia la arquitectura monolítica; Microservicios se refiere a un término que está ganando popularidad en la actualidad, el cual se refiere a un enfoque orientado al desarrollo de aplicaciones pequeñas, que colaboran juntos a través de peticiones HTTP a sus APIs.

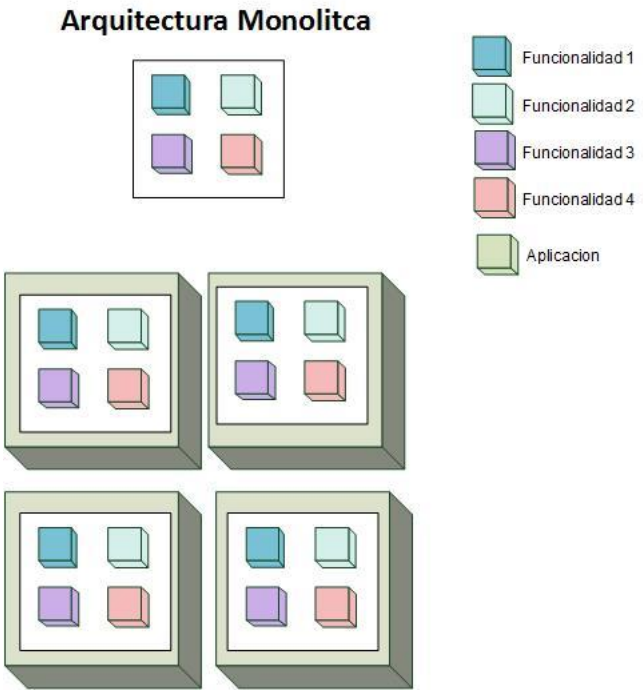


Figura 1: Arquitectura Monolítica
Fuente: El autor
Elaboracion: El autor

1.1. Microservicios.

Es preciso definir el término Microservicios, el cual, ha surgido como un prometedor estilo arquitectónico que se utiliza para la construcción de aplicaciones cuyo principal objetivo es dividir una aplicación en pequeños componentes. A continuación se exponen algunas definiciones dadas por autores.

1.1.1. Definiciones.

Los microservicios (MS), según (Knoche, 2016), se centran explícitamente en la estructura interna de una aplicación y los enfoques tradicionales orientados a servicios, cuyo objetivo es centrarse en el negocio y los servicios granulares. Este enfoque hace de ellos una opción prometedora para la modernización de sistemas de software monolíticos para dividirlos en un conjunto de servicios que interactúan entre sí.

Killalea (2016), menciona que los MS son una aproximación a la construcción de sistemas distribuidos en los cuales los servicios están expuestos solo a través de API's; los propios servicios tienen un alto grado de cohesión interna en torno a un contexto especifico y bien delimitado y el acoplamiento entre ellos está suelto, es decir estos servicios suelen ser simples, pero pueden estar compuestos y formar parte de aplicaciones más complejas. Gadea & Ionescu (2016) mencionan que los microservicios son un nuevo tipo de arquitectura de software, que componen una aplicación como un conjunto de servicios individuales e independientes, en el cual cada servicio está dedicado a una sola capacidad de negocio; estos servicios pueden ser implementados utilizando diversas tecnologías; permitiendo implementaciones flexibles, con tiempos de desarrollo más cortos.

Connor (2016) agrega que MS "es un estilo de desarrollo en el que un sistema se divide en una serie de pequeños componentes cooperantes. Normalmente, estos componentes interactúan a través de una interfaz directa de punto a punto, por ejemplo, http."

En la **Figura 2** se representa una arquitectura de microservicios en la que se representa cada funcionalidad como un servidor, lo que al momento de replicar resulta más productivo**,** ya que solo se replican las funcionalidades que se necesiten y sean necesarios para una aplicación concreta, más no toda la aplicación.

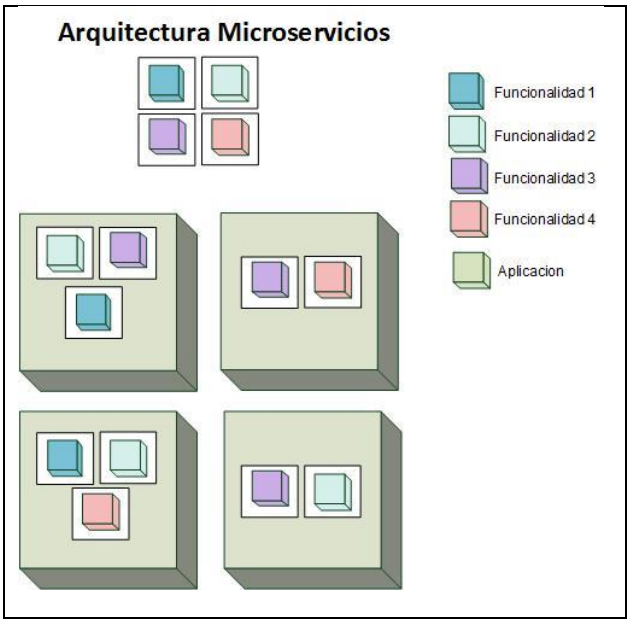


Figura 2: Arquitectura de Microservicios
Fuente: El autor
Elaboración: El autor

Para explicar de forma gráfica la arquitectura monolítica, en la **Figura 3** se representa un ejemplo de la misma, la cual posee como punto de acceso entre cliente y aplicación un navegador que permite acceder a la aplicación, donde luego las operaciones pasan primeramente por un balanceador de carga, un balanceador de carga es un hardware o software que se encarga de asignar o 'balancear' las solicitudes que llegan de los clientes al servidor; además las 4 funcionalidades de la aplicación se encuentran en el mismo servidor y se tiene una única línea de comunicación con la base datos; al estar las 4 funcionalidades en el mismo servidor, dependen la una de la otra y si una falla toda la aplicación deja de funcionar.

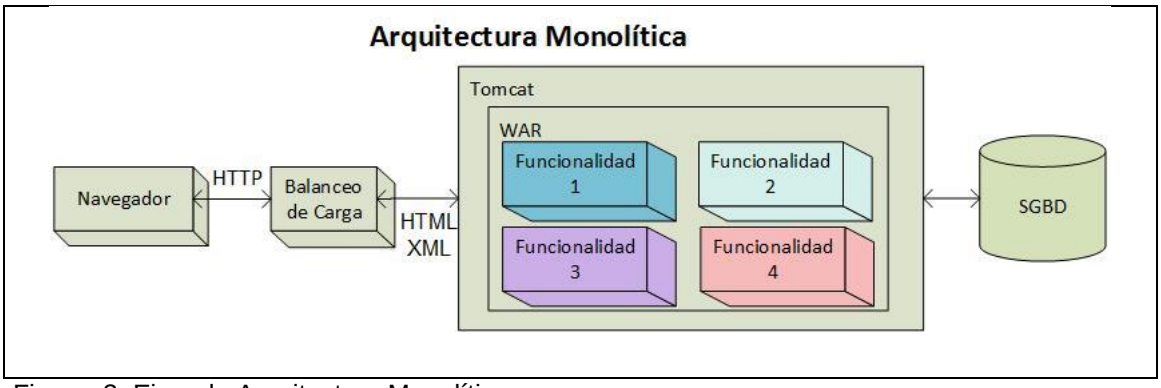


Figura 3: Ejemplo Arquitectura Monolítica.
Fuente: El autor
Elaboración: El autor

Por otra parte una arquitectura de Microservicios la cual se expone en la **Figura 4,** posee un navegador que permite el acceso a la aplicación a través de un API

Gateway; un API Gateway es el único punto de entrada para todos los clientes, encargado de gestionar las solicitudes de los mismos, enviando estas solicitudes al servicio apropiado; la cual permite el acceso a las 4 funcionalidades del MS; cada funcionalidad es independiente de las demás, ya que cada una se encuentra en un servidor, además posee comunicación independiente con la base de datos. Al estar las 4 aplicaciones en diferentes servidores, y por lo tanto ser independientes, al momento en que una deje de funcionar, la aplicación sigue funcional, sin ningún inconveniente.

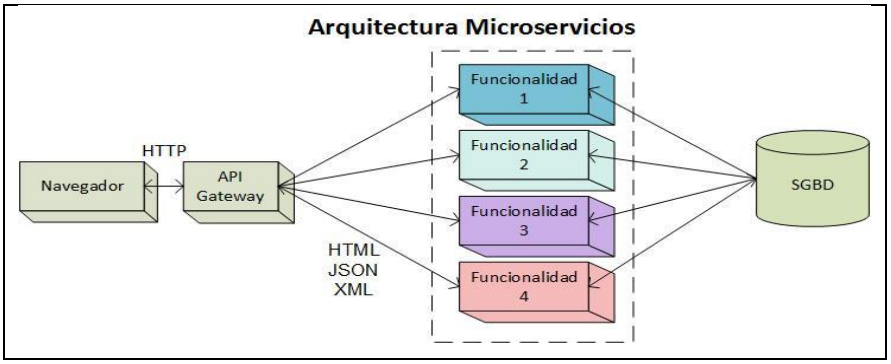


Figura 4: Ejemplo Arquitectura Microservicios
Fuente: El autor
Elaboración: El autor

1.1.2. Características de los Microservicios

Tomando como referencia las **Figura 2** y **Figura 4** se puede exponer algunas características que poseen los microservicios entre las que destacan:

- *Tolerancia a fallos:* La arquitectura de Microservicios posee mecanismos de tolerancia a fallos, lo cual permite que los fallos que pueden afectar a un microservicio no afecten al resto gracias al aislamiento de contenedores, ya que los servicios pueden fallar en cualquier momento, permitiendo detectar los fallos de forma rápida y, si es posible, restaurar automáticamente los servicios, siendo esto un parámetro que indica que se está trabajando con aplicaciones aisladas y por ende se puede aplicar la interoperabilidad. (Hasselbring, 2016).

- *Componetizacion:* Hasselbring (2016) menciona que “el uso de microservicios como componentes permite que las aplicaciones tengan que ser diseñadas de manera que puedan tolerar el fracaso de los servicios individuales, por el motivo de que los servicios pueden fallar en cualquier momento, es importante ser capaz de detectar los fallos de forma rápida y, si es posible, restaurarlos automáticamente.” (p. 1).

- *Separacion clara de funciones:* Killalea (2016) afirma que la característica principal detrás de microservicios es la separación clara de funciones, la cual centra la atención de cada servicio en algunos aspectos bien definidos de la aplicación en general, además pueden estar compuestos con la articulación flexible entre los servicios, pudiendo ser desplegados de una manera independiente; brindando una clara separación de las capas, que necesitan mínimo acoplamiento, con potencial para una mayor tasa de intercambio lo que conduce a una mayor agilidad en los negocios y la velocidad de ingeniería. Cuando se realiza la separación clara de funciones, debe existir comunicación entre las mismas, esto con el fin de que funcione como un todo.

- *Tamaño y despliegue:* Hasselbring (2016) menciona que las arquitecturas de microservicios proporcionan pequeños servicios que pueden ser desplegados y escalados de forma independiente el uno del otro, además explica que ésta arquitectura hace énfasis en la coordinación de menos transacciones entre los servicios. Para realizar estas transacciones se debe tener en cuenta la forma en que se puede realizar las mismas, teniendo relación con la interoperabilidad.

- *Autonomía:* Permite a los propietarios de la arquitectura, tomar diferentes decisiones con respecto a los problemas de la construcción de sistemas en las áreas de persistencia, consistencia y concurrencia, brindando así a los propietarios de los servicios mayor autonomía, y puede dar lugar a una adopción más rápida de nuevas tecnologías. (Killalea, 2016).

- *Persistencia de los datos:* Respecto a la persistencia de los datos, los microservicios individuales pueden emplear la *persistencia* Polyglott, gracias a que las arquitecturas de microservicios son cloud native, otorgando como resultado una elasticidad automatizada y rápida. (Hasselbring, 2016). Esto es muy útil al momento de tener varias tecnologías, en este caso de base de datos; y la forma en que se accede a las mismas, puede variar entre una y otra.

- *Modularidad:* En microservicios, cada servicio se ejecuta en su propio proceso y ocupa sólo una pequeña cantidad de almacenamiento de datos y la lógica de negocio. Cada servicio puede utilizar la tecnología que mejor le sirve al caso de uso incluyendo lenguajes de programación o DBMS diferentes, además los microservicios son relativamente pequeños y por lo tanto más fáciles de entender especialmente para un desarrollador. (Renz, Hoffmann, Staubitz, & Meinel, 2016). Por lo tanto una aplicación desarrollada con Microservicios por

más grande que sea, está divida en pequeñas partes, según la funcionalidad y posee comunicación entre las mismas.

- *Independencia:* Como se mencionó anteriormente, los microservicios están construidos alrededor de las capacidades de negocio y toman una implementación de la pila completa de software para esa área de negocios; además los microservicios permiten que cada servicio gestione su propia base de datos, incluso con los sistemas de gestión de bases de datos diferentes (persistencia Polyglott con consistencia eventual, como se mencionó anteriormente); aparte de los datos, el código tampoco es compartido entre los microservicios para evitar dependencias; sólo se recomienda la reutilización de código de la arquitectura de software de código abierto, como sugerencia y para conseguir una granularidad adecuada, se propone una descomposición vertical a lo largo de servicios del negocio (Hasselbring, 2016). Esto permite tener una idea clara de la estructura que tomara la aplicación y las relaciones entre las funcionalidades (que son totalmente independientes unas de otras), además, esta descomposición especifica las formas de comunicación e intercambio de datos que se va a utilizar en la estructura.

- *Colaborativos*: Davies (2010) afirma que: “Los microservicios se pueden utilizar para compartir piezas de contenido basada en el contexto (opiniones, recomendaciones, imágenes, audio, entre otros), que se recaba de forma automática, por las capacidades del terminal, o de forma manual, por la entrada directa del usuario.” (p.2); lo colaborativo también se extiende a la codificación, monitoreo y utilización de los Microservicios.

- *Automatización y carga operativa:* Killalea (2016) indica que “a medida que más ingenieros de software han seguido este modelo (…), así como el desarrollo de microservicios, han conducido a una amplia adopción de una serie de prácticas que permiten una mayor automatización y una menor carga operativa. Entre ellas se encuentran implementación contínua, capacidad virtualizada o en contenedores, elasticidad automatizada.” (p.6).

- *Monitoreo:* Hasselbring (2016) expone que los microservicios ponen mucho énfasis en la supervisión en tiempo real de la aplicación, comprobando dos indicadores técnicos, por ejemplo, el número de solicitudes por segundo a la base de datosy las métricas de negocio relevantes tales como el número de solicitudes por minuto ; además, este monitorero puede proporcionar un

sistema de alerta temprana de algo que va mal encaminado. Estas aplicaciones no son aisladas y son utilizadas por otras aplicaciones, por lo que continuamente se las debe supervisar.

1.1.3. Ventajas / Desventajas

Como todo estilo arquitectónico, los Microservicios poseen puntos a favor y en contra, desde el punto de vista del negocio y técnico. Entre las *ventajas* se encuentran:

- **Punto de vista del Negocio:**

  - *Escalabilidad:* Killalea (2016) menciona que "los MS proporcionan un modelo eficaz para las organizaciones que escalan mucho, más allá de los límites del contacto personal" (p.5); es decir que es ideal cuando se necesita implementar aplicaciones en organizaciones que geográficamente están dispersas, permitiendo una comunicación fluida.

  - *Toma de decisiones:* Permite a la organización asumir enfoques muy diferentes a las expectativas de gobierno de TI, que este tiene de los diferentes servicios; es decir, se asegura que la carga más grave o pesada, se concentre en un número muy pequeño de los servicios, liberando al resto para tener una mayor tasa de innovación, sin cargas por tales preocupaciones. (Killalea, 2016). Es decir que permite a la gerencia de la organización gestionar los recursos de mejor manera y que no sean desperdiciados, y poder destinarlos a otras actividades.

- **Punto de vista Técnico:**

  - *Modularidad:* Connor (2016) expone que "...un sistema altamente modular y con diversos componentes es más fácil de mantener que una jerarquía de clases tradicional..." (p. 3). Como se indicaba anteriormente una de las características de los Microservicios es la modularidad, la cual tiene relación directa con la interoperabilidad.

  - *Retroalimentación:* A cada servicio (generalmente en aplicaciones de tamaño considerable) se le asigna un equipo de desarrolladores, y este equipo es completamente responsable del servicio (alcance de la funcionalidad, la arquitectura para la construcción de éste, así como utilizarla.), esto hace que los desarrolladores entren en contacto con el

software, y a la vez con el cliente, siendo el bucle de retroalimentación de los clientes esencial para mejorar la calidad del servicio. (Killalea, 2016). Para la interoperabilidad, también se debe designar un equipo específico, según la cantidad de sistemas implicados.

- *Despliegue Rápido:* Connor (2016) refiere como ventaja a la capacidad de desplegar rápidamente a los servicios a un sistema de producción, ya que los servicios son entidades independientes, por lo tanto, sólo el servicio bajo análisis tiene que someterse a pruebas rigurosas y el resto del sistema no se ve afectado. Es decir el ahorro de tiempo es significativo, ya que no se pone a prueba a todo el sistema, sino solo a la funcionalidad especificada.

- *Tecnologías variadas:* Una ventaja relevante con MS, es que los consumidores de los mismos no deben preocuparse por cómo los datos persisten detrás de un conjunto de API's de los que dependen, siendo posible cambiar un mecanismo de persistencia con otro sin que los consumidores sean notificados. (Killalea, 2016). En consecuencia, los consumidores no deben preocuparse por ese aspecto, ya que los Microservicios se encargan del mismo, esto además incluye los mecanismos para acceder o usar estas tecnologías.

- *Independencia:* Los Microservicios son unidades altamente cohesivas de código que son más fáciles de gestionar de forma aislada, reduciendo la carga sobre los desarrolladores y si se aplican de manera responsable puede conducir a códigos más simples, con menos defectos. (Connor et al., 2016).

Entre las *desventajas* tenemos las siguientes:

- **Punto de vista del Negocio:**

  - *Cantidad y costo:* Ser capaz de desplegar rápidamente pequeñas unidades independientes es una gran ventaja para el desarrollo, pero pone una presión adicional sobre las operaciones ya que media docena de aplicaciones se convierten ahora en cientos de pequeños microservicios, para muchas organizaciones se hace complicado y costoso manejar tal cantidad de herramientas que cambian rápidamente. (Fowler, 2015). La

interoperabilidad se ve comprometida tambien, ya que se hace mas compleja según aumenta la cantidad de microservicios.

- *Cultura organizacional:* adicional a la parte técnica, Fowler (2015) agrega que para hacer todo esto efectivamente, también es necesario introducir una cultura de devops: mayor colaboración entre desarrolladores, operaciones y todos los demás involucrados en la entrega de software. Es decir el cambio cultural, el cual es siempre un obstáculo, especialmente en organizaciones antiguas y de gran tamaño o trayectoria.

- **Punto de vista Técnico:**
  - *Infraestructura DevOps:* Connor (2016) menciona que se necesita una infraestructura DevOps (Development + Operations, es una cultura o movimiento que automatiza el proceso de entrega del software y los cambios en la infraestructura. Su objetivo es ayudar a crear un entorno donde la construcción, prueba y lanzamiento de un software pueda ser más rápido y con mayor fiabilidad) más sofisticada, porque se requiere normalmente la construcción de un despliegue de servicios de pipeline, y por lo tanto toca hacer uso de las tecnologías de nube y de contenedores las cuales permiten la construcción de esos pipeline, ésta tecnología es la que está impulsando la adopción de estos procesos híper agiles ajustados. Aquí la implementación de la interoperabilidad está totalmente presente.

  - *Gestion de datos*: Se debe abordar el problema de la gestión de datos. Ya que como se mencionó anteriormente, cada microservicio tiene su propia base de datos privada, pudiendo ser SQL o NoSQL (Richardson, 2015a); por lo tanto se necesita establecer diversos mecanismos de interoperabilidad para las mismas.

  - *Tiempo de respuesta:* Las llamadas remotas pueden ser lentas, Fowler (2015) indica que si se posee un servicio el cual llama a media docena de servicios remotos, y cada uno de los cuales llama a otra media docena de servicios remotos, estos tiempos de respuesta se suman lo que ocasiona características de latencia significativas. Este resultado principalmente se origina por un mal diseño de mecanismos de interoperabilidad.

  - *Desarrollo de consultas y transacciones:* Richardson (2015), agrega que el desarrollo de transacciones comerciales que actualizan entidades que son

propiedad de múltiples servicios, es un desafío, al igual que la implementación de consultas para recuperar datos de múltiples servicios. Relacionado con el problema de la gestión de los datos, de la misma manera se necesita establecer diversos mecanismos de interoperabilidad para estas transacciones.

- *Asincronia*: Otro punto a tomar en cuenta es la asincronía, si se hace seis llamadas asincrónicas en paralelo, llega a ser tan lento como la llamada más lenta en lugar de la suma de sus latencias. Esto puede tener un rendimiento grande, pero la programación asíncrona es difícil de conseguir, y mucho más difícil de depurar, la mayoría de las aplicaciones de microservicios necesitan asincronía para obtener un rendimiento aceptable. (Fowler, 2015). Tiene algo relacion con el problema de tiempos de respuesta.

- *Fiabilidad:* Fowler (2015) La fiabilidad también se encuentra comprometida, ya que el programador espera que las llamadas de función en proceso funcionen, pero una llamada remota puede fallar en cualquier momento, además con muchos microservicios, hay aún más potenciales puntos de falla, el programador tiene la tarea adicional de determinar las consecuencias del fracaso para cada llamada remota. Los mecanismos de interoperabilidad deben tener en cuenta estos posibles problemas.

- *Persistencia de los datos:* Al momento de la gestión de Datos, se debe tener cuidado en la persistencia de los mismos. “Los microservicios introducen problemas de consistencia eventuales debido a la gestión descentralizada de datos.” (Fowler, 2015, p.1). Problemas con la consistencia de los datos, debido al uso de bases de datos diferentes.

1.1.4. Comparación entre Arquitecturas Monolíticas vs Microservicios

Como se mencionó en el apartado **1.1.1**, la arquitectura de Microservicios es una alternativa frente a la Monolítica, la cual se utiliza al momento de realizar aplicaciones sin importar el tamaño de las mismas; a continuación se presenta una tabla comparativa (**Tabla 1**) entre estas dos arquitecturas, en la cual se tienen varios ámbitos de comparación, con el objetivo de identificar los enfoques que adopta cada una de ellas.

Tabla 1: Comparación entre Arquitecturas Monolíticas con Microservicios.

ÁMBITO	MONOLÍTICOS	MICROSERVICIOS
Autonomía	Es difícil eliminar una funcionalidad con seguridad, por el motivo de que otras funcionalidades dependen de ésta.	Una implementación de métodos que corresponden a una funcionalidad se puede eliminar con seguridad, por el motivo de que es independiente y otras funcionalidades no dependen de ésta.
	Pocas aplicaciones son verdaderamente autónomas, por lo general hay otras aplicaciones, con las que están relacionados.	Cada aplicación es totalmente autónoma, y no depende de otras.
Despliegue	Las distintas aplicaciones hacen despliegues coordinados.	Cada aplicación se despliega independientemente.
Cambios	El costo de los cambios en todos los niveles es alto. Un cambio de una sola funcionalidad significa una redistribución de toda la aplicación.	Los cambios son fáciles de implementar y no costosos ya que están dirigidos a servicios específicos.
	Un cambio en una funcionalidad tiene consecuencias inesperadas o requiere un cambio en otras funcionalidades, ya que son dependientes entre sí.	Los cambios solo afectan a la funcionalidad donde se realizan los mismos, cada funcionalidad es independiente.
Persistencia	Los servicios comparten un almacén de persistencia.	Cada funcionalidad puede poseer su propio almacén de persistencia.
	No se puede cambiar la capa de persistencia de una funcionalidad sin afectar a otras funcionalidades.	Como posee su propio almacén de persistencia, los cambios solo afectan a esa funcionalidad.
	No evitan por completo los problemas de inconsistencia, pero sufren mucho menos de ellos, ya que la cantidad de bases de datos es reducida y	Los problemas de persistencia, están más presentes, por el motivo de que existen muchas bases de datos.

	compartida.	
	Se puede actualizar un gran grupo de datos en una sola transacción, por el motivo de que las bases de datos son pocas.	Se requiere múltiples recursos para actualizar además de transacciones distribuidas, ya que se usan muchas bases de datos y de diferentes tecnologías.
Equipo Desarrollo	El equipo de desarrollo necesita conocer profundamente los diseños y esquemas de las otras funcionalidades, deben familiarizarse con el diseño y la composición de toda la aplicación.	El equipo de desarrollo solo conoce los diseños y esquemas de la funcionalidad que es responsable. Cada funcionalidad puede ser manejada por un equipo separado, por lo que esto favorece la separación de las preocupaciones y responsabilidades.
Tamaño	La mayoría de las aplicaciones son demasiado grandes para modernizarlas en un solo paso. Como consecuencia, tales modernizaciones se llevan a cabo gradualmente en varios pasos, o incrementos, a lo largo de un camino de la modernización.	Las aplicaciones son pequeñas, por lo que se las puede actualizar en un solo paso.
Escalabilidad	La escalabilidad es limitada y complicada, la aplicación completa tiene que ser replicada, por el motivo de que las funcionalidades de la misma son dependientes entre ellas.	La escalabilidad se puede realizar mediante la distribución de las funcionalidades a través de servidores, replicando solamente las que son necesarias, cada funcionalidad se puede implementar de forma independiente, lo que hace que sea más fácil de implementar continuamente nuevas versiones de las funcionalidades.
Comunicación	Todas las funcionalidades pertenecen a una sola aplicación, no hay sobrecarga de comunicación.	Las llamadas remotas son necesarias para la comunicación entre las funcionalidades, siendo factores indirectos en el

		rendimiento de la aplicación.
Enfoque	Hay un enfoque en los niveles y la integración a través de niveles.	Hay un foco en las necesidades del negocio y la comunicación entre los equipos.
Fallos	Un fallo en una funcionalidad puede o da de baja a toda la aplicación y dificultar la experiencia del usuario.	Un fallo en una funcionalidad no significa necesariamente una falla a toda la aplicación, sólo puede fallar esa funcionalidad específica, ya que las funcionalidades se desacoplan entre sí.
Entrega Continua	La entrega continua es una habilidad valiosa, casi siempre vale la pena el esfuerzo para conseguirla, pero no es obligatoria.	La entrega continua se convierte en esencial para una configuración seria de microservicios. No hay forma de manejar docenas de servicios sin la automatización y colaboración que fomenta la entrega continúa. La complejidad operacional también se incrementa debido al aumento de las demandas en la gestión de estos servicios y el monitoreo.

Fuente: El autor
Elaboración: El autor

En base a la **Tabla 1,** se puede mencionar que tanto la Arquitectura de Microservicios, como la Monolítica tienen sus puntos a favor y en contra; se puede concluir que no son arquitecturas perfectas, pero que se las usa según lo que se busque implementar y teniendo en cuenta esta comparación, que detalla los puntos de vista de cada una; por ejemplo si se desea desarrollar una aplicación que involucre múltiples tecnologías lo ideal sería implementar MS, y tener en cuenta las consideraciones que conlleva esta arquitectura, frente a la Monolítica.

1.2. Modelos Arquitectónicos para el diseño de Microservicios

Al diseñar una aplicación, cada decisión arquitectónica tiene ventajas como inconvenientes. Si los beneficios de un enfoque superan los inconvenientes depende en gran medida del contexto del proyecto en particular. Un patrón arquitectónico es una manera ideal de describir una solución a un problema en un contexto dado junto con sus compensaciones. Dentro de microservicios se puede trabajar con varios patrones que otorgan solución a diversos problemas entre los que se incluyen la comunicación entre servicios, el registro de servicio y el descubrimiento de servicios.

1.2.1. Patrones utilizados en la construcción de Microservicios.

Para entender la forma de trabajar de MS, Richardson (2014) propone un ejemplo el cual indica que una aplicación maneja peticiones (solicitudes y mensajes a través del protocolo HTTP), acceso a bases de datos (relacional y no relacional), intercambio de mensajes con otros sistemas, y retorno de una respuesta en formato HTML/JSON/XML. Estos servicios se comunican utilizando protocolos síncronos como HTTP/REST, se desarrollan y despliegan independientemente uno del otro.

Los requerimientos para el ejemplo propuesto por Richardson (2014) son los siguientes: existe un equipo de desarrolladores trabajando en la aplicación, los nuevos miembros del equipo deben ser productivos, la aplicación debe ser fácil de entender y modificar, el despliegue continuo debe ser una característica de la aplicación, se debe ejecutar varias copias de la aplicación en varias máquinas para satisfacer los requisitos de escalabilidad y disponibilidad y finalmente aprovechar las tecnologías emergentes (marcos, lenguajes de programación, etc.)

Según los requerimientos mencionados en el ejemplo, la solución es implementar una arquitectura de microservicios como se observa en la **Figura 5,** para ello, el arquitecto de la aplicación descompone funcionalmente la aplicación en un conjunto de servicios colaborativos, en el que cada servicio ejecuta un conjunto de funciones estrechamente relacionadas, pero independientes entre sí.

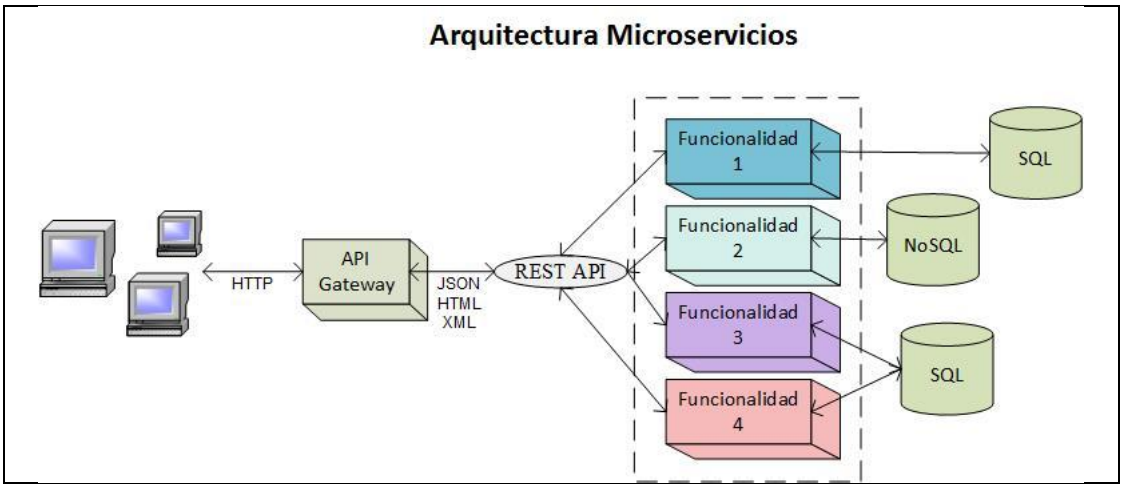


Figura 5: Ejemplo de Arquitectura de Microservicios
Fuente: El autor
Elaboración: El autor

Bajo este contexto, aparece Scale Cube (ver **Figura 6**) que es un modelo de escalabilidad tridimensional muy útil el cual sirve para descomponer funcionalmente la aplicación en un conjunto de servicios colaborativos. Según este modelo existen 3 o ejes (X, Y y Z).

- *Eje X:* Richardson (2014b) menciona que consiste en ejecutar varias copias de una aplicación detrás de un equilibrador de carga; es decir si hay N copias de una aplicación, cada copia maneja 1/N de la carga, siendo un enfoque sencillo y comúnmente usado para escalar una aplicación, un inconveniente de este enfoque es que debido a que cada copia potencialmente accede a todos los datos, la caché requiere más memoria para ser eficaz, además este enfoque no aborda los problemas de incrementar el desarrollo y la complejidad de la aplicación.

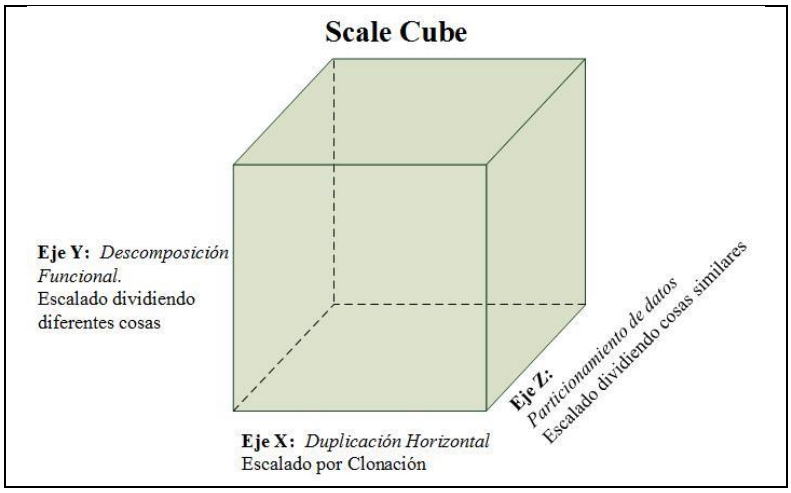


Figura 6: Scale Cube
Fuente: El autor
Elaboración: El autor

- *Eje Y:* A diferencia del eje X y el eje Z, que consisten en ejecutar varias copias idénticas de la aplicación, el escalado del eje del eje Y divide la aplicación en múltiples y diferentes servicios, en el cual cada uno es responsable de una o más funciones estrechamente relacionadas, existen 2 formas para dividir la aplicación: una consiste en utilizar la descomposición basada en verbo y definir los servicios que implementan un solo caso de uso, como la comprobación; el otro método es descomponer la aplicación por sustantivo y crear servicios responsables de todas las operaciones relacionadas con una entidad en particular, como la gestión de clientes, una aplicación puede utilizar una combinación de descomposición basada en verbo y sustantivo. (Richardson, 2014c).

Se puede realizar una combinación entre los escalados X y Y. **(Figura 7).** En la cual primeramente se hace un escalado de tipo Y, separando las 3 funcionalidades, y a partir de estas se crean las copias de cada funcionalidad, teniendo como resultado un escalado tipo X.

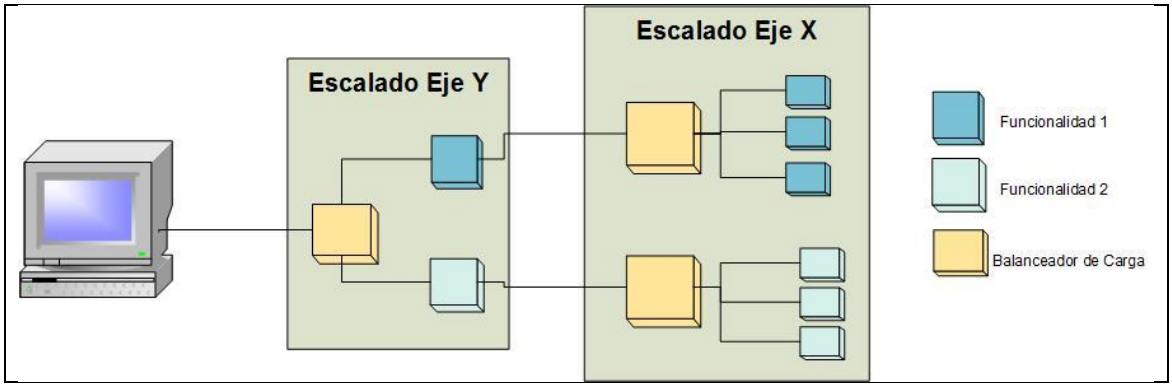


Figura 7: Escalado Eje Y, y Eje X
Fuente: El autor
Elaboración: El autor

- *Escalado del eje Z*: Según Richardson (2014b) cada servidor ejecuta una copia idéntica del código, en este sentido, es similar a la escala de eje X; la gran diferencia es que cada servidor es responsable de sólo un subconjunto de los datos; las divisiones en el eje Z se utilizan comúnmente para escalar las bases de datos; los datos son particionados a través de un conjunto de servidores basados en un atributo de cada registro; el escalado del eje Z también se puede aplicar a las aplicaciones. (Ver **Figura 8**).

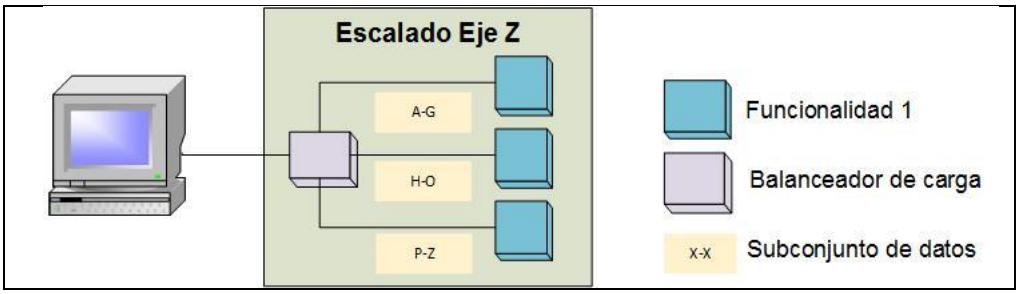


Figura 8: Escalado Eje Z
Fuente: El autor
Elaboración: El autor

Luego de analizar los 3 tipos de escalados existentes, para la arquitectura de MS se utiliza la escalada de eje Y, ya que este es ideal al momento de separar una aplicación según sus funcionalidades.

Villamizar (2015), agrega que en el estilo de microservicios, cada uno de ellos es desarrollado y probado independientemente por un equipo de desarrollo, utilizando el método más apropiado, con bases de código independientes y además el equipo se encarga de implementar, escalar, operar y actualizar el microservicio.

Frente a los microservicios hay un conjunto de aplicaciones Gateway, cada una ofrece servicios a tipos específicos de usuarios finales. Cada Gateway expone sus servicios utilizando diferentes interfaces (web, móvil, cliente) y protocolos tales como: HTTP, SOAP/HTTP y REST/HTTP respectivamente (Villamizar, 2015). Los microservicios suelen exponer sus servicios a los Gateway (no a los usuarios finales), los mismos que reciben solicitudes de usuarios finales, consumen uno o varios microservicios y envían los resultados a los usuarios finales, además cada Gateway es desarrollado, probado, desplegado, escalado, operado y actualizado de forma independiente por un equipo y típicamente no tienen capas de persistencia; además cada microservicio / gateway se desarrolla utilizando diferentes lenguajes de programación (Java, .NET, PHP, Ruby, Phyton, Scala, etc.) y tecnologías que manejan persistencia de datos (SQL, No-SQL, etc.).

En la **Figura 9,** se observa una arquitectura donde se involucra dos microservicios, ambos exponen su servicio principal utilizando REST como protocolo. Los servicios expuestos por cada microservicio son consumidos por el Gateway. El Gateway se desarrolla como una aplicación web ligera que recibe la solicitud de los usuarios finales (navegadores), obtiene el resultado que consume uno o más microservicios y devuelve los resultados, además cada equipo de desarrollo tiene asignada una tarea específica.

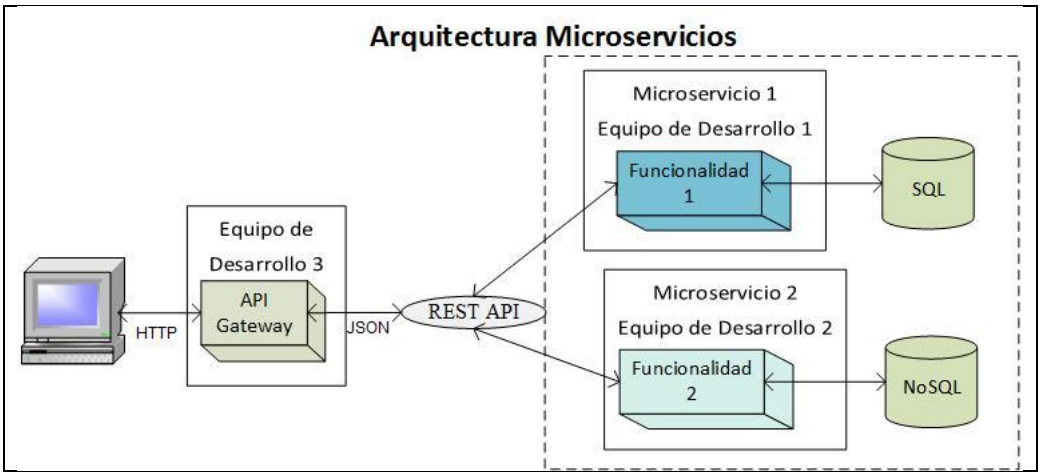


Figura 9: Arquitectura de Microservicios
Fuente: El autor
Elaboración: El autor

El protocolo utilizado entre los navegadores y el Gateway es HTTP; el protocolo para intercambio de mensajes usado entre el Gateway y cada microservicio es JSON. Cada funcionalidad posee una base de datos; el Gateway no almacena ninguna información, por lo que no necesita una capa de persistencia.

Existen 9 patrones de diseño propuestos por Richardson (2014), relacionados con microservicios, como se observa en la **Figura 10.**

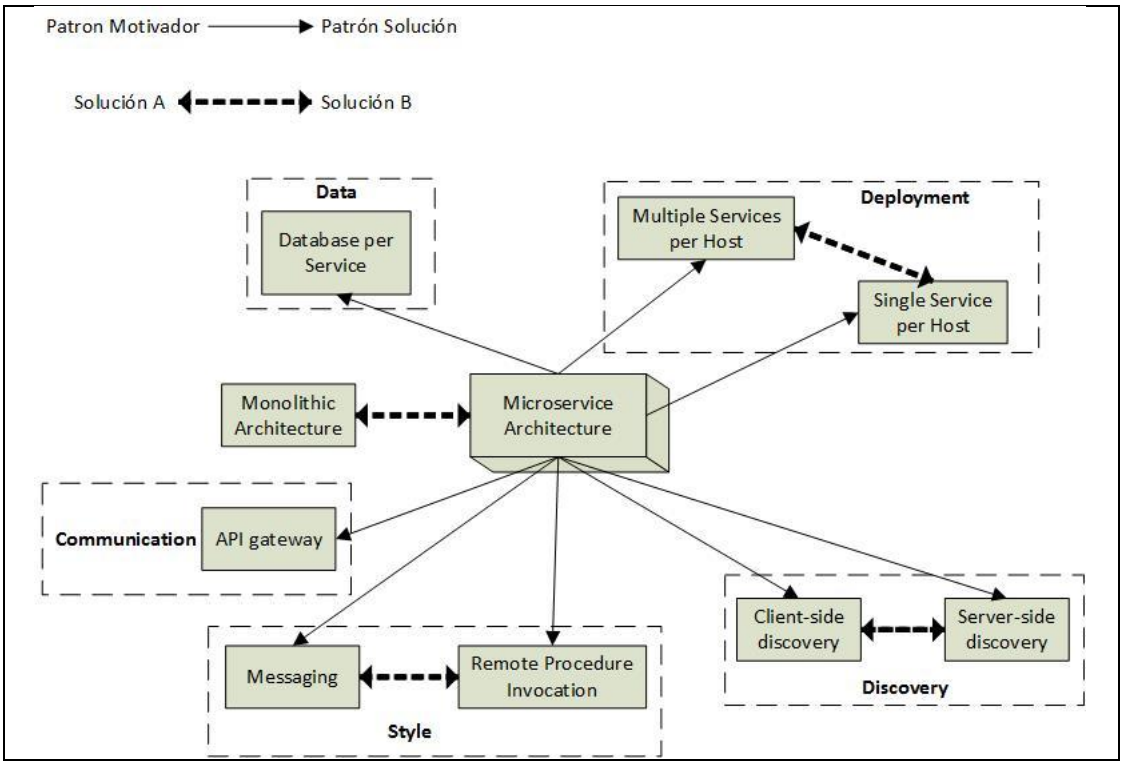


Figura 10: Patrones relacionados con Microservicios
Fuente: Richardson (2014a)
Elaboración: El autor

A continuación se describe los patrones que se observan en la **Figura 10**:

- El patrón *API Gateway* se lo utiliza para definir cómo los clientes acceden a los servicios en una arquitectura de microservicios, es el único punto de entrada para todos los clientes. Se encarga de gestionar las solicitudes de los clientes direccionándolas al servicio apropiado.
- Richardson (2014a) indica que el API Gateway se encarga de proporcionar la API óptima para cada cliente, además reduce el número de solicitudes de ida y vuelta. Por ejemplo, permite a los clientes recuperar datos de varios servicios con un solo viaje de ida y vuelta; menos solicitudes también significa menos gastos generales y mejora la experiencia del usuario. Además oculta a los clientes de cómo la aplicación se divide en microservicios, y también oculta las ubicaciones de las instancias de servicio. Para finalizar simplifica mueve la lógica para llamar a varios servicios, desde el cliente a la puerta de enlace API.
- Los patrones *Client-side Discovery* y *Server-side Discovery* se utilizan para encaminar las solicitudes de un cliente a una instancia de servicio disponible en una arquitectura de microservicios**.**
- *Client-side Discovery*: Al realizar una solicitud a un servicio, el cliente obtiene la ubicación de una instancia de servicio consultando un registro de servicio, que conoce las ubicaciones de todas las instancias de servicio; este patrón tiene menos piezas móviles y saltos de red en comparación con *Server-side Discovery.* (Richardson, 2015b)*.*
- *Server-side Discovery*: Richardson (2015c) indica que al hacer una solicitud a un servicio, el cliente realiza una solicitud a través de un enrutador (equilibrador de carga) que se ejecuta en una ubicación bien conocida, el enrutador consulta un registro de servicios, que puede estar incorporado en el enrutador, y reenvía la solicitud a una instancia de servicio que se encuentre disponible. En comparación *Client-side Discovery*, el código del cliente es más sencillo ya que no tiene que tratar con el descubrimiento de servicios, porque un cliente simplemente hace una solicitud al enrutador y este último es el encargado de realizar este descubrimiento.
- Los patrones de *Messaging* y de *Remote Procedure Invocation* son dos maneras diferentes en las que los servicios pueden comunicarse.
- El patrón *Database per Service* describe cómo cada servicio tiene su propia base de datos**.** Mantiene los datos persistentes de cada microservicio privados

para ese servicio y accesibles sólo a través de su API. Richardson (2016a), indica que este patrón ayuda a asegurar que los servicios estén ligeramente acoplados, que los cambios en la base de datos de un servicio no afectan a ningún otro servicio, y que cada servicio puede utilizar el tipo de base de datos que mejor se adapte a sus necesidades, pero siempre teniendo en cuenta que la complejidad de gestión de múltiples bases de datos SQL y NoSQL.

- Los patrones *Single Service per Host* y *Multiple Services per Host* son dos estrategias de despliegue de Microservicios.
- *Single Service per Host:* En este patrón Richardson (2016c) indica que se trata de Implementar cada instancia de servicio único en su propio host, en la cual las instancias de servicios están aisladas unas de otras, por lo tanto no hay posibilidad de conflictos de recursos o dependencia de las versiones. Una instancia de servicio sólo puede consumir como máximo los recursos de un único host, por lo que es fácil de monitorear, administrar y reasignar cada instancia de servicio.
- *Multiple Services per Host:* Ejecuta varias instancias de diferentes servicios en un host (máquina física o virtual), hay varias formas de implementar una instancia de servicio según Richardson (2016b) en un host compartido, como son: Implementar cada instancia de servicio como un proceso JVM (Java Virtual Machime), o implementar varias instancias de servicio en la misma JVM.

1.2.2. Atributos de calidad y su relación con MS: Interoperabilidad

Todo proyecto de software tiene como objetivo cumplir con aspectos de calidad, seguridad, fiabilidad, mantenibilidad, interoperabilidad, etc.

La calidad del software se puede observar como un atributo, en el cual se refiere a características medibles, es decir cosas que se pueden comparar para conocer estándares, como longitud, color, propiedades; teniendo relevada importancia en el desarrollo de sistemas; la medida en que la aplicación posee una combinación deseada de atributos de calidad como usabilidad, rendimiento, fiabilidad y seguridad indica el éxito del diseño y la calidad general de la aplicación de software.

En referencia a los atributos de calidad, Microsoft (2009) afirma que son los factores generales que afectan el comportamiento de las aplicaciones en tiempo de ejecución, el diseño del sistema, y la experiencia del usuario. Representan áreas de preocupación que tienen el potencial de generar un amplio impacto a una aplicación a través de las capas. Algunos de estos atributos están relacionados con el diseño

general del sistema, mientras que otros son específicos para tiempos de ejecución, diseño, o aspectos centrados en el usuario.

Al diseñar aplicaciones para cumplir con cualquiera de los atributos de calidad, es necesario considerar el impacto potencial en otros requisitos. La importancia o prioridad de cada atributo de calidad difiere del contexto del sistema, la interoperabilidad como *atributo de calidad* según Ducq & Chen (2008) es la capacidad de dos (o más) aplicaciones para intercambiar información y utilizar recíprocamente su funcionalidad, la cual no es un estado binario y con el fin de mejorarla entre dos aplicaciones particulares, es necesario desarrollar métricas para medir el grado de interoperabilidad que puede ser evaluado. La

Como atributo de calidad, la interoperabilidad según Ducq & Chen (2008) es la capacidad de dos (o más) aplicaciones para intercambiar información y utilizar recíprocamente su funcionalidad, la cual no es un estado binario y con el fin de mejorarla entre dos aplicaciones particulares, es necesario desarrollar métricas para medir el grado de interoperabilidad que puede ser evaluado.

Tabla 2, categoriza los atributos de calidad en cuatro áreas específicas relacionadas con el diseño, el tiempo de ejecución, el sistema y las cualidades del usuario, cada uno de los cuales tiene un impacto en la arquitectura de microservicios, en la cual cada uno está presente en mayor o en menor grado, según la funcionalidad y tamaño de la aplicación, adaptada de los atributos dados por Microsoft (2009).

Como atributo de calidad, la interoperabilidad según Ducq & Chen (2008) es la capacidad de dos (o más) aplicaciones para intercambiar información y utilizar recíprocamente su funcionalidad, la cual no es un estado binario y con el fin de mejorarla entre dos aplicaciones particulares, es necesario desarrollar métricas para medir el grado de interoperabilidad que puede ser evaluado.

Tabla 2: Atributos de calidad

CATEGORÍA	ATRIBUTO DE CALIDAD
Cualidades del diseño	Integridad Conceptual
	Mantenibilidad
	Reusabilidad
Cualidades en tiempo de ejecución	Disponibilidad
	Interoperabilidad
	Manejabilidad
	Rendimiento
	Confiabilidad
	Escalabilidad
	Seguridad

Cualidades del Sistema	Soportabilidad
	Testeabilidad
Cualidades del Usuario	Usabilidad

Fuente: Microsoft (2009)
Elaboración: El autor

Microsoft (2009), menciona que la interoperabilidad, considera a los protocolos de comunicación, interfaces y formatos de datos como claves al momento de implementar una aplicación. La estandarización es también un aspecto importante a considerar en el diseño de una aplicación interoperable. La interoperabilidad es primordial en Microservicios, por el motivo de que al ser funcionalidades que no dependen entre sí, pero si comunican e intercambian datos entre ellas, es necesario establecer la forma de interacción (interoperabilidad) entre las funcionalidades. Las cuestiones clave para la interoperabilidad que propone Microsoft (2009) son:

- ***La interacción con las aplicaciones existentes o externas que utilizan diferentes formatos de datos*****:** considera como se puede habilitar las aplicaciones para interoperar, mientras evoluciona por separado o incluso se sustituye. Por ejemplo, utilizar la orquestación con adaptadores para conectarse con aplicaciones externas o heredadas y traducir datos entre aplicaciones; o utilizar un modelo de datos canónicos para manejar la interacción con un gran número de formatos de datos diferentes.

- ***Difuminación de límites*****: que permite a los artefactos de una aplicación "desactivar" a otra**: Considera la posibilidad de aislar sistemas mediante interfaces de servicio y / o capas de asignación. Por ejemplo, exponer los servicios utilizando interfaces basadas en XML o tipos estándar para apoyar la interoperabilidad con otros sistemas. Diseñar los componentes para que sean cohesivos y tengan un acoplamiento bajo para maximizar la flexibilidad y facilitar el reemplazo y la reutilización.

- ***La falta de adherencia a los estándares:*** Estar enterado de los estándares formales y de facto, para el dominio en el que se esté trabajando, y considerar el uso de uno de ellos en lugar de crear algo nuevo y propietario.

Aunque las aplicaciones se consideran autónomas en cuanto a su funcionalidad y funcionamiento, colaboran con otras aplicaciones para contribuir a las metas de la aplicación superior. Por lo tanto, como mencionan Stary & Wachholder (2015) la interoperabilidad entre éstas aplicaciones requiere reconocer y superar su naturaleza

autónoma, lo que significa que son necesarios para que las aplicaciones sean diversas y emergentes en el comportamiento.

La interoperabilidad como una parte inherente de la conectividad de las aplicaciones es crucial cuando se trata de la interacción aplicaciones y colaboración. En el caso de que las aplicaciones autónomas no sean interoperables; la conectividad, la diversidad, y la emergencia no están habilitadas.

La **Figura 11**(a) expone por ejemplo que ambas funcionalidades de la aplicación están integradas, y comparten partes comunes. En el segundo caso **Figura 11**(b) sin embargo, las funcionalidades implicadas no tienen dependencias funcionales, sino más bien la capacidad de cooperar entre sí para lograr un objetivo común. Las aplicaciones involucradas están de acuerdo en una forma común de interacción para colaborar e intercambiar información, ambas aplicaciones han acordado una forma común de interacción manteniendo su independencia en términos de operación.

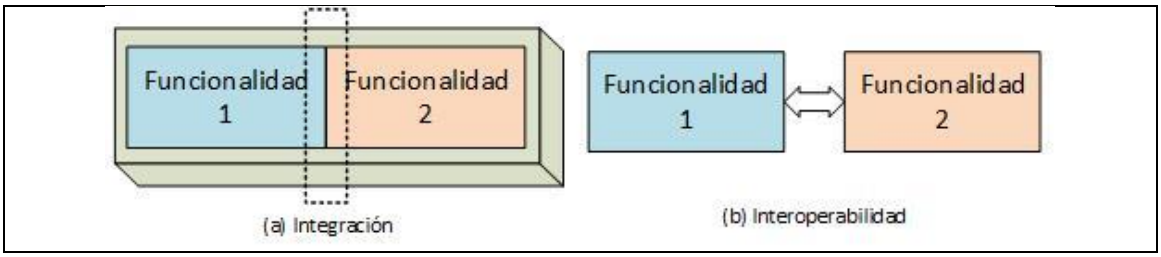


Figura 11: Integración del sistema, frente a Interoperabilidad.
Fuente: Microsoft (2009)
Elaboración: El autor

Stary & Wachholder (2015), mencionan que la interoperabilidad es esencial no sólo para superar el aislamiento de la aplicación, sino también para permitir que las aplicaciones sean diversas y emergentes en el comportamiento; sin embargo, sólo se da cuando las aplicaciones pueden acordar un estándar antes de su implementación, en entornos dinámicos en los que no existe un estándar común la interoperabilidad no puede garantizarse.

Finalmente se puede decir que la interoperabilidad es la capacidad de dos (o más) aplicaciones para intercambiar información y utilizar recíprocamente su funcionalidad; son autónomas en cuanto a su funcionalidad y funcionamiento, es decir no tienen dependencias funcionales; las aplicaciones están de acuerdo en una forma común de interacción para colaborar e intercambiar información, manteniendo siempre su independencia en términos de operación; la interoperabilidad permite la conectividad, interacción, colaboración, y diversidad de las aplicaciones.

En el presente trabajo se implementará y evaluará el grado de interoperabilidad existente en una aplicación construida bajo la arquitectura de MS.

1.2.3. Validación de Interoperabilidad en MS:

El hecho de que se pueda mejorar la interoperabilidad, como mencionaba Ducq & Chen (2008), significa que existen métricas para medir el grado de interoperabilidad. Medir la interoperabilidad permite conocer las fortalezas y debilidades para interoperar y priorizar acciones para mejorar su capacidad de asociación.

A continuación se mencionan tres tipos de mediciones de interoperabilidad expuestas por Ducq & Chen (2008):

a) Medida potencial de interoperabilidad:

Se refiere a la identificación de un conjunto de características que tienen impacto en la interoperabilidad. El objetivo es evaluar la potencialidad de un sistema para adaptarse y acomodarse dinámicamente para superar posibles barreras al interactuar con un tercer socio.

Se proponen cinco niveles que caracterizan la potencialidad:

1. ***aislado***, lo que representa una incapacidad total para interoperar;

2. ***inicial***, donde la interoperabilidad requiere fuertes esfuerzos que afectan a la asociación;

3. ***ejecutable***, donde la interoperabilidad es posible incluso si el riesgo de encontrar problemas es alto;

4. ***conectable***, donde la interoperabilidad es fácil incluso si los problemas pueden aparecer para la asociación distante;

5. ***interoperable***, que considera la evolución de los niveles de interoperabilidad y donde el riesgo de resolver problemas es débil.

b) Medida de compatibilidad de interoperabilidad:

La medición de la compatibilidad de interoperabilidad para Ducq & Chen (2008) debe realizarse durante la etapa de ingeniería, es decir, cuando se conocen los sistemas de la interoperación. La medida se hace con respecto a las barreras identificadas a la

interoperabilidad, por ejemplo, se pueden realizar las siguientes preguntas para saber si existe incompatibilidad entre dos sistemas.

- *Compatibilidad conceptual:* Sintáctica: ¿es la información a ser intercambiada expresada con la misma sintaxis? Semántica: ¿la información que se va a intercambiar tiene el mismo significado? Esto quiere decir que si la información a intercambiar es entendible y tiene el significado adecuado según el contexto en que se esté tratando.
- *Compatibilidad organizacional:* Personas: ¿las autoridades / responsabilidades están claramente definidas en ambas partes? Organización: ¿son compatibles las estructuras organizativas?
- *Compatibilidad tecnológica:* Plataforma: ¿son compatibles las tecnologías de plataforma de TI? Comunicaciones: ¿utilizan los socios los mismos protocolos de intercambio?

A partir de esas preguntas, se puede diseñar una tabla (**Tabla 3**), en la cual, si se detecta una incompatibilidad, el coeficiente 1 se asigna al problema de interoperabilidad y la barrera que se considera. Por el contrario, el coeficiente 0 se dará cuando no se detectan incompatibilidades.

Tabla 3: Medida de compatibilidad de interoperabilidad

ÁMBITO	**BARRERAS**					
	Conceptual		Organizacional		Tecnología	
	Sintáctica	Semántica	Responsabilidad Autoridades	Organización	Plataforma	Comunicaciones
Negocios	1	1	0	1	0	0
Procesos	1	1	1	1	1	1
Servicios	1	0	0	0	1	0
Datos	0	0	0	1	1	1

Fuente: Ducq & Chen (2008)
Elaboración: El autor

c)Medida de rendimiento operativo de interoperabilidad:

La medición del rendimiento debe realizarse durante la fase operativa, es decir, en tiempo de ejecución, para evaluar la capacidad de interoperación. Los criterios clásicos como el costo, el retraso y la calidad pueden utilizarse para medir el desempeño con respecto a las barreras y preocupaciones durante un ciclo de interoperabilidad básica (intercambio y uso de información).

El tiempo de interoperabilidad corresponde a la duración entre la fecha en que se solicita la información y la fecha en que se utiliza la información solicitada, el cual se puede descomponerse en varios periodos de tiempo, por lo tanto cuanto más fina sea

esta descomposición, la definición de debilidades de la interoperabilidad será más exacta.

La calidad de la interoperabilidad toma en consideración tres tipos de calidad según Ducq & Chen (2008):

- Del intercambio: establece si el intercambio se realiza correctamente, es decir, si la información enviada a un socio tiene éxito
- Del uso: representa el número de información recibida por un socio en comparación con el número de información solicitada. Un mayor número de información recibida (dificultad para introducir toda la información) o menos (escasez de información) en el número de información solicitada significa un deficiencia.
- De la conformidad: corresponde a la explotación de la información, es decir, si la información recibida es explotable o no.

Por su parte Rezaei (2014) indica que para alcanzar la interoperabilidad se requiere resolución a distintos niveles:

- **Interoperabilidad técnica:** Se logra entre los sistemas de comunicaciones electrónicas o los elementos de equipos electrónicos de comunicaciones, cuando los servicios o la información pueden ser intercambiados directa y satisfactoriamente entre ellos y sus usuarios. La interoperabilidad técnica suele estar asociada con componentes de hardware, sistemas y plataformas que permiten la comunicación de máquina a máquina. Este tipo de interoperabilidad se centra en los protocolos de comunicación y la infraestructura necesaria para que esos protocolos funcionen.
- **Interoperabilidad sintáctica:** se define como la capacidad de intercambiar datos. La interoperabilidad sintáctica generalmente se asocia con formatos de datos. Los mensajes transferidos por protocolos de comunicación deben poseer una sintaxis y codificación bien definidas.
- **Interoperabilidad semántica:** se define como la capacidad de operar en esos datos de acuerdo con la semántica acordada. La interoperabilidad semántica está normalmente relacionada con la definición de contenido, y se ocupa de la interpretación humana en lugar de la máquina de este contenido. Por lo tanto, la interoperabilidad a este nivel denota que existe un entendimiento común entre las

personas con respecto a la definición del contenido (información) que se intercambia.

- **Interoperabilidad organizacional:** se refiere a la capacidad de las organizaciones de comunicar y transferir datos significativos (información) a pesar del uso de una variedad de sistemas de información sobre diferentes tipos de infraestructuras, posiblemente a través de diversas regiones geográficas y culturas. La interoperabilidad organizativa se basa en la interoperabilidad exitosa de los aspectos técnicos, sintácticos y semánticos.

Al momento de hablar de interoperabilidad Rezaei (2014), identifica los problemas de la misma y los clasifican en cuatro niveles de granularidad (**Figura 12**).

El primer nivel de granularidad, de los problemas de interoperabilidad consiste en la interoperabilidad de los datos, la interoperabilidad de los procesos, la interoperabilidad de las normas, la interoperabilidad de los objetos, la interoperabilidad de los sistemas informáticos y la interoperabilidad cultural.

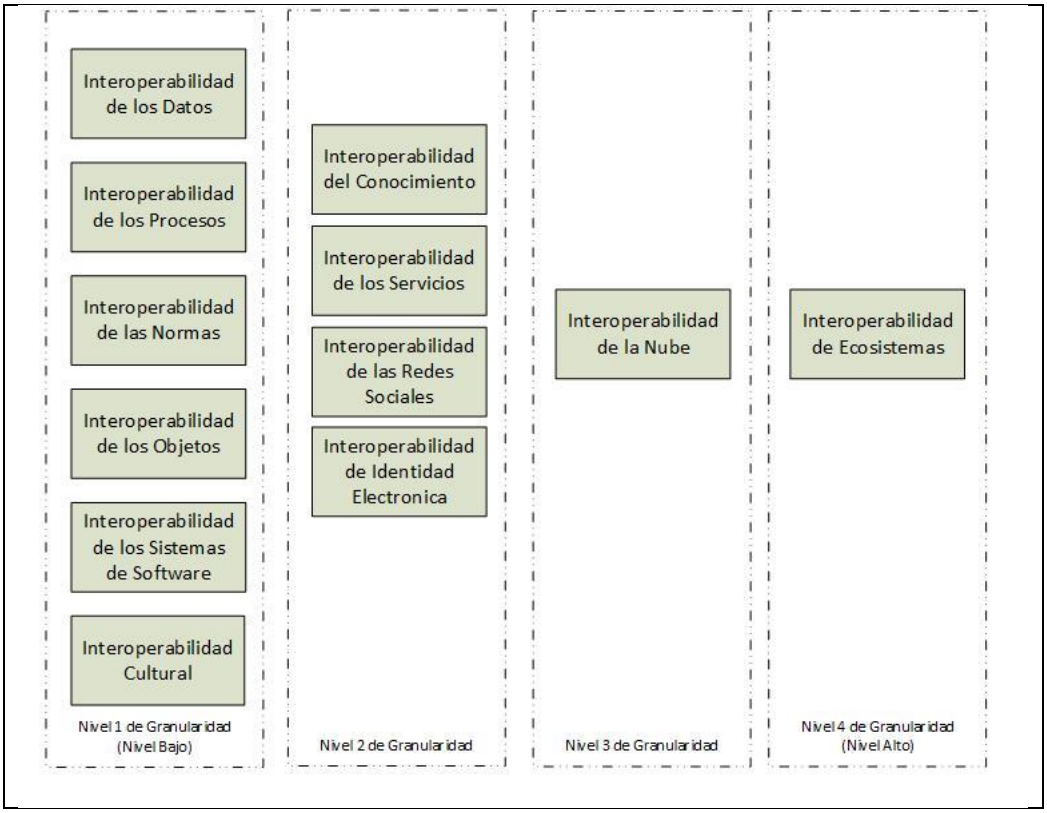


Figura 12: Problemas de Interoperabilidad
Fuente: Rezaei (2014).
Elaboración: El autor

El segundo nivel de granularidad de los problemas de interoperabilidad se centra en la interoperabilidad del conocimiento, que consiste en elementos que surgen de la

interoperabilidad de los datos, la interoperabilidad de los procesos, la interoperabilidad de las normas y la interoperabilidad cultural.

La interoperabilidad de los servicios, que incorpora los hechos de la interoperabilidad de los procesos, la interoperabilidad de los datos, la interoperabilidad de las normas y la interoperabilidad de los sistemas informáticos. La interoperabilidad de las redes sociales, que consiste en elementos que surgen de la interoperabilidad cultural y la interoperabilidad de los datos, y la interoperabilidad de la identificación electrónica, que está fuertemente relacionada con la interoperabilidad de los objetos, la interoperabilidad de los sistemas de software Y la interoperabilidad de reglas.

El tercer nivel de granularidad incluye: Interoperabilidad de la nube, que toma elementos de la interoperabilidad de los servicios, de la interoperabilidad del conocimiento y de la interoperabilidad de la identidad e intenta infundirlos con las características de la nube.

El cuarto nivel de granularidad de los problemas de interoperabilidad implica la interoperabilidad de los ecosistemas, que se ocupa de las empresas virtuales y digitales y está relacionada con la interoperabilidad de las nubes y la interoperabilidad de las redes sociales.

Por su parte Lambolais, Courbis, Luong, & Phan (2011) indican que un sistema es interoperable si, bajo un conjunto dado de condiciones, sus dispositivos son capaces de establecer, sostener y, si es necesario, derribar un enlace de comunicación manteniendo un cierto nivel de funcionamiento; un conjunto de partes es interoperable cuando está "bloqueado", es decir, cualquier punto de interacción puede ser alcanzado, la comunicación no será bloqueada, y cada parte alcanzará uno de sus estados finales.

Reemplazar en un sistema un componente por otro, no garantiza la interoperabilidad del sistema, de esta manera la interoperabilidad debe comprobarse de dos maneras: analizar todo el comportamiento de la arquitectura o comparar el nuevo componente sustituido según su primera implementación o su contexto. Además agrega que una vez definida una especificación de arquitectura, la interoperabilidad pueden ser automáticamente construida. (Lambolais, Courbis, Luong, & Phan, 2011).

En la **Tabla 4** se presenta las metricas referentes a interoperabilidad y sus respectivas descripciones, propuesta por Lambolais, Courbis, Luong, & Phan (2011).

Tabla 4: Métricas para evaluación de Interoperabilidad

METRICAS	DESCRIPCION
Conectividad	Se puede medir directamente contando el número de mensajes iniciados por todas las unidades participantes y el número de mensajes recibidos para la red o enlace de datos. En la medida en que el enlace está en funcionamiento continuo, la conectividad muestreada de esta manera es representativa de la conectividad de red. Si la red funciona intermitentemente, entonces la muestra debe ser cuidadosamente seleccionada y probada para asegurar que se alcanza el nivel de confianza requerido.
Capacidad	Es la velocidad a la cual los datos pueden ser pasados a través del tiempo. Dados sus parámetros de funcionamiento, se puede calcular una velocidad máxima de datos para cualquier sistema o grupo de sistemas.
Sobrecarga del Sistema	Se produce cuando hay que intercambiar más datos de los que el sistema es capaz de transmitir. Normalmente, la sobrecarga se coloca en una cola y luego se transmite cuando hay capacidad disponible. Por lo tanto, la medida de la sobrecarga del sistema es la suma de los mensajes que quedan en las colas después de su período de transmisión asignado para todos los nodos del sistema.
Subutilización	Esto ocurre cuando la tasa de datos del sistema / carga de mensajes es menor que su capacidad total, pero los mensajes están esperando en las colas que se van a transmitir. Esto ocurre cuando la asignación de transmisión a nodos seleccionados es menor que la requerida para borrar la cola al final de un período de transmisión. Del mismo modo, otros nodos no utilizan todo su tiempo asignado.
Subcapacidad	Se produce cuando los mensajes permanecen en colas y la velocidad de datos del sistema es máxima.
Latencia de datos	Es el tiempo transcurrido desde el momento del evento hasta el momento de su recepción por parte del usuario. Para propósitos analíticos, la latencia se divide a menudo en segmentos más pequeños. Esta división es útil en situaciones que implican un sensor remoto y un procesamiento intermedio para reducir los datos a una forma utilizable (mensaje de seguimiento) antes de pasar los datos al usuario.
Información Interpretación y utilización	Después de haber pasado los datos e interpretado correctamente, el siguiente paso sería verificar que se tome la acción correcta. La verificación de la acción tomada implica una revisión de la lógica asociada con cada opción que es posible en respuesta a un mensaje u acción del operador.

Fuente: Lambolais, Courbis, Luong, & Phan (2011).
Elaboración: El autor.

1.3. Modelos para evaluar interoperabilidad en MS

A continuación se presentan los modelos que permiten evaluar la interoperabilidad, así como una breve descripción de los mismos, para finalmente establecer un cuadro comparativo entre los mismo para conocer cuál es el que más adecuado para nuestra aplicación.

1.3.1. Quantification of interoperability methodology (QoIM)

Constituye la base para los niveles de los sistemas de información basados con interoperabilidad.

El enfoque de la medición de la interoperabilidad es único porque asocian la interoperabilidad con medidas de eficacia. Su objetivo era evaluar los problemas de interoperabilidad para tres áreas de misión: vigilancia de área amplia, orientación en el horizonte y guerra electrónica.

Ford Thomas C., Colombi John M., Graham Scott R., (2007), mencionan además que la interoperabilidad de sistemas, unidades o fuerzas puede ser factorizada en un conjunto de componentes que pueden cuantificar la interoperabilidad e identificaron los siete componentes necesarios como lenguajes, estándares, entorno, procedimientos, requisitos, factores humanos y medios. Para cada componente se asigna una medida de la función de la lógica de la eficacia y la utilizan para crear una tabla de verdad que es llenada con la simulación de acontecimientos discretos.

1.3.2. Military communications and information systems interoperability (MCISI)

Es usado para modelar matemáticamente la interoperabilidad de los Sistemas de Comunicaciones e Información (CIS); El cual es un proceso de múltiples etapas que combina requerimientos operacionales, datos CIS, estándares, interfaces e instalaciones de modelado. Se utiliza un cubo de color para visualizar el modelo MCISI; un eje del cubo representaba el nivel de comando; el segundo indica los servicios CIS y el tercer eje representa el medio de transmisión. (Metrics et al., 2007).

Las intersecciones tenían sus propios colores: rojo que no representaba ninguno, amarillo que significaba parcial y verde que indicaba interoperabilidad completa de un servicio específico a través de un medio específico a un nivel de comando especificado.

Ford Thomas C., Colombi John M., Graham Scott R., (2007), explican que una serie de puntos representan un conjunto de sistemas dentro de un entorno multidimensional, mientras que las características de los sistemas constituyen las coordenadas de los puntos. Entonces se definió una "distancia" normalizada entre dos puntos como d (A, B), e indica que en los casos en que d (A, B) = 0, entonces los sistemas A y B adquirieron plena interoperabilidad y si D (A, B)> 1, entonces la interoperabilidad de los dos sistemas se redujo.

1.3.3. Levels of information systems interoperability (LISI)

El LISI es un procedimiento para definir, medir, evaluar y certificar el grado de interoperabilidad requerido o logrado por y entre organizaciones o aplicaciones; éste modelo LISI se centra en mejorar los niveles de interoperabilidad de la complejidad dentro de los sistemas.

Polyakov & Ksenofontov (2001) exponen cinco niveles de interoperabilidad (0-4):

- Nivel 0 - Interoperabilidad aislada.
- Nivel 1 - Interoperabilidad conectada
- Nivel 2 - Interoperabilidad funcional: los sistemas
- Nivel 3 - Interoperabilidad basada en el dominio
- Nivel 4 - Interoperabilidad basada en la empresa

El modelo LISI, trabaja con 3 tipos de métricas:

- *Nivel genérico de interoperabilidad:* Se calcula para sistemas individuales y se expresa como un valor calculado matemáticamente, haciendo una comparación entre un sistema único por un lado y el modelo de capacidades LISI por el otro.
- *Nivel esperado de interoperabilidad:* Se define como el nivel genérico más bajo de ambos sistemas, es decir, el nivel donde se espera la interoperabilidad de ambos sistemas entre sí.
- *Nivel específico de interoperabilidad:* Es el valor métrico calculado entre los dos sistemas, resultantes de la comparación entre las alternativas de implementación que cada uno de los sistemas ha utilizado con respecto a las capacidades registradas. (Polyakov & Ksenofontov, 2001)

1.3.4. Organizational interoperability maturity model for C2 (OIM).

Se definen cinco niveles de madurez organizacional en este modelo indicadas por Rezaei (2014):

- Nivel 0: *Independiente:* Incluye las organizaciones que normalmente podrían trabajar sin ninguna interacción, excepto por contacto personal.
- Nivel 1: *Ad Hoc*: En este nivel hay algunos objetivos generales compartidos, sin embargo, las aspiraciones individuales de la organización tienen prioridad, y las organizaciones siguen siendo totalmente distintas.
- Nivel 2: *Colaborativo*: Los marcos reconocidos están preparados para apoyar la interoperabilidad, y también se identifican objetivos compartidos.
- Nivel 3: *Integrado*: Incluye objetivos compartidos y sistemas de valores, un entendimiento común y una preparación para interoperar, como una doctrina detallada de que existe una experiencia significativa en su uso.
- Nivel 4: *Unificado*: Comparte las metas organizacionales, los sistemas de valores, la estructura / estilo de comando y las bases de conocimiento en todo el sistema.

1.3.5. Interoperability assessment methodology (IAM)

Se basa en la idea de medición y cuantificación de un conjunto de componentes del sistema de interoperabilidad. Este modelo introdujo nueve componentes, que eran requisitos, conectividad de nodos, elementos de datos, protocolos, flujo de información, utilización de información, interpretación, latencia y estándares, los nueve componentes inlcuyen una respuesta "sí / no". (Metrics et al., 2007)

1.3.6. Stoplight

Este modelo tiene como objetivo ayudar a los responsables de la toma de decisiones a determinar si el sistema heredado que utilizan puede satisfacer los requisitos operativos y de interoperabilidad de la adquisición. (Metrics et al., 2007)

Está diseñado como matriz bidimensional, de tal manera que "se cumple con los requisitos operativos (sí / no)" se muestra en filas y "se cumple con los requisitos de adquisición (sí / no)" aparece en las columnas de la matriz. Las intersecciones matriciales están en rojo, amarillo, naranja y verde. La asignación de este conjunto jerárquico de colores depende de cuán bien se cumpla cada requisito específico. Este método ayuda a desarrollar un sistema de cuatro colores de evaluación de interoperabilidad.

1.3.7. Enterprise interoperability maturity model:

Se define un conjunto de niveles de madurez y un conjunto de áreas de preocupación; Rezaei (2014) indica, que cada área de preocupación sería definida por un conjunto de objetivos y metas relevantes para las cuestiones de interoperabilidad y colaboración.

Dependiendo de la ausencia, o de la presencia de los indicadores de madurez, se definiría la interoperabilidad y el nivel de madurez de la colaboración para cada área de preocupación.; Simultáneamente, si la compañía pretende alcanzar el próximo nivel de madurez, se debe considerar el estado To-Be.

Las seis áreas de preocupación en el modelo de madurez de interoperabilidad de la empresa se definen de la siguiente manera: Estrategia y Procesos Empresariales, Organización y Competencias, Productos y servicios, Sistemas y tecnología, Ambiente Legal, Seguridad y Confianza, Modelado Empresarial

Los cinco niveles de madurez se identifican de la siguiente manera: Realizado, Modelado, Integrado, Interoperable, Optimización.

1.3.8. The layered interoperability score (i-Score)

Este método utiliza los datos de arquitectura actuales y puede implicar más de un tipo de interoperabilidad. Lo que distingue a este método indica Rezaei (2014), es el mecanismo que utiliza para determinar un límite superior empírico de interoperabilidad para aquellos sistemas que soportan el proceso operativo. Este método puede aceptar capas personalizadas que permitan al analista compensar la medición por un número ilimitado de factores de rendimiento relacionados con la interoperabilidad. Dicho conjunto de factores puede incluir ancho de banda, tasa de capacidad de misión, protocolos, efectos atmosféricos o probabilidad de conexión, entre otros.

Este método es útil para la medición no tradicional de la interoperabilidad como mediciones de interoperabilidad de políticas u organizaciones.

1.3.9. Government interoperability maturity matrix:

El modelo según Rezaei (2014) ofrece a las administraciones un método sencillo de autoevaluación que puede utilizarse para evaluar la situación actual de las administraciones en materia de interoperabilidad del gobierno electrónico y las medidas necesarias para mejorar su posicionamiento en relación con la implantación del sistema y la prestación de servicios. Este modelo de madurez amplía los tres tipos de interoperabilidad, con el fin de identificar varios atributos de interoperabilidad que

deben tenerse en cuenta con la intención de evaluar el posicionamiento de la organización en interoperabilidad de gobierno electrónico. Posee 5 niveles de madurez (son los mismos del apartado **1.3.4.)**

Se definen tres dimensiones principales de interoperabilidad: Interoperabilidad Organizacional, Interoperabilidad Semántica, Interoperabilidad Técnica.

1.3.10. System-of-systems Interoperability (SOSI):

SOSI (System-of-systems interoperability), se desarrolló para permitir a los investigadores llevar a cabo más eficazmente la investigación de interoperabilidad de sistemas de sistemas. El modelo SoSI se basa en tres tipos de interoperabilidad (operacional, constructiva y programática) y las actividades asociadas a cada uno. SoSI carece de métricas para medir específicamente la interoperabilidad dentro de un sistema de sistema, pero proporciona un marco en el cual un analista puede usar sus propias métricas para medir la interoperabilidad (Metrics et al., 2007). El informe técnico en el que SoSI se introduce contiene un resumen de LISI, OIM, NMI, LCIM, LCI y SoSI.

1.3.11. Comparación y Evaluación de modelos de Interoperabilidad:

Después de exponer los modelos para evaluar la interoperabilidad, se realiza un análisis comparativo de los mismos, teniendo en cuenta las siguientes métricas propuestas por Rezaei (2014):

- *Interoperabilidad de datos:* Describe la capacidad de los datos de ser universalmente accesibles, reutilizables y comprensibles por todas las partes en las transacciones abordando la falta de entendimiento común causada por el uso de diferentes representaciones, diferentes propósitos, diferentes contextos y diferentes enfoques dependientes de la sintaxis.
- *Interoperabilidad de procesos:* Se define como la capacidad de alinear procesos de diferentes entidades (empresas), para que puedan intercambiar datos y realizar negocios de una manera transparente.
- *Interoperabilidad de las reglas:* es la habilidad de las entidades para alinear y hacer coincidir sus reglas legales y de negocios para realizar transacciones automatizadas legítimas que también son compatibles con las reglas internas de operación de negocios entre sí.
- *Interoperabilidad de objetos:* se refiere a la interconexión en red y la cooperación de objetos cotidianos. La interoperabilidad de dispositivos o

componentes de hardware puede considerarse como un caso particular del dominio de interoperabilidad de objetos.

- *Interoperabilidad de los sistemas de software:* Se refiere a la capacidad de un sistema empresarial o un producto para trabajar con otros sistemas o productos empresariales sin el esfuerzo especial de las partes interesadas. Esto se puede lograr con arquitecturas alternativas de TI y soluciones, incluyendo el desarrollo personalizado de API, intermediación orientada a mensajes, implementaciones de arquitectura orientada a servicios o gateways de software independientes.
- *Interoperabilidad cultural:* Es el grado en que el conocimiento y la información están anclados a un modelo unificado de significado a través de las culturas. Los sistemas empresariales que tengan en cuenta los aspectos de interoperabilidad cultural pueden ser utilizados por grupos transnacionales en diferentes lenguas y culturas con el mismo dominio de interés de una manera rentable y eficiente.
- *Interoperabilidad del conocimiento:* es la capacidad de dos o más entidades diferentes para compartir sus activos intelectuales, aprovechar al instante el conocimiento mutuo y utilizarlo, y ampliarlos mediante la cooperación.
- *Interoperabilidad de los servicios:* puede definirse como la capacidad de una empresa para registrar dinámicamente, agregar y consumir servicios compuestos de una fuente externa, como un socio comercial o un proveedor de servicios basado en Internet, de manera uniforme.

Según las métricas aplicadas a los modelos de interoperabilidad se tiene la **Tabla 5**, en la cual:

(+) Indica que el modelo de evaluación de la interoperabilidad ha adoptado un enfoque para este criterio, sin juzgar si este enfoque proporciona cobertura total o parcial para la cuestión.

(–) Se refiere a la falta de un enfoque tangible de esta cuestión

Tabla 5: Comparativa entre Modelos de Interoperabilidad

Evaluación de interoperabilidad	Quantification of interoperability methodology	Military communications and information systems interoperability	Levels of information systems interoperability	Organizational interoperability maturity model for C2	Interoperability assessment methodology	Stoplight	Enterprise interoperability maturity model	The layered interoperability score	Government interoperability maturity matrix	System-of-systems Interoperability
Interoperabilidad de datos	+	+	+	+	+	+	+	+	+	+
Interoperabilidad de procesos	–	–	+	+	–	+	+	+	+	–
Interoperabilidad de las reglas	–	–	+	+	–	–	+	+	–	–
Interoperabilidad de Objetos	–	+	+	+	+	–	+	+	–	–
Interoperabilidad de Sistemas de Software	+	+	+	+	+	+	+	+	+	+
Interoperabilidad Cultural	+	–	–	+	–	–	+	–	–	–
Interoperabilidad del Conocimiento	–	–	+	+	–	–	+	–	+	–
Interoperabilidad de los servicios	–	+	+	–	–	+	+	–	+	+

Fuente: El autor.
Elaboración: El autor.

Según las comparaciones realizadas se tomó como puntos fuertes las siguientes métricas:

- Interoperabilidad de Datos
- Interoperabilidad de Sistemas de Software
- Interoperabilidad de los Servicios

Se selecciona estas métricas, por lo que las mismas están estrechamente relacionadas con aspectos técnicos de la aplicación a desarrollar, como son: el acceso a los datos (Interoperabilidad de datos), capacidad para trabajar con otros sistemas mediante APIs o Gateways (Interoperabilidad de los sistemas de software), el consumo de servicios de una fuente externa (Interoperabilidad de los servicios).

Por otra parte, las que no poseen tanto peso, tienen más relación con el entorno empresarial, éstas son: la Interoperabilidad de procesos, Interoperabilidad de las reglas, Interoperabilidad cultural, e Interoperabilidad del conocimiento; y finalmente una relacionada con la interoperabilidad a nivel de hardware como es la Interoperabilidad de objetos; se las excluyó, ya que la aplicación a desarrollar está enfocada a ser una aplicación pequeña, por lo que no se utiliza hardware considerable y además no interviene el entorno empresarial.

Según el análisis realizado, se ha decidido utilizar los modelos LISI (Levels of information systems interoperability), SOSI (System-of-systems Interoperability) y Stoplight ya que cumplen con los requisitos mencionados y además no son estrictamente para un entorno empresarial, como algunos de los otros modelos (ver **Anexo B**: Selección de modelo de Interoperabilidad**).**

Todos los patrones descritos en el apartado 1.2.1, pueden ser implementados para garantizar estas 3 métricas de interoperabilidad, para este caso se usará los patrones *API Gateway, Server-side Discovery, Single Service per Host.*

CAPITULO 2
MODELOS DE EVALUACION DE INTEROPERABILIDAD

En busca de un modelo o combinación de modelos que permita evaluar la interoperabilidad en Microservicios, en el apartado **1.3.11** se realizó un análisis de los modelos que nos permiten realizar esta evaluación, por lo que se decidió utilizar los modelos LISI (Levels of information systems interoperability), SOSI (System-of-systems Interoperability) y Stoplight; en este capítulo se describe a detalle dichos modelos (definición, características, que y como mide la interoperabilidad) y se selecciona lo que se implementará de cada uno, para finalmente proponer una combinación de los mismos, que nos permita realizar una evaluación de la interoperabilidad según nuestras necesidades.

2.1. Modelo LISI para evaluar interoperabilidad en MS

2.1.1 Definición

El Modelo de Referencia de Niveles de Interoperabilidad de Sistemas de Información (LISI en inglés) es un procedimiento para definir, medir, evaluar y certificar el grado de interoperabilidad requerido o logrado por y entre organizaciones o aplicaciones (Rezaei et al., 2014).

Best Manufacturing Practices (2008) menciona además que este modelo ampliado para incluir definiciones detalladas de capacidades, opciones y criterios de implementación, puede soportar rigurosas evaluaciones de interoperabilidad de sistemas y evaluaciones comparativas.

2.1.2 Características

Según Best Manufacturing Practices (2008) y Kasunic & Anderson, (2004) el Modelo de Referencia LISI, posee las siguientes características:

- Facilitar una comprensión común de la interoperabilidad y sus facilitadores en cada nivel de sofisticación de la interacción sistema-sistema (Best Manufacturing Practices, 2008).
- Facilitar una comprensión común de la interoperabilidad y el conjunto de capacidades que permite cada nivel lógico de interacción sistema-sistema (Kasunic & Anderson, 2004).
- Transforma los niveles de interoperabilidad en capacidades requeridas (procedimientos, aplicaciones, infraestructura, datos) que forman la base para hacer comparaciones entre sistemas heterogéneos (Best Manufacturing Practices, 2008).

- Proporciona un modelo de madurez de interoperabilidad y las capacidades necesarias asociadas como base para realizar comparaciones entre sistemas heterogéneos y sistemas individuales maduros (Kasunic & Anderson, 2004).
- Proporcionar una metodología para evaluar y mejorar la interoperabilidad guiando los requisitos y el análisis de la arquitectura, el desarrollo de sistemas, la adquisición, el campo y la inserción tecnológica (Kasunic & Anderson, 2004).

2.1.3 ¿Qué mide?

Se centra en mejorar los niveles de interoperabilidad de los sistemas, permitiendo evaluar la interoperabilidad de los sistemas, es además una disciplina (Kasunic & Anderson, 2004).

LISI extiende la definición de interoperabilidad más allá de la capacidad de mover datos de un sistema a otro - considera la capacidad de intercambiar y compartir servicios entre sistemas. LISI se enfoca en aumentar los niveles de sofisticación para la interacción de sistema a sistema; es decir, umbrales de capacidades que los sistemas exhiben a medida que mejoran su capacidad para interactuar con otros sistemas (Best Manufacturing Practices, 2008). Las capacidades específicas necesarias para alcanzar cada nivel se describen en términos de cuatro atributos: procedimientos, aplicaciones, infraestructura y datos, que están representados por el acrónimo "PAID".

LISI evalúa el nivel de interoperabilidad alcanzado entre sistemas (no entre usuarios). Una vez que los problemas de interoperabilidad de sistema a sistema se han aislado, la capacidad de abordar los problemas de interoperabilidad de los usuarios está muy mejorada (Kasunic & Anderson, 2004).

2.1.4 ¿Cómo mide?

LISI presenta una estructura lógica y una disciplina o "modelo de madurez" para mejorar la interoperabilidad de forma incremental entre sistemas de información (Best Manufacturing Practices, 2008).

El modelo LISI proporciona una línea de base de umbrales de capacidad (PAID), además según menciona Best Manufacturing Practices (2008) se encarga de proporcionar el vocabulario y el marco común necesarios para discutir la interoperabilidad entre sistemas.

2.1.4.1 Niveles de LISI:

El Modelo LISI está orientado por niveles, que representan grados de sofisticación cada vez mayores, requeridos para lograr interacciones entre sistemas de información (Best Manufacturing Practices, 2008). Se definen cinco niveles, en la

Tabla 6 se presenta una visión general del *Modelo de Madurez de Interoperabilidad de LISI*. Este modelo identifica y resume los niveles, a través de los cuales un sistema debe progresar lógicamente para mejorar sus capacidades de interoperar.

Tabla 6: Visión general del modelo de madurez de interoperabilidad LISI.

Nivel	Intercambio de Información
4 Empresa Manipulación Interactiva: Datos compartidos y aplicaciones	Información y aplicaciones globales distribuidas Interacciones simultáneas con datos complejos Colaboración avanzada Actualización de la base de datos global activada por eventos
3 Dominio Datos compartidos; Aplicaciones "separadas"	Bases de datos distribuidas. Colaboración Sofisticada, ejemplo: Cuadro Operativo Común
2 Funcional Funciones comunes mínimas; Datos y aplicaciones separados.	Intercambio de productos heterogéneo Colaboración básica Colaboración de grupo. Por ejemplo: intercambio de imágenes anotadas, mapas con superposiciones.
1 Conectado Conexión electrónica; Datos y aplicaciones separados	Intercambio heterogéneo de productos Colaboración básica. Ejemplo: voz FM, Enlaces de datos tácticos, archivos de texto, transferencias, mensajes, correo
0 Aislado No conectado	Gateway Manual. Ejemplo: disquete, cinta, intercambio de impresoras

Fuente: (Kasunic & Anderson, 2004)
Elaboración: El autor

2.1.4.2 Atributos PAID

LISI categoriza los factores que influyen en la capacidad de interoperabilidad de los sistemas de información, en cuatro atributos clave que comprenden el dominio de la interoperabilidad: Procedimientos, Aplicaciones, Infraestructura y Datos (Best Manufacturing Practices, 2008). Estos atributos, denominados colectivamente como PAID, abarcan toda la gama de consideraciones de interoperabilidad. Ayudan a definir los conjuntos de características para el intercambio de servicios en cada nivel de sofisticación. La prescripción de capacidades de cada nivel debe cubrir los cuatro atributos habilitantes de interoperabilidad denominados PAID, siendo crítica para mover la interoperabilidad más allá de la simple conexión entre sistemas.

- *Procedimientos:* Según Rezaei (2014) aquí se incluyen numerosas formas de controles operativos y directrices documentadas que influyen en todos los aspectos de la integración del sistema, el desarrollo y la funcionalidad operacional. Además se refieren a la guía y estándares de arquitectura, políticas y procedimientos y doctrina que permiten el intercambio de información entre sistemas. Se trata de enfocarse en las muchas formas de guía que impactan en la interoperabilidad del sistema, incluyendo doctrina, misión, arquitecturas y estándares (Best Manufacturing Practices, 2008).
- *Aplicación:* Rezaei (2014) y Kasunic & Anderson (2004) indican que incluyen la misión del sistema, que es el propósito fundamental de la construcción del sistema y los requisitos funcionales del sistema. Estos atributos indican aplicaciones que permiten procesar, intercambiar y manipular, además representan los aspectos funcionales del sistema. Estas funciones se manifiestan en los componentes de software del sistema, desde los procesos individuales hasta las suites de aplicaciones integradas (Best Manufacturing Practices, 2008).
- *Infraestructura:* Best Manufacturing Practices (2008) y Rezaei (2014) Soporta el establecimiento y uso de una conexión entre aplicaciones o sistemas, estos atributos incluyen los entornos que permiten la interacción, tales como servicios del sistema, redes, hardware, etc. Es necesaria para apoyar las operaciones de los sistemas, la cual contiene cuatro subcomponentes que también se definen en términos de niveles crecientes de sofisticación (Kasunic & Anderson, 2004).
- Los atributos de *Datos* según Rezaei (2014) se centran en los procesos de información del sistema y contienen tanto el formato de datos (sintaxis) como su contenido o significado (semántica). Estos atributos de datos de interoperabilidad incluyen protocolos y formatos que permiten la información y

los intercambios de datos, aquí se incluye los formatos y estándares de datos que soportan la interoperabilidad en todos los niveles. Incorpora toda la gama de estilos y formatos de texto simple a modelos de datos empresariales (Best Manufacturing Practices, 2008). Finalmente Kasunic & Anderson (2004) agregan que son las estructuras de datos e información utilizadas para soportar tanto las aplicaciones funcionales como la infraestructura del sistema.

En la

Tabla 7 se presenta una relación entre los niveles de interoperabilidad con los atributos PAID, permitiendo conocer la relación general entre los mismos.

Tabla 7: PAID y niveles de interoperabilidad

	PROCEDIMIENTOS	**APLICACIONES**	**INFRAESTRUCTURA**	**DATOS**
NIVEL 0	El sistema tiene procedimientos establecidos localmente que rigen el control de acceso. Un usuario debe acceder directamente al sistema para compartir información con otros sistemas.	Funcionalmente independientes en la mayoría de los sistemas aislados. Los datos resultantes son importantes pero la capacidad de manipular consistentemente esos datos no entra en juego.	Principalmente independiente entre sistemas. La mayoría del intercambio de información es por acceso físico. Como máximo, un sistema aislado puede intercambiar datos por medios físicos comunes tales como discos o cintas.	Modelos de datos privados
NIVEL 1	Más allá del control de acceso simple, la mayoría de las cuales aún se relacionan principalmente con políticas locales o de nivel de sitio.	Independientes entre sistemas, pero utilizan controladores e interfaces comunes.	Soportar conexiones simples de peer to peer para permitir la transferencia de datos local consistente con los procedimientos locales establecidos.	Pueden existir modelos de datos locales, pero suelen ser específicos de un programa en particular. Informes simples o gráficos son un ejemplo.
NIVEL 2	Centrarse en el nivel del programa individual, especificando las implementaciones que los programas deben soportar.	Incluyen la automatización de escritorio y la capacidad de intercambiar algunos datos estructurados. Los programas de automatización de oficinas son un ejemplo. Las interfaces web son significativas.	Interactúan con otros sistemas en el área local a través de LANs. Estas LAN pueden utilizar protocolos (como TCP / IP) que soporten redes de área extensa.	Es posible que existan estructuras de datos avanzadas, pero todavía apoyan principalmente aplicaciones individuales (modelos de datos de programa). Cada vez son más comunes los formatos de datos entre los programas.
NIVEL 3	Se centran en la interacción del dominio donde un dominio puede abarcar muchas áreas geográficas, pero se centra en un área funcional (C2, inteligencia, logística).	Avanzado más allá de los programas individuales, se admite la capacidad básica de colaboración de grupo, como el seguimiento de las revisiones de documentos o la gestión del flujo de	Las redes de infraestructura son globales. En este nivel, la interacción tiene lugar en partes del espacio global de información, aunque no en todas.	Los *datos* definidos modelos de datos existen y se entienden entre las aplicaciones, sin embargo, sólo representan un dominio particular.

		trabajo.		
NIVEL 4	Procedimientos a nivel de empresa, basados en el entendimiento a nivel empresarial de tareas.	Integradas en el espacio de información distribuido común. Varios usuarios pueden acceder a las mismas instancias de datos de toda la empresa.	Redes globales que soportan topologías multidimensionales. Estas redes pueden tener diferentes áreas basadas en seguridad o control de acceso, pero están integradas apropiadamente para soportar las necesidades de los usuarios.	Modelos de datos empresariales apoyan la integración de aplicaciones. Hay una comprensión común de los datos en toda la empresa.

Fuente: Best Manufacturing Practices (2008)
Elaboración: El autor

Rezaei (2014) indica el modelo LISI (ver **Tabla 8**) en el que los cinco niveles de interoperabilidad se ilustran en filas y cuatro columnas, demostrando que los atributos de LISI contienen Procedimientos, Aplicaciones, Infraestructura y Datos (PAID). La amplia clasificación de intersecciones de nivel / atributo facilita el abordaje de las capacidades específicas requeridas.

Tabla 8: El modelo LISI

<table>
<tr><th colspan="3" rowspan="2">NIVEL (Ambiente)</th><th colspan="4">Atributos de Interoperabilidad</th></tr>
<tr><th>Procedimientos</th><th>Aplicaciones</th><th>Infraestructura</th><th>Datos</th></tr>
<tr><td rowspan="3">Nivel Empresa</td><td rowspan="3">4</td><td>c</td><td>Empresas Multinacionales</td><td rowspan="2">Interactivo (aplicaciones cruzadas)</td><td rowspan="3">Topologías Multidimensionales</td><td rowspan="2">Modelo interempresarial</td></tr>
<tr><td>b</td><td>Empresa intergubernamental</td></tr>
<tr><td>a</td><td>Empresa DoD</td><td>Objeto completo (Cortar y pegar)</td><td>Modelo Empresa</td></tr>
<tr><td rowspan="3">Nivel de Dominio</td><td rowspan="3">3</td><td>c</td><td rowspan="3">Dominio
Servicio / Agencia
Doctrina, Procedimientos, Formación, etc.</td><td>Datos Compartidos (Situación Muestra los Intercambios Directos de DB)</td><td rowspan="3">WAN</td><td>DBMS</td></tr>
<tr><td>b</td><td>Colaboración de grupo (Tableros blancos, VTC)</td><td rowspan="2">Modelos de Dominio</td></tr>
<tr><td>a</td><td>Texto completo (Cortar y pegar)</td></tr>
<tr><td rowspan="3">Nivel funcional</td><td rowspan="3">2</td><td>c</td><td rowspan="2">Entorno operativo común (DIICOE Level5) Conformidad</td><td>Navegador web</td><td rowspan="2">LAN</td><td rowspan="3">Modelos del programa Y formatos de datos avanzados</td></tr>
<tr><td>b</td><td>Operaciones básicas (Documentos, Mapas, Informes, Imágenes Hojas de cálculo, Datos)</td></tr>
<tr><td>a</td><td>Programa Estándar Procedimientos, Entrenamiento, etc.</td><td>Adv. Mensajería (Parsers, E-Mail +)</td><td>Redes</td></tr>
<tr><td>Nivel Conectado</td><td>1</td><td>d</td><td>Queja de Normas (JTA, IEEE)</td><td>Mensajería básica</td><td>Dos vías</td><td>Formatos de Datos</td></tr>
</table>

				(Texto sin formato, E-mail sin adjuntos)		Básicos
		c		Transferencia de archivos de datos		
		b	Perfil de seguridad	Interacción Simple (Charla del texto, voz, fax, alejado, Acceso, Telemetría)		
		a			Una vía	
Nivel Aislado	**0**	**d**	Procedimientos de Intercambio de Medios	N/A	Media removible	Formatos de medios
		c	Control de acceso manual: Nato Nivel 3		Reentrada Manual	Datos privados
		b	Control de acceso manual: Nato Nivel 2			
		a	Control de acceso manual: Nato Nivel 1			
		0	Interoperabilidad no conocida			

Fuente: (Rezaei et al., 2014)
Elaboración: El autor

2.1.4.3 Métricas

La métrica LISI representa cuantitativamente el *grado de interoperabilidad* obtenido de los sistemas. Existen varios estilos de métrica para la interoperabilidad del modelo LISI basados en la naturaleza, el objetivo y la estrategia que utilizan para realizar la comparación y mostrar los resultados. En la **Tabla 9** se muestra una configuración típica de varias opciones disponibles para la explicación de las métricas LISI.

Tabla 9: Métricas de interoperabilidad del Modelo LISI

Tipo de Métrica	Niveles	Sub-Niveles
G= Genérico E= Esperado S= Especifico	4= Empresa 3= Dominio 2= Funcional 1= Conectado 0= Aislado	Varía según los niveles
Nivel LISI (Forma Corta) G2 **Nivel LISI (con sub-nivel) G2b**		

Fuente: (Rezaei et al., 2014)
Elaboración: El autor

Como se describe en la **Tabla 9**, hay tres tipos de métricas LISI basadas en tres tipos de relaciones que se miden. La distinción principal entre estos tres tipos es la comparación de un sistema único con el modelo LISI (genérico) y los dos casos diferentes donde dos o más sistemas se comparan entre sí (esperados y específicos) estos tres tipos de métricas tratan sobre:

- *Nivel genérico de interoperabilidad:* Se calcula para sistemas individuales y se expresa como un valor calculado matemáticamente, haciendo una comparación entre un sistema único por un lado y el modelo de capacidades LISI por el otro. Está determinado por el nivel más alto en el modelo de capacidades LISI donde se implementan todas las capacidades PAID (sin dependencia de ninguna alternativa de implementación), esto implica la necesidad de que un sistema tenga implementaciones para cada capacidad individual a través de los atributos PAID (Rezaei et al., 2014).
- *Nivel esperado de interoperabilidad:* Se define como el nivel genérico más bajo de ambos sistemas, es decir, el nivel donde se espera la interoperabilidad de ambos sistemas entre sí. Este nivel esperado se especifica sobre la base de que cualquiera de los dos sistemas debe ser capaz de interoperar a un cierto nivel en el caso de que cada uno de ellos posea el conjunto de capacidades genéricas necesarias para lograr los tipos de intercambio de información de ese nivel (Rezaei et al., 2014).
- *Nivel específico de interoperabilidad:* Es el valor métrico calculado entre los dos sistemas, resultantes de la comparación entre las alternativas de implementación que cada uno de los sistemas ha utilizado con respecto a las capacidades registradas. El nivel específico es el nivel más alto en el que dos sistemas realizan implementaciones interoperables documentadas a través de todos los aspectos de PAID. Puede diferir del nivel esperado debido a la adición de elementos a las tablas de opciones de LISI y / o consideraciones de criterios diferentes de implementación técnica (Polyakov & Ksenofontov, 2001).

Entre los productos de evaluación LISI más importantes según Rezaei (2014) tenemos:

- La *herramienta de recopilación de datos de interoperabilidad*: para reunir la información pertinente necesaria y evaluar la interoperabilidad de los sistemas de información, el modelo LISI utiliza un Cuestionario de Interoperabilidad.
- *Perfiles de interoperabilidad*: los datos recogidos por el Cuestionario LISI se asignan a la plantilla del Modelo de Capacidades LISI utilizando los perfiles de Interoperabilidad.

Las opciones de implementación son, por lo tanto, capturadas por los perfiles para cada una de las capacidades PAID existentes en el sistema o sistemas que están bajo evaluación según Rezaei (2014); por supuesto, en un formato que sea capaz de facilitar la comparación requerida en las escalas de sistema a nivel así como de

sistema a sistema, y las métricas del sistema se derivan de los perfiles. El perfil de interoperabilidad de un sistema se ha representado en el **Anexo C**: Ejemplo de perfil de interoperabilidad de los sistemas, como un ejemplo ficticio. En este ejemplo, el nivel de interoperabilidad genérico del sistema es 2c, el nivel más alto en el que se implementa una capacidad para cada uno de los atributos PAID (Rezaei et al., 2014).

Kasunic & Anderson (2004) mencionan que LISI se aplica a lo largo del ciclo de vida del sistema de información, desde el análisis de requisitos hasta el desarrollo de sistemas, la adquisición, el campo y la posterior mejora y modificación. Específicamente, el Modelo de Referencia de LISI está diseñado para apoyar el desarrollo y el análisis de las arquitecturas, ayudando a identificar, al principio, problemas, brechas y deficiencias que pueden estar presentes en cualquier arquitectura de tecnología de la información.

2.1 Modelo Stoplight para evaluar interoperabilidad en MS

2.2.1 Definición

Es un modelo simple llamado modelo de semáforo (**Tabla 10**) que asigna un código de un solo color a un sistema, es posiblemente el primer modelo para abordar directamente la relación entre los requisitos y la interoperabilidad. (Metrics et al., 2007).

Tabla 10: Modelo Stoplight

		Cumple con los requisitos de adquisición?	
		SI	NO
Cumple con los requisitos operativos?	SI	Verde	Amarillo
	NO	Naranja	Rojo

Fuente: (Rezaei et al., 2014)
Elaboración: El autor

2.2.2 Características

Según las consideraciones de Metrics et al. (2007) y Rezaei et al. (2014) se extraen las siguientes características:

- Modelo simple y sencillo de usar.
- Matriz de intersecciones de 2 parámetros (requisitos de adquisición y requisitos operativos).
- Para cada requisito hay 2 estados (Cumple o no cumple).
- Solo tiene 4 posibles estados (Verde, Naranja, Amarillo y Rojo).

2.2.3 ¿Qué mide?

Con el objetivo de ayudar a los responsables de la toma de decisiones determina si el sistema que utilizan puede satisfacer los requisitos operativos y de interoperabilidad de la adquisición (Metrics et al., 2007).

2.2.4 ¿Cómo mide?

Rezaei et al., (2014) menciona que "la interoperabilidad es notoriamente difícil de medir", aun así, introdujeron este modelo para medirlo. Como matriz, se ha diseñado de tal manera que "se cumple con los requisitos operativos (sí / no)" se muestra en

filas y "se cumple con los requisitos de adquisición (sí / no)" aparece en las columnas de la matriz. Las intersecciones matriciales están en rojo, amarillo, naranja y verde. La asignación de este conjunto jerárquico de colores depende de cuán bien se cumpla cada requisito específico. Además indican un ejemplo de una codificación de color (**Tabla 11**) que puede adaptarse en un cronograma para representar las futuras mejoras de la interoperabilidad del plan (Rezaei et al., 2014)

Tabla 11: Definición e implicaciones del modelo Stoplight.

Verde	El sistema cumple su conjunto de requisitos de interoperabilidad y no tiene problemas de interoperabilidad conocidos	Sistema de campo sin problemas conocidos que cumpla con todos los requisitos documentados.
Amarillo	El sistema no cumple con su conjunto de requisitos de interoperabilidad, pero no tiene problemas de interoperabilidad conocidos.	Los requisitos documentados no reflejan el uso operacional del sistema.
Rojo	El sistema no cumple con su conjunto de requisitos de interoperabilidad, pero no tiene problemas de interoperabilidad conocidos	Mejora, migración y / o planes de acción deben ser puestos en marcha
Naranja	El sistema cumple su conjunto de requisitos de interoperabilidad, pero tiene problemas de interoperabilidad conocidos	Revisar los requisitos y determinar si los requisitos son adecuados

Fuente: (Rezaei et al., 2014)
Elaboración: El autor

2.3 Modelo SOSI para evaluar interoperabilidad en MS

2.3.1. Definición

Levine et al. (2003) menciona que es un modelo simple que representa la amplia gama de actividades que son necesarias para lograr la interoperabilidad:

- *Gestión del programa:* Son actividades realizadas para gestionar la adquisición de un sistema, se centra en los contratos, incentivos y prácticas tales como la gestión de riesgos. (Meyers, Levine, Morris, Place, & Plakosh, 2004)
- *Construcción del Sistema:* Son las actividades realizadas para crear y mantener un sistema, el foco está en la arquitectura, las normas y los estándares.
- *Sistema Operacional:* Son las actividades realizadas para operar un sistema, se centra en las interacciones con otros sistemas, con las personas y la distribución de datos. El usuario final se considera parte del sistema operacional.

2.3.2. Características

Levine et al. (2003) y Meyers, Levine, Morris, Place, & Plakosh (2004) concuerdan que:

- Modelo Simple pero robusto.
- Mide específicamente 3 tipos de interoperabilidad (no abordados por los otros modelos).
- Sugieren que el concepto de interoperabilidad Backplane es necesario.

2.3.3. ¿Qué mide?

Levine et al. (2003) menciona que este modelo mide 3 tipos de interoperabilidad en específico (no se los describe, ni se los coloca en la **Tabla 5,** por el motivo de que ***solo*** este modelo cumple con la medición de los mismos):

- **Interoperabilidad programática:**

Son las actividades relacionadas con la gestión de un programa en el contexto de otro programa. Normalmente, los programas se gestionan en un aislamiento cómodo, con poca necesidad de considerar las funciones de gestión de otros programas. Como

resultado, la interoperabilidad se define típicamente por una especificación común y se logra a través de los esfuerzos del *personal técnico* (Levine et al., 2003).

Debido a las limitaciones de las especificaciones y a la falta de incentivos para que los programas puedan examinar más allá de ellas, la interoperabilidad resultante suele ser menor de lo que se desea. Para remediar esta situación, se necesitan enfoques y técnicas de gestión que permitan superar las brechas entre los programas y las perspectivas aisladas.

El logro de la interoperabilidad programática exige una gestión del riesgo coherente y conjunta (tanto para el hardware como para el software) entre las Oficinas de Gestión del Programa (Program Management Offices) (PMO), en donde el riesgo será compartido, y distribuido, a través de los programas.

- **Interoperabilidad Constructiva:**

La interoperabilidad constructiva se refiere a las actividades relacionadas con la construcción y el mantenimiento de un sistema en el contexto de otro sistema. La interoperabilidad constructiva incluye el uso común de arquitectura, estándares, especificaciones de datos, protocolos de comunicación, y lenguajes para construir sistemas interoperables. Levine et al. (2003) menciona que dos factores limitan las concepciones actuales de la interoperabilidad constructiva, en primer lugar, una suposición común sostiene que la atención a tales mecanismos (arquitectura, normas, etc.) por sí solo dará lugar a la interoperabilidad entre sistemas. En segundo lugar, estos mismos mecanismos captan sólo parte de la rica semántica que debe ser compartida para la interacción de sistemas sofisticados.

Lograr una interoperabilidad constructiva exige nuevas perspectivas sobre el uso de estándares y arquitecturas de software. El simple uso de una norma no garantizará la interoperabilidad del sistema. La definición conjunta de las normas y de la arquitectura del sistema proporcionará un aspecto crítico para la construcción de cada sistema.

- **Interoperabilidad Operacional:**

La interoperabilidad operacional se refiere a las actividades relacionadas con el funcionamiento de un sistema en el contexto de otros sistemas. Estas actividades incluyen:

- Doctrina que rige la forma en que se utiliza el sistema.

- Convenciones sobre la forma en que el usuario interpreta la información derivada de los sistemas de interoperación (es decir, la semántica de la interoperación).

- Estrategias para capacitar al personal en el uso de sistemas interoperables.

Levine et al. (2003) hace énfasis en que el logro de la interoperabilidad operacional exige la imposición de requisitos en el contexto más amplio, el software asociado con cada sistema individual debe satisfacer muchos de estos requisitos.

- **Interoperabilidad Backplane:**

La visión de interoperabilidad entre PMOs es extensible a un sistema que cruza los límites de la Oficina Ejecutiva del Programa (Program Executive Office) (PEO) a través del uso de backplanes programáticos, constructivos y operativos. Estos backplanes definen un conjunto consistente de prácticas y técnicas que conducen a una interoperación exitosa que puede ser empleada para cualquier número de sistemas y PEOs (Levine et al., 2003).

2.3.4. ¿Cómo mide?

Cuando consideramos la interacción entre más de 2 programas, una premisa clave del trabajo de SOSI según Levine et al.(2003) que indica que para tener interoperabilidad entre sistemas operacionales, se debe introducir y abordar el alcance completo de la interoperabilidad entre las organizaciones que participan en la adquisición de sistemas. La interoperabilidad entre múltiples sistemas requiere el desarrollo de la interoperabilidad entre las PMO y los sistemas que controlan. Esta interoperabilidad se logrará más eficazmente mediante la interoperación en la gestión del programa, la construcción del sistema y los niveles operacionales. Cada dimensión representa un tipo de interoperabilidad.

Para lograr estos backplanes de interoperabilidad, cada sistema debe ser visto como una unidad. Este punto de vista contrasta con la práctica actual, en la que el incentivo del administrador del programa está vinculado a un sistema particular, con una atención mínima al contexto más amplio en el que el sistema residirá. Las actividades que damos por sentadas como parte del desarrollo del sistema (por ejemplo, Gestión de requisitos, definición de la arquitectura, procesos de desarrollo y gestión y definición de la semántica operativa) también deben realizarse en el sistema.

En el modelo SOSI, las cuestiones programáticas, constructivas y operacionales deben ser gestionadas a lo largo del ciclo de vida.

2.4 Comparación de modelos de Interoperabilidad seleccionados

Después de exponer los modelos para evaluar la interoperabilidad seleccionados, se realiza un análisis comparativo de los mismos (similar al realizado en la **Tabla 5**), en la **Tabla 12** en el cual se observa que aspectos de la interoperabilidad son considerados por cada modelo; teniendo así aspectos en común, como también únicos pertenecientes a cada modelo que no son cubiertos por los demás.

Tabla 12: Comparativa entre Modelos de Interoperabilidad seleccionados

Evaluación de interoperabilidad	Levels of information systems interoperability	Stoplight	System-of-systems Interoperability
Interoperabilidad de datos	+	+	+
Interoperabilidad de Sistemas de Software	+	+	+
Interoperabilidad de los servicios	+	+	+
Interoperabilidad Programática	-	-	+
Interoperabilidad Constructiva	-	-	+
Interoperabilidad Operacional	-	-	+
Interoperabilidad Backplane	-	-	+
Relación existente entre Requisitos con Interoperabilidad	-	+	-

Fuente: El autor
Elaboración: El autor

CAPITULO 3:
MODELO PROPUESTO PARA LA EVALUACION DE INTEROPERABILIDAD EN MICROSERVICIOS

En el presente capítulo se presenta una propuesta para el modelo de evaluación de interoperabilidad, tomando como referencia el capítulo anterior, en el cual se analizaron los modelos que permiten evaluar la interoperabilidad, permitiendo realizar una combinación de los mismos.

3.1 Propuesta del modelo para evaluación de interoperabilidad en Microservicios

3.1.1 Justificación

Todos los modelos descritos en el capítulo 2 proporcionan una representación *parcial* de algún aspecto, característica o dimensión para evaluar la interoperabilidad; por lo tanto es necesario implementar una combinación de los modelos, que nos permita realizar una evaluación de la interoperabilidad que abarque lo más completamente posible nuestras necesidades, según las aplicaciones desarrolladas.

3.1.2 Elementos

A continuación se expone lo extraído de cada modelo analizado (LISI, Stoplight, SOSI), y lo que se utilizará para evaluar la interoperabilidad de las aplicaciones; se toma como referencia la **Tabla 12** y el **Anexo B**: Selección de modelo de Interoperabilidad:

- **LISI:** De este se extrae el Modelo de Madurez de Interoperabilidad para clasificar el nivel que posee la aplicación, además los atributos PAID que son las capacidades específicas necesarias para alcanzar cada nivel, abarcando así una amplia gama de consideraciones de interoperabilidad, con ese objetivo se usará la plantilla de la **Tabla 8**, y las métricas de la **Tabla 9**.
- **Stoplight:** De este modelo se realiza una combinación respecto a los que se expone en la matriz (**Tabla 10**) y la tabla de especificación de colores (**Tabla 11**).
- **SOSI:** Se usara lo relacionado a la *interoperabilidad constructiva* (se lo divide en 5 ítems) que corresponde a las actividades que se debe tener en cuenta para crear y mantener un sistema, la misma que indica que se debe hacer uso de una arquitectura, estándares, especificaciones de datos, protocolos de comunicación, y lenguajes para construir sistemas interoperables.

En base a lo extraído y analizado de cada modelo se propone la **Tabla 13**, la cual será utilizada para evaluar la interoperabilidad existente en las aplicaciones.

Tabla 13: Propuesta de modelo de evaluación

<table>
<tr><th>MODELO</th><th>NUMERO DE ITEM</th><th colspan="6">DETALLE ITEM</th><th>RESULTADO</th></tr>
<tr><td rowspan="15">LISI</td><td rowspan="15">Ítem 1</td><td colspan="3">NIVEL (Ambiente)</td><td colspan="4">Atributos de Interoperabilidad</td><td rowspan="15"></td></tr>
<tr><td colspan="3"></td><td>Procedimientos</td><td>Aplicaciones</td><td>Infraestructura</td><td>Datos</td></tr>
<tr><td rowspan="3">Nivel Empresa</td><td rowspan="3">4</td><td>c</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td><td></td></tr>
<tr><td>b</td></tr>
<tr><td>a</td><td></td></tr>
<tr><td rowspan="3">Nivel de Dominio</td><td rowspan="3">3</td><td>c</td><td rowspan="3"></td><td></td><td rowspan="3"></td><td></td></tr>
<tr><td>b</td><td></td></tr>
<tr><td>a</td><td></td><td></td></tr>
<tr><td rowspan="3">Nivel funcional</td><td rowspan="3">2</td><td>c</td><td rowspan="3"></td><td></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>b</td><td></td></tr>
<tr><td>a</td><td></td><td></td></tr>
<tr><td rowspan="4">Nivel Conectado</td><td rowspan="4">1</td><td>d</td><td rowspan="3"></td><td></td><td rowspan="4"></td><td rowspan="4"></td></tr>
<tr><td>c</td><td></td></tr>
<tr><td>b</td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td>a</td><td></td></tr>
<tr><td rowspan="5">Nivel Aislado</td><td rowspan="5">0</td><td>d</td><td colspan="2"></td><td rowspan="5"></td><td></td><td></td></tr>
<tr><td>c</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>b</td></tr>
<tr><td>a</td></tr>
<tr><td>0</td><td colspan="4">Interoperabilidad no conocida</td></tr>
<tr><td rowspan="3">Stoplight</td><td rowspan="3">Ítem 1</td><td colspan="2"></td><td colspan="4">Cumple con los requisitos de adquisición?</td><td rowspan="3"></td></tr>
<tr><td colspan="2"></td><td colspan="2">SI</td><td colspan="2">NO</td></tr>
<tr><td rowspan="2">Cumple con los requisitos operativos?</td><td>SI</td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td>NO</td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td rowspan="5">SOSI</td><td>Ítem 1</td><td>Arquitectura</td><td colspan="6"></td><td rowspan="5"></td></tr>
<tr><td>Ítem 2</td><td>Implementación estándares</td><td colspan="6"></td></tr>
<tr><td>Ítem 3</td><td>Especificaciones/formato de datos</td><td colspan="6"></td></tr>
<tr><td>Ítem 4</td><td>Protocolos de comunicación</td><td colspan="6"></td></tr>
<tr><td>Ítem 5</td><td>Patrones/Lenguajes</td><td colspan="6"></td></tr>
</table>

Fuente: El autor
Elaboración: El autor

3.1.3 Proceso de Medición

3.1.3.1 Entrada

Se tiene las aplicaciones desarrolladas y de las cuales se necesita evaluar el nivel de interoperabilidad que poseen, por lo tanto cada aplicación es una entrada.

3.1.3.2 Proceso

Consiste en aplicar el modelo propuesto en la **Tabla 13** a cada aplicación para identificar la interoperabilidad existente en cada una.

3.1.3.3 Salida

Luego de realizar la evaluación de la interoperabilidad a cada aplicación según el modelo propuesto; por cada modelo seleccionado (LISI, Stoplight, SOSI) son posibles los siguientes resultados (como se observa en la **Tabla 14**), cabe que mencionar que por LISI y Stoplight solo es posible 1 resultado, mientras que en SOSI por su parte pueden ser varios a la vez. Además el resultado obtenido en SOSI, está realizado según algunas consideraciones mencionadas en el apartado **1.1.2.3.**

Tabla 14: Resultados según modelo propuesto

	RESULTADOS	EJEMPLO
LISI	**Métrica:** - Genérico (G) - Esperado (E) - Especifico (S) **Nivel:** - 4= Empresa - 3= Dominio - 2= Funcional - 1= Conectado - 0= Aislado **Subnivel** - Letras desde la “a” hasta la “d” (según lo permita el nivel)	G3b
Stoplight	Colores: - Verde - Amarillo - Rojo - Naranja	Color Rojo
SOSI	Medida potencial de interoperabilidad: - Aislado - Inicial - Ejecutable - Conectable - Interoperable Interoperabilidad Técnica Interoperabilidad Sintáctica Interoperabilidad Semántica Interoperabilidad Organizacional	*Medida potencial de interoperabilidad:* Conectable Interoperabilidad Técnica Interoperabilidad Semántica

Fuente: El autor
Elaboración: El autor

CAPITULO 4:
DISEÑO, IMPLEMENTACION y VALIDACION

En este capítulo se presenta el proceso de implementación de los prototipos los cuales son una aplicación web y los cambios (tecnología, características, etc.) que va tomando desde una arquitectura monolítica hasta conseguir una aplicación de arquitectura de MS; para finalmente realizar la aplicación del modelo a las aplicaciones, obtener los resultados de la interoperabilidad y realizar las respectivas comparaciones.

4.1 Aplicaciones desarrolladas.

Para la implementación se tomó como base a una aplicación web desarrollada de forma individual denominada *"Reservas de citas* médicas"; para el desarrollo se ha tomado en cuenta el criterio de Santis, Florez, Nguyen, & Rosa (2016) los cuales mencionan que para construir una aplicación basada en Microservicios, existen varias técnicas, una de ellas es partir de una aplicación Monolítica, para identificar, separar y construir gradualmente las funcionalidades en servicios preferentemente basados en REST.

Además Richardson & Smith (2016) coinciden en que para construir una aplicación basada en Microservicios es recomendable empezar con una aplicación Monolítica; existen varias estrategias, de las cuales se seguirá la *Estrategia 3: Extracción de Servicios,* la misma que consiste en identificar las funcionalidades/módulos de la aplicación monolítica, para luego extraerlas, construirlas e implementarlas como servicios independientes (así mismo recomiendan el uso de RESTful), todo este proceso se lo realiza incrementalmente es decir una funcionalidad a la vez. Para realizar el paso de la aplicación monolítica a RESTful se tomó en consideración, el escalado en el eje Y, explicado y mencionado en el apartado **1.1.2.1,** en el que se hace referencia al *Scale Cube*.

A continuación se describen las características que dicha aplicación va tomando en el proceso.

4.1.1 Aplicación Monolítica Inicial (PHP)

4.1.1.1 Características y Tecnologías

Las características, y tecnologías con sus respectivas versiones que posee la aplicación inicial se describen en la **Tabla 15,** en la cual se observa que no se hace uso de Patrones, ni de Arquitectura, o Store Procedures en la construcción de la misma**.**

Tabla 15: Características Aplicación Monolítica (PHP)

Recurso	Tecnología	Versión
Patrones	X	
Arquitectura	X	
IDE	SublimeText	3.0.0
BDD	MySQL	5.0.11
Store Procedures	X	
Servidor Web	Apache	2.4.9
Lenguaje de Programación	PHP	5.5.12
Otros	CSS, HTML, Javascript	

Fuente: El autor
Elaboración: El autor

4.1.1.2 Funcionalidades

Las *funcionalidades* que realiza la aplicación se exponen a continuación, se puede visualizar las capturas de pantalla de su funcionamiento en el **Anexo D**:.

- Registro de usuarios.
- Login (Ingreso de usuarios al sistema, se maneja 2 roles (Cliente y Administrador)).
- CRUD especialidades Médicas (Realizado por el Administrador).
- CRUD de Doctores (Realizado por el Administrador).
- Vinculación de Especialidades a Doctores. (Realizado por el Administrador)
- CRUD de Citas Médicas (Realizado por el Cliente).

4.1.1.3 Implementación

Para detallar el desarrollo y despliegue de la aplicación web (PHP) se consideran 3 niveles de implementación:

- Datos.
- Codificación.
- Despliegue (Servidor).

4.1.1.3.1 Datos

Referente a la gestión de la base de datos MySQL, se utiliza la herramienta *WampServer (Es similar a Xampp, la diferencia con este último, es que WampServer es exclusivamente para Windows),* la cual implementa el servidor Apache y además proporciona una interfaz web para la gestión de la base de datos; se diseñó una base de datos según el *diagrama Entidad-Relación* que se puede observar en el **Anexo E**: Diagrama Entidad-Relación "Citas Médicas"**.**

4.1.1.3.2 Codificación

Se procedió a realizar la codificación utilizando las herramientas que se mencionan en la **Tabla 15.** Al ser una aplicación que **NO** implementa una arquitectura, se la dividió internamente a la aplicación en carpetas; las cuales interactúan entre sí para cumplir con las funcionalidades del software, como se expone en la **Figura 13**.

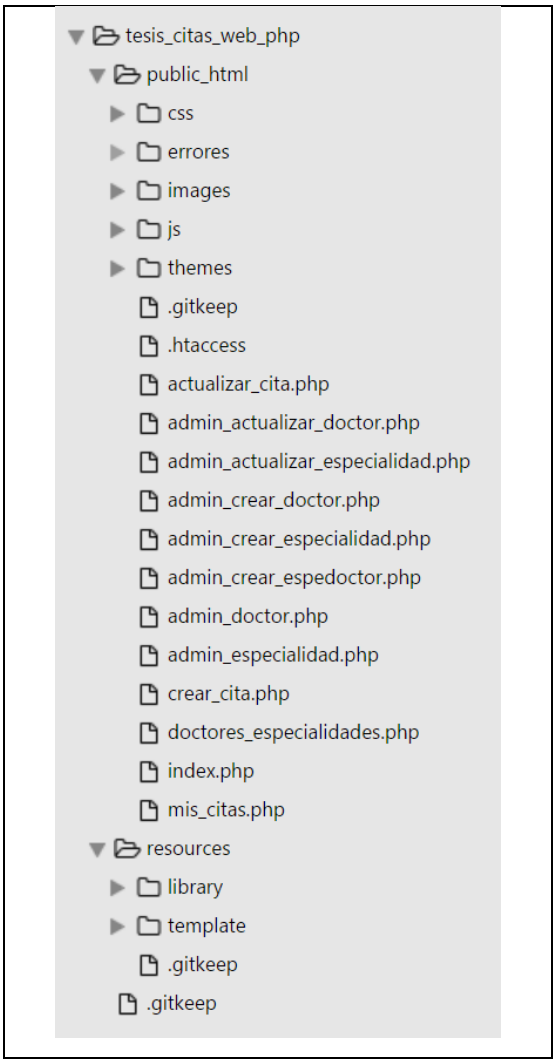


Figura 13: Estructura de Aplicación WEB PHP
Fuente: El autor
Elaboración: El autor

Seguidamente se especifica las implicaciones que conllevó la construcción de la aplicación.

- **Acceso a Datos.**

Definido en la codificación como la carpeta *library,* realiza la interacción con la base de datos. Estos archivos se encargan de realizar el CRUD según sea solicitado, en la **Figura 14** y la **Figura 15** aprecia la conexión entre el software y la base de datos.

```php
<?php
    /*DATOS DE LA BASE DE DATOS MYSQL*/
    $db_host="127.0.0.1";
    $db_user="root";
    $db_pass="";
    $db_data="tesis_citas_medicas";
    $tipo_conexion="mysql";
?>
```

Figura 14: Archivo *config.php* para conexión con base de datos.
Fuente: El autor
Elaboración: El autor

```php
<?php
    include("config.php");

    if($tipo_conexion=="mysql"){
        include("/library/class_mysql.php");
        $miconexion = new DB_mysql;
        $miconexion->conectar($db_data,$db_host,$db_user,$db_pass);
?>
```

Figura 15: Archivo *conexion.php* para conexión con base de datos.
Fuente: El autor
Elaboración: El autor

- **Presentación.**

Definido en la codificación como las carpetas *public_html, template* son las encargadas de la presentación de las interfaces, captura y validación de datos del usuario en la **Figura 16**, se observa una porción de código del formulario de registro.

```php
<aside class="registro">
    <h3>Registro a la APP</h3>

    <form action="<?php echo $url_site; ?>resources/library/guardar_registro.php" method="post">
      <br><input type="text" name="cedula" value="" required placeholder="Cedula"><br>

      <br><input type="text" name="nombre" value="" required placeholder="Nombre"><br>

      <br><input type="text" name="apellido" value="" required placeholder="Apellido"><br>

      <br><input type="text" name="usuario" value="" required placeholder="Usuario"><br>

      <br><input type="password" name="pass" value="" required placeholder="Contraseña" ><br>
    <!--TIPO DE ROL 1=ADMINISTRADOR; 2=PACIENTE  -->
      <br><input type="hidden" name="rol" value="2" required placeholder="Rol" >

      <input class="btn btn-warning" type="submit" value="Registrar">
    </form>
</aside>
```

Figura 16: Archivo *formulario.php* para recolección de datos desde el usuario.
Fuente: El autor
Elaboración: El autor

4.1.1.3.3 Despliegue (Servidor)

Como se mencionó anteriormente para la aplicación web (php), se usa el servidor Apache (integrado en WampServer), en el cual utilizamos la configuración por defecto, permitiendo levantar la aplicación (colocando el proyecto en el directorio

C:\wamp\www) en el *localhost* (http://127.0.0.1/tesis_monolitica_web). En la **Figura 17** se observa el respectivo diagrama de despliegue.

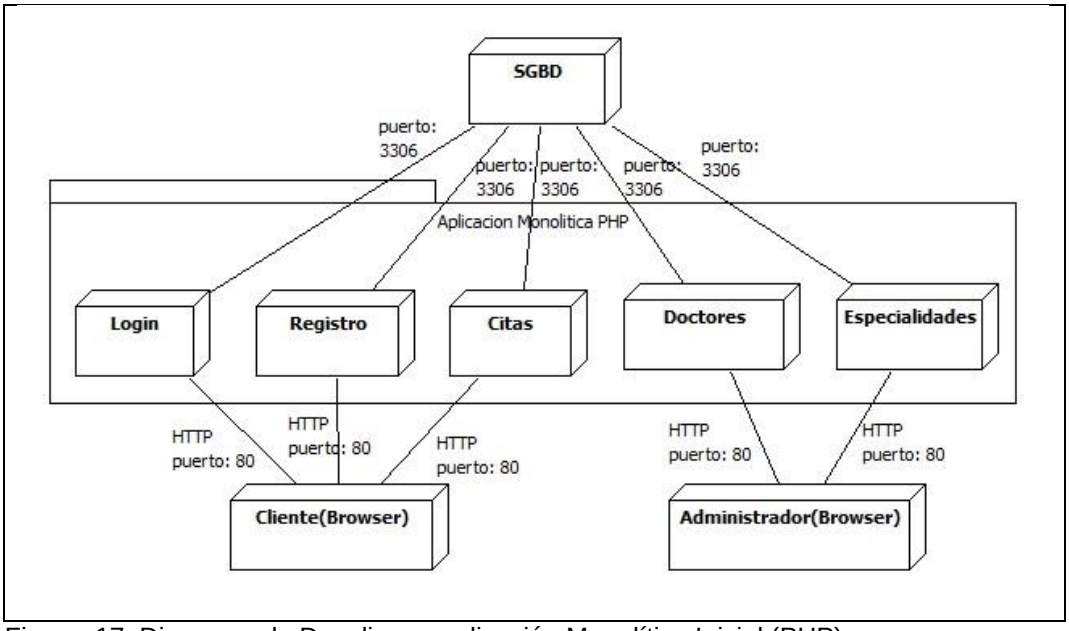


Figura 17: Diagrama de Despliegue aplicación Monolítica Inicial (PHP).
Fuente: El autor
Elaboración: El autor

4.1.2 Aplicación Monolítica (JAVA)

4.1.2.1 Características y Tecnologías

A la aplicación monolítica realizada en PHP, se procedió a migrarla a una tecnología Java, obteniendo como resultado lo que se observa en la **Tabla 16**, en la cual se indica entre otros recursos, que se hace uso de una arquitectura en 3 capas (layers); y así mismo las tecnologías que se utilizan en la aplicación permiten un mejor desempeño de la misma.

Tabla 16: Características Aplicación Monolítica (JAVA)

Recurso	Tecnología	Versión
Patrones	X	
Arquitectura	3 layers (capas)	
IDE	Netbeans	8.0.2
BDD	MySQL	5.0.11
Store Procedures	X	
Servidor Web	Glassfish	4.1.2
Lenguaje de Programación	Java EE (JSP, Servlets)	5.5.12
Otros	CSS, HTML, Javascript	

Fuente: El autor
Elaboración: El autor

4.1.2.2 Funcionalidades

Las *funcionalidades* que realiza la aplicación son las siguientes (se mantienen las mismas que la aplicación en PHP, ver **Anexo D**: Funcionalidades Aplicación Web (PHP, Java))

- Registro de usuarios.
- Login (Ingreso de usuarios al sistema, se maneja 2 roles (Cliente y Administrador)).
- CRUD especialidades Médicas (Realizado por el Administrador).
- CRUD de Doctores (Realizado por el Administrador).
- Vinculación de Especialidades a Doctores. (Realizado por el Administrador)
- CRUD de Citas Médicas (Realizado por el Cliente).

4.1.2.3 Implementación

Para detallar el desarrollo y despliegue de la aplicación web (Java) se consideran 3 niveles de implementación:

- Datos.
- Codificación.
- Despliegue (Servidor).

4.1.2.3.1 Datos

Se conserva la misma base de datos mencionada en el apartado **4.1.1.3.1.** Seguidamente se realizó la conexión lógica entre el software y la base de datos, dando como resultado un enlace activo para la codificación de la aplicación web. En la **Figura 18** se observa dicha conexión con las tablas alojadas en la base de datos.

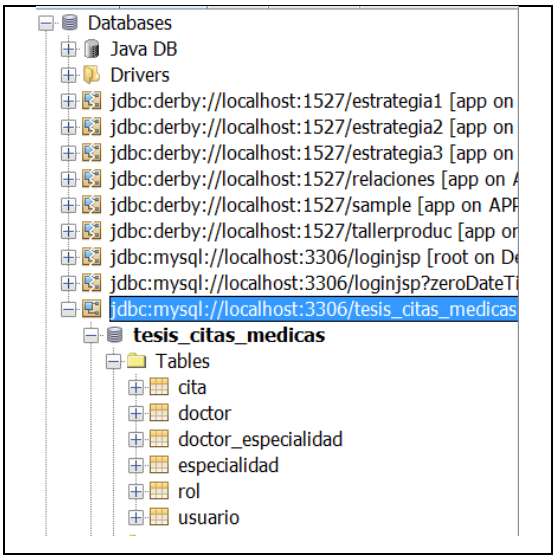


Figura 18: Conexión entre el software y la base de datos.
Fuente: El autor
Elaboración: El autor

4.1.2.3.2 Codificación

Se procedió a realizar la codificación utilizando las herramientas que se mencionan en la **Tabla 16.** Al ser una aplicación que implementa una arquitectura 3 capas, se la dividió internamente a la aplicación en las mismas; las cuales interactúan entre sí para cumplir con las funcionalidades del software, como se expone en la **Figura 19.**

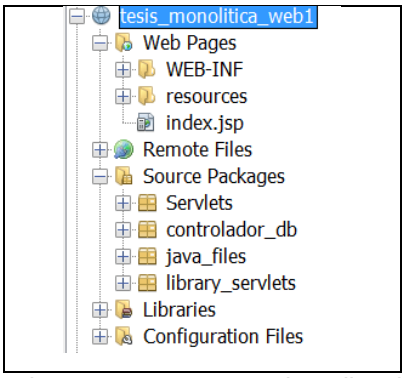


Figura 19: Estructura de aplicación Web Java.
Fuente: El autor
Elaboración: El autor

Seguidamente se especifica las implicaciones que conllevó la construcción de la aplicación.

- **Capa de Acceso a Datos.**

Definido en la codificación como *controlador_db* y *library_servlets* como se observa en la **Figura 20**, realiza la interacción con la base de datos. Estos archivos se encargan de realizar el CRUD según sea solicitado, en la **Figura 21** se aprecia la conexión con la base de datos.

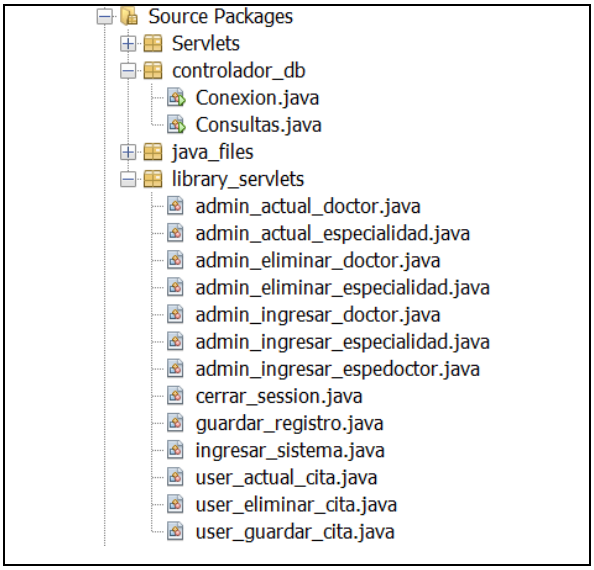


Figura 20: Capa de Acceso a Datos.
Fuente: El autor
Elaboración: El autor

```java
public class Conexion {

    private String USERNAME = "root";
    private String PASSWORD = "";
    private String HOST = "127.0.0.1";
    private String PORT = "3306";
    private String DATABASE = "tesis_citas_medicas";
    private String CLASSNAME = "com.mysql.jdbc.Driver";
    private String URL = "jdbc:mysql://" + HOST + ":" + PORT + "/" + DATABASE;
    private Connection con;

    public Conexion(){
        try {
            Class.forName(CLASSNAME);
            con = DriverManager.getConnection(URL, USERNAME, PASSWORD);

        } catch (ClassNotFoundException e) {
            System.err.println("ERROR "+e);
        } catch (SQLException e){
             System.err.println("Error "+e);
        }
    }
    public Connection getConexion(){
        return con;
    }
```

Figura 21: Configuración de Persistencia.
Fuente: El autor
Elaboración: El autor

Las mencionadas configuraciones permiten establecer una conexión directamente a la base de datos empleada.

- **Capa de Lógica de Negocio.**

Definido en la codificación como *java_files* **(Figura 22)** son las encargadas de realizar operaciones con los datos; con la aplicación se realizaran operaciones referentes al encriptado y desencriptado de la contraseña como se observa en la **Figura 23.**

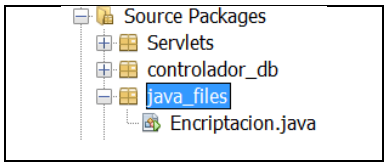


Figura 22: Capa de Lógica de Negocio.
Fuente: El autor
Elaboración: El autor

```
public String encriptar(String texto) {
    try {
        Cipher cipher = Cipher.getInstance(cI);
        SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes(), alg);
        IvParameterSpec ivParameterSpec = new IvParameterSpec(iv.getBytes());
        cipher.init(Cipher.ENCRYPT_MODE, skeySpec, ivParameterSpec);
        byte[] encrypted = cipher.doFinal(texto.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    } catch (Exception e) {
        return "";
    }
}

public String desencriptar(String texto) {
    try {
        Cipher cipher = Cipher.getInstance(cI);
        SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes(), alg);
        IvParameterSpec ivParameterSpec = new IvParameterSpec(iv.getBytes());
        byte[] enc = Base64.getDecoder().decode(texto);
        cipher.init(Cipher.DECRYPT_MODE, skeySpec, ivParameterSpec);
        byte[] decrypted = cipher.doFinal(enc);
        return new String(decrypted);
    } catch (Exception e) {
        return "";
    }
}
```

Figura 23: Archivo de encriptación.
Fuente: El autor
Elaboración: El autor

- **Capa de Presentación.**

Definido en la codificación como *resources,* y *servlets* son las encargadas de la presentación de las interfaces, captura y validación de datos del usuario **(Figura 24).**

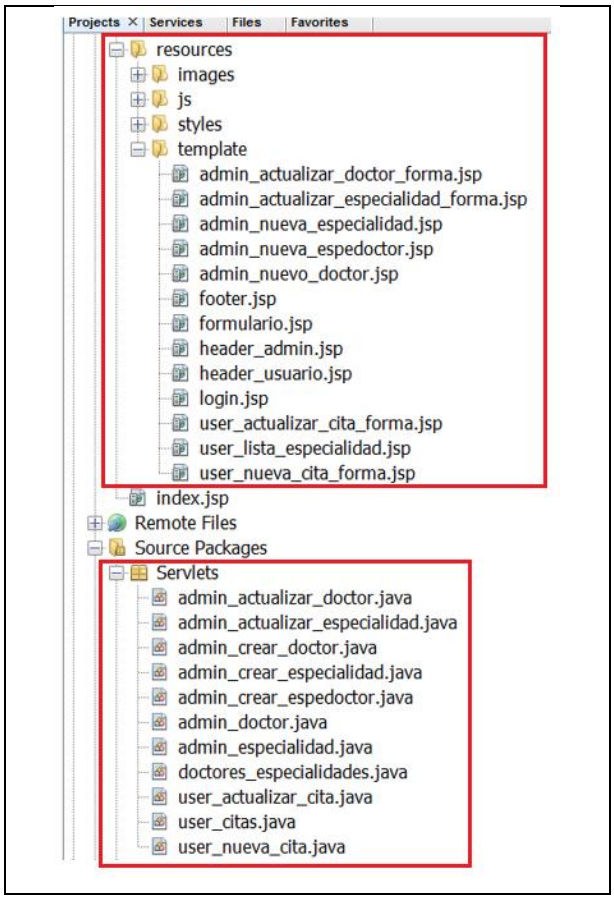


Figura 24: Capa de presentación
Fuente: El autor
Elaboración: El autor

4.1.2.3.3 Despliegue (Servidor)

Para la aplicación con Java se utiliza el servidor Glassfish (integrado en Netbeans), en el cual utilizamos la configuración por defecto, se realiza el respectivo *Deploy* de la aplicación, permitiendo levantar la misma en el localhost (*localhost:8080/tesis_monolitica_web1)*. El diagrama de despliegue se mantiene invariable como se observa en la **Figura 17**.

4.1.3 Aplicación RESTful (JAVA)

4.1.3.1 Características y Tecnologías

Posteriormente, con la aplicación Monolítica desarrollada en Java, se procedió a extraer determinadas funcionalidades de la misma (aplicando escalado en el eje Y, según indica el Scale Cube **1.1.2.1**), para su implementación con RESTful (**Tabla 18**), cabe mencionar que solo se manejaran peticiones de tipo GET; en la **Tabla 17** se

observan las características que posee la aplicación; entre lo que se destaca la implementación del patrón Facade.

Tabla 17: Características Aplicación RESTful (JAVA)

Recurso	Tecnología	Versión
Patrones	Facade	
Arquitectura	n layers (capas)	
IDE	Netbeans	8.0.2
BDD	MySQL	5.0.11
Store Procedures	SI (6)	
Servidor Web	Glassfish	4.1.2
Lenguaje de Programación	Java EE (JSP, Servlets)	5.5.12
Otros	CSS, HTML, Javascript, RESTful	

Fuente: El autor
Elaboración: El autor

4.1.3.2 Funcionalidades

Las *funcionalidades* que realiza la aplicación son las siguientes:

- Obtener citas médicas, según cedula del cliente, fecha, cedula del doctor, o especialidad.
- Obtener doctores según la especialidad o todos.

Dichas funcionalidades implementadas en RESTful se representarían tal y como se observa en la **Tabla 18.**

Tabla 18: Identificación de Servicios RESTful

	URI	Parámetro	Recurso
Citas	http://host/aplicación/ms/cita/{cedula}/paciente	Cedula(Cliente)	Lista todas las citas pertenecientes al paciente con la cedula ingresada. (Para sacar todas las citas existentes se ingresa como cedula el valor de -1)
	http://host/aplicación/ms/cita/{fecha}/fecha	Fecha	Lista todas las citas en una fecha establecida. En la fecha se manejara, el formato (31122017).
	http://host/aplicación/ms/cita/{cedula}/doctor	Cedula (Doctor).	Lista todas las citas pertenecientes a un doctor establecido.
	http://host/aplicación/ms/cita/{especialidad}/especialidad	Especialidad (Id)	Lista todas las citas con una especialidad establecida.
Doctores	http://host/aplicación/ms/doctor	No requiere parámetro	Lista todos los doctores
	http://host/aplicación/ms/doctor/{especialidad}	Especialidad (Id)	Busca/Lista todos los doctores que tengan una especialidad establecida.

Fuente: El autor
Elaboración: El autor

Para el módulo de Citas se crearon 4 servicios RESTful con sus respectivos parámetros de entrada; para el módulo de Doctores por su parte 2. Los servicios RESTful referentes a Citas, retornan información de varias tablas (5) existentes en la base de datos, como se observa en la **Tabla 19**.

Tabla 19: Columnas y datos (Modulo Cita)

COLUMNA	TIPO DE DATOS
Id_cita(cita)	Int (clave principal)
Paciente(cita)	Int (clave Foránea)
Doctor(cita)	Int (clave Foránea)
Especialidad(cita)	Int (clave Foránea)
Fecha(cita)	String (varchar)
Hora(cita)	String (varchar)
cedula (usuario)	Int
nombres(usuario)	String (varchar)
apellidos(usuario)	String(varchar)
username(usuario)	String(varchar)
password(usuario)	String(varchar)
rol(usuario)	Int (clave foránea)
genero(usuario)	String(varchar)
nombre(Rol)	String(varchar)
Nombre(especialidad)	String(varchar)
cedula (doctores)	Int
nombres(doctores)	String (varchar)
apellidos(doctores)	String(varchar)
genero(doctores)	String(varchar)

Fuente: El autor
Elaboración: El autor

Por su parte en el módulo de Doctores se retornan los valores de 3 tablas, como se indica en **Tabla 20**.

Tabla 20: Columnas y datos (Modulo Doctor)

COLUMNA	TIPO DE DATOS
id_doctor(doctores)	Int (clave principal)
cedula (doctores)	Int
nombres(doctores)	String (varchar)
apellidos(doctores)	String(varchar)
genero(doctores)	String(varchar)
Id_doctor_especialidad(doctor_especialidad)	Int (clave principal)
especialidad(doctor_especialidad)	Int (clave Foránea)
Nombre(especialidad)	String(varchar)

Fuente: El autor
Elaboración: El autor

Cada uno de los servicios RESTful están diseñados para generar las respuestas en formato XML, HTML, JSON, y se lo retorna según se lo solicite en la petición. Así mismo se ha creído conveniente la implementación *de Store Procedures* (o llamados Procedimientos Almacenados) en la base de datos, y su relación con cada servicio RESTful (**Tabla 21**), indicando sus respectivos parámetros de entrada.

Tabla 21: Relación entre Procedimientos Almacenados y Servicios RESTful.

Procedimiento Almacenado	Parámetro	Identificador RESTful
listar_cita_por_cedula(cedula)	Cedula(Paciente)	http://host/aplicación/ms/cita/{cedula}/paciente
listar_cita_por_fecha(fecha)	Fecha	http://host/aplicación/ms/cita/{fecha}/fecha
listar_cita_por_doctor(cedula)	Cedula (Doctor).	http://host/aplicación/ms/cita/{cedula}/doctor
listar_cita_por_especialidad(especialidad)	Especialidad (Id)	http://host/aplicación/ms/cita/{especialidad}/especialidad
listar_doctor ()	No requiere parámetro	http://host/aplicación/ms/doctor
listar_doctor_por_especialidad(especialidad)	Especialidad (Id)	http://host/aplicación/ms/doctor/{especialidad}

Fuente: El autor
Elaboración: El autor

4.1.3.3 Implementación

Para detallar el desarrollo y despliegue de la aplicación RESTful (Java) se consideran 3 niveles de implementación:

- Datos.
- Codificación.
- Despliegue (Servidor).

4.1.3.3.1 Datos

Se conserva la misma base de datos mencionada en el apartado **4.1.1.3.1**, a la cual se le agregaron los Store Procedures mencionados en la **Tabla 21,** como se observa en la **Figura 25**, para esto se usó la herramienta *MySQL Workbench***.**

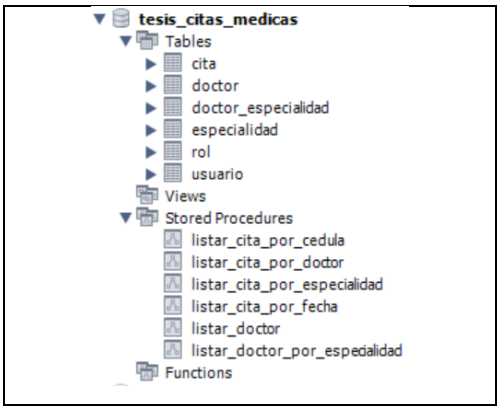


Figura 25: Procedimientos almacenados en la Base de Datos.
Fuente: El autor
Elaboración: El autor

En la **Figura 26**, se observa la estructura para la creación de un Store Procedure implementada en *MySQL Workbench***.**

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `listar_cita_por_cedula`(in in_cedula varchar(20))
BEGIN
/*
    CREADO:
        Nombres: Jose Luis Cueva Tacuri
        Fecha: 03/07/2017

    DESCRIPCION: Store Procedure que lista todas las citas pertenecientes al usuario con la cedula ingresada.
                 (Para sacar todas las citas existentes se ingresa como cedula el valor de -1)

        INPUT: El parametro 'cedula' de tipo varchar.

        OUTPUT: Un resulset con todos los indicados en la sentencia select.
*/
SELECT ci.id_cita ID_CITA, ci.usuario ci_usuario, ci.doctor ci_doctor,ci.especialidad ci_especialidad, ci.fecha ci_fecha,ci.hora ci_hora,
            doc.id_doctor ID_DOCTOR, doc.cedula doc_cedula, doc.nombres doc_nombres, doc.apellidos doc_apellidos ,doc.genero doc_genero,
            es.id_especialidad ID_ESPECIALIDAD, es.nombre es_nombre,
            us.id_usuario ID_USUARIO,us.cedula us_cedula,us.nombres us_nombres, us.apellidos us_apellidos,us.username us_username, us.password
            rol.id_rol ID_ROL, rol.nombre rol_nombre
    from cita as ci
    join usuario as us ON   ci.usuario=us.id_usuario
    join doctor as doc ON ci.doctor=doc.id_doctor
    join especialidad es ON ci.especialidad=es.id_especialidad
    join rol rol ON us.rol=rol.id_rol
    where (us.cedula=in_cedula or in_cedula='-1');
END
```

Figura 26: Creación de Procedimientos almacenados en la Base de Datos.
Fuente: El autor
Elaboración: El autor

4.1.3.3.2 Codificación

Se procedió a realizar la codificación utilizando las herramientas que se mencionan en la **Tabla 17**. Al ser una aplicación que implementa una arquitectura n capas, se la dividió internamente a la aplicación en las mismas; las cuales interactúan entre sí para cumplir con las funcionalidades del software, como se expone en la **Figura 27**.

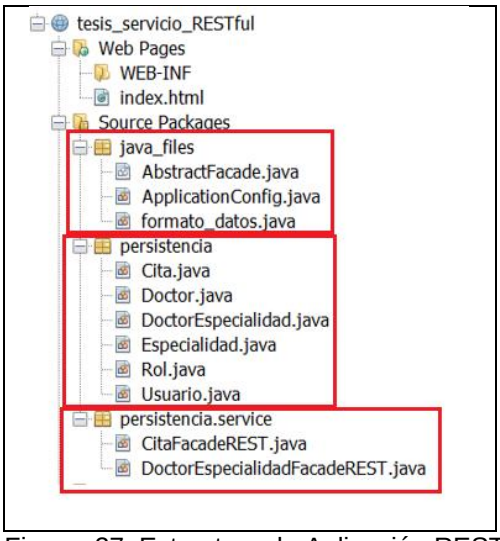


Figura 27: Estructura de Aplicación RESTful Java.
Fuente: El autor
Elaboración: El autor

Seguidamente se especifica las implicaciones que conllevó la construcción de la aplicación.

- **Capa de Acceso a Datos.**

Definido en la codificación como *persistencia,* este paquete de programación contiene mapeada la base de datos; es decir se construyeron clases independientes de programación para cada tabla a nivel de base de datos, las clases codificadas se relacionan de la misma forma que en la base de datos relacional. Este componente realiza la interacción con la base de datos, es decir las solicitudes de datos; así mismo de interactuar directamente con las configuraciones de persistencia desplegada mediante JPA reflejadas en la **Figura 28.**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1 Resource Definitions//EN" "http://glassfish.org
<resources>
    <jdbc-connection-pool allow-non-component-callers="false" associate-with-thread="false" connection-creation-retry-attempts="0"
        <property name="serverName" value="localhost"/>
        <property name="portNumber" value="3306"/>
        <property name="databaseName" value="tesis_citas_medicas"/>
        <property name="User" value="root"/>
        <property name="Password" value=""/>
        <property name="URL" value="jdbc:mysql://localhost:3306/tesis_citas_medicas?zeroDateTimeBehavior=convertToNull"/>
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource enabled="true" jndi-name="citas_medicas" object-type="user" pool-name="mysql_tesis_citas_medicas_rootPool"/>
</resources>
```

Figura 28: Configuración de Persistencia.
Fuente: El autor
Elaboración: El autor

Al momento de utilizar procedimientos almacenados se debe realizar el mapeo de los datos obtenidos a las Entity Class respectivas, se lo realizó a través de anotaciones, una muestra del mapeo implementado se observa en la **Figura 29.**

```
@Entity
@Table(name = "cita")
@XmlRootElement
@SqlResultSetMapping(
        name = "MapeoListarCita",
        entities = {
            @EntityResult(entityClass = Cita.class,
                    fields = {
                        @FieldResult(name = "id_cita", column = "ID_CITA"),
                        @FieldResult(name = "usuario", column = "ci_usuario"),
                        @FieldResult(name = "doctor", column = "ci_doctor"),
                        @FieldResult(name = "especialidad", column = "ci_especialidad"),
                        @FieldResult(name = "fecha", column = "ci_fecha"),
                        @FieldResult(name = "hora", column = "ci_hora"),}
            ),
            @EntityResult(entityClass = Doctor.class,
                    fields = {
                        @FieldResult(name = "id_doctor", column = "ID_DOCTOR"),
                        @FieldResult(name = "cedula", column = "doc_cedula"),
                        @FieldResult(name = "nombres", column = "doc_nombres"),
                        @FieldResult(name = "apellidos", column = "doc_apellidos"),
                        @FieldResult(name = "genero", column = "doc_genero"),}
            ),
```

Figura 29: Mapeo de Consulta | Clase Cita
Fuente: El autor
Elaboración: El autor

Las mencionadas configuraciones permiten establecer una conexión directa con la base de datos empleada.

- **Capa de Lógica de Negocio.**

Definido en la codificación como *java_files* donde se escribieron 3 clases de programación que contienen la lógica necesaria para cumplir con las funcionalidades de los servicios web RESTful.

- **AbstractFacade**:

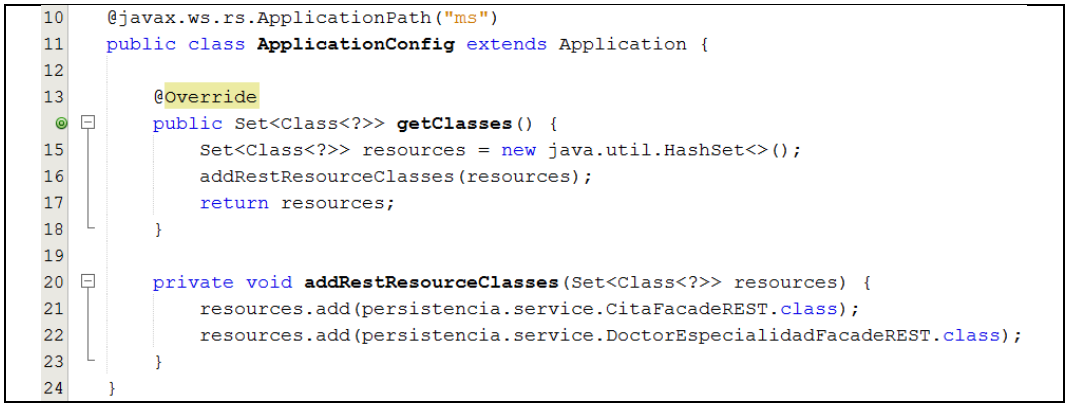
Esta clase contiene la lógica para obtener los recursos del repositorio de datos; es la fachada del patrón de diseño Facade. Esta clase posee las funcionalidades que resolverán las invocaciones de cada identificador previamente definido en la **Tabla 18.** La **Figura 30**, presenta una muestra de la codificación desarrollada (se tiene un método para cada funcionalidad en un formato especificado de retorno, pudiendo ser XML, Json, o HTML; en donde se realiza la llamada a los procedimientos almacenados y se utiliza el mapeo ya definido por cada clase).

```
48  public String ListarCitaPorFechaHTML(Object fecha) {
49      StoredProcedureQuery storedProcedure = getEntityManager().createStoredProcedureQuery("listar_cita_por_fecha", "MapeoLista
50              .registerStoredProcedureParameter("fecha", String.class, ParameterMode.IN);
51
52      storedProcedure.setParameter("fecha", fecha);
53      return formato.CitaGenerarHTML(storedProcedure.getResultList());
54  }
55
56  public List<T> ListarCitaPorDoctorXMLJSON(Object cedula) {
57      StoredProcedureQuery storedProcedure = getEntityManager().createStoredProcedureQuery("listar_cita_por_doctor", "MapeoList
58              .registerStoredProcedureParameter("cedula", String.class, ParameterMode.IN);
59
60      storedProcedure.setParameter("cedula", cedula);
61      return formato.CitaGenerarListaClases(storedProcedure.getResultList());//ME RECUPERA EL RESULTADO;
62  }
63
64  public String ListarCitaPorDoctorHTML(Object cedula) {
65      StoredProcedureQuery storedProcedure = getEntityManager().createStoredProcedureQuery("listar_cita_por_doctor", "MapeoList
66              .registerStoredProcedureParameter("cedula", String.class, ParameterMode.IN);
67
68      storedProcedure.setParameter("cedula", cedula);
69      return formato.CitaGenerarHTML(storedProcedure.getResultList());//ME RECUPERA EL RESULTADO;
70  }
```

Figura 30: Codificación | Clase AbstractFacade
Fuente: El autor
Elaboración: El autor

- **ApplicationConfig**:

En esta se declaran las clases que responderán a la invocación de los servicios web, además se especifica las clases que harán uso de la clase AbstractFacade la misma que se expone en la **Figura 31.**



```
10  @javax.ws.rs.ApplicationPath("ms")
11  public class ApplicationConfig extends Application {
12
13      @Override
14      public Set<Class<?>> getClasses() {
15          Set<Class<?>> resources = new java.util.HashSet<>();
16          addRestResourceClasses(resources);
17          return resources;
18      }
19
20      private void addRestResourceClasses(Set<Class<?>> resources) {
21          resources.add(persistencia.service.CitaFacadeREST.class);
22          resources.add(persistencia.service.DoctorEspecialidadFacadeREST.class);
23      }
24  }
```

Figura 31: Codificación | Clase ApplicationConfig
Fuente: El autor
Elaboración: El autor

- **formato_datos**:

Esta clase contiene la codificación correspondiente al formato de los datos de retorno, pudiendo ser una *Lista de Clases* (que después se la puede convertir fácilmente en Json o XML), o en una tabla *HTML,* una porción de código se observa en la **Figura 32.**

```java
public class formato_datos {

    public List<Cita> CitaGenerarListaClases(List<Object[]> resultado) {
        List<Cita> ListaCitas = new ArrayList<>();
        resultado.stream().forEach((record) -> {
            Cita citas = (Cita) record[0];
            Doctor doctor = (Doctor) record[1];
            Especialidad espe = (Especialidad) record[2];
            Usuario us = (Usuario) record[3];
            Rol rol = (Rol) record[4];
            citas.setDoctor(doctor);
            citas.setEspecialidad(espe);
            us.setRol(rol);
            citas.setUsuario(us);
            ListaCitas.add((Cita) citas);
        });
        return ListaCitas;
    }

    public String CitaGenerarHTML(List<Object[]> resultado) {
        String CitasHTML = "<!DOCTYPE html><html><head>"
                + "<style>table, th, td { border: 1px solid black; border-collapse: collapse;</style>"
                + "<title>Listar Citas</title></head>"
                + "<body><h1>Listar Citas</h1>";
```

Figura 32: Codificación | Clase formato_datos
Fuente: El autor
Elaboración: El autor

– **Capa de Servicio/Presentación.**

Corresponde al paquete definido en la codificación como *persistencia.service* contiene las clases para implementar los servicios web. Estas clases son parte del patrón de diseño implementado, invocan funcionalidades de la fachada AbstractFacade, las clases escritas son:

- CitaFacadeREST
- DoctorEspecialidadFacadeREST

En la **Figura 33** se observa la invocación de métodos necesarios para resolver las peticiones de los clientes (según el formato solicitado), la porción de código expuesta muestra 3 métodos propiedad de la clase CitaFacadeREST, en el cual los dos primeros métodos realizan la misma funcionalidad, la diferencia radica en el formato de la petición/respuesta de los datos (el primero es para XML y Json, mientras que el segundo es para HTML).

```
26      @GET
27      @Path("{cedula}/paciente")
28      @Produces({"application/xml" + ";charset=utf-8", "application/json" + ";charset=utf-8"})
29      public List<Cita> ListarCitaPorCedulaXMLJSON(@PathParam("cedula") String cedula) {
30          return super.ListarCitaPorCedulaXMLJSON(cedula);
31      }
32
33      @GET
34      @Path("{cedula}/paciente")
35      @Produces({"text/html"})//Este vale para HTML!!!! para metodo de tipo String
36      public String ListarCitaPorCedulaHTML(@PathParam("cedula") String cedula) {
37          return super.ListarCitaPorCedulaHTML(cedula);
38      }
39
40      @GET
41      @Path("{fecha}/fecha")
42      @Produces({"application/xml" + ";charset=utf-8", "application/json" + ";charset=utf-8"})
43      public List<Cita> ListarCitaPorFechaXMLJSON(@PathParam("fecha") String fecha) {
44          return super.ListarCitaPorFechaXMLJSON(fecha);
45      }
```

Figura 33: Codificación | Clase CitaFacadeREST
Fuente: El autor
Elaboración: El autor

4.1.3.3.3 Despliegue (Servidor)

Para la aplicación de RESTful con Java, se utiliza el servidor Glassfish (integrado en Netbeans), en el cual utilizamos la configuración por defecto, se realiza el respectivo *Deploy* de la aplicación, permitiendo levantar los servicios y consumirlos según las URI's definidas en la **Tabla 18**. El diagrama de despliegue de la aplicación se observa en la **Figura 35**. La estructura al momento de la implementación con el patrón *Facade* se observa en la **Figura 34**, en el cual las aplicaciones hacen uso del servicio RESTful, el cual internamente posee la implementación del patrón mencionado:

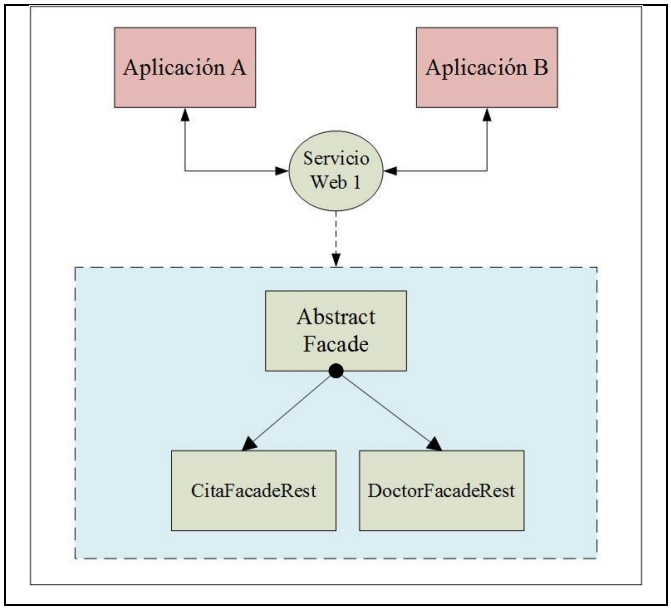


Figura 34: Diagrama de Despliegue aplicación RESTful (JAVA).
Fuente: El autor
Elaboración: El autor

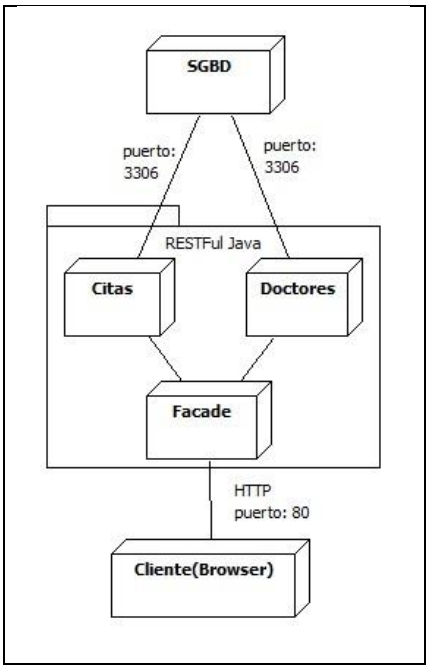


Figura 35: Diagrama de Despliegue aplicación RESTful (JAVA).
Fuente: El autor
Elaboración: El autor

4.1.4 Aplicación RESTful (NODE)

4.1.4.1 Características y Tecnologías

Posteriormente, con la aplicación RESTful desarrollada en Java, se procedió a realizar la migración de tecnología a Node.js, esto debido a que es un lenguaje ligero y por lo tanto es más óptimo y brinda facilidades para posteriormente implementar Microservicios. En la **Tabla 22** se observa las tecnologías/herramientas utilizadas en este proceso.

Tabla 22: Características Aplicación MS (NODE).

Recurso	Tecnología	Versión
Patrones	Adaptación de Facade	
Arquitectura	n layers (capas)	
IDE	SublimeText	3.0.0
BDD	MySQL	5.0.11
Store Procedures	SI (6)	
Servidor Web	Node.js	6.9.2
Lenguaje de Programación	Node.js	6.9.2
Otros	CSS, HTML, Javascript, RESTful, Express (framework Node)	

Fuente: El autor
Elaboración: El autor

4.1.4.2 Funcionalidades

Se conserva la misma estructura mencionada en el apartado **0.** Además se debe establecer los puertos de funcionamiento de cada servicio RESTful como se observa en la **Tabla 23**.

Tabla 23: Relación puerto con servicio RESTful.

Puerto	Aplicación node	Identificador RESTful
1400	cita_paciente	http://host/aplicación/ms/cita/{cedula}/paciente
1500	cita_fecha	http://host/aplicación/ms/cita/{fecha}/fecha
1600	cita_doctor	http://host/aplicación/ms/cita/{cedula}/doctor
1700	cita_especialidad	http://host/aplicación/ms/cita/{especialidad}/especialidad
1800	doctor	http://host/aplicación/ms/doctor
1900	doctor_especialidad	http://host/aplicación/ms/doctor/{especialidad}

Fuente: El autor
Elaboración: El autor

4.1.4.3 Implementación

Para detallar el desarrollo y despliegue de la aplicación RESTful (Node) se consideran 3 niveles de implementación:

- Datos.
- Codificación.
- Despliegue (Servidor).

Primeramente se realizó la instalación de Node y las herramientas relacionadas que permiten el desarrollo y despliegue de la aplicación (ver **Anexo F**: Instalación de Node.js en Windows 8.1**).**

4.1.4.3.1 Datos

Se utiliza la misma base de datos mencionada en el apartado **4.1.3.3.1.**

4.1.4.3.2 Codificación

Se procedió a realizar la codificación utilizando las herramientas que se mencionan en la **Tabla 22**. Se realizó una aplicación por cada RESTful para proporcionar independencia entre los mismos. Al ser aplicaciones que implementa una arquitectura n capas, se la dividió internamente a cada aplicación en las mismas; las cuales interactúan entre sí para cumplir con las funcionalidades del software, como se expone en la **Figura 36** (se observan las 6 aplicaciones Node, la estructura que poseen es la misma que posee *cita_doctor*).

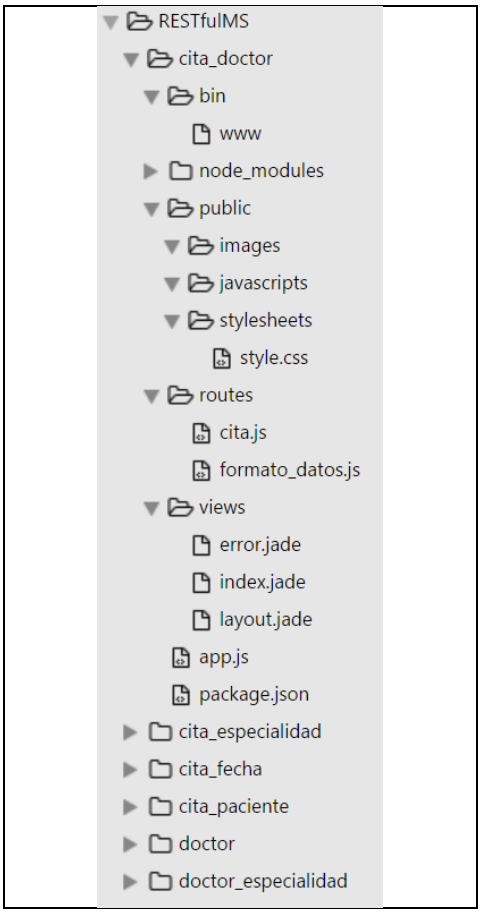


Figura 36: Estructura de Aplicaciones RESTful Node.
Fuente: El autor
Elaboración: El autor

Seguidamente se especifica las implicaciones que conllevó la construcción de la aplicación *cita_doctor* (el procedimiento y estructura es similar para las otras 5 aplicaciones) para la creación de la aplicación ver el **Anexo G**: Desarrollo de Aplicación Node.js.

- **Acceso a Datos.**

Definido en la codificación como *app.js,* este archivo de programación realiza la interacción con la base de datos, es decir las solicitudes de datos; una porción de este archivo se observa en la **Figura 37**.

```
//Conexion con MYSQL
var mysql = require('mysql');
var connection=require('express-myconnection');

var app = express();

//Crear conexion SQL
app.use(connection(mysql,{
host:"localhost",
user:"root",
password:"",
database:"tesis_citas_medicas"
},'request'));
```

Figura 37: Acceso a base de Datos | Archivo app.js
Fuente: El autor
Elaboración: El autor

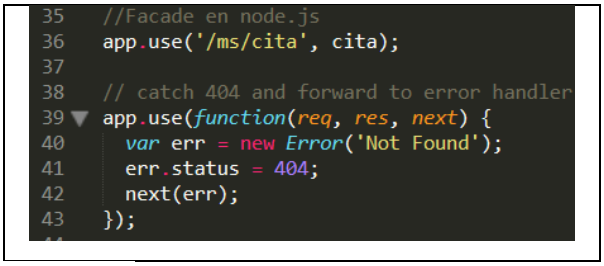
La mencionada configuración permite establecer una conexión directamente a la base de datos empleada.

– **Lógica de Negocio.**

Definido en la codificación como la carpeta *routes*, se escribieron 2 archivos de programación que contienen la lógica necesaria para cumplir con las funcionalidades de los servicios web RESTful.

- **app.js**:

Este archivo contiene la lógica para obtener los recursos del repositorio de datos; es la fachada del patrón de diseño Facade (cabe mencionar que es una adaptación del mismo). Este archivo posee las funcionalidades que resolverán las invocaciones de *cada identificador* previamente definido en la **Tabla 23**. La **Figura 38**, presenta una muestra de la codificación desarrollada, en donde se direcciona al archivo *cita.*

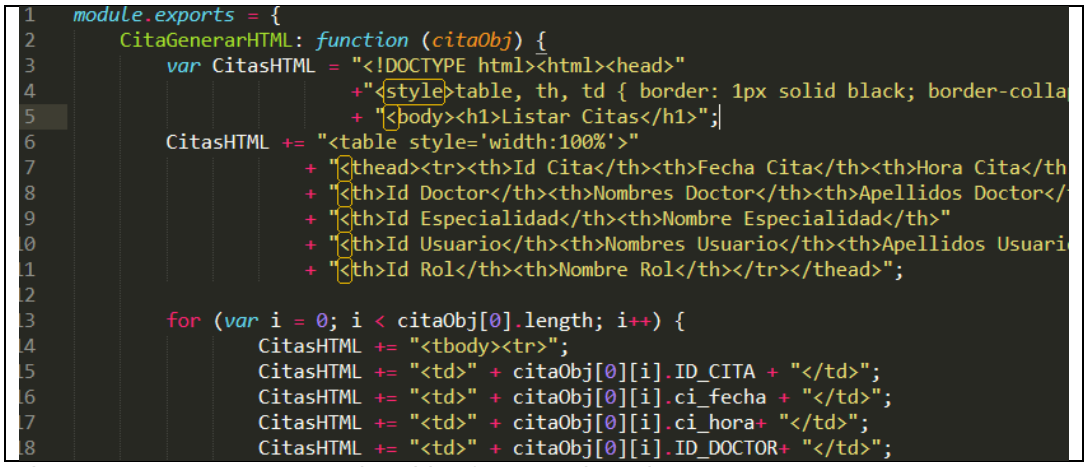


```
//Facade en node.js
app.use('/ms/cita', cita);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});
```

Figura 38: Funcionalidad Facade | Archivo app.js
Fuente: El autor
Elaboración: El autor

- **formato_datos.js**:

Este archivo contiene la codificación correspondiente al formato de los datos de retorno en una tabla *HTML* (debido a la inexistencia de una librería que los genere automáticamente como es el caso de XML, y Json) una porción del código se observa en la **Figura 39**.



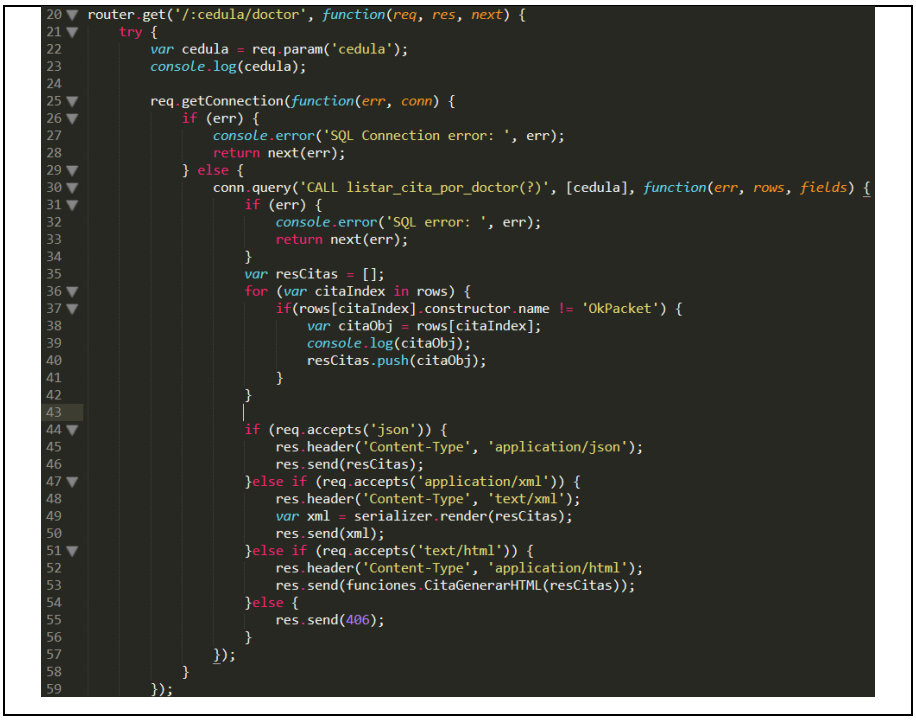
```
module.exports = {
    CitaGenerarHTML: function (citaObj) {
        var CitasHTML = "<!DOCTYPE html><html><head>"
                        +"<style>table, th, td { border: 1px solid black; border-colla
                        + "<body><h1>Listar Citas</h1>";
        CitasHTML += "<table style='width:100%'>"
                    + "<thead><tr><th>Id Cita</th><th>Fecha Cita</th><th>Hora Cita</th
                    + "<th>Id Doctor</th><th>Nombres Doctor</th><th>Apellidos Doctor</
                    + "<th>Id Especialidad</th><th>Nombre Especialidad</th>"
                    + "<th>Id Usuario</th><th>Nombres Usuario</th><th>Apellidos Usuari
                    + "<th>Id Rol</th><th>Nombre Rol</th></tr></thead>";

        for (var i = 0; i < citaObj[0].length; i++) {
                CitasHTML += "<tbody><tr>";
                CitasHTML += "<td>" + citaObj[0][i].ID_CITA + "</td>";
                CitasHTML += "<td>" + citaObj[0][i].ci_fecha + "</td>";
                CitasHTML += "<td>" + citaObj[0][i].ci_hora+ "</td>";
                CitasHTML += "<td>" + citaObj[0][i].ID_DOCTOR+ "</td>";
```

Figura 39: Formato a HTML | Archivo formato_datos.js
Fuente: El autor
Elaboración: El autor

– **Capa de Servicio/Presentación.**

- **cita.js**:

Este archivo implementa los servicios web, y es parte del patrón de diseño implementado. La **Figura 40**, presenta una porción de la codificación desarrollada, en donde se ejecuta el servicio RESTful (el cual hace uso de los procedimientos almacenados creados en el apartado **4.1.3.3.1)**, y se retorna los datos según el formato solicitado (XML, HTML, Json).



```
router.get('/:cedula/doctor', function(req, res, next) {
    try {
        var cedula = req.param('cedula');
        console.log(cedula);

        req.getConnection(function(err, conn) {
            if (err) {
                console.error('SQL Connection error: ', err);
                return next(err);
            } else {
                conn.query('CALL listar_cita_por_doctor(?)', [cedula], function(err, rows, fields) {
                    if (err) {
                        console.error('SQL error: ', err);
                        return next(err);
                    }
                    var resCitas = [];
                    for (var citaIndex in rows) {
                        if(rows[citaIndex].constructor.name != 'OkPacket') {
                            var citaObj = rows[citaIndex];
                            console.log(citaObj);
                            resCitas.push(citaObj);
                        }
                    }

                    if (req.accepts('json')) {
                        res.header('Content-Type', 'application/json');
                        res.send(resCitas);
                    }else if (req.accepts('application/xml')) {
                        res.header('Content-Type', 'text/xml');
                        var xml = serializer.render(resCitas);
                        res.send(xml);
                    }else if (req.accepts('text/html')) {
                        res.header('Content-Type', 'application/html');
                        res.send(funciones.CitaGenerarHTML(resCitas));
                    }else {
                        res.send(406);
                    }
                });
            }
        });
```

Figura 40: Servicio RESTful | Archivo cita.js
Fuente: El autor
Elaboración: El autor

4.1.4.3.3 Despliegue (Servidor)

Para la aplicación de RESTful con Node, se crea un pequeño servidor http (una funcionalidad de Node y que facilita el framework Express a través del archivo *www*) en el cual se utiliza la configuración por defecto y colocamos el puerto que el servicio va a utilizar (como se observa en la **Figura 41**), se realiza la ejecución de la aplicación (ver **Anexo H**: Ejecución de Aplicación Node.js) permitiendo levantar el servicio y consumirlos según las URIs y los puertos definidos en la **Tabla 23.** El diagrama de despliegue de la aplicación se observa en la **Figura 42**.

```
7   var app = require('../app');
8   var debug = require('debug')('serviciorestful:server');
9   var http = require('http');
10
11  /**
12   * Get port from environment and store in Express.
13   */
14
15  //PUERTO A USAR!!!!
16  var port = normalizePort(process.env.PORT || '1600');
17  app.set('port', port);
18
19  /**
20   * Create HTTP server.
21   */
22
23  var server = http.createServer(app);
24
25  /**
26   * Listen on provided port, on all network interfaces.
27   */
28
29  server.listen(port);
30  server.on('error', onError);
31  server.on('listening', onListening);
```

Figura 41: Configuración del Servidor | Archivo www
Fuente: El autor
Elaboración: El autor

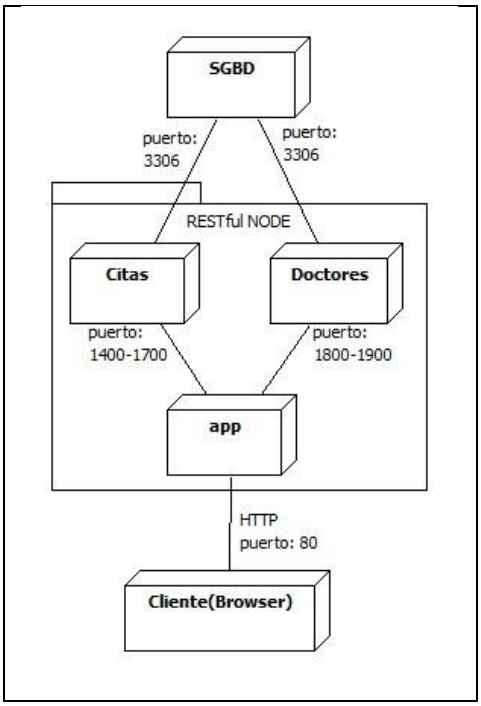


Figura 42: Diagrama de Despliegue aplicación RESTful (NODE).
Fuente: El autor
Elaboración: El autor

4.1.5 Aplicación Microservicios (NODE)

4.1.5.1 Características y Tecnologías

Posteriormente, con la aplicación RESTful desarrollada en Node, se procedió a implementarla como MS, en la **Tabla 24** se observa las tecnologías/herramientas utilizadas en este proceso; el patrón *Server-side Discovery* es implementado también por el API Gateway, el funcionamiento de los patrones aquí mencionados se describió en el apartado **1.1.2.1.**

Tabla 24: Características Aplicación Microservicios (Node)

Recurso	Tecnología	Versión
Patrones	Api Gateway (Alternativa a Facade) *(Communication),* Single Service per Host *(Deployment),* Server-side Discovery *(Discovery)*	
Arquitectura	n layers (capas)	
IDE	SublimeText	3.0.0
BDD	MySQL	5.0.11
Store Procedures	SI (6)	
Servidor Web	Node.js	6.9.2
Lenguaje de Programación	Node.js	6.9.2
Otros	Docker, CSS, HTML, Javascript, RESTful, Express (framework Node).	

Fuente: El autor
Elaboración: El autor

4.1.5.2 Funcionalidades

Se conserva la misma estructura mencionada en el apartado **4.1.4.2,** pero se cambia un poco el identificador RESTful como se observa en la **Tabla 25,** esto por el motivo de la implementación del patrón *API Gateway* para que *p*ueda direccionar adecuadamente las peticiones al MS adecuado y a través del patrón *Server-side Discovery* pueda encontrar el correcto y disponible.

Tabla 25: Relación puerto con servicio RESTful.

Puerto	Aplicación node	Identificador RESTful (Docker con Gateway)
1400	cita_paciente	http://host/aplicación/ms/cita_p/{cedula}/paciente
1500	cita_fecha	http://host/aplicación/ms/cita_f/{fecha}/fecha
1600	cita_doctor	http://host/aplicación/ms/cita_d/{cedula}/doctor
1700	cita_especialidad	http://host/aplicación/ms/cita_e/{especialidad}/especialidad
1800	doctor	http://host/aplicación/ms/doctor_t
1900	doctor_especialidad	http://host/aplicación/ms/doctor_e/{especialidad}

Fuente: El autor
Elaboración: El autor

4.1.5.3 Implementación

Para detallar el desarrollo y despliegue de la aplicación Microservicios (Node) se consideran 3 niveles de implementación:

- Datos.
- Codificación.
- Despliegue (Servidor).

Primeramente se realizó la instalación de Docker, teniendo en cuenta la arquitectura que posee (ver **Anexo I**: Arquitectura de Docker) y las herramientas relacionadas que permiten el desarrollo y despliegue de la aplicación (ver **Anexo J**: Instalación y Configuración de Docker en Windows 8.1), a partir de lo cual se obtuvo el respectivo diagrama de componentes (**Figura 43**).

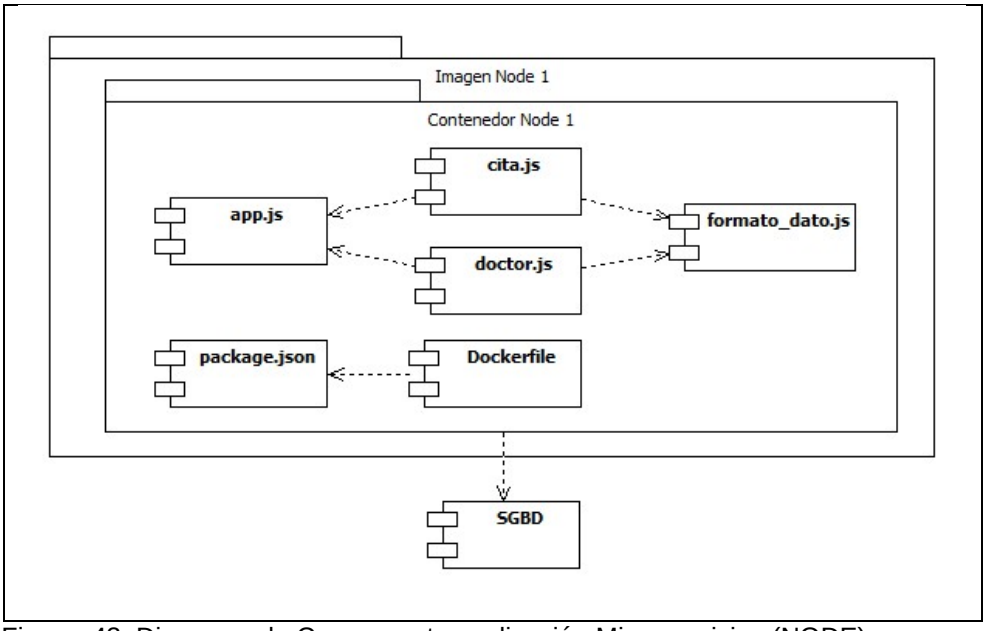


Figura 43: Diagrama de Componentes aplicación Microservicios (NODE).
Fuente: El autor
Elaboración: El autor

La arquitectura a implementar se observa en la **Figura 44**, en la cual se tiene el repositorio web de Docker del cual se utilizan solo 2 imágenes de la gran cantidad existente (**Figura 44 - 1)**, las mismas que se las descarga en el host Docker local para poder usarlas (**Figura 44 - 2)**, a partir de ellas se crean los respectivos contenedores (**Figura 44 - 3)**, para que finalmente un cliente las pueda utilizar (**Figura 44 - 4)**, todo esto controlado por el Daemon de Docker, el cual también tiene comunicación con la base de datos, siendo esta externa a Docker; al momento de que un cliente quiere acceder a un MS proporcionado por un contenedor que se encuentra en el host Docker, siempre se comunica con el Daemon de Docker, el cual es el encargado de

controlar y comunicar todos los componentes pertenecientes al Host Docker local entre ellos, como con componentes externos (como son los clientes, la Base de datos, o el repositorio Web de Docker como se observa en el gráfico.)

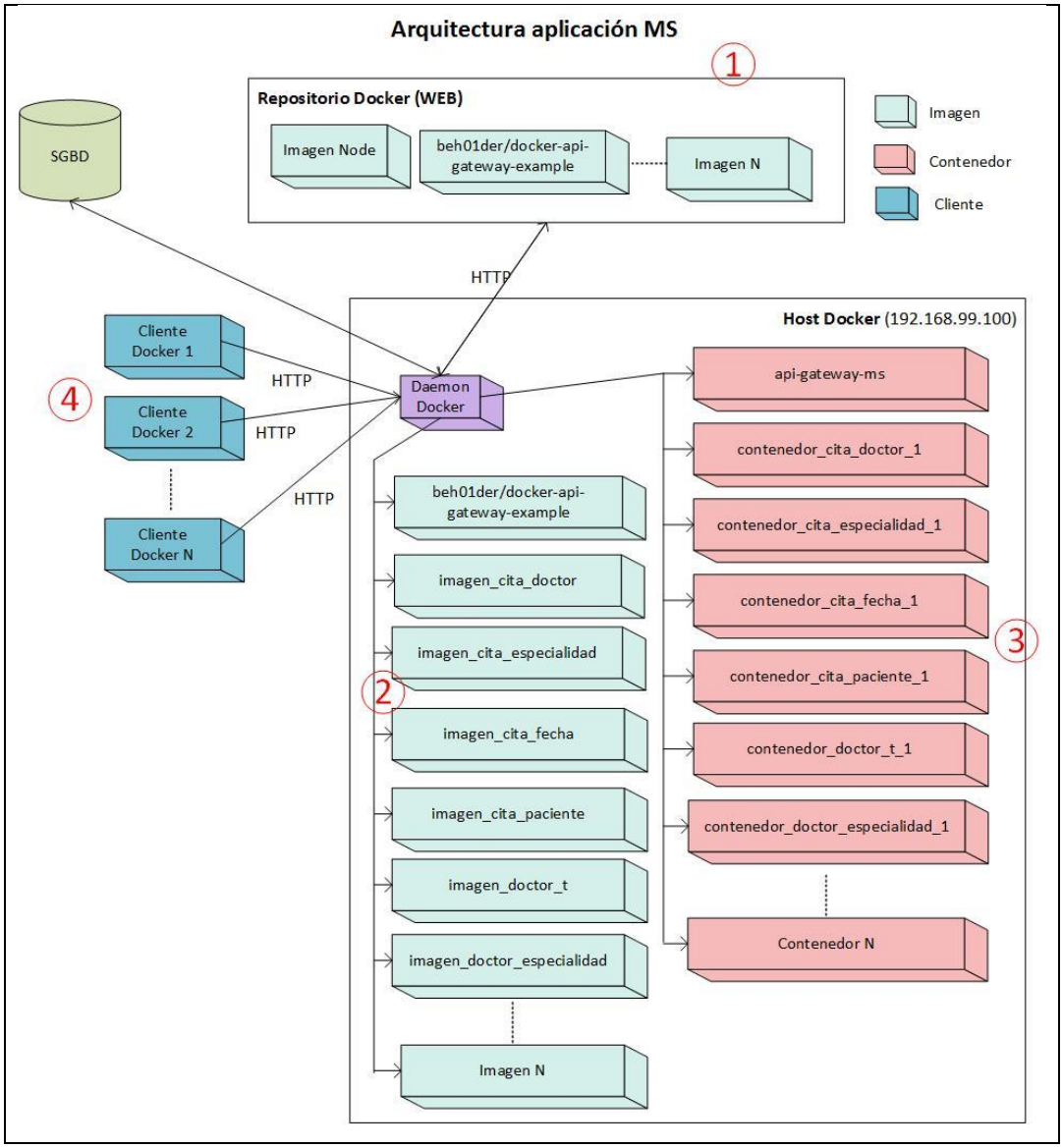


Figura 44: Arquitectura de aplicación Microservicios (NODE) propuesta.
Fuente: El autor
Elaboración: El autor

4.1.5.3.1 Datos

Se utiliza la misma base de datos mencionada en el apartado **4.1.3.3.1.**

4.1.5.3.2 Codificación

Se utiliza las mismas aplicaciones generadas en el apartado **4.1.4.3.2,** a las cuales se les agrega el archivo de configuración *Dockerfile* (el cual no posee una extensión) como se expone en la **Figura 45** (se observan las 6 aplicaciones Node, la estructura que poseen es la misma que posee *cita_doctor* cada una posee un archivo *Dockerfile*).

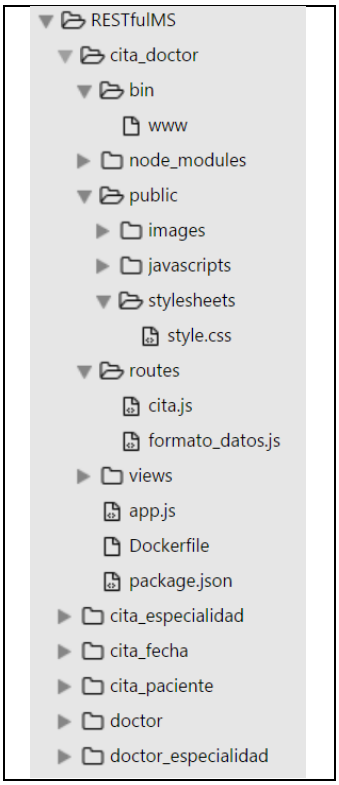


Figura 45: Estructura de aplicación MS (Node)
Fuente: El autor
Elaboración: El autor

Seguidamente se especifica las implicaciones que conllevó la implementación de la aplicación *cita_doctor* (el procedimiento y estructura es similar para las otras 5 aplicaciones) en Docker, para la creación de la aplicación ver **Fuente:** El autor

Elaboración: El autor

Anexo K: Desarrollo de Aplicación en Docker**.**

- **Acceso a Datos.**

Definido en la codificación como *app.js,* este archivo de programación realiza la interacción con la base de datos, es decir las solicitudes de datos; una porción de este archivo se observa en la **Figura 46**.

```
//Conexion con MYSQL
var mysql = require('mysql');
var connection=require('express-myconnection');

var app = express();

//Crear conexion SQL
app.use(connection(mysql,{
host:"192.168.99.1",
user:"root",
password:"",
database:"tesis_citas_medicas"
},'request'));
```

Figura 46: Acceso a Base de Datos | Archivo app.js
Fuente: El autor
Elaboración: El autor

La mencionada configuración permite establecer una conexión directamente a la base de datos empleada (ya no es *localhost*, ya que por el uso de contenedores se manejan direcciones IPs).

- **Lógica de Negocio.**

Definido en la codificación como la carpeta *routes*, se escribieron 2 archivos de programación que contienen la lógica necesaria para cumplir con las funcionalidades de los servicios web RESTful.

- **app.js**:

Este archivo contiene la lógica para obtener los recursos del repositorio de datos; es la fachada del patrón de diseño Facade. Este archivo posee las funcionalidades que resolverán las invocaciones de *cada identificador* previamente definido en la **Tabla 25.** La **Figura 47**, presenta una muestra de la codificación desarrollada, en donde se direcciona al archivo *cita.*

```
//Facade en node.js
app.use('', cita);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});
```

Figura 47: Funcionalidad Facade | Archivo app.js
Fuente: El autor
Elaboración: El autor

- **formato_datos.js**:

Se mantiene sin cambios (ver **Figura 39**)

- **Capa de Servicio/Presentación.**
  - **cita.js**:

Se mantiene sin cambios (ver **Figura 40**)

4.1.5.3.3 Despliegue (Servidor)

Similar al apartado **4.1.4.3.3,** con la diferencia de que se realiza la ejecución con Docker (ver **Anexo L**: Ejecución de Aplicación en Docker) permitiendo levantar los servicios y consumirlos según las URIs definidas en la **Tabla 25**, sin la necesidad de indicar los puertos en la URI para el consumo, esto por la intervención del *API* Gateway (ver **Anexo M**: API Gateway) . El diagrama de despliegue de la aplicación se observa en la **Figura 48**.

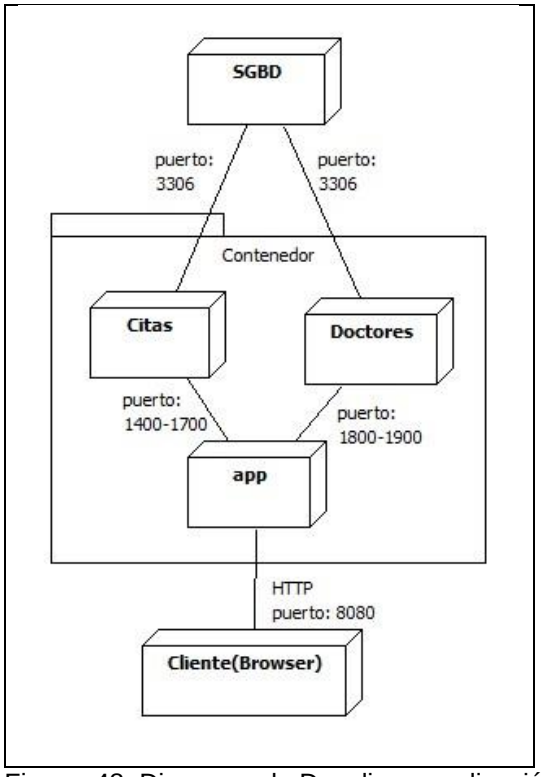


Figura 48: Diagrama de Despliegue aplicación Microservicios (NODE).
Fuente: El autor
Elaboración: El autor

4.2 Validación de la aplicación del Modelo Propuesto

Los resultados del modelo definido y propuesto en el **CAPITULO 3**:**,** con el que se evaluó a las aplicaciones se indican a continuación, cabe mencionar que en el Modelo *LISI*, para clasificar a la aplicación en un nivel de interoperabilidad se debe cumplir con los 4 atributos PAID en dicho nivel; además el resultado obtenido en SOSI, está realizado según algunas consideraciones mencionadas en el apartado **1.1.2.3.**

Según se observa en la **Tabla 26** la interoperabilidad alcanzada por la aplicación Monolítica codificada con PHP de acuerdo a cada modelo es:

1. Con el modelo LISI se obtiene que la aplicación posee un nivel Aislado: Nivel no conocido de interoperabilidad (G0) según lo diferentes atributos identificados en la misma.

2. Con el modelo Stoplight se obtiene un resultado que se encuentra clasificado como *Color Naranja*, esto debido a que la aplicación cumple su conjunto de requisitos de interoperabilidad (intercambio de datos/información básico), pero tiene problemas de interoperabilidad conocidos (debido a que solo implementa 1 formato de datos a intercambiar (HTML)).

3. A través del modelo SOSI se toma en cuenta el cumplimiento de los 5 ítems mencionados, y a partir de los cuales se les asigna la interoperabilidad correspondiente que cumplen (La *Medida potencial de interoperabilidad es* Inicial, debido a se requiere fuertes esfuerzos que afectan a la asociación; y además se aplica la *Interoperabilidad Semántica,* debido a que la información es entendible y tiene el significado adecuado según el contexto*).*

Tabla 26: Evaluación de aplicación Monolítica PHP

<table>
<tr><th>MODELO</th><th>NUMERO DE ITEM</th><th colspan="7">DETALLE ITEM</th><th>RESULTADO</th></tr>
<tr><td rowspan="14">LISI</td><td rowspan="14">Ítem 1</td><td colspan="3" rowspan="2">NIVEL (Ambiente)</td><td colspan="4">Atributos de Interoperabilidad</td><td rowspan="14">G0</td></tr>
<tr><td colspan="2">Procedimientos</td><td>Aplicaciones</td><td>Infraestructura</td><td>Datos</td></tr>
<tr><td rowspan="3">Nivel Empresa</td><td rowspan="3">4</td><td>c</td><td colspan="2">N/A</td><td rowspan="3">N/A</td><td rowspan="3">N/A</td><td rowspan="2">N/A</td></tr>
<tr><td>b</td><td colspan="2">N/A</td></tr>
<tr><td>a</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td rowspan="3">Nivel de Dominio</td><td rowspan="3">3</td><td>c</td><td colspan="2" rowspan="3">N/A</td><td>N/A</td><td rowspan="3">N/A</td><td>SQL</td></tr>
<tr><td>b</td><td>N/A</td><td rowspan="2">N/A</td></tr>
<tr><td>a</td><td>N/A</td></tr>
<tr><td rowspan="2">Nivel funcional</td><td rowspan="2">2</td><td>c</td><td colspan="2" rowspan="2">N/A</td><td>Chrome, Firefox, IE</td><td rowspan="2">IP LAN</td><td rowspan="3">.php
HTML</td></tr>
<tr><td>b</td><td>Operaciones básicas con Interfaz WEB (Informes, Datos)</td></tr>
<tr><td>Nivel Conectado</td><td>1</td><td>a</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td rowspan="4">Nivel Conectado</td><td rowspan="4">1</td><td>d</td><td colspan="2" rowspan="2">Web Interface Design Tips</td><td>N/A</td><td rowspan="3">Modem</td><td rowspan="3">Informes simples</td></tr>
<tr><td>c</td><td>N/A</td></tr>
<tr><td>b</td><td colspan="2" rowspan="2">N/A</td><td rowspan="2">Interacción Simple</td></tr>
<tr><td>a</td><td>N/A</td></tr>
<tr><td rowspan="4">Nivel Aislado</td><td rowspan="4">0</td><td>d</td><td colspan="2">De inicio de Sesión</td><td rowspan="4">N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td>c</td><td rowspan="3">Control de acceso manual</td><td>N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td>b</td><td>N/A</td><td rowspan="2">N/A</td><td rowspan="2">N/A</td></tr>
<tr><td>a</td><td>N/A</td></tr>
<tr><td colspan="3">0</td><td colspan="4">Interoperabilidad no conocida</td></tr>
<tr><td rowspan="4">Stoplight</td><td rowspan="4">Ítem 1</td><td colspan="3"></td><td colspan="4">Cumple con los requisitos de adquisición?</td><td rowspan="4">Color Naranja.</td></tr>
<tr><td colspan="3"></td><td colspan="2">SI</td><td colspan="2">NO</td></tr>
<tr><td rowspan="2">Cumple con los requisitos operativos?</td><td>SI</td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td>NO</td><td colspan="2">X</td><td colspan="2"></td></tr>
<tr><td rowspan="5">SOSI</td><td>Ítem 1</td><td>Arquitectura</td><td colspan="7">No posee</td><td rowspan="5">Medida potencial de interoperabilidad = Inicial
Interoperabilidad Semántica</td></tr>
<tr><td>Ítem 2</td><td>Implementación frameworks</td><td colspan="7">No posee</td></tr>
<tr><td>Ítem 3</td><td>Especificaciones/formato de datos</td><td colspan="7">HTML</td></tr>
<tr><td>Ítem 4</td><td>Protocolos de comunicación</td><td colspan="7">HTTP</td></tr>
<tr><td>Ítem 5</td><td>Patrones/Lenguajes</td><td colspan="7">No implementa Patrones, utiliza el lenguaje PHP.</td></tr>
</table>

Fuente: El autor
Elaboración: El autor

Por su parte el nivel de interoperabilidad alcanzado por la aplicación *RESTful Java* se observa en la **Tabla 27,** en la cual:

1. El modelo LISI nos indica que la aplicación posee un *Nivel Funcional* de interoperabilidad (G2c) según los diferentes atributos identificados en la misma.

2. Con el modelo Stoplight se obtiene un resultado que se encuentra clasificado como *Color Verde*, esto debido a que la aplicación cumple su conjunto de requisitos de interoperabilidad (intercambio de datos/información básico), además posee 3 formatos de intercambio diferentes (por lo tanto no posee problemas de interoperabilidad conocidos).

3. A través del modelo SOSI se toma en cuenta el cumplimiento de los 5 ítems mencionados, y a partir de los cuales se les asigna la interoperabilidad correspondiente que cumplen (La *Medida potencial de interoperabilidad* es Ejecutable, debido a que la interoperabilidad es posible incluso si el riesgo de encontrar problemas es alto (debido a que se implementan en una sola ubicación física); además se aplica la *Interoperabilidad Técnica por* el motivo de que la información puede ser intercambiada directa y satisfactoriamente, la *Interoperabilidad Sintáctica* referente a la implementación de varios formatos de datos los cuales poseen una sintaxis y codificación bien definidas y *la Interoperabilidad Semántica*, debido a que la información es entendible y tiene el significado adecuado según el contexto)

Tabla 27: Evaluación de aplicación RESTful Java

<table>
<tr><th>MODELO</th><th>NUMERO DE ITEM</th><th colspan="7">DETALLE ITEM</th><th>RESULTADO</th></tr>
<tr><td rowspan="15">LISI</td><td rowspan="15">Ítem 1</td><td colspan="3" rowspan="2">NIVEL (Ambiente)</td><td colspan="4">Atributos de Interoperabilidad</td><td rowspan="15">G2c</td></tr>
<tr><td>Procedimientos</td><td>Aplicaciones</td><td>Infraestructura</td><td>Datos</td></tr>
<tr><td rowspan="3">Nivel Empresa</td><td rowspan="3">4</td><td>c</td><td>N/A</td><td rowspan="2">N/A</td><td rowspan="2">N/A</td><td>N/A</td></tr>
<tr><td>b</td><td>N/A</td><td>N/A</td></tr>
<tr><td>a</td><td>N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td rowspan="3">Nivel de Dominio</td><td rowspan="3">3</td><td>c</td><td rowspan="3">N/A</td><td>Datos Compartidos (Situación Muestra los Intercambios Directos de DB)</td><td rowspan="3">N/A</td><td>SQL</td></tr>
<tr><td>b</td><td>N/A</td><td rowspan="2">N/A</td></tr>
<tr><td>a</td><td>N/A</td></tr>
<tr><td rowspan="3">Nivel funcional</td><td rowspan="3">2</td><td>c</td><td rowspan="3">Se centra en el nivel del servicio individual (especificando las funcionalidades que los servicios deben soportar)</td><td>Chrome, Firefox, IE</td><td rowspan="3">IP LAN</td><td rowspan="3">XML HTML JSON</td></tr>
<tr><td>b</td><td>Operaciones básicas sin Interfaz WEB (Informes, Datos)</td></tr>
<tr><td>a</td><td>N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td rowspan="4">Nivel Conectado</td><td rowspan="4">1</td><td>d</td><td rowspan="2">RESTful Design Tips</td><td>Mensajería básica (Texto sin formato)</td><td rowspan="3">Modem</td><td rowspan="3">Informes simples</td></tr>
<tr><td>c</td><td>N/A</td></tr>
<tr><td>b</td><td rowspan="2">N/A</td><td rowspan="2">Interacción Simple</td></tr>
<tr><td>a</td><td>N/A</td></tr>
<tr><td rowspan="5">Nivel Aislado</td><td rowspan="5">0</td><td>d</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td>c</td><td>N/A</td><td>N/A</td><td rowspan="4">N/A</td><td rowspan="4">N/A</td><td rowspan="4"></td></tr>
<tr><td>b</td><td>N/A</td></tr>
<tr><td>a</td><td>N/A</td></tr>
<tr><td>0</td><td colspan="5">Interoperabilidad no conocida</td></tr>
<tr><td rowspan="2">Stoplight</td><td rowspan="2">Ítem 1</td><td colspan="3"></td><td colspan="4">Cumple con los requisitos de adquisición?</td><td rowspan="2">Color Verde</td></tr>
<tr><td colspan="3"></td><td colspan="2">SI</td><td colspan="2">NO</td></tr>
</table>

		Cumple con los requisitos operativos?	**SI**	X		
			NO			
SOSI	**Ítem 1**	Arquitectura	N layers (capas)			**Medida potencial de interoperabilidad =** Ejecutable **Interoperabilidad Técnica** **Interoperabilidad Sintáctica** **Interoperabilidad Semántica**
	Ítem 2	Implementación frameworks	No posee			
	Ítem 3	Especificaciones/formato de datos	HTML, JSON, XML			
	Ítem 4	Protocolos de comunicación	HTTP			
	Ítem 5	Patrones/Lenguajes	Patrón Facade. Lenguaje JAVA EE			

Fuente: El autor
Elaboración: El autor

Finalmente en la **Tabla 28** se observa la interoperabilidad alcanzada por la aplicación de MS codificada con Node, en la cual:

1. Según el modelo LISI se obtiene que la aplicación posee un *Nivel Funcional* de interoperabilidad (G2c) según los diferentes atributos identificados en la misma.

2. Atraves del modelo Stoplight se obtiene un resultado que se encuentra clasificado como *Color Verde*, esto debido a que la aplicación cumple su conjunto de requisitos de interoperabilidad (intercambio de datos/información básico), además posee 3 formatos de intercambio diferentes (por lo tanto no posee problemas de interoperabilidad conocidos).

3. Con el modelo SOSI se toma en cuenta el cumplimiento de los 5 ítems mencionados, y a partir de los cuales se les asigna la interoperabilidad correspondiente que cumplen (La *Medida potencial de interoperabilidad* es Interoperable, debido a que considera la evolución de los niveles de interoperabilidad y donde el riesgo de resolver problemas es débil (debido a que se implementan en diferentes ubicaciones, por la implementación de Docker); además se aplica la *Interoperabilidad Técnica por* el motivo de que la información puede ser intercambiada directa y satisfactoriamente, la *Interoperabilidad Sintáctica* referente a la implementación de varios formatos de datos los cuales poseen una sintaxis y codificación bien definidas y *la Interoperabilidad Semántica*, debido a que la información es entendible y tiene el significado adecuado según el contexto).

Tabla 28: Evaluación de aplicación Microservicios (NODE)

<table>
<tr><th>MODELO</th><th>NUMERO DE ITEM</th><th colspan="7">DETALLE ITEM</th><th>RESULTADO</th></tr>
<tr><td rowspan="15">LISI</td><td rowspan="15">Ítem 1</td><td colspan="3" rowspan="2">NIVEL (Ambiente)</td><td colspan="6">Atributos de Interoperabilidad</td><td rowspan="15">G2c</td></tr>
<tr><td colspan="2">Procedimientos</td><td>Aplicaciones</td><td>Infraestructura</td><td colspan="2">Datos</td></tr>
<tr><td rowspan="3">Nivel Empresa</td><td rowspan="3">4</td><td>c</td><td colspan="2">N/A</td><td>N/A</td><td rowspan="3">N/A</td><td colspan="2">N/A</td></tr>
<tr><td>b</td><td colspan="2">N/A</td><td rowspan="2"></td><td colspan="2"></td></tr>
<tr><td>a</td><td colspan="2">N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
<tr><td rowspan="3">Nivel de Dominio</td><td rowspan="3">3</td><td>c</td><td colspan="2" rowspan="3">N/A</td><td>Datos Compartidos (Intercambios Directos de DB)</td><td rowspan="3">IP WAN (Posible Mediante Docker)</td><td colspan="2">SQL</td></tr>
<tr><td>b</td><td>N/A</td><td colspan="2"></td></tr>
<tr><td>a</td><td>N/A</td><td colspan="2"></td></tr>
<tr><td rowspan="2">Nivel funcional</td><td rowspan="2">2</td><td>c</td><td colspan="2" rowspan="2">Se centra en el nivel del microservicio individual (especificando las funcionalidades que los microservicios deben soportar)</td><td>Chrome, Firefox, IE</td><td rowspan="2">IP LAN</td><td colspan="2" rowspan="2">XML
HTML
JSON</td></tr>
<tr><td>b</td><td>Operaciones básicas sin Interfaz WEB (Informes, Datos)</td></tr>
<tr><td>a</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td rowspan="4">Nivel Conectado</td><td rowspan="4">1</td><td>d</td><td colspan="2" rowspan="2">RESTful Design Tips</td><td>Mensajería básica (Texto sin formato)</td><td rowspan="3">Modem</td><td colspan="2" rowspan="3">Informes simples</td></tr>
<tr><td>c</td><td></td></tr>
<tr><td>b</td><td colspan="2" rowspan="2">N/A</td><td rowspan="2">Interacción Simple</td></tr>
<tr><td>a</td><td>N/A</td></tr>
<tr><td rowspan="5">Nivel Aislado</td><td rowspan="5">0</td><td>d</td><td colspan="2">N/A</td><td rowspan="5">N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
<tr><td>c</td><td>N/A</td><td>N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
<tr><td>b</td><td colspan="2">N/A</td><td rowspan="3">N/A</td><td rowspan="3">N/A</td></tr>
<tr><td>a</td><td colspan="2">N/A</td></tr>
<tr><td>0</td><td colspan="6">Interoperabilidad no conocida</td></tr>
<tr><td rowspan="3">Stoplight</td><td rowspan="3">Ítem 1</td><td colspan="2"></td><td colspan="6">Cumple con los requisitos de adquisición?</td><td rowspan="3">Color Verde</td></tr>
<tr><td colspan="3"></td><td colspan="3">SI</td><td colspan="2">NO</td></tr>
<tr><td colspan="2">Cumple con los</td><td>SI</td><td colspan="3">X</td><td colspan="2"></td></tr>
</table>

		requisitos operativos?	**NO**			
SOSI	**Ítem 1**	Arquitectura	N layers (capas)			**Medida potencial de interoperabilidad =** Interoperable **Interoperabilidad Técnica** **Interoperabilidad Sintáctica** **Interoperabilidad Semántica**
	Ítem 2	Implementación frameworks	Express (framework Node).			
	Ítem 3	Especificaciones/formato de datos	HTML, JSON, XML			
	Ítem 4	Protocolos de comunicación	HTTP			
	Ítem 5	Patrones/Lenguajes	Patrones: API Gateway, Single Service per Host, Server-side Discovery. Lenguaje: Node.js			

Fuente: El autor
Elaboración: El autor

Según estos resultados, en la **Tabla 29** se muestra un cuadro comparativo de las aplicaciones (en orden de menor a mayor, en cuanto a interoperabilidad alcanzada); a través de la cual se observa que en *LISI* todas poseen el nivel genérico de interoperabilidad (G), por el motivo de que se trata de la comparación entre un sistema con el modelo PAID; según esto se observa que la aplicación de la interoperabilidad de la aplicación Monolítica (PHP) es Básica, mientras que RESTful Java y MS (Node) son más completas, con la diferencia de que en esta ultima la *Medida potencial de interoperabilidad* es la máxima posible (Interoperable), frente a la de RESTful Java que es intermedia (Ejecutable).

Tabla 29: Comparativa de Interoperabilidad de aplicaciones

	Aplicación Monolítica (PHP)	**RESTful Java**	**MS (Node)**
LISI	G0	G2c	G2c
Stoplight	Color Naranja	Color Verde	Color Verde
SOSI	Medida potencial de interoperabilidad = Inicial Interoperabilidad Semántica	Medida potencial de interoperabilidad = Ejecutable Interoperabilidad Técnica Interoperabilidad Sintáctica Interoperabilidad Semántica	Medida potencial de interoperabilidad = Interoperable Interoperabilidad Técnica Interoperabilidad Sintáctica Interoperabilidad Semántica

Fuente: El autor
Elaboración: El autor

CONCLUSIONES

Al finalizar el presente trabajo de titulación se concluye:

- Los Microservicios permiten convertir una aplicación grande (Monolítica) como un conjunto de pequeñas aplicaciones (microservicios) que se pueden desarrollar, implementar, escalar, operar y monitorear independientemente.

- Los microservicios se comunican mediante una interfaz bien definida, normalmente REST, siendo HTTP el método más común para la comunicación entre microservicios al ser un mecanismo de peso ligero, permitiendo responder a las solicitudes, teniendo en cuenta que tanto el cliente como el servicio tienen que estar disponibles para que esto suceda correctamente.

- Los Microservicios aplican la separación clara de funciones, en la cual cada uno realiza una única función; pudiendo ser desplegados de una manera independiente, esto es posible a la implementación de contenedores (Docker), los cuales proporciona estas características, permitiendo que cada MS sea totalmente independiente.

- Para actualizar, reparar o reemplazar un Microservicio, no es necesario reconstruir la aplicación completa, basta con cambiar la parte que lo necesita; además solo el microservicio bajo análisis tiene que someterse a pruebas rigurosas y el resto de la aplicación no se ve afectada, por lo tanto el ahorro de tiempo es significativo, ya que no se pone a prueba toda la aplicación, sino solo a la funcionalidad especificada; estas características son posibles debido a la implementación de contenedores (Docker) en los cuales cada uno realiza una funcionalidad, y se los gestiona individualmente.

- Las aplicaciones de Microservicios se consideran autónomas en cuanto a su funcionalidad y funcionamiento, pero colaboran con otras aplicaciones para contribuir a las metas de la aplicación *superior* (conjunto de aplicaciones de Microservicios).

- La interoperabilidad es primordial en MS, debido a que el intercambio de datos entre las aplicaciones es constante, al ser funcionalidades que no dependen entre sí, por lo tanto es necesario establecer la forma de interacción

(interoperabilidad) entre las aplicaciones, mediante protocolos de comunicación y formatos de datos.

- Los modelos que permiten evaluar la interoperabilidad proporcionan una vista parcial de la misma, por lo que fue necesario proponer una combinación de los más adecuados para el presente trabajo siendo estos LISI, SOSI, Stoplight.

- La interoperabilidad es manejada a distintos niveles como son: Técnica (se centra en los protocolos de comunicación y la infraestructura necesaria para que esos protocolos funcionen), Sintáctica (se asocia con formatos de datos, que deben poseer una sintaxis y codificación bien definidas.), y Semántica (la capacidad de operar con los datos intercambiados de acuerdo con la semántica acordada);

- La tecnología Docker es útil para el desarrollo y despliegue de aplicaciones de microservicios, ya que permite el uso de imágenes y contenedores, con lo cual las incompatibilidades de tecnologías y hardware se “eliminan”.

- El componente principal de Docker es el Daemon, el cual es el encargado de gestionar el acceso y salida de las peticiones realizadas al Host Docker Local.

- Los contenedores (en Docker) siempre se crean a partir de una imagen; además a partir de una imagen se pueden crear más de un contenedor; por lo tanto para eliminar una imagen se debe primero eliminar los contenedores creados a partir de la misma.

- La implementación de Restart Policies a los contenedores permite a los mismos “levantarse” automáticamente en caso de que “caigan” por fallas; siendo de mucha utilidad al momento de proporcionar disponibilidad de los MS, como se implementó en la aplicación de MS codificado con Node.

- Los *fallos* o *caídas* de un contenedor no afectan al funcionamiento de la aplicación, gracias al aislamiento proporcionado por los contenedores, siendo tolerantes a fallos.

- El error *502 Bad Gateway* (implementado en el API Gateway), permite conocer al cliente que la URL solicitada no se encuentra registrada; es decir existe un error en la URL, al momento de acceder al Microservicio.

- La implicación de teorías de arquitectura de software, beneficia al desarrollo de software; permitiendo desarrollar aplicaciones más interoperables, como es el caso de las aplicaciones desarrolladas en este trabajo.

- La aplicación monolítica (tanto en PHP, como en Java EE) es una aplicación compleja, de tamaño considerable; por lo tanto es complicado la implementación de cambios y detección de errores ágilmente; y además para realizarlo se la tiene q dar de baja completamente, hasta culminar los cambios planteados.

- El desarrollo de servicios web RESTful permite total independencia de tecnologías y lenguajes de programación; pues su lenguaje de respuesta es universal y permite la intercomunicación entre distintos aplicativos de software que implemente cualquier tecnología, siendo primordiales para lograr la interoperabilidad.

- La aplicación desarrollada en RESTful JAVA, implementa el patrón Facade el cual contribuye a la estructuración del código fuente, mantiene interfaces de comunicación aisladas, además tiene relación con la interoperabilidad ya que existe un único punto de accedo (*fachada)* desde el cual se direcciona al servicio solicitado para retornar una respuesta.

- El desarrollo de RESTful en Node.js es un paso necesario para construir una aplicación basada en MS, debido a que Node es una tecnología “liviana” que permite la creación de aplicaciones Web que consumen pocos recursos, y por lo tanto útil para la implementación de funcionalidades pequeñas.

- Las aplicaciones en RESTful Node.js implementadas con DOCKER, tienen un único punto de acceso HTTP (API Gateway) donde cada MS es mapeado y tiene su propia ruta a la cual se realizan las solicitudes/respuestas HTTP a clientes externos; además del cambio dinámico de las direcciones al momento de levantar/reiniciar/deploy un MS.

- El patrón API Gateway (en la aplicación RESTful Node .js implementada en Docker), está configurado dinámicamente y es el único componente disponible (visible) para usuarios externos; por lo tanto todas las peticiones tienen que pasar el API Gateway, el cual se encarga de realizar la redirección al servicio solicitado, en caso de existir.

- La implementación del API Gateway es primordial en una arquitectura de MS; este patrón implementa también a Server side Discovery ya que realiza el descubrimiento de servicios existentes en el Docker Local dinámicamente.

- La aplicación de XML, JSON, y HTML como formato de intercambio de datos permite gestionar contenidos más livianos, e interoperables para las aplicaciones.

RECOMENDACIONES

Al finalizar el presente trabajo de titulación se recomienda:

- Para iniciar con el desarrollo de MS, se recomienda usar una única base de datos, debido a problemas con la consistencia de los mismos; hasta ir familiarizándose con este tipo de Arquitectura y poder implementar la denominada persistencia Polyglott.

- Utilizar las herramientas que proporciona la tecnología Docker para el desarrollo y despliegue de aplicaciones (sean o no MS), debido a la baja demanda de recursos que realizan debido a la implementación de solo las funcionalidades necesarias.

- La implementación (por lo menos al inicio), de la tecnología Node, para el desarrollo de aplicativos web basados en la tecnología de MS, debido a que esta tecnología tiene un bajo consumo de recursos, pero es lo necesariamente óptima y útil para la realización de este tipo de aplicativos.

- Revisar las configuraciones de todos los firewalls existentes en el equipo para evitar bloqueos en la comunicación de las aplicaciones.

- La implementación de patrones, frameworks y de varios formatos de intercambio de datos, para asegurar la interoperabilidad de las aplicaciones.

- Al ser un tema “relativamente” nuevo, existe documentación que se está generando y publicando recientemente, ya sea para respaldar o criticar este tipo de arquitectura, se recomienda periódicamente realizar la búsqueda de información nueva.

BIBLIOGRAFÍA:

Alexandre Eleutério Santos Lourenço. (2017). DOCKER: Using Containers to implement a Microservices Architecture. Retrieved August 22, 2017, from https://alexandreesl.com/2016/01/08/docker-using-containers-to-implement-a-microservices-architecture/

Best Manufacturing Practices. (2008). LISI Model: Levels Of Information Systems Interoperability (LISI) Reference Model. Retrieved March 20, 2017, from http://www.bmpcoe.org/library/books/lisi model/

Chausenko, A. (2016). API Gateway for Dockerized Microservices. Retrieved July 18, 2017, from https://memz.co/api-gateway-microservices-docker-node-js/

Connor, R. V. O., Elger, P., & Clarke, P. M. (2016). Exploring the impact of situational context – A case study of a software development process for a microservices architecture, 6–10. http://doi.org/10.1145/2904354.2904368

Davies, M., Fensel, A., Carrez, F., Narganes, M., Urdiales, D., Danado, J., … Danado, J. (2010). Defining user-generated services in a semantically-enabled mobile platform. *iiWAS2010 - 12th International Conference on Information Integration and Web-Based Applications and Services*, 333–340. http://doi.org/10.1145/1967486.1967539

Docker. (2017). Docker. Retrieved August 22, 2017, from https://www.docker.com/

Ducq, Y., & Chen, D. (2008). How to measure interoperability: Concept and Approach. *14th International Conference on …*. Retrieved from http://www.ami-communities.net/pub/bscw.cgi/S4e4a7754/d494846/070 - 17.doc

Fowler, M. (2015). Microservice Trade-Offs. Retrieved November 15, 2016, from http://martinfowler.com/articles/microservice-trade-offs.html

Gadea, C., & Ionescu, D. (2016). A Reference Architecture for Real-time Microservice API Consumption. *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, 2:1–2:6. http://doi.org/10.1145/2904111.2904115

Hasselbring, W. (2016). Microservices for Scalability: Keynote Talk Abstract. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 133–134. http://doi.org/10.1145/2851553.2858659

Kasunic, M., & Anderson, W. (2004). Measuring Systems Interoperability Challenges and Opportunities, (April), 63.

Kasunic, M., Polyakov, a V, & Ksenofontov, M. a. (2001). Measuring Systems Interoperability, (41).

Killalea, T. O. M. (2016). The Hidden Dividends of Microservices, (june), 1–10. http://doi.org/10.1145/2956641.2956643

Knoche, H. (2016). Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices, 121–124. http://doi.org/10.1145/2851553.2892039

Lambolais, T., Courbis, A. L., Luong, H. V., & Phan, T. L. (2011). *Interoperability analysis of systems*. *IFAC Proceedings Volumes (IFAC-PapersOnline)* (Vol. 18). IFAC. http://doi.org/10.3182/20110828-6-IT-1002.03523

Levine, L., Meyers, B. C., & Place, P. R. H. (2003). Proceedings of the System of Systems Interoperability Workshop (February 2003). *Software Engineering Institute*, (February).

Metrics, T. C., Concepts, C., Ford, A. T. C., Colombi, J. M., Graham, S. R., Jacques, D. R., … Scott R. Graham, D. R. J. T. C. F. J. M. C. (2007). Submission to the Call for Papers : 12th ICCRTS " Adapting C2 to the 21 st Century ." *Information Systems*, (937), 28.

Meyers, B. C., Levine, L., Morris, E., Place, P., & Plakosh, D. (2004). SOSI : System of Systems Interoperability. *Software Engineering Institute*, (January).

Microsoft. (2009). Chapter 16: Quality Attributes. Retrieved December 10, 2016, from https://msdn.microsoft.com/en-us/library/ee658094.aspx

Node. (2017). Node.js. Retrieved August 22, 2017, from https://nodejs.org/es/

Renz, J., Hoffmann, D., Staubitz, T., & Meinel, C. (2016). Using A / B Testing in MOOC Environments. *Lak*, 304–313. http://doi.org/10.1145/2883851.2883876

Rezaei, R., Chiew, T. K., Lee, S. P., & Shams Aliee, Z. (2014). Interoperability evaluation models: A systematic review. *Computers in Industry*, *65*(1), 1–23. http://doi.org/10.1016/j.compind.2013.09.001

Richardson, C. (2014a). API gateway pattern. Retrieved January 26, 2017, from http://microservices.io/patterns/apigateway.html

Richardson, C. (2014b). Pattern: Microservices Architecture. Retrieved November 15, 2016, from http://microservices.io/patterns/microservices.html

Richardson, C. (2014c). The Scale Cube. Retrieved December 9, 2016, from http://microservices.io/articles/scalecube.html

Richardson, C. (2015a). Event-driven microservices. Retrieved November 27, 2016, from http://eventuate.io/whyeventdriven.html

Richardson, C. (2015b). Pattern: Client-side service discovery. Retrieved January 26, 2017, from http://microservices.io/patterns/client-side-discovery.html

Richardson, C. (2015c). Pattern: Server-side service discovery. Retrieved January 26, 2017, from http://microservices.io/patterns/server-side-discovery.html

Richardson, C. (2016a). Pattern: Database per service. Retrieved January 26, 2017, from http://microservices.io/patterns/data/database-per-service.html

Richardson, C. (2016b). Pattern: Multiple service instances per host. Retrieved January 26, 2017, from http://microservices.io/patterns/deployment/multiple-services-per-host.html

Richardson, C. (2016c). Pattern: Single Service Instance per Host. Retrieved January 26, 2017, from http://microservices.io/patterns/deployment/single-service-per-host.html

Richardson, C., & Smith, F. (2016). Microservices. From Design to Deployment, *1*, 80.

Santis, S. De, Florez, L., Nguyen, D. V, & Rosa, E. (2016). Evolve the Monolith to Microservices with Java and Node. *IBM Redbooks*, *1*, 132.

Stary, C., & Wachholder, D. (2015). System-of-systems support - A bigraph approach to interoperability and emergent behavior. *Data and Knowledge Engineering*, *00*, 1–18. http://doi.org/10.1016/j.datak.2015.12.001

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 Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube. *10th Computing Colombian Conference*, 583–590. http://doi.org/10.1109/ColumbianCC.2015.7333476

ANEXOS

Anexo A: Clasificación de Bibliografía

Se realizó la clasificación de la bibliografía resultante, de la búsqueda realizada (**Tabla 30**).

Tabla 30: Clasificación de Bibliografía.

TIPO DE DOCUMENTO								
Parcial usables	**Journal**	**Conference (Paper)**	**Web Page**	**Chapter**	**Book**		**INCLUIDOS**	**NO INCLUIDOS**
29	29	0	0	0	0		14	15
34	6	28	0	0	0		26	8
28	28	0	0	0	18		4	24
22	22	0	0	76	2		1	21
113						**Total**	**45**	**68**
	NO							
	SI							

Fuente: El autor
Elaboración: El autor

Anexo B: Selección de modelo de Interoperabilidad

Según los modelos que se identificaron en la sección **1.3,** se procedió a clasificarlos según el área a la que son orientados (**Tabla 31**).

Tabla 31: Clasificación de Modelos de Interoperabilidad.

AREA	MODELOS
General	Quantification of interoperability methodology
Defensa	Military communications and information systems interoperability
Desarrollo de software e ingeniería de sistemas	Levels of information systems interoperability
Defensa, Empresarial	Organizational interoperability maturity model for C2
General	Interoperability assessment methodology
General	Stoplight
Empresarial	Enterprise interoperability maturity model
General	The layered interoperability score
Empresarial	Government interoperability maturity matrix
General	System-of-Systems Interoperability

Fuente: El autor
Elaboración: El autor

Anexo C: Ejemplo de perfil de interoperabilidad de los sistemas

Tabla 32: Ejemplo de implementación de LISI

<table>
<tr><td colspan="3" rowspan="2">LEVEL
(Environment)</td><td colspan="4">Interoperability Attributes</td></tr>
<tr><td>Procedures</td><td>Applications</td><td>Infrastructure</td><td>Data</td></tr>
<tr><td rowspan="3">Enterprise Level (Universal)</td><td rowspan="3">4</td><td>c</td><td></td><td rowspan="3"></td><td rowspan="3"></td><td></td></tr>
<tr><td>b</td><td></td><td></td></tr>
<tr><td>a</td><td></td><td></td><td></td></tr>
<tr><td rowspan="3">Domain Level (Integrated)</td><td rowspan="3">3</td><td>c</td><td rowspan="3">Service-Approved MSN & ORD, WAN addressing Scheme</td><td></td><td rowspan="3">TCP/IP WAN, NFS, SNMP, ISDN card</td><td rowspan="2">MIDB, SQL</td></tr>
<tr><td>b</td><td></td><td></td></tr>
<tr><td>a</td><td></td><td></td></tr>
<tr><td rowspan="3">Functional Level (Distributed)</td><td rowspan="3">2</td><td>c</td><td rowspan="2">DII COE Complaint, Windows-std File name extensions</td><td>IE 4.0</td><td rowspan="2">IP LAN NES, NTP, X500</td><td rowspan="3">NIFT 2, USMTF, x-400, wks, xls, DTED, DBDB, .ppt, .doc, RPF, CGM, JBIG, JPEG, HTML, VPF</td></tr>
<tr><td>b</td><td>MS Office, Access, CMTK, SD, MPEG Viewer</td></tr>
<tr><td>a</td><td>On-line Documentation</td><td>Eudora</td><td>TIBS, LINK 16 & 22</td></tr>
<tr><td rowspan="4">Connected Level (Peer-to-Peer)</td><td rowspan="4">1</td><td>d</td><td rowspan="2">Windows Interface Design Guide (JTA)</td><td></td><td rowspan="2">HF Data, Modem, Kermit, STU III, GSM Cellular</td><td rowspan="3">MPEG1.2 GKS, .wmf</td></tr>
<tr><td>c</td><td>FTP</td></tr>
<tr><td>b</td><td>ITU-T Rec X509, Mil Std 2045-28500 Securty Labels</td><td rowspan="2">Chat 2.0, Win32 API, PPS</td></tr>
<tr><td>a</td><td>GBS</td></tr>
<tr><td rowspan="5">Isolated Level (Manual)</td><td rowspan="5">0</td><td>d</td><td colspan="2">Login Procedures</td><td rowspan="4"></td><td></td><td></td></tr>
<tr><td>c</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>b</td></tr>
<tr><td>a</td></tr>
<tr><td>0</td><td colspan="5">NO KNOWN INTEROPERABILITY</td></tr>
</table>

Fuente: (Kasunic & Anderson, 2004)
Elaboración: El autor

Anexo D: Funcionalidades Aplicación Web (PHP, Java)

- Registro de usuarios.

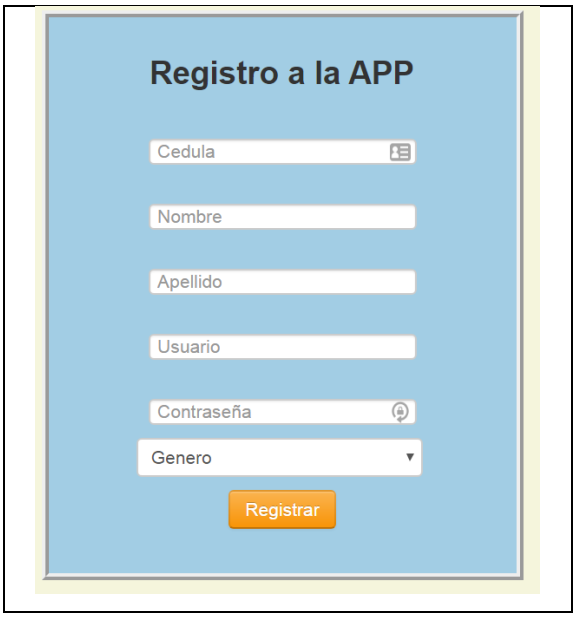


Figura 49: Plantilla de Registro
Fuente: El autor
Elaboración: El autor

- Login (Ingreso de usuarios al sistema, se maneja 2 roles (Cliente y Administrador)).

Figura 50: Plantilla de Login
Fuente: El autor
Elaboración: El autor

- CRUD especialidades Médicas (Realizado por el Administrador).

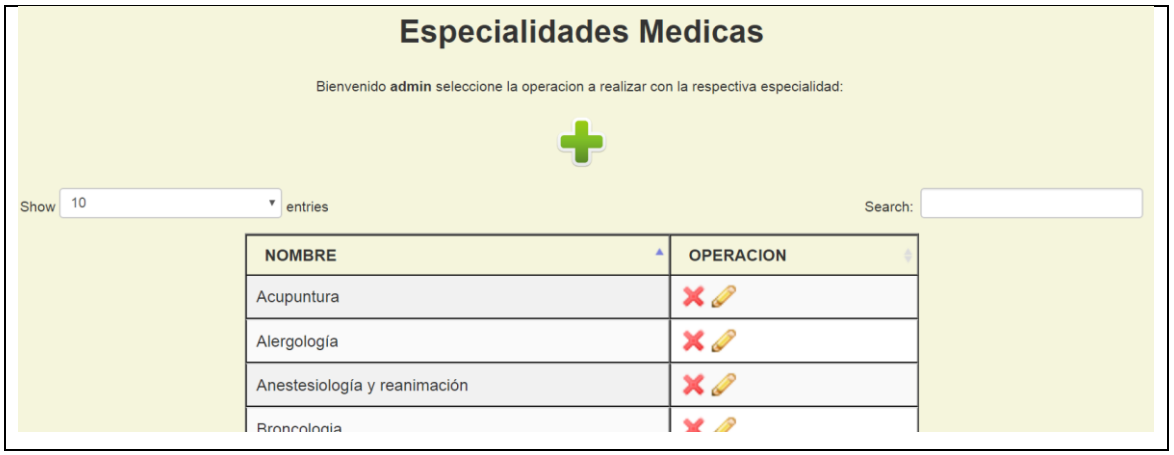


Figura 51: Página Principal de Especialidades Médicas
Fuente: El autor
Elaboración: El autor

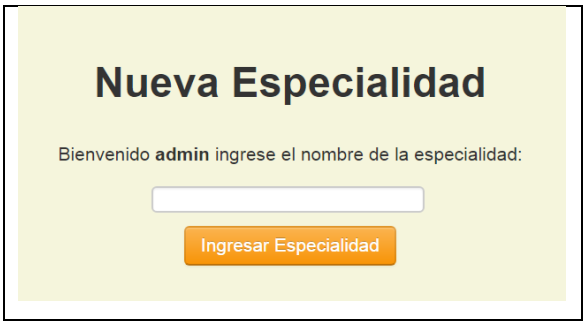


Figura 52: Pantalla para el ingreso de nueva especialidad.
Fuente: El autor
Elaboración: El autor

Figura 53: Pantalla para actualizar especialidad.
Fuente: El autor
Elaboración: El autor

- CRUD de Doctores (Realizado por el Administrador).

Figura 54: Página Principal de Doctores
Fuente: El autor
Elaboración: El autor

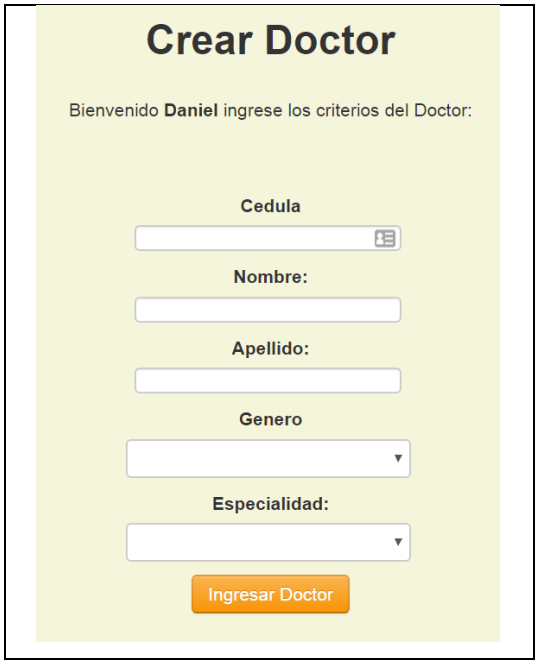


Figura 55: Pantalla para ingresar un nuevo Doctor.
Fuente: El autor
Elaboración: El autor

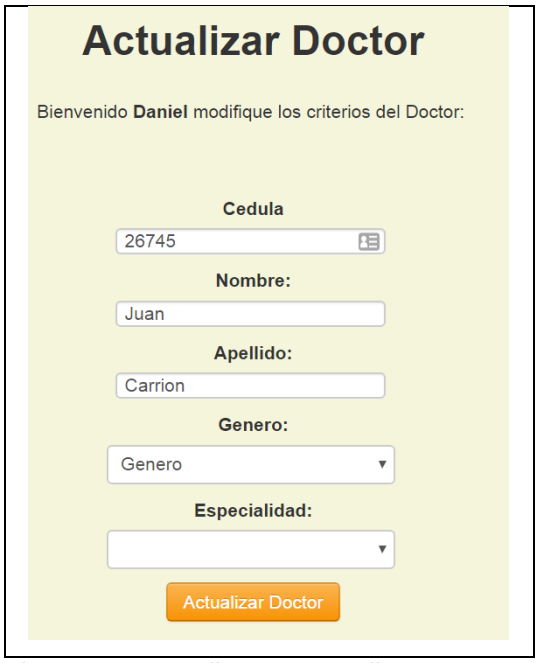


Figura 56: Pantalla para actualizar un Doctor.
Fuente: El autor
Elaboración: El autor

- Vinculación de Especialidades a Doctores. (Realizado por el Administrador)

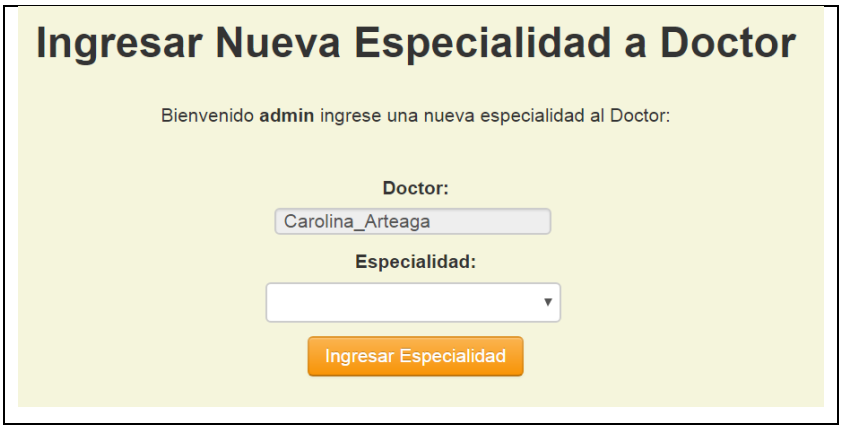


Figura 57: Pantalla para asignar una especialidad a un doctor.
Fuente: El autor
Elaboración: El autor

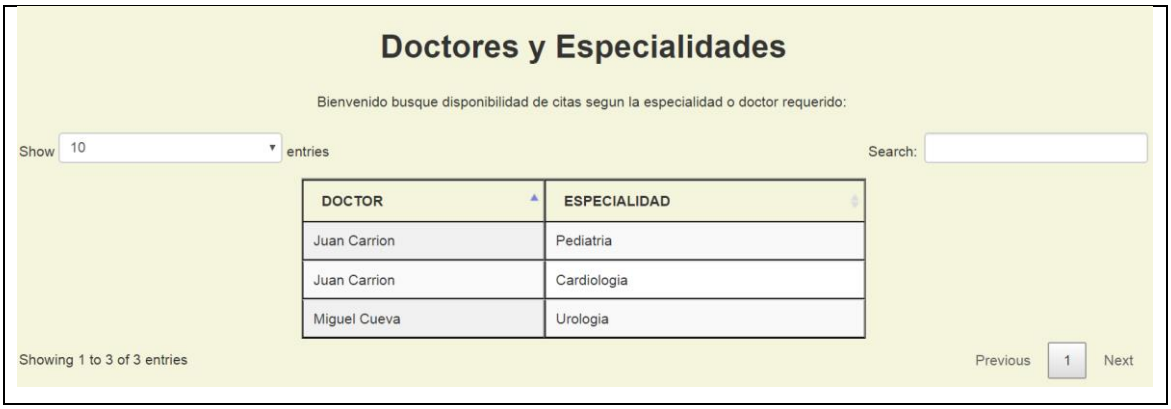


Figura 58: Página Principal de Doctores con Especialidades.
Fuente: El autor
Elaboración: El autor

- CRUD de Citas Médicas (Realizado por el Cliente).

Figura 59: Pantalla para ingresar una nueva cita.
Fuente: El autor
Elaboración: El autor

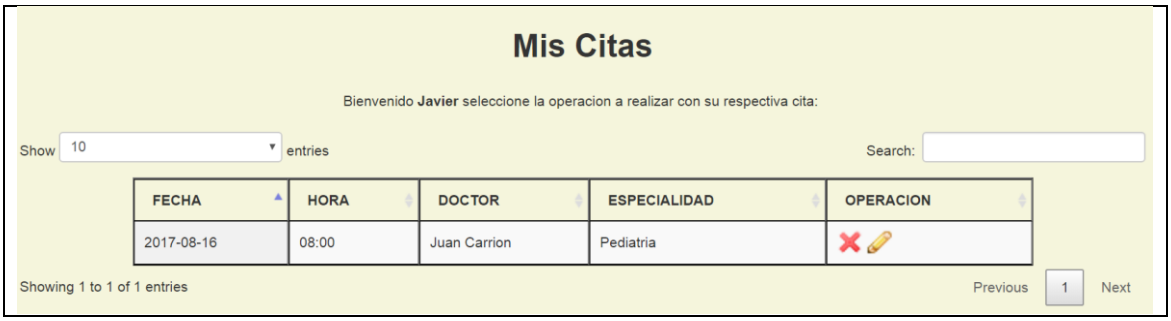


Figura 60: Página Principal de Citas Médicas.
Fuente: El autor
Elaboración: El autor

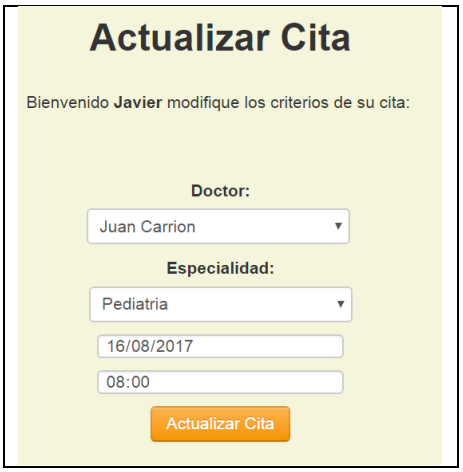


Figura 61: Pantalla para actualizar cita.
Fuente: El autor
Elaboración: El autor

Anexo E: Diagrama Entidad-Relación “Citas Médicas”

Se posee 6 tablas:

- **Usuario:** Guarda los datos del usuario ingresado. Tiene una relación de *muchos* (usuario) a *uno* (rol).
- **Rol:** Al momento de crear un usuario se almacena su respectivo rol (pudiendo ser admin, o paciente).
- **Cita:** Guarda los datos referentes a las citas del paciente. Tiene una relación de *uno* (usuario) a *muchos* (cita); *muchos* (cita) a *uno* (especialidad); y *muchos* (cita) a *uno* (doctor).
- **Doctor:** Guarda información referente al doctor. Tiene una relación de uno (doctor) a muchos (doctor_especialidad).
- **Especialidad:** Guarda información de las especialidades médicas. Tiene una relación de uno (especialidad) a muchos (doctor_especialidad).
- **Doctor_especialidad:** Tabla de unión que permite la relación entre doctor y especialidad.

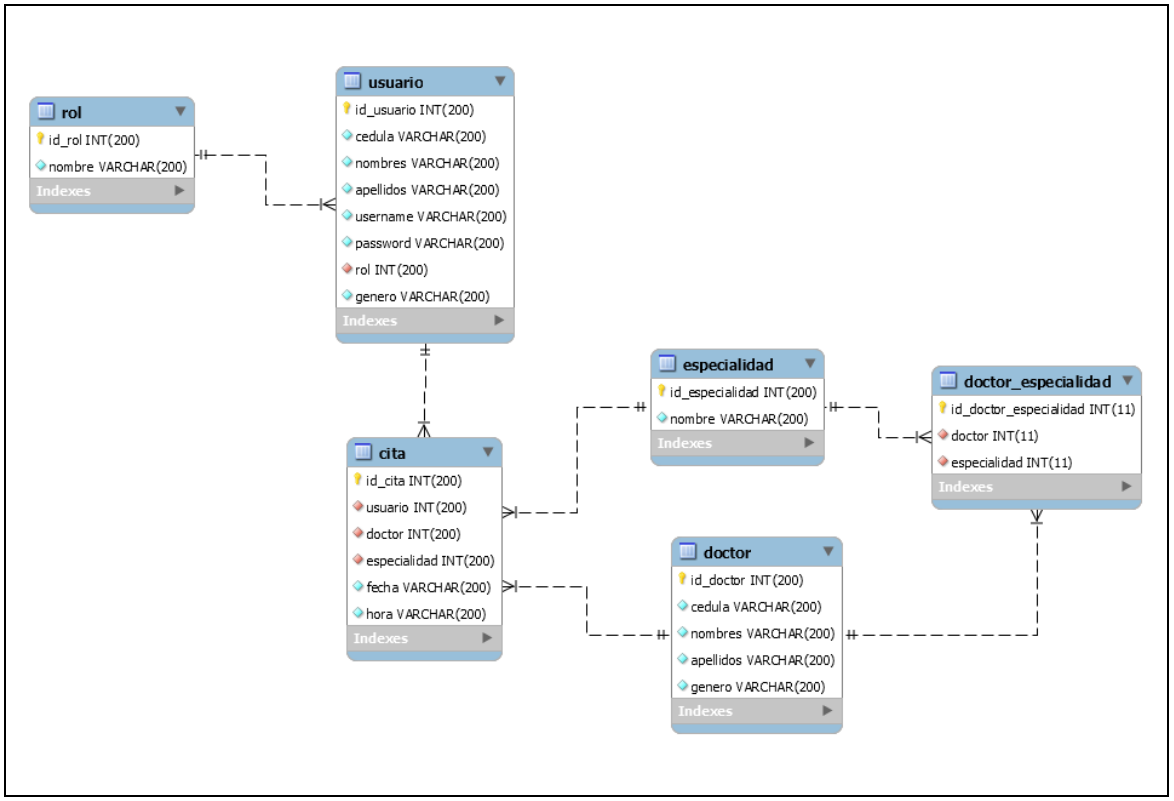


Figura 62: Diagrama Entidad-Relación “Citas Médicas”
Fuente: El autor
Elaboración: El autor

Anexo F: Instalación de Node.js en Windows 8.1

- Se descarga el instalador (**LTS**) de la página oficial *(https://nodejs.org/en/)*

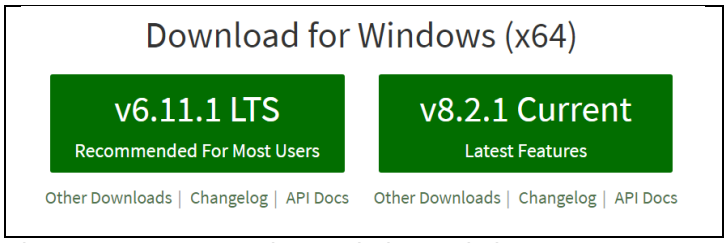


Figura 63: Descarga de Instalador Node.js
Fuente: Node (2017)
Elaboración: El autor

- Se ejecuta el instalador (.msi)
- Se sigue los pasos del Wizard (**Figura 64**), y la instalación de paquetes por defecto (incluido npm).

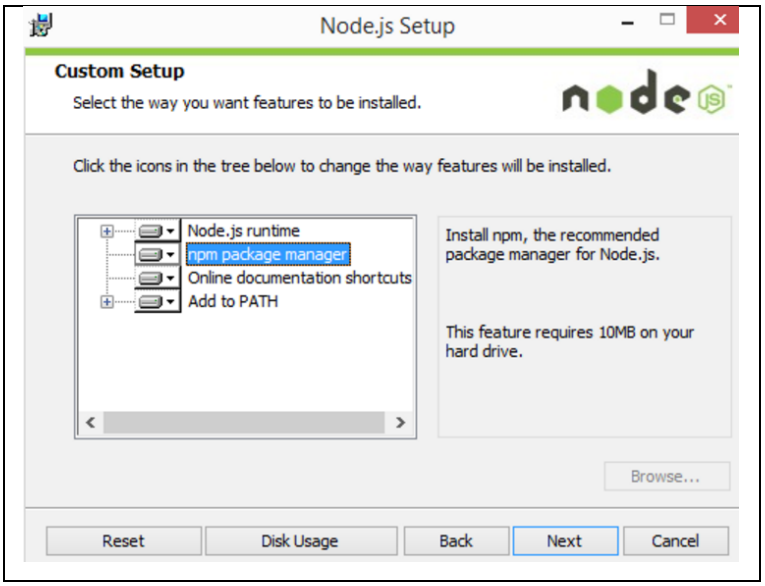


Figura 64: Descarga de Instalador Node.js
Fuente: Node (2017)
Elaboración: El autor

- Se reinicia la computadora.
- Para comprobar la instalación correcta, se inicia la consola de Windows (cmd), y se ejecutan los comandos: *"node -v"* y *"npm version"*. (**Figura 65**)

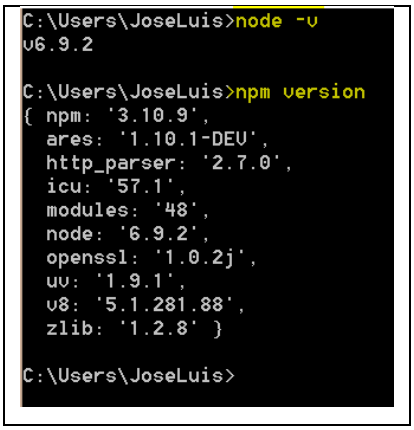


Figura 65: Comandos verificación Node
Fuente: El autor
Elaboración: El autor

- Para ejecutar nuestro primer hola mundo creamos un archivo .js (en este caso *"hola.js"*), en codificación Node y lo ejecutamos con el comando *"node hola.js":*

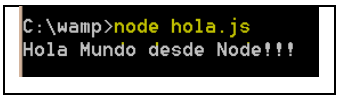


Figura 66: Hola mundo Node.js
Fuente: El autor
Elaboración: El autor

- Después de realizar la instalación de Node procedemos a instalar un framework de Node .js, en este caso denominado Express, el cual proporciona un conjunto solido de características para aplicaciones Web; para esto utilizados el siguiente comando en la consola de Windows *"npm install -g express-generator"*.

Figura 67: Instalación framework Express
Fuente: El autor
Elaboración: El autor

Anexo G: Desarrollo de Aplicación Node.js

- Se procede a ubicarse en el directorio en donde se crearan los proyectos Node, y se crean los proyectos de uno en uno con el comando "*express nombreproyecto",* el cual nos genera una carpeta con los archivos básicos para una aplicación web (**Figura 68**); finalmente ingresamos a la carpeta creada.

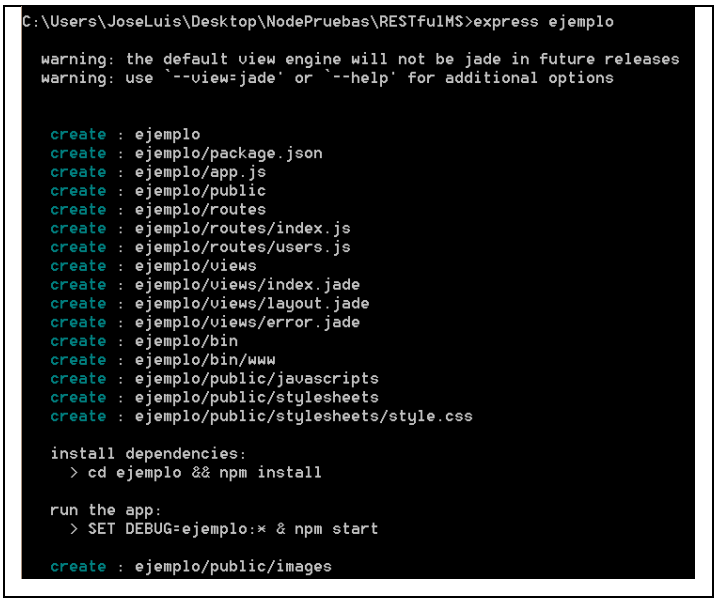


Figura 68: Creación de proyecto Express.
Fuente: El autor
Elaboración: El autor

- Se ejecuta el comando *"npm install"*, para instalar todas las dependencias establecidas en el archivo *"package.json" (***Figura 69***)*.

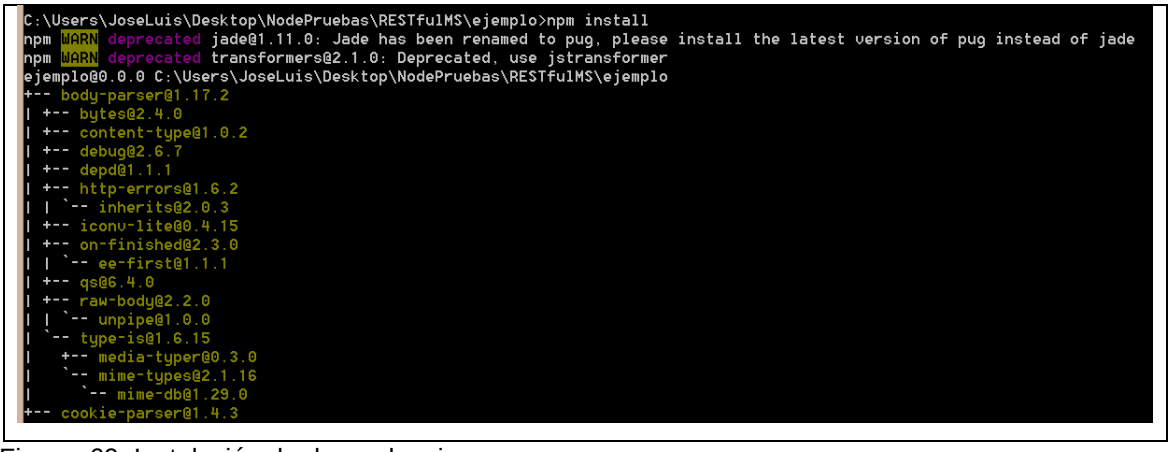


Figura 69: Instalación de dependencias.
Fuente: El autor
Elaboración: El autor

- Luego procedemos a modificar el archivo *app.js,* en el cual se agregó la conexión con MySQL. (ver **Figura 37**).

- Posteriormente se procedió a agregar las dependencias adicionales que son necesarias para la aplicación, esto se realiza en el archivo *"package.json"*, tales dependencias (o librerías) son: *"mysql"* y *"express-myconnection"* para la conexión con la base de datos y *"easyxml"* útil para realizar la conversión de los datos al formato XML (**Figura 70**).

```
{
  "name": "cita_doctor",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.17.1",
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.3",
    "express": "~4.15.2",
    "jade": "~1.11.0",
    "morgan": "~1.8.1",
    "serve-favicon": "~2.4.2",
    "mysql": "",
    "express-myconnection": "",
    "easyxml": ""
  }
}
```

Figura 70: Agregación de dependencias adicionales.
Fuente: El autor
Elaboración: El autor

- Se procede a instalar las nuevas dependencias agregadas ejecutando el comando *"npm install".*
- Finalmente se realiza la codificación del servicio RESTful, de los archivos *app.js*, *formato_datos.js, cita.js (o doctor.js), y* www (como se observa en el apartado **4.1.4.3.2**).

Anexo H: Ejecución de Aplicación Node.js

- Después de realizar la codificación de la aplicación RESTful en Node procedemos a realizar la respectiva ejecución, para esto nos ubicamos en el directorio donde se encuentra la aplicación (atraves de la consola de Windows), y ejecutamos el comando *"npm start".*
- A continuación abrimos el navegador e ingresamos a la dirección asignada al servicio RESTful con su respectivo puerto y parámetros **(4.1.4.2)** (en este ejemplo es *http://127.0.0.1:1600/ms/cita/777777/doctor*),
- Y nos retorna por defecto en el formato Json (al ser la primera opción), como se observa en la **Figura 71**.

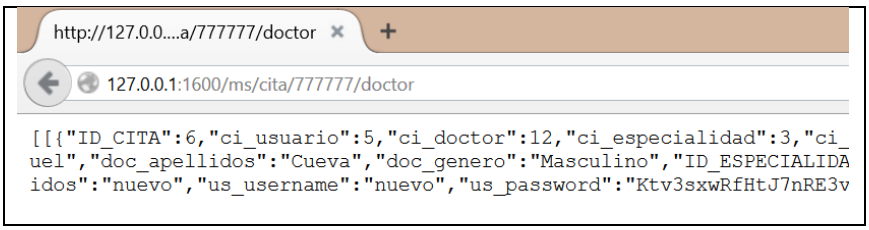


Figura 71: Verificación de RESTful en el navegador.
Fuente: El autor
Elaboración: El autor

- Así mismo en la consola nos muestra el resultado de la consulta realizada al servicio RESTful (**Figura 72**).

```
C:\Users\JoseLuis\Desktop\NodePruebas\RESTfulMS\cita_doctor>npm start

> serviciorestful@0.0.0 start C:\Users\JoseLuis\Desktop\NodePruebas\RESTfulMS\cita_doctor
> node ./bin/www

express deprecated req.param(name): Use req.params, req.body, or req.query instead routes\cita.js:22:26
777777
[ RowDataPacket {
    ID_CITA: 6,
    ci_usuario: 5,
    ci_doctor: 12,
    ci_especialidad: 3,
    ci_fecha: '2017-06-25',
    ci_hora: '20:20',
    ID_DOCTOR: 12,
    doc_cedula: '777777',
    doc_nombres: 'Miguel',
    doc_apellidos: 'Cueva',
    doc_genero: 'Masculino',
    ID_ESPECIALIDAD: 3,
    es_nombre: 'Urologia',
    ID_USUARIO: 5,
    us_cedula: 'nuevo',
    us_nombres: 'nuevo',
    us_apellidos: 'nuevo',
    us_username: 'nuevo',
    us_password: 'Ktv3sxwRfHtJ7nRE3vqoDQ==',
    us_rol: 2,
    us_genero: 'Masculino',
    ID_ROL: 2,
    rol_nombre: 'paciente' } ]
GET /ms/cita/777777/doctor 200 1195.058 ms - 473
```

Figura 72: Resultado de consulta a RESTful (consola Windows).
Fuente: El autor
Elaboración: El autor

- Cabe mencionar que al ser *cada servicio RESTful una aplicación* se debe realizar todo el proceso aquí mencionado para su ejecución para *cada aplicación.*

Anexo I: Arquitectura de Docker

La arquitectura de Docker es una arquitectura Cliente/Servidor, la misma que posee varios componentes/partes, en la cual se tiene el repositorio web de Docker en donde se encuentran las imágenes **(Figura 73 - 1)**, de las cuales se descarga en nuestro host Docker para poder usarlas las que sean necesarias**(Figura 73 - 2)**, a partir de las mismas se crean los respectivos contenedores **(Figura 73 - 3)**, para que finalmente un cliente las pueda utilizar **(Figura 73 - 4)**, todo esto controlado por el Daemon de Docker, el cual también tiene comunicación con la base de datos, siendo esta externa a Docker.

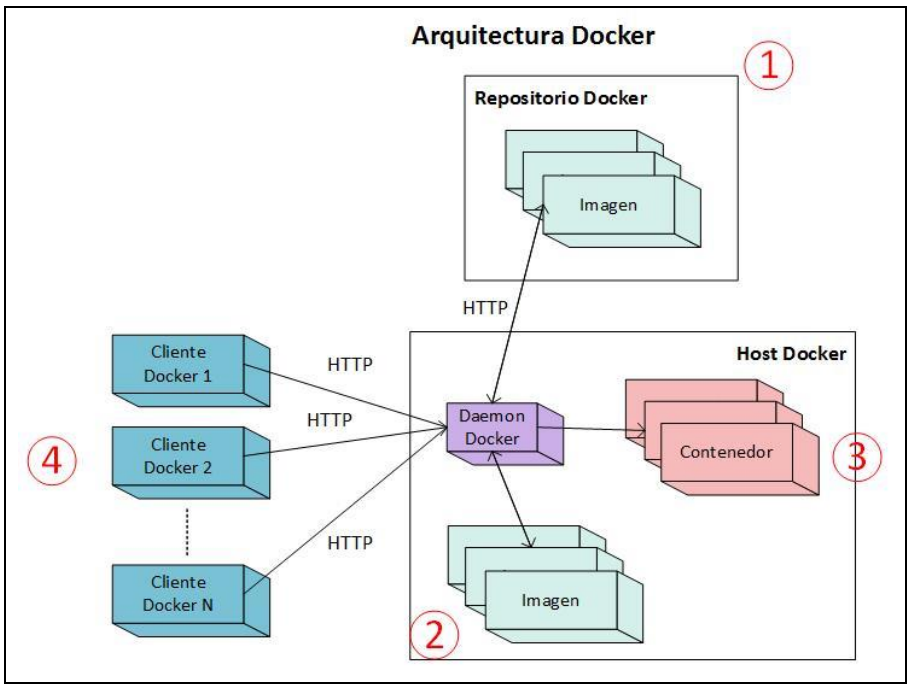


Figura 73: Arquitectura Base de Docker
Fuente: Alexandre Eleutério Santos Lourenço (2017)
Elaboración: El autor

Anexo J: Instalación y Configuración de Docker en Windows 8.1

- Se procede a descargar *"Docker Toolbox"* (*https://www.docker.com/products/docker-toolbox*), la cual nos permite utilizar Docker en sistemas Windows antiguos que no cumplen con los requisitos mínimos del sistema (el requisito que no se cumple es que el Sistema Operativo tiene que ser Windows 10) para la aplicación *Docker* para Windows (**Figura 74**).

Figura 74: Pantalla de descarga de Docker
Fuente: Docker (2017)
Elaboración: El autor

- Se verifica que la opción de *Virtualización* este habilitada. (Para esto iniciamos el *administrador de tareas* y nos colocamos en la pestaña de *Performance* **Figura 75**).

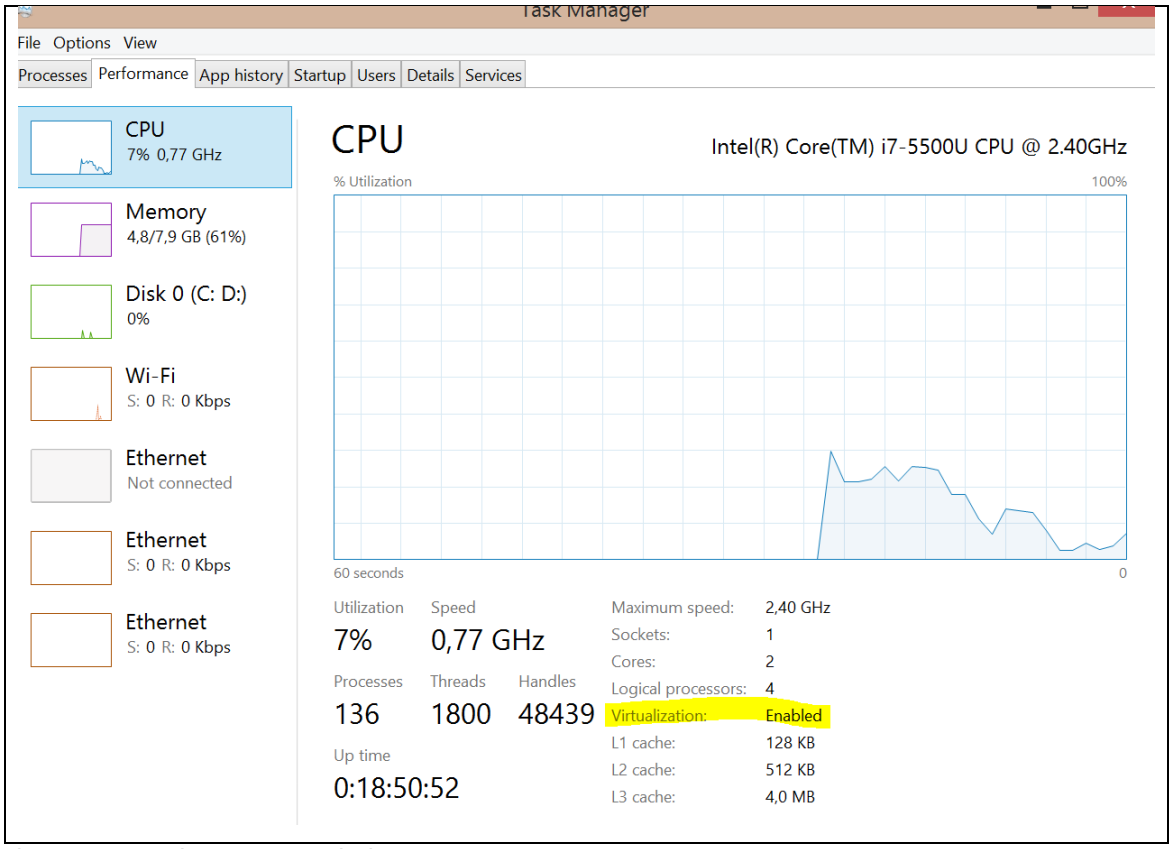


Figura 75: Task Manager Windows
Fuente: El autor
Elaboración: El autor

- Luego se procede a ejecutar el instalador (.exe), previamente descargado.
- Se seleccionan las configuraciones y paquetes por defecto.

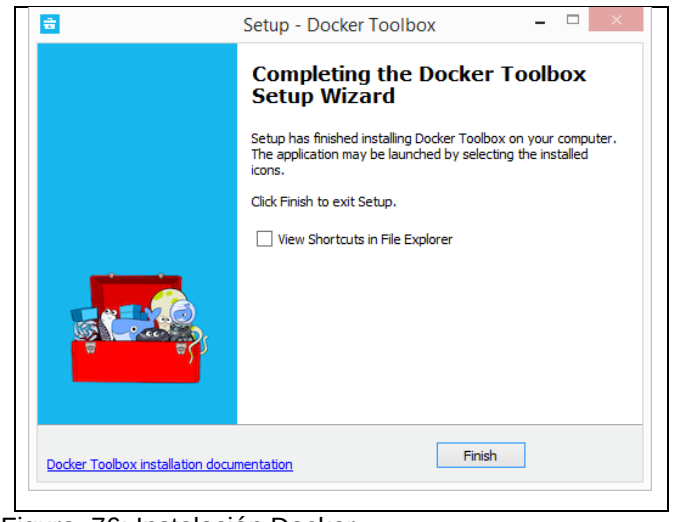


Figura 76: Instalación Docker.
Fuente: Docker (2017)
Elaboración: El autor

- Luego de concluida la instalación en el escritorio se crean 3 iconos, como resultado de la misma: *VirtualBox* (para gestión de las máquinas virtuales), *Docker Quickstart Terminal* (consola para la ejecución de comandos de Docker) y Kitematic (alternativa a la consola, la cual proporciona una interfaz gráfica).

Figura 77: Iconos resultado de instalación de Docker.
Fuente: El autor
Elaboración: El autor

- Se procede a ejecutar el archivo *"Docker Quickstart Terminal"*, al ser la primera vez que se ejecuta realiza algunas configuraciones (por lo que requiere un poco de tiempo para culminar, en este caso ya no es la primera vez que se lo ejecuta por lo que los mensajes informativos son cortos). Al momento de terminar estas configuraciones nos aparece el signo **$** en el terminal para poder ejecutar nuestros comandos (**Figura 78**).

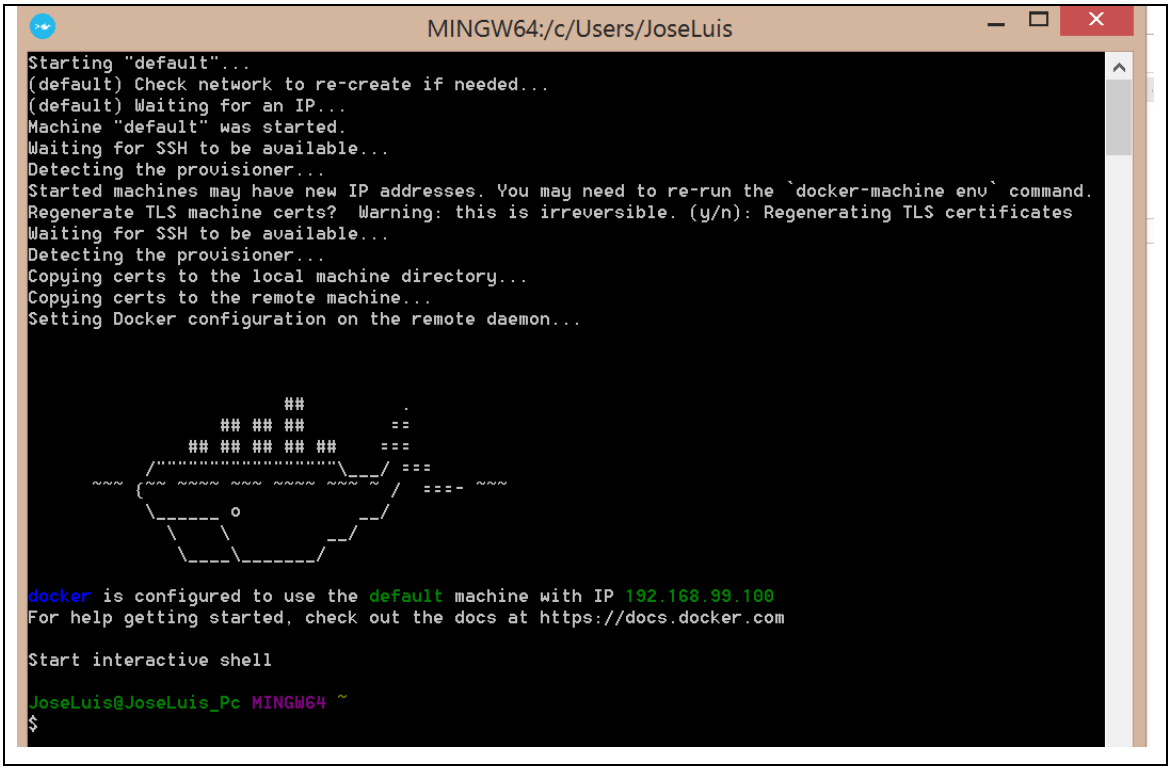


Figura 78: Docker Quickstart Terminal de Docker.
Fuente: El autor

Elaboración: El autor

- Para comprobar el correcto funcionamiento se verifica la versión instalada con el comando *docker version.*(**Figura 79**)

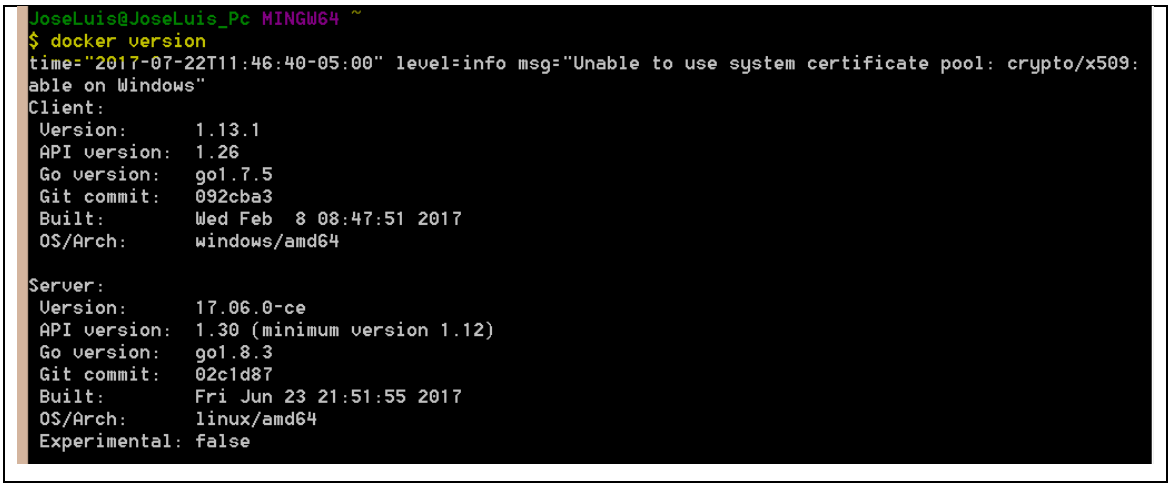


Figura 79: Versión de Docker
Fuente: El autor
Elaboración: El autor

- Se puede comprobar también la IP asignada con el comando *(docker-machine ls).*

Figura 80: IP asignada a Docker
Fuente: El autor
Elaboración: El autor

- Para ejecutar nuestro primer “Hola mundo” desde Docker (**Figura 81**), ingresamos el comando *“docker run hello-world”;* el cual nos descarga la imagen desde el repositorio de Docker y la ejecuta (en este caso la imagen ya estaba descargada anteriormente, por lo que solo procede a ejecutarla).

```
JoseLuis@JoseLuis_Pc MINGW64 ~
$ docker run hello-world
time="2017-07-22T11:49:45-05:00" level=info msg="Unable to use system certificate pool: crypto/x509:
able on Windows"

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Figura 81: Hello World en Docker
Fuente: El autor
Elaboración: El autor

- Una alternativa a la consola es la utilización de *Kinematic* (como se mencionó anteriormente), para esto abrimos esta herramienta, y seleccionamos el contenedor que queremos ejecutar y hacemos click en *"START"* y nos muestra el resultado de la ejecución del contenedor (en *"CONTAINER LOGS", ver* **Figura 82**).

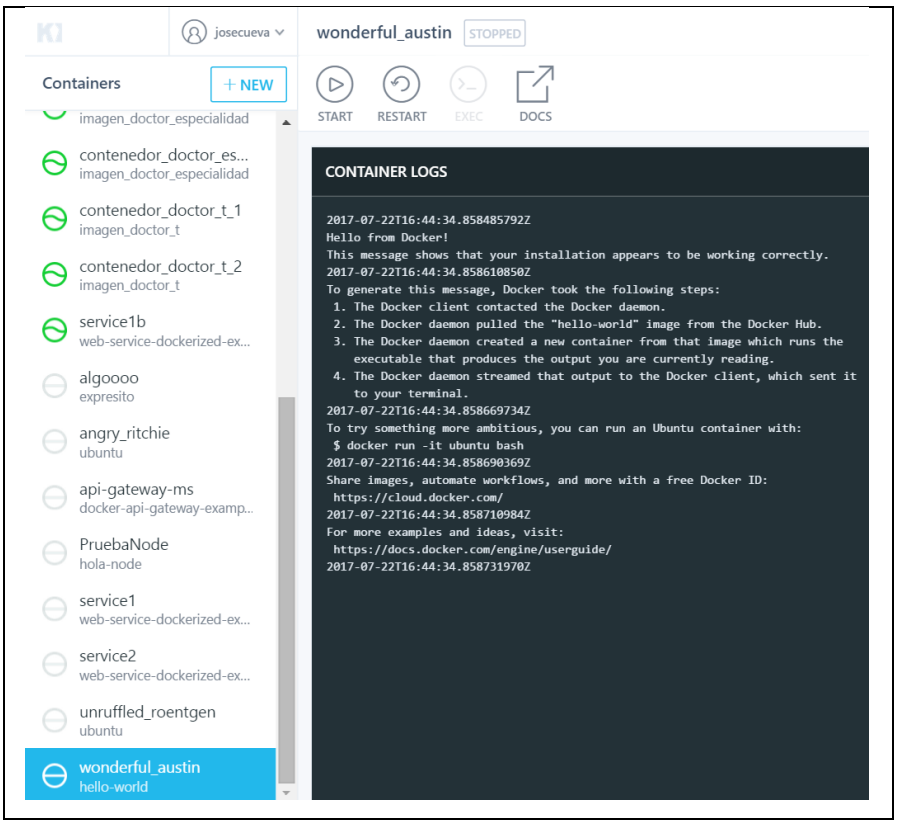


Figura 82: Contanier Logs en *Kinematic*.
Fuente: El autor
Elaboración: El autor

Anexo K: Desarrollo de Aplicación en Docker

- Con las aplicaciones realizadas en Node.js, lo que se procede es a *"Dockerizarlas",* esto es generar (*build*) una imagen Docker que contenga y ejecute una aplicación Node y a partir de las cuales se crean (*Run*) los respectivos contenedores, como se observa en la **Figura 83** (este proceso se realiza por cada aplicación RESTful, generando por lo tanto 6 imágenes, y 12 contenedores; 2 por cada imagen), como se mencionan en el apartado **4.1.5.3.2,** lo que se realiza es la creación un archivo de configuración denominado *Dockerfile* (y **Figura 84**), en el cual (se mantiene la estructura para los demás Dockerfiles, modificándose solo los parámetros *EXPOSE* y *ENV PORT*) se realiza lo siguiente (cabe mencionar que este archivo no posee una extensión):
  - *Línea 1:* indica la imagen a descargar desde el repositorio que posee Docker en la web, en este caso descarga una imagen de *node*, que posee la etiqueta *boron*, esta imagen contiene todo lo necesario para ejecutar aplicaciones Node.

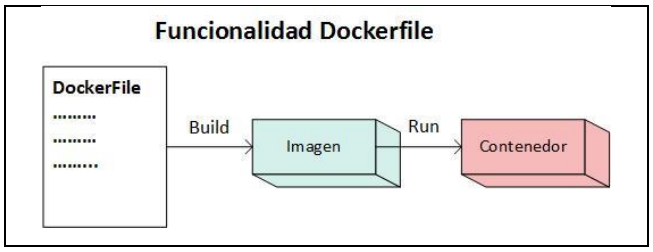


Figura 83: Funcionalidad del Dockerfile
Fuente: El autor
Elaboración: El autor

Dockerfile

```
1  FROM node:boron
2
3  # Create app directory
4  RUN mkdir -p /usr/src/app
5  WORKDIR /usr/src/app
6
7  # Install app dependencies
8  COPY package.json /usr/src/app/
9  RUN npm install
10
11 # Bundle app source
12 COPY . /usr/src/app
13
14 EXPOSE 1600
15 ENV PORT 1600
16
17 CMD [ "npm", "start" ]
```

Figura 84: Contenido Dockerfile.
Fuente: El autor
Elaboración: El autor

- *Línea 4:* crea el directorio especificado dentro de la imagen.
- *Línea 5:* establece al directorio, como directorio raíz para el código de la aplicación.
- *Línea 8:* copia el archivo package.json, en el directorio creado (dentro de la imagen).
- *Línea 9:* Instalada las dependencias que indica el archivo package.json, en la imagen.
- *Línea 12:* copia todos los archivos de la aplicación RESTful al directorio raíz (de la imagen).
- *Línea 14 y 15:* Indican el puerto por él se puede acceder a la aplicación.
- *Línea 17:* indica el comando Node.js mediante el cual se ejecuta la aplicación.

– A continuación de realizar la configuración del *Dockerfile*, se configura el acceso a la base de datos de la aplicación, para esto se modifica el archivo app.js (como se observa en el apartado **4.1.5.3.2**).

– Luego se procede a crear las imágenes por cada RESTful (este paso se realiza por cada RESTful, asignándole un nombre diferente a cada una); para esto primeramente nos colocamos en el directorio en donde se encuentra el Dockerfile (configurado anteriormente) y ejecutamos el comando (*docker build -t nombre_imagen -f ./Dockerfile .)* y procede a la generación de la imagen respectiva (ejecutando cada línea que se codifico en el Dockerfile una a continuación de otra).

– Atraves del comando *docker images,* se puede observar las imágenes creadas (ver **Figura 85**)

Figura 85: Imágenes en Docker.
Fuente: El autor
Elaboración: El autor

- Posteriormente se procede a crear los contenedores necesarios (se ejecuta el comando las veces que se requiera un nuevo contenedor):

docker run -d --name <nombre de contenedor> -e SERVICE_NAME==<nombre de servicio>-l=api_route='<ruta de servicio>' --expose <numero de puerto> -v <Direccion de aplicacion>:/usr/src/app <nombre de imagen>

- En el cual se indica el nombre que tendrá el contenedor, el nombre del servicio y la ruta (utilizada posteriormente por el API Gateway) por la cual se accede al mismo; además del puerto por el que está activo, así mismo el directorio en donde se encuentra la aplicación en Windows y el directorio de ejecución en Docker (esto se hace con el motivo de que los cambios en la aplicación sean más rápidos al momento de verificarlos), y finalmente la imagen a utilizar.
- Posterioremente se procede primero a ejecutar el comando:

docker run -d --name api-gateway-ms -v /var/run/docker.sock:/var/run/docker.sock -p 80:8080 beh01der/docker-api-gateway-example

- *El* cual se encarga de descargar una imagen del API Gateway (*beh01der/docker-api-gateway-example*) que se encuentra en el repositorio web de Docker, y crea un contenedor al cual se le asigna el nombre de *api-gateway-ms;* que se va a ejecutar en el puerto 80 de Docker y externamente se va a acceder atraves del puerto 8080.
- Se realizó por lo tanto, la creación de 7 imágenes (incluido el API Gateway) y 13 contenedores (2 por cada imagen RESTful y 1 del API Gateway).

REINICIO DE MS AUTOMÁTICO

- Esta funcionalidad que Docker implementa, la denomina restart policies, se la aplica a un contenedor utilizando uno de los siguientes comandos:

Para crear un nuevo contenedor con la configuración mencionada:

docker run -d --name <nombre de contenedor> -e SERVICE_NAME=<nombre de servicio> -l=api_route='<ruta de servicio>' --expose <numero de puerto> -v <Direccion de aplicacion>:/usr/src/app ***--restart=always*** *<nombre de imagen>*

O para modificar la configuración de un contenedor ya creado (esta configuración se aplicó a todos los contenedores):

docker update --restart=always < nombre de contenedor >

- Existen varias banderas, en nuestro caso utilizaremos *"always"*, la cual se encarga de reiniciar el contenedor si este se detiene (por algún fallo), o si se reinicia Docker (por lo que no se requiere de inician cada contenedor, por el motivo de que se inician conjuntamente con Docker). Se debe tener en cuenta que las *restart policies* se ignoran cuando se detiene manualmente el contenedor, y solo se aplican a contenedores.
- Para verificar que se está aplicando, se ejecuta el comando "*docker inspect <nombre de contenedor>"* en el cual se observa la configuración del contenedor en el apartado de *"HostConfig"* y en el subapartado "*RestartPolicy":* (en la **Figura 86** se observa la configuración antes de aplicar el comando, mientas que la **Figura 87** es el resultado de aplicarlo)

Figura 86: Configuración Restart Policy (Antes)
Fuente: El autor
Elaboración: El autor

Figura 87: Configuración Restart Policy (Después)
Fuente: El autor
Elaboración: El autor

- También se puede verificar ejecutando el comando "*docker ps –a"* y observando el estado del contenedor, en este caso se reinicia porque no se ha iniciado correctamente:

Figura 88: Reinicio de contenedor
Fuente: El autor
Elaboración: El autor

- Finalmente se procedió a crear los respectivos privilegios a la Base de datos existente en WAMP para que pueda ser accedida por los MS (ver **Figura 89**)

Figura 89: Privilegios a Base de datos (WAMP)
Fuente: El autor
Elaboración: El autor

Anexo L: Ejecución de Aplicación en Docker

- Se debe tener en cuenta que el orden de ejecución del API Gateway y los MS no influye en el funcionamiento; siempre y cuando ambos estén en ejecución al momento de acceder a un MS específico.
- Para visualizar el estado de los contenedores se ejecuta el comando *docker ps -a ;* en el cual se detalla información como el *ID del contenedor*, la *imagen* a la que pertenece, el *estado* del contenedor (en es esta caso están todos *UP* es decir en ejecución), los *puertos* en los que se está ejecutando, el nombre del contenedor, entre otra información (**Figura 90**), esta misma visualización se la puede realizar también en la herramienta Kinematic (**Figura 90**, **Figura 91**, **Figura 92**, **Figura 93**, **Figura 94**)

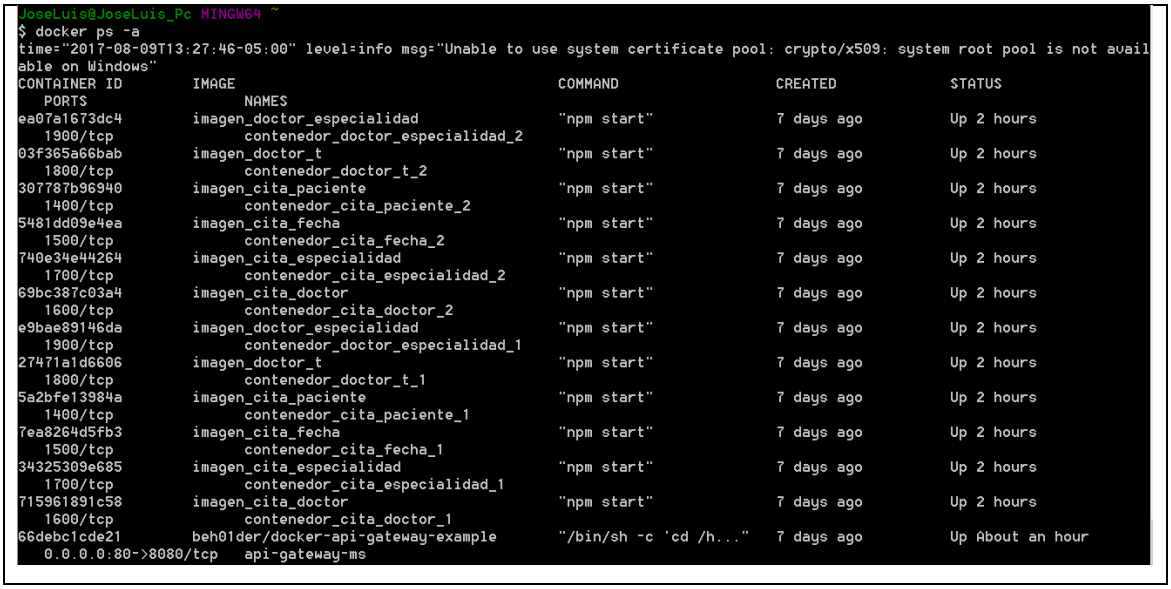


Figura 90: Información de los contenedores Docker (Consola).
Fuente: El autor
Elaboración: El autor

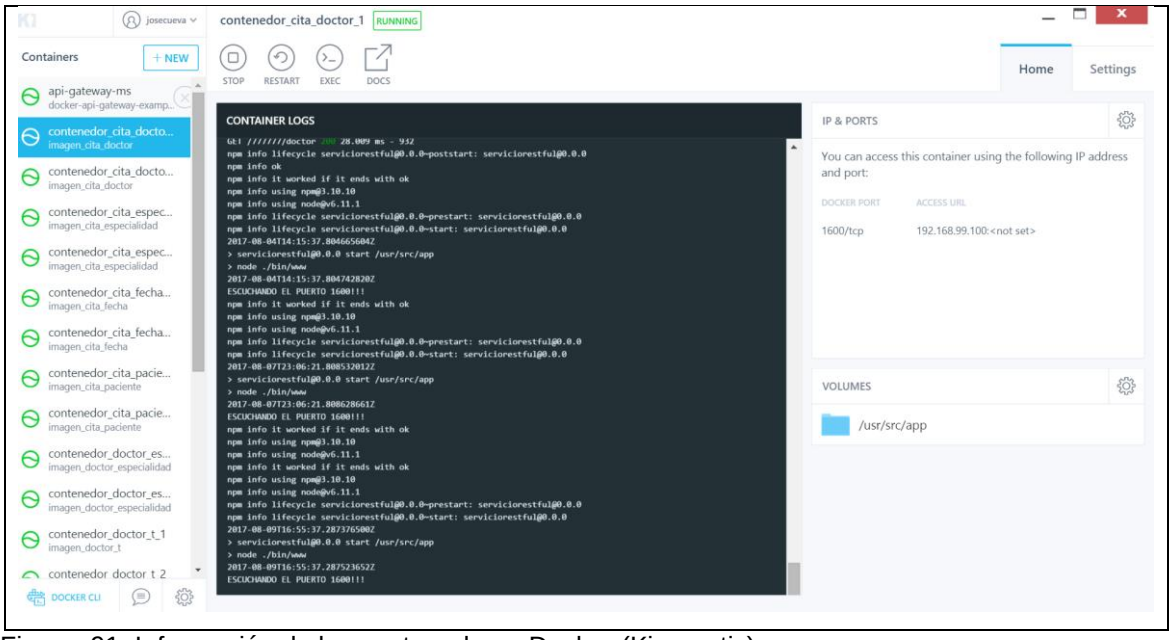


Figura 91: Información de los contenedores Docker (Kinematic).
Fuente: El autor
Elaboración: El autor

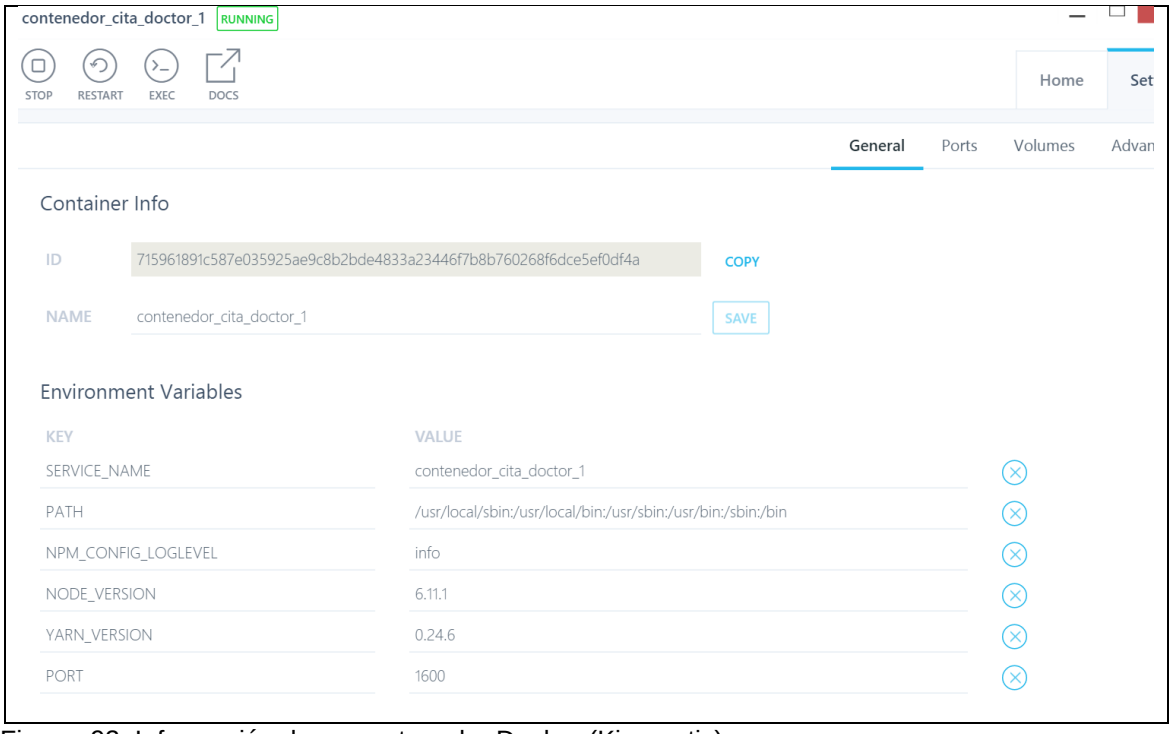


Figura 92: Información de un contenedor Docker (Kinematic).
Fuente: El autor
Elaboración: El autor

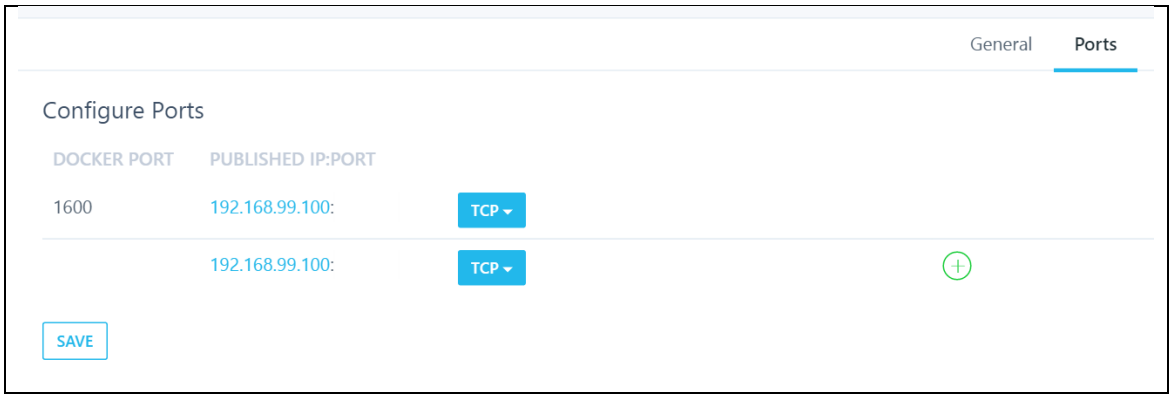


Figura 93: Información de los puertos de un contenedor Docker (Kinematic).
Fuente: El autor
Elaboración: El autor

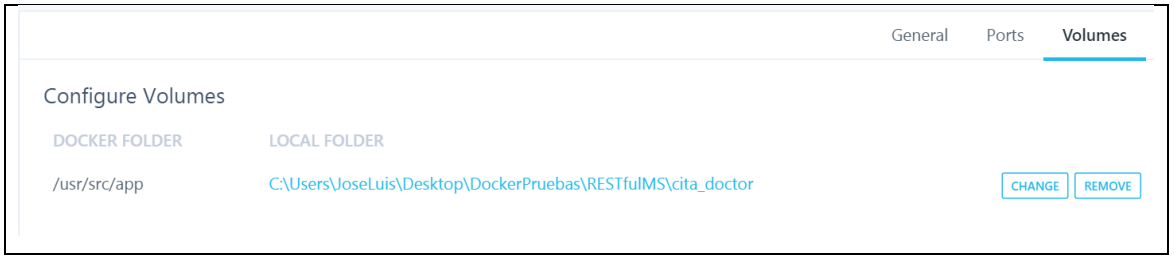


Figura 94: Información de los directorios de un contenedor Docker (Kinematic).
Fuente: El autor
Elaboración: El autor

- Cada contenedor tiene una ip asignada en Docker, se la visualiza ejecutando el comando *docker inspect <nombre_contenedor>*; en el apartado *bridge* de *Networks*, la misma que es usada por el API Gateway para realizar las respectivas redirecciones (**Figura 95**).

Figura 95: Configuración de red Contenedor Docker.
Fuente: El autor
Elaboración: El autor

- Finalmente antes de realizar el consumo de los MS se debe bajar el firewall (tanto de Windows como del Antivirus, para poder acceder a la base de datos).

Anexo M: API Gateway

- Para este componente se tomó como base una implementación simple en Node.js del patrón API Gateway desarrollado por Chausenko (2016).
- Se ejecutó el siguiente comando:

docker run -d --name api-gateway-ms -v /var/run/docker.sock:/var/run/docker.sock -p 80:8080 beh01der/docker-api-gateway-example

- El cual se encarga de descargar la imagen desde el repositorio de Docker en la web y crea un contenedor denominado *api-gateway-ms* para nuestro uso, además de especificar los puertos de ejecución, y los puertos de UNIX para hacer uso del contenedor.
- Así mismo se le aplico la *restart policies* al contenedor para el reinicio automático: *docker update --restart=always api-gateway-ms.*
- La aplicación usa los módulos/librerías de npm: *node-docker-monitor* and *http-proxy.* (**Figura 96**)

Figura 96: Librerías del archivo service.js.
Fuente: El autor
Elaboración: El autor

- El API Gateway reacciona a los eventos de Docker cuando un contenedor se *levanta (onContainerUp)* o se detiene *(onContainerDown)* como se observa en la **Figura 97**, creando o removiendo las *routing rules* para los mismos; como se mencionó anteriormente, para indicar que un contenedor será manejado por el Gateway se le debe colocar la etiqueta **api_route**, permitiendo obtener toda la información de dicho contenedor y apartir de esta definir la URL del mismo y agregar una nueva *route* a la colección del API Gateway; cuando un contenedor se da de baja se remueve automáticamente la *route* correspondiente a dicho contendor.

```
monitor({
    onContainerUp: function (containerInfo, docker) {
        if (containerInfo.Labels && containerInfo.Labels.api_route) {
            // register a new route if container has "api_route" label defined
            var container = docker.getContainer(containerInfo.Id);
            // get running container details
            container.inspect(function (err, containerDetails) {
                if (err) {
                    console.log('Error getting container details for: %j', containerInfo, err);
                } else {
                    try {
                        // prepare and register a new route
                        var route = {
                            apiRoute: containerInfo.Labels.api_route,
                            upstreamUrl: getUpstreamUrl(containerDetails)
                        };

                        routes[containerInfo.Id] = route;
                        console.log('Registered new api route: %j', route);
                    } catch (e) {
                        console.log('Error creating new api route for: %j', containerDetails, e);
                    }
                }
            });
        }
    },

    onContainerDown: function (container) {
        if (container.Labels && container.Labels.api_route) {
            // remove existing route when container goes down
            var route = routes[container.Id];
            if (route) {
                delete routes[container.Id];
                console.log('Removed api route: %j', route);
            }
        }
    }
}, dockerOpts);
```

Figura 97: Porción de código del archivo service.js (1)
Fuente: El autor
Elaboración: El autor

- Además se implementa una funcionalidad de proxy, la cual cuando una solicitud de HTTP es recibida, se encarga de encontrar la ruta apropiada, en caso de que exista, se actualiza la URL y se la redirecciona al contenedor destino. Si la ruta no existe, retorna el código *502 "Bad gateway"* al cliente (**Figura 98**)

```
77  console.log('API gateway is listening on port: %d', httpPort);
78  server.listen(httpPort);
79
80  // create proxy
81  var proxy = httpProxy.createProxyServer();
82  proxy.on('error', function (err, req, res) {
83      returnError(req, res);
84  });
85
86  // proxy HTTP request / response to / from destination upstream service if route matches
87  function handleRoute(route, req, res) {
88      var url = req.url;
89      var parsedUrl = parseurl(req);
90
91      if (parsedUrl.path.indexOf(route.apiRoute) === 0) {
92          req.url = url.replace(route.apiRoute, '');
93          proxy.web(req, res, { target: route.upstreamUrl });
94          return true;
95      }
96  }
97
98  // generate upstream url from containerDetails
99  function getUpstreamUrl(containerDetails) {
100     var ports = containerDetails.NetworkSettings.Ports;
101     for (id in ports) {
102         if (ports.hasOwnProperty(id)) {
103             return 'http://' + containerDetails.NetworkSettings.IPAddress + ':' + id.split('/')[0];
104         }
105     }
106 }
107
108 // send 502 response to the client in case of an error
109 function returnError(req, res) {
110     res.writeHead(502, {'Content-Type': 'text/plain'});
111     res.write('Bad Gateway for: ' + req.url);
112     res.end();
113 }
```

Figura 98: Porción de código del archivo service.js (2)
Fuente: El autor
Elaboración: El autor

- Finalmente se tiene la configuración del respectivo *Dockerfile:* se agrega el directorio de trabajo, se instalan las dependencias y se indica el comando que iniciará a la aplicación. (**Figura 99**)

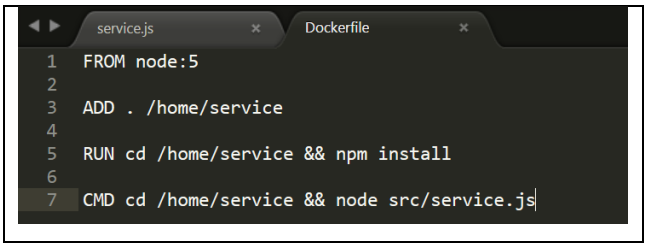


Figura 99: Configuración de archivo Dockerfile.
Fuente: El autor
Elaboración: El autor

- En la **Figura 100** (similar a la expuesta en la **Figura 4**) se posee la estructura que implementa el API Gateway; el cliente hace una solicitud a un MS, para lo cual se debe acceder primeramente al API Gateway el que es el encargado de re direccionar la solicitud a la dirección correcta.

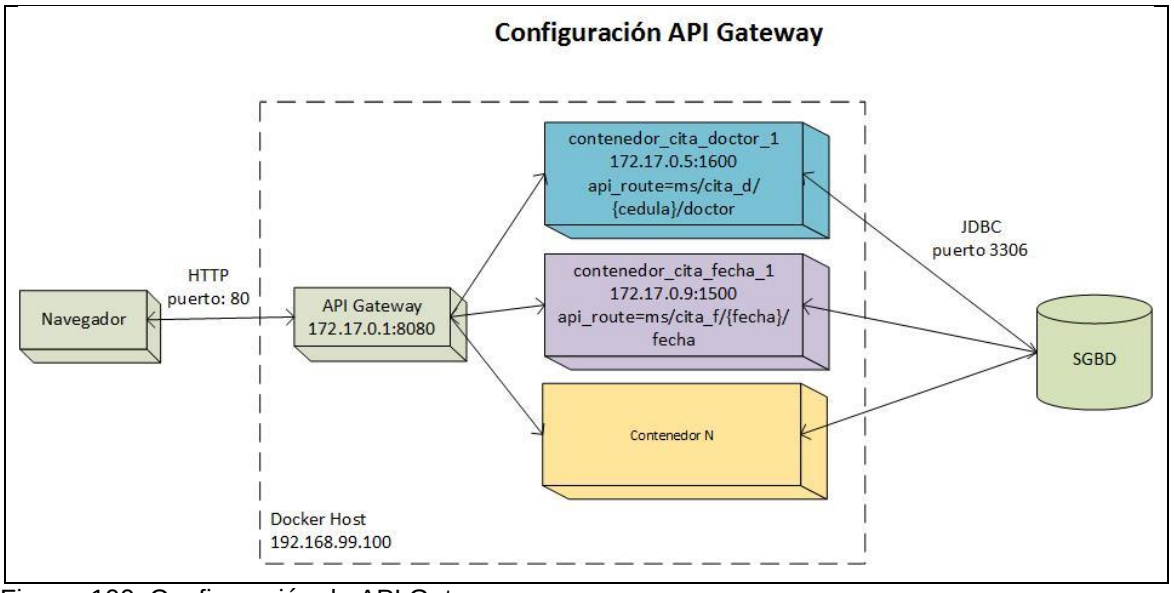


Figura 100: Configuración de API Gateway.
Fuente: El autor
Elaboración: El autor

- En la **Figura 101** se observa el funcionamiento del API Gateway a través de los logs del contenedor (también se lo puede observar en la consola mediante el comando *docker logs api-gateway-ms*), en la cual al iniciarse, realiza la conexión y establece el puerto para la comunicación, siendo esto correcto, procede a encontrar los servicios respectivos a los cuales se les ha asignado la *apiRoute,* y los registra; cuando se detiene a un contenedor automáticamente lo detecta y lo elimina del registro, asi mismo cuando uno nuevo se inicia (y tiene asignada la *apiRoute* respectiva) se agrega automáticamente.

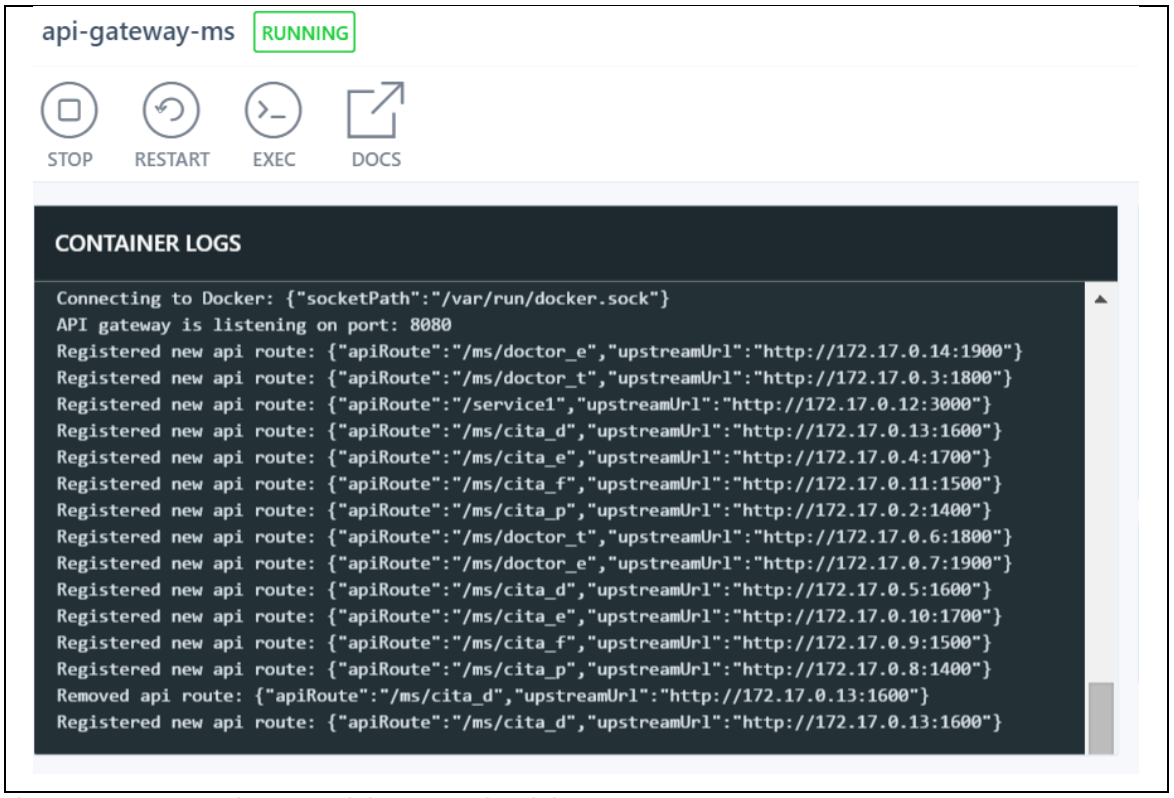


Figura 101: Contanier Logs del contenedor del API Gateway.
Fuente: El autor
Elaboración: El autor

Anexo N: Aplicación de Consumo

- Para verificar la correcta funcionalidad de las aplicaciones desarrolladas, se ha creído conveniente realizar una pequeña aplicación de consumo en JAVA.
- Consumo de RESTful desarrollado en JAVA:
    - En la **Figura 102** se observa una porción del código implementado, que nos permite consumir (en este caso) 2 servicios RESTful (consulta de citas por cedula del paciente, y por fecha de la cita), y además se especifica el formato de respuesta esperado (por cada solicitud), en el cual se colocaron los 3 desarrollados como son: Json, XML y HTML.

```
17          Client cliente = ClientBuilder.newClient();
18          WebTarget objetivo = cliente.target("http://localhost:8080/tesis_servicio_RESTful/ms/cita/1233451/paciente");
19          System.out.println("CITAS POR CEDULA ==== RECUPERACION DE DATOS EN JSON");
20          System.out.println(objetivo.request(MediaType.APPLICATION_JSON).get(String.class));
21          System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN XML");
22          System.out.println(objetivo.request(MediaType.APPLICATION_XML).get(String.class));
23          System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN HTML");
24          System.out.println(objetivo.request(MediaType.TEXT_HTML).get(String.class));
25
26          Client cliente2 = ClientBuilder.newClient();
27          WebTarget objetivo2 = cliente2.target("http://localhost:8080/tesis_servicio_RESTful/ms/cita/25062017/fecha");
28          System.out.println("\n\nCITAS POR FECHA ==== RECUPERACION DE DATOS EN JSON");
29          System.out.println(objetivo2.request(MediaType.APPLICATION_JSON).get(String.class));
30          System.out.println("\nCITAS POR FECHA ==== RECUPERACION DE DATOS EN XML");
31          System.out.println(objetivo2.request(MediaType.APPLICATION_XML).get(String.class));
32          System.out.println("\nCITAS POR FECHA ==== RECUPERACION DE DATOS EN HTML");
33          System.out.println(objetivo2.request(MediaType.TEXT_HTML).get(String.class));
```

Figura 102: Porción de código de consumo para RESTful JAVA
Fuente: El autor
Elaboración: El autor

    - En la **Figura 103** se observa una porción de la respuesta proporciona de la ejecución del código, en la cual se muestran los resultados en los formatos solicitados.

```
Output
Java DB Database Process | GlassFish Server | tesis_cliente_RESTful (run)
run:
CITAS POR CEDULA ==== RECUPERACION DE DATOS EN JSON
[{"doctor":{"apellidos":"Carrion","cedula":"26745","genero":"Masculino","idDoctor":7,"nombres":"Juan"},"especialidad":{"idEspecialidad":

CITAS POR CEDULA ==== RECUPERACION DE DATOS EN XML
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><citas><cita><doctor><apellidos>Carrion</apellidos><cedula>26745</cedula><genero>

CITAS POR CEDULA ==== RECUPERACION DE DATOS EN HTML
<!DOCTYPE html><html><head><style>table, th, td { border: 1px solid black; border-collapse: collapse;</style><title>Listar Citas</title>


CITAS POR FECHA ==== RECUPERACION DE DATOS EN JSON
```

Figura 103: Porción del resultado del consumo (RESTful JAVA)
Fuente: El autor
Elaboración: El autor

- Consumo de RESTful desarrollado en Node.js:
    - En la **Figura 104** se observa una porción del código implementado, que nos permite consumir (en este caso) 2 servicios RESTful (consulta

de citas por cedula del paciente, y por fecha de la cita), y además se especifica el formato de respuesta esperado (por cada solicitud), en el cual se colocaron los 3 desarrollados como son: Json, XML y HTML.

```
76          WebTarget objetivo_node = cliente.target("http://localhost:1400/ms/cita/1233451/paciente");
77          System.out.println("CITAS POR CEDULA ==== RECUPERACION DE DATOS EN JSON");
78          System.out.println(objetivo_node.request(MediaType.APPLICATION_JSON).get(String.class));
79          System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN XML");
80          System.out.println(objetivo_node.request(MediaType.APPLICATION_XML).get(String.class));
81          System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN HTML");
82          System.out.println(objetivo_node.request(MediaType.TEXT_HTML).get(String.class));
83
84          WebTarget objetivo_node2 = cliente.target("http://localhost:1500/ms/cita/25062017/fecha");
85          System.out.println("CITAS POR FECHA ==== RECUPERACION DE DATOS EN JSON");
86          System.out.println(objetivo_node2.request(MediaType.APPLICATION_JSON).get(String.class));
87          System.out.println("\nCITAS POR FECHA ==== RECUPERACION DE DATOS EN XML");
88          System.out.println(objetivo_node2.request(MediaType.APPLICATION_XML).get(String.class));
89          System.out.println("\nCITAS POR FECHA ==== RECUPERACION DE DATOS EN HTML");
90          System.out.println(objetivo_node2.request(MediaType.TEXT_HTML).get(String.class));
```

Figura 104: Porción de código de consumo para RESTful NODE
Fuente: El autor
Elaboración: El autor

- En la **Figura 105** se observa una porción de la respuesta proporciona de la ejecución del código, en la cual se muestran los resultados en los formatos solicitados.

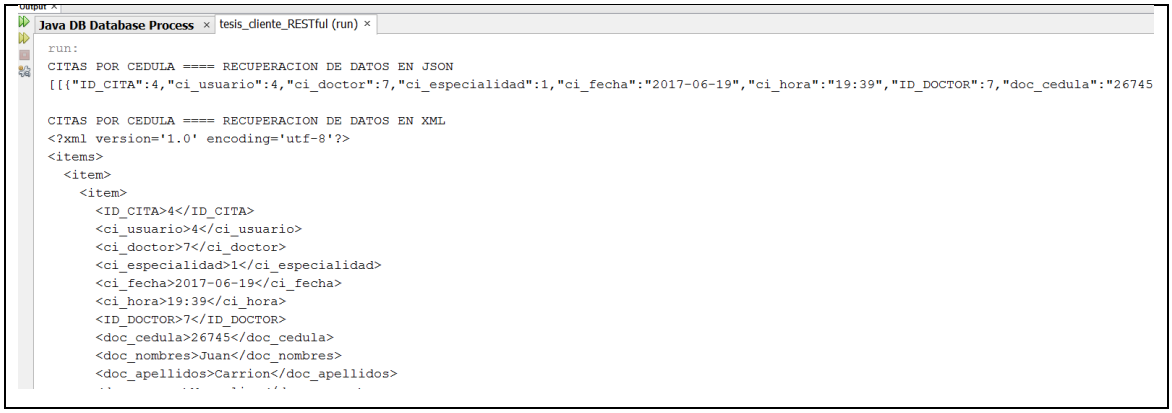


Figura 105: Porción del resultado del consumo (RESTful NODE)
Fuente: El autor
Elaboración: El autor

- Consumo de RESTful desarrollado en Docker (recordar BAJAR los 2 firewalls tanto del Antivirus como del sistema Operativo):
  - En la **Figura 106** se observa una porción del código implementado, que nos permite consumir (en este caso) 2 servicios RESTful (consulta de citas por cedula del paciente, y por fecha de la cita), y además se especifica el formato de respuesta esperado (por cada solicitud), en el cual se colocaron los 3 desarrollados como son: Json, XML y HTML.

```
125            WebTarget objetivo_docker1 = cliente.target("http://192.168.99.100/ms/cita_p/1233451/paciente");
126            System.out.println("CITAS POR CEDULA ==== RECUPERACION DE DATOS EN JSON");
127            System.out.println(objetivo_docker1.request(MediaType.APPLICATION_JSON).get(String.class));
128            System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN XML");
129            System.out.println(objetivo_docker1.request(MediaType.APPLICATION_XML).get(String.class));
130            System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN HTML");
131            System.out.println(objetivo_docker1.request(MediaType.TEXT_HTML).get(String.class));
132
133            WebTarget objetivo_docker2 = cliente.target("http://192.168.99.100/ms/cita_f/25062017/fecha");
134            System.out.println("CITAS POR FECHA ==== RECUPERACION DE DATOS EN JSON");
135            System.out.println(objetivo_docker2.request(MediaType.APPLICATION_JSON).get(String.class));
136            System.out.println("\nCITAS POR FECHA ==== RECUPERACION DE DATOS EN XML");
137            System.out.println(objetivo_docker2.request(MediaType.APPLICATION_XML).get(String.class));
138            System.out.println("\nCITAS POR FECHA ==== RECUPERACION DE DATOS EN HTML");
139            System.out.println(objetivo_docker2.request(MediaType.TEXT_HTML).get(String.class));
140
```

Figura 106: Porción de código de consumo para RESTful MS
Fuente: El autor
Elaboración: El autor

- En la **Figura 107** se observa una porción de la respuesta proporciona de la ejecución del código, en la cual se muestran los resultados en los formatos solicitados.

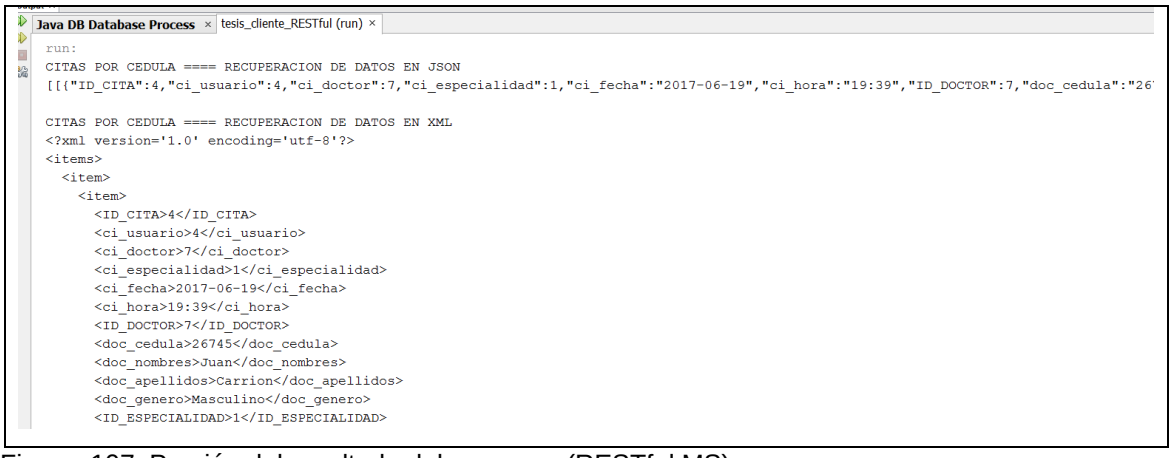


Figura 107: Porción del resultado del consumo (RESTful MS)
Fuente: El autor
Elaboración: El autor

- En la **Figura 108** se observa el log de un contenedor (contenedor_cita_doctor_2) al procesar una solicitud.

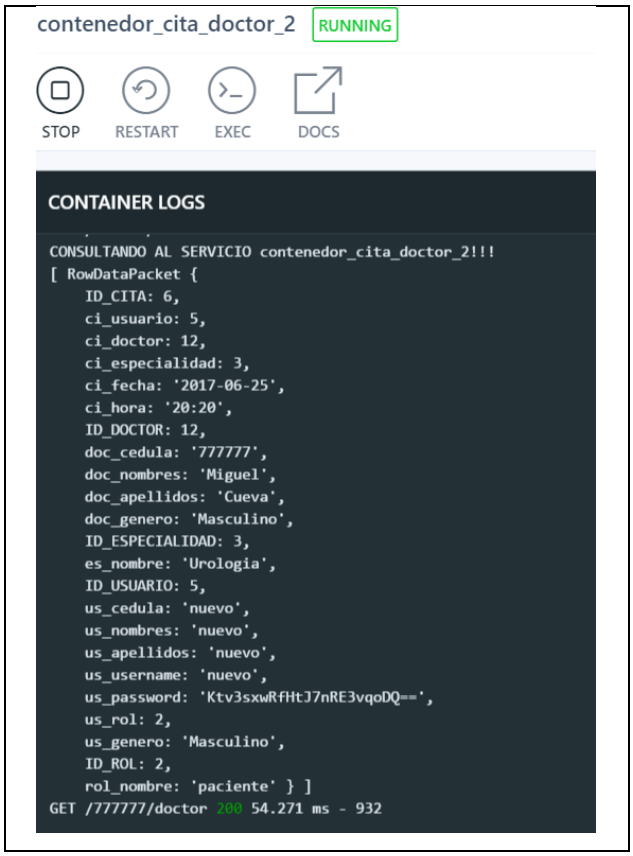


Figura 108: Contanier Log del contenedor *contenedor_cita_doctor_2*
Fuente: El autor
Elaboración: El autor

- En la **Figura 109** se observa el log del contenedor del API Gateway (api-gateway-ms) en el cual se encuentran registradas automáticamente las *api route* de los MS activos, en el cual indica la apiRoute, la dirección IP (manejada por Docker) y el puerto asignado a cada contenedor.

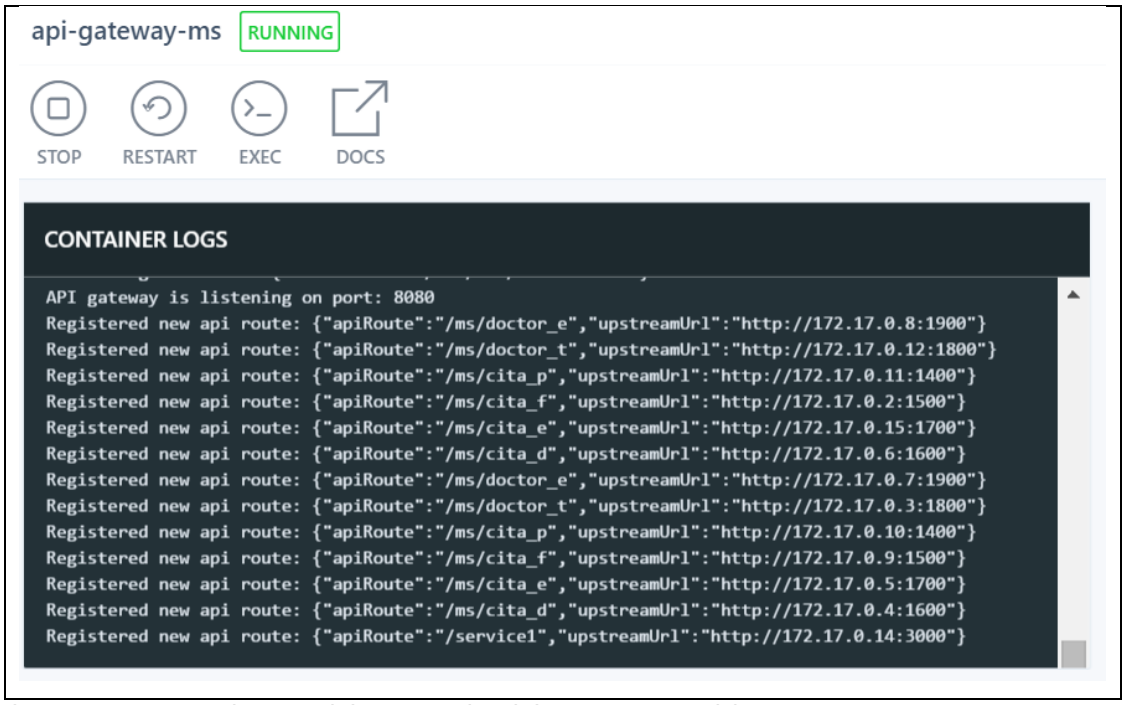


Figura 109: Contanier Log del contenedor del API Gateway (1)
Fuente: El autor
Elaboración: El autor

- Se procedió a dar de baja a un contenedor (contenedor_cita_doctor_2) esto con la finalidad de comprobar que se re direcciona a un contenedor activo; el contenedor es eliminado automáticamente del API Gateway como se observa en la **Figura 110** al detectar que el contenedor ya no está disponible.

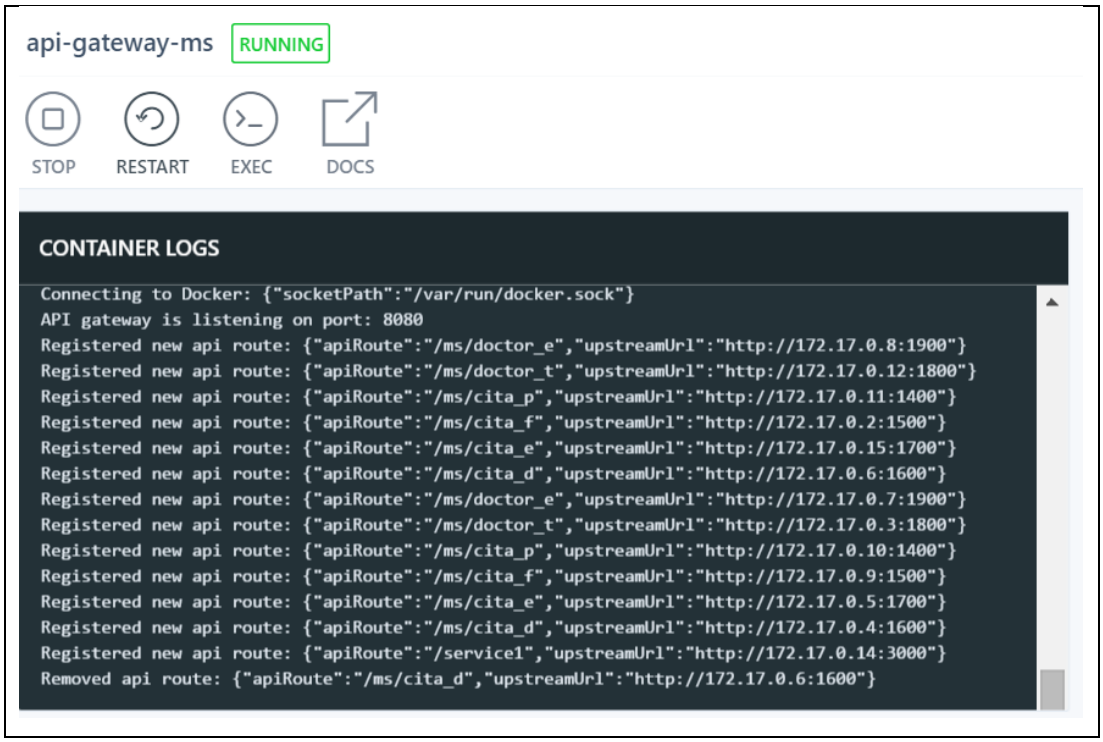


Figura 110: Contanier Log del contenedor del API Gateway (2)
Fuente: El autor
Elaboración: El autor

- En este caso lo re direccionó automáticamente al contenedor contenedor_cita_doctor_1 (ver **Figura 111**)

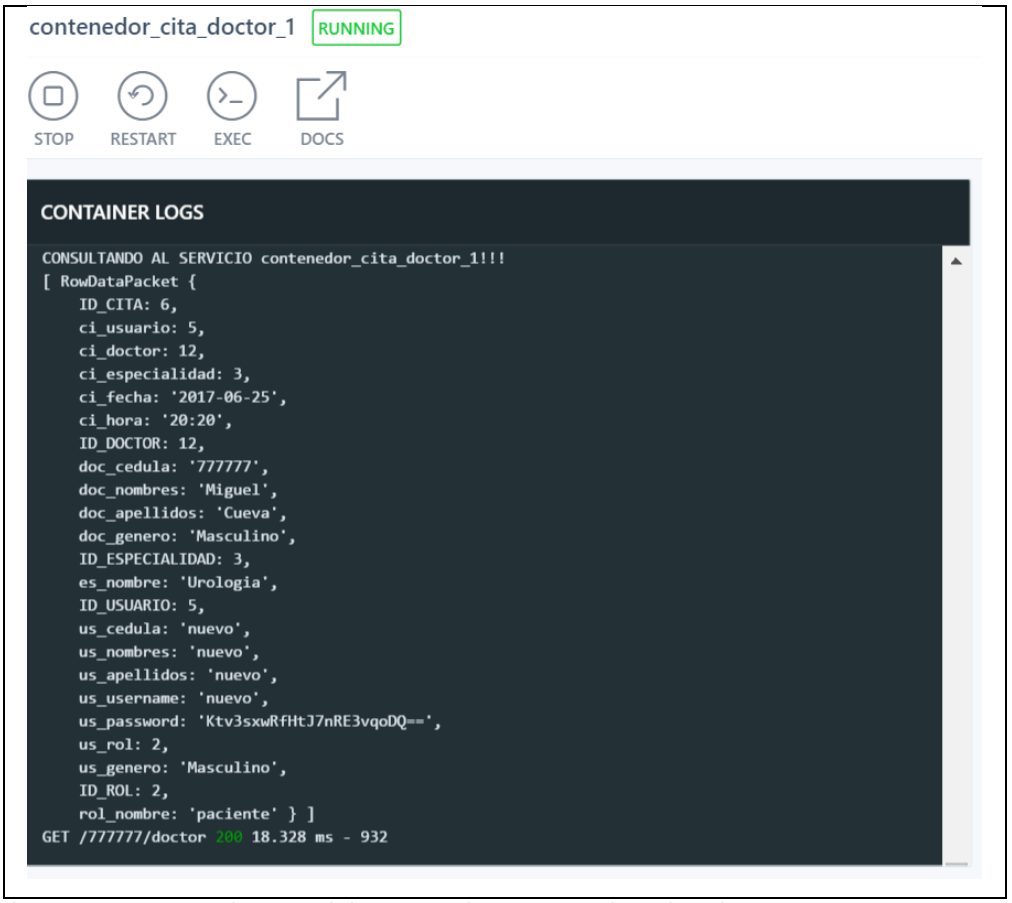


Figura 111: Contanier Log del contenedor *contenedor_cita_doctor_1*
Fuente: El autor
Elaboración: El autor

- Al momento de la ejecución del código, el resultado se mantiene inmutable (**Figura 112**).

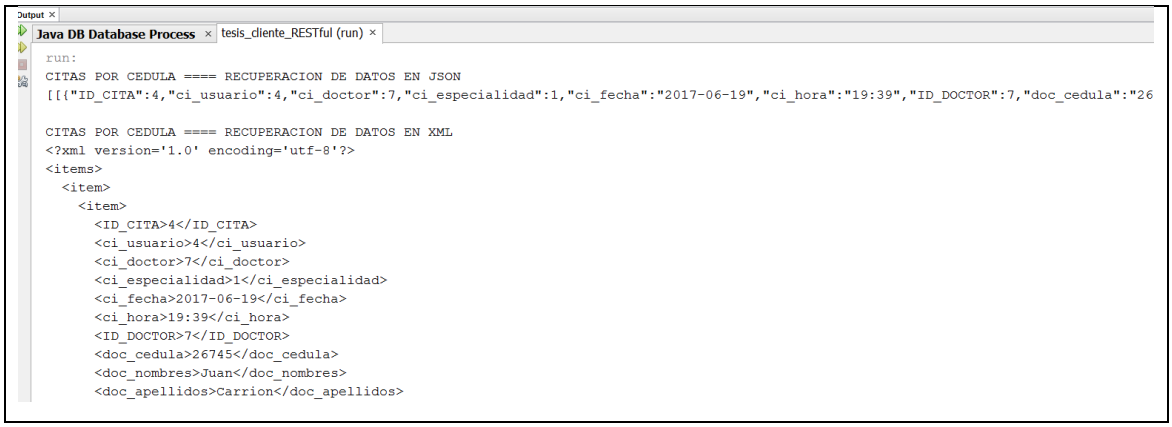


Figura 112: Porción del resultado del consumo (RESTful MS)
Fuente: El autor
Elaboración: El autor

- Al ingresar a una URL que no se encuentra registrada en el API Gateway (**Figura 113**), este último nos retorna un mensaje de *Bad Gateway 500* (implementada en el Gateway), el cual nos indica de que la solicitud no se puede realizar por que no se encuentra la dirección (**Figura 114**)

```
125             WebTarget objetivo_docker1 = cliente.target("http://192.168.99.100/mss/cita_p/1233451/paciente");
126             System.out.println("CITAS POR CEDULA ==== RECUPERACION DE DATOS EN JSON");
127             System.out.println(objetivo_docker1.request(MediaType.APPLICATION_JSON).get(String.class));
128             System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN XML");
129             System.out.println(objetivo_docker1.request(MediaType.APPLICATION_XML).get(String.class));
130             System.out.println("\nCITAS POR CEDULA ==== RECUPERACION DE DATOS EN HTML");
131             System.out.println(objetivo_docker1.request(MediaType.TEXT_HTML).get(String.class));
```

Figura 113: Porción de código de consumo para RESTful MS
Fuente: El autor
Elaboración: El autor

```
Output ×
Java DB Database Process × | tesis_cliente_RESTful (run) ×
CITAS POR CEDULA ==== RECUPERACION DE DATOS EN JSON
Exception in thread "main" javax.ws.rs.ServerErrorException: HTTP 502 Bad Gateway
        at org.glassfish.jersey.client.JerseyInvocation.createExceptionForFamily(JerseyInvocation.java:1124)
        at org.glassfish.jersey.client.JerseyInvocation.convertToException(JerseyInvocation.java:1104)
        at org.glassfish.jersey.client.JerseyInvocation.translate(JerseyInvocation.java:882)
        at org.glassfish.jersey.client.JerseyInvocation.lambda$invoke$1(JerseyInvocation.java:766)
        at org.glassfish.jersey.client.JerseyInvocation$$Lambda$94/78204644.call(Unknown Source)
        at org.glassfish.jersey.internal.Errors.process(Errors.java:315)
        at org.glassfish.jersey.internal.Errors.process(Errors.java:297)
        at org.glassfish.jersey.internal.Errors.process(Errors.java:228)
        at org.glassfish.jersey.process.internal.RequestScope.runInScope(RequestScope.java:414)
        at org.glassfish.jersey.client.JerseyInvocation.invoke(JerseyInvocation.java:764)
        at org.glassfish.jersey.client.JerseyInvocation$Builder.method(JerseyInvocation.java:427)
        at org.glassfish.jersey.client.JerseyInvocation$Builder.get(JerseyInvocation.java:323)
        at tesis_cliente_restful.Tesis_cliente_RESTful.main(Tesis_cliente_RESTful.java:127)
Java Result: 1
BUILD SUCCESSFUL (total time: 5 seconds)
```

Figura 114: Resultado del consumo (RESTful MS)
Fuente: El autor
Elaboración: El autor