COMUNICACIONES
SECRETARÍA DE INFRAESTRUCTURA, COMUNICACIONES Y TRANSPORTES
INSTITUTO MEXICANO DEL TRANSPORTE

Arquitectura de microservicios para optimizar el acceso a datos del SiT Log Lab

Cesar Jaime Montiel Moctezuma
Miguel Gastón Cedillo Campos
Marisol Barrón Bastida
Bernardo Hernández Sánchez

Publicación Técnica No. 733
Querétaro, México
2023

ISSN 0188-7297

Esta investigación fue realizada en la Coordinación de Transporte Integrado y Logística del Instituto Mexicano del Transporte, por el Dr. Cesar Jaime Montiel Moctezuma, Dr. Miguel Gastón Cedillo Campos, la Mtra. Marisol Barrón Bastida y el Mtro. Bernardo Hernández Sánchez.

Este trabajo es el producto final del proyecto de investigación interna TI_07_21 Implementación de una arquitectura de microservicios para mejorar el acceso a la información de los sistemas del Laboratorio Nacional en Sistemas de Transporte y Logística (SiT Log Lab).

Se agradece la revisión y aportaciones del Dr. Carlos Martner Peyrelongue, coordinador de Transporte Integrado y Logística del IMT, cuyas observaciones y contribuciones mejoraron la calidad de este documento.

Así mismo, se agradece el apoyo brindado por el Laboratorio Nacional de Sistemas de Transporte y Logística (SiT-LOG Lab), cuyas capacidades tecnológicas facilitaron el buen desarrollo de este proyecto.

Tabla de Contenido

Índice de Figuras

Página

Sinopsis

En la actualidad, la importancia de la implementación de nuevas tecnologías en el sector transporte ha incrementado debido al auge que ha tenido la industria 4.0 y la nueva tendencia de la revolución digital que se encuentra redefiniendo muchos aspectos de la industria y la logística. El Laboratorio Nacional en Sistemas de Transporte y Logística es una unidad estratégica de investigación aplicada con sede principal en el Instituto Mexicano del Transporte y con colaboración de otras instituciones de investigación a nivel nacional e internacional. Dentro del Laboratorio se han implementado una gran cantidad de proyectos de investigación con productos relacionados a sistemas informáticos que mejoran la toma de decisiones en el sector logístico. Sin embargo, estos proyectos fueron elaborados como una arquitectura monolítica y no cuentan con la capacidad para comunicarse entre sí. Por esta razón, este trabajo propone e implementa la idea de microservicios para migrar los sistemas actuales, con el objetivo de crear un sistema que permita la compatibilidad y comunicación entre proyectos, que sea escalable, resiliente, seguro, y ofreciendo una estrategia sencilla para desarrollar nuevos proyectos. Se explican las características de los microservicios, técnicas de migración de sistemas, y las tecnologías utilizadas para la nueva arquitectura.

Abstract

Implement new technologies on research projects have increased their importance due to the rise of Industry 4.0 and new trends of digital revolution that redefine many aspects of logistics. The National Laboratory for Transportation Systems and Logistics is a strategic research unit with headquarters at the Mexican Institute of Transportation; it has collaboration with national and international research centers. This Laboratory has the infrastructure and equipment for scientific and technological development and it contributes to the training of human resources in transport, logistic and supply chain areas. It has implemented several research projects with national and international importance, and some computational systems have been developed from these results to improve decision making in logistics sector. However, these projects were developed as a monolithic architecture and don't have the capacity to communicate each other. For this reason, this work proposes and implements the idea of microservices as an option to migrate current systems, with the objective of creating a system to allow the communication between projects, scalable, resilient, secure, and as a simple strategy to develop new applications. Furthermore, this work defines and describe the advantages and disadvantages of microservices architecture, systems migration techniques, and the technologies implemented into the proposed architecture.

Introducción

El Laboratorio Nacional en Sistemas de Transporte y Logística (SiT-LOG Lab) es una unidad estratégica de investigación aplicada dentro del IMT con colaboración de otras instituciones de investigación a nivel nacional e internacional, que busca reforzar la estructura y equipamiento para el desarrollo científico y tecnológico, a través de la innovación y formación de recursos humanos en las áreas de transporte, logística y cadena de suministro. La investigación realizada más destacada del SiT-LOG Lab se enfoca en logística urbana y de autotransporte nacional a través de proyectos que implementan nuevas tecnologías como Big Data, Inteligencia Artificial, Drones, Internet de las Cosas, Realidad Aumentada, entre otras tecnologías.

En la actualidad la importancia de la implementación de nuevas tecnologías en el sector transporte ha incrementado debido al auge que ha tenido la industria 4.0 (Ramírez, 2019) y la nueva tendencia de la revolución digital que se encuentra redefiniendo muchos aspectos de la industria y la logística. Con base en dicha tendencia es necesario actualizar los sistemas de información para que puedan acoplarse de manera eficiente con los sistemas informáticos existentes en la actualidad, para mejorar las soluciones tecnológicas ofrecidas a través de los distintos proyectos y aumentar la motivación para el desarrollo de las líneas de investigación que han surgido en los últimos años.

Si bien actualmente ya se cuenta con plataformas informáticas que ofrecen avances tecnológicos en el sector transporte llamados Sistemas Inteligentes de Transporte, pero la forma de acceder a esta información, hace que la toma de decisiones no sea rápida, debido a que se necesita una interacción directa de un usuario final con el sistema. Regularmente se accede a dichos sistemas a través de diferentes medios, desde teléfonos hasta computadora portátil o de escritorio, sin embargo, en la mayoría de los casos no tienen la capacidad de comunicarse con otros sistemas inteligentes. Una solución a este problema es desarrollar en cada uno de los proyectos la capacidad de poder interactuar con otros sistemas a través de una comunicación autónoma, flexible y escalable.

Actualmente el Laboratorio Nacional en Sistemas de Transporte y Logística se encuentra trabajando en una plataforma, la cual, genera indicadores en diferentes temas como lo son la confiabilidad de los

tiempos de viaje del autotransporte de carga, zonas de riesgo de robo, información meteorológica, entre otros temas importantes para la infraestructura carretera (Transporte, 2018). Sin embargo, aún no tiene la capacidad para interactuar con otros sistemas informáticos pertenecientes a actores principales que utilizan el proyecto para mejorar la toma de decisiones.

Este proyecto se vincula con los siguientes puntos del Plan Nacional de Desarrollo 2019-2024 (Diario Oficial de la Federación, 2019):

3.3.2 Impulsar el desarrollo y adopción de nuevas tecnologías en los sectores productivos y la formación de capacidades para aprovecharlas, vinculando la investigación con la industria y los usuarios.

Con base en lo anterior, también está relacionado con las estrategias prioritarias del Programa Sectorial de Comunicaciones y Transportes 2020-2024 (Secretaria de Comunicaciones y Transportes, 2020), específicamente la estrategia 2.6.5 “Fomentar la implementación de sistemas inteligentes de transporte” y la 3.4.1 “Identificar, analizar y experimentar tecnologías y herramientas de telecomunicaciones, radiodifusión e información para fomentar el desarrollo tecnológico de México”.

Este proyecto tiene como objetivo mejorar la arquitectura de software a través de una arquitectura de microservicios para optimizar la generación de información en los distintos proyectos desarrollados dentro del SiT-LOG Lab, y mejorar la capacidad de comunicación con otros sistemas informáticos, aproximándonos así, a la ideología de sistemas inteligentes en el sector transporte.

En cuanto al alcance del proyecto, se contempla el desarrollo e implementación de los microservicios para los proyectos del laboratorio, teniendo en cuenta la configuración de la infraestructura de computo necesaria para la ejecución y la lógica de negocio de cada proyecto para agilizar y facilitar la migración a la nueva arquitectura.

La metodología para esta investigación se inicia con el método inductivo, mediante el estudio de la información referente a migración de sistemas computacionales de arquitecturas monolíticas a arquitectura de microservicios: antecedentes, características, ventajas y desventajas de este paradigma. Posteriormente, el análisis de diferentes metodologías de desarrollo que sirvan para la implementación y codificación en los sistemas informáticos actuales. Finalmente, pruebas de conectividad entre microservicios, y documentación que sirva como referencia para el uso de los mismos, conclusiones y recomendaciones para otros sistemas.

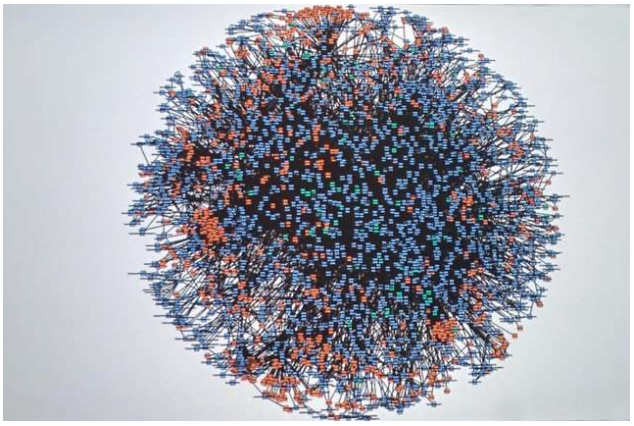
1. Antecedentes

La mayoría de las grandes empresas, además de las nuevas incorporaciones, construyen sus sistemas utilizando una arquitectura monolítica. Ésta es la mejor manera debido a que para la fase inicial de los proyectos es mucho más rápido desarrollar y configurar un proyecto monolítico, sin embargo, después de un tiempo, surgen problemas debido a que dichos proyectos no tienen la maduración suficiente, y con el crecimiento del sistema, el código se vuelve más complicado, la arquitectura más compleja, y se necesitan más desarrolladores para mantenerla. Al mismo tiempo, se comienza a perder la velocidad, flexibilidad y agilidad para reaccionar a las nuevas necesidades, que se convierten en nuevas funcionalidades, del sistema original. El Laboratorio Nacional en Sistemas de Transporte y Logística no es exento a esto, los proyectos desarrollados en las primeras fases fueron relevantes con resultados bastante confiables, pero al escalar poco a poco los proyectos, comenzaron a generar problemas. Además, ya que el objetivo original de los proyectos eran un resultado específico, no se preveía la comunicación y transferencia de información entre dichos sistemas, es por eso, que no son compatibles para compartir datos que podrían ser de utilidad entre estos.

En el entorno empresarial actual, con el incremento del avance tecnológico y la implementación de soluciones digitales, para las operaciones diarias en diferentes instituciones y organizaciones, debido al COVID-19, es necesario mantener actualizados los sistemas actuales para cubrir las necesidades de tecnologías de la información, y en muchos casos, para sistemas “obsoletos”, hablando tecnológicamente, la necesidad de migración de estos sistemas heredados a nuevas tecnologías y herramientas se ha convertido en una prioridad. Las nuevas necesidades, han provocado que las nuevas aplicaciones deban implementarse en un entorno ágil, lo que conlleva a un desarrollo rápido y con capacidad de respuesta, mientras se mantiene de manera óptima el rendimiento, la seguridad y la rentabilidad.

De esta necesidad aparece la nueva arquitectura de microservicios, aunque no es un tema nuevo en el ámbito tecnológico, ha ganado un gran impacto en los últimos años. Los microservicios nacieron como un apoyo para la construcción e implementación de aplicaciones comerciales de manera rápida y efectiva en empresas desde los años 2005-2010 (Divante,

2021), donde grandes empresas como Amazon y Netflix empezaron a apostar en la transición de sus sistemas a una arquitectura de microservicios. Aunque en ese entonces no se desarrollaban sistemas monolíticos de gran tamaño, la mayoría de los componentes y servicios que se desarrollaban estaba estrechamente relacionados entre sí, y eso provocaba, que cambios importantes en el código, quedaran atascados en el proceso de implementación durante largos lapsos antes de que pudieran ser utilizados por sus clientes finales, Figura 1.1.



Fuente: CTO Amazon Werner Vogels. Twitter (Werner Vogels, 2016).

Figura 1.1 Gráfica en Tiempo Real dependencias Amazon 2008

Los microservicios, o arquitectura de microservicios, resolvieron estos problemas ofreciendo mayor flexibilidad en el desarrollo de aplicaciones, sistemas escalables y con costos operativos más bajos, además de reducir la complejidad del proceso de desarrollo de software.

Actualmente, el mercado de los microservicios en la nube, se espera que tenga un incremento de 683.2 millones de dólares registrados en el 2018 a 1880 millones de dólares para el año 2023, a una tasa de crecimiento anual compuesto del 22.4% (MarketsAndMarkets, 2018). Los principales factores que impulsan dicho mercado son las transformaciones digitales, la proliferación de esta arquitectura y la ideología de negocios orientados al cliente. Existe una gran cantidad de empresas a nivel mundial que utilizan y ofrecen ecosistemas basados en microservicios. Existen un gran número de empresas que han migrado sus sistemas a esta arquitectura, algunos ejemplos son: Amazon, BestBuy, Coca Cola, eBay, Netflix, Spotify, Uber, entre otros. El aumento de la demanda de arquitectura de microservicios en la nube se debe a sus diversas características tales como: menos tiempo de desarrollo, fácil implementación, reutilización en diferentes proyectos, mejor aislamiento de fallas, funciona bien con unidades estandarizadas (contenedores), integración con servicios de terceros, además, es fácil de entender.

En la logística y la cadena de suministro también se ha comenzado a utilizar este tipo de arquitectura, debido a que muchas empresas que ofrecen servicios logísticos, ya cuentan con sistemas que facilitan la gestión de las operaciones., Han comenzado a integrar este tema como actualizaciones y mejoras sobre sus aplicaciones actuales. Por ejemplo, en la logística, de manera general se encarga de los procesos de coordinación, gestión y transporte para los bienes que serán distribuidos hacia un cliente. Para cada uno de estos procesos se realizan actividades interdisciplinarias que involucran a varios departamentos dentro de una organización, para eso ya se pueden identificar aplicaciones que apoyan a toda la cadena de suministro. Entre el software que se puede identificar es el dedicado para la gestión de la cadena de suministro (*Supply Chain Management*-SCM), de relaciones con los clientes (*Customer Relationship Management*-CRM), para la planeación de recursos (*Enterprise Resource Planning* ERP), los sistemas de planificación (*Advanced Planning and Scheduling*-APS), entre otros. Todos estos sistemas tienen algo en común, tienen un conjunto de módulos de software con diferentes funcionalidades, pero relacionados entre sí. Dentro de estos se puede fácilmente integrar una arquitectura de microservicios para optimizar el sistema y brindarles la capacidad de escalado e integración con otras aplicaciones de manera sencilla.

2. Microservicios

Un microservicio es un término que comenzó a principios del 2005, con la presentación del Dr. Peter Rodgers en su presentación de la conferencia "*Web Services Edge*" donde utilizó el término "*Micro-Web-Services*", en la cual demuestra su pensamiento en contra del pensamiento convencional de ese entonces, cuando la arquitectura SOAP[1](*Simple Object Access Protocol*)-SOA[2](*Service Oriented Architecture*) (Microsoft, 2014; World Wide Web Consortium [W3C], 2007) estaba en un su auge, entonces abogó por los servicios REST[3](*Representational Transfer Protocol*) (Fielding & Taylor, 2000), donde analizó y definió a los componentes de software como microservicios web (Rodgers, 2005).

Posteriormente, Juval Löwy utilizó el término como una construcción de sistemas en los cuales cada una de sus clases eran servicios, además, definió la necesidad de tecnologías que pudieran admitir un uso tan granular de los servicios, de esta idea extendió el uso de *Windows Communication Foundation*[4] (WCF) (Microsoft, 2021) para realizar esta actividad, cada clase definirla y tratarla como un servicio mientras se mantiene el modelo de programación convencional de clases.

El término se utilizó de una manera más formal en un taller para arquitectos de desarrollo de software en Venecia (Dragoni et al., 2017) y se definió como un estilo de arquitectura en común para muchas aplicaciones, pero hasta mayo del 2012, fue cuando se definió el nombre apropiado como microservicios. Por otro lado, este término también fue utilizado en otras presentaciones de la 33va Conferencia para *Java Masters* por James Lewis (Lewis, 2012). Por otro lado, Adrian Cockcroft (Cockcroft, 2013) en una conferencia representando a Netflix, describió este enfoque como el pionero en el estilo de desarrollo para sistemas web. Esta idea ha

[1] Protocolo de Acceso a Objetos Simples que define como dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.

[2] Arquitectura Orientada a Servicios que define a un servicio como una representación lógica de una actividad de negocio que tiene un resultado específico.

[3] Protocolo de Transferencia de Estado Representacional define una arquitectura para obtener datos o indicar operaciones de estos datos, sin las abstracciones de los protocolos basados en patrones de intercambio de mensajes como SOAP.

[4] *Framework* enfocado al desarrollo de aplicaciones orientadas a servicios propuesta por Microsoft.

sido tomada por muchos empresas y por una gran parte de la comunidad de desarrolladores (Lewis & Fowler, 2014). Algunas definiciones que se pueden encontrar en la web son:

- "El término de Arquitectura de Microservicios como una forma particular para el diseño de las aplicaciones de software como conjunto de servicios implementables de forma independiente. Aunque no hay definiciones precisas para el estilo de la arquitectura, existen algunas características en común: las capacidades de negocio de la organización, despliegue automatizado, inteligencia en los nodos finales de las redes, y control descentralizado de lenguajes y datos" (Lewis & Fowler, 2014).
- "Los microservicios son una propuesta como arquitectura para construir sistemas, los cuales conforman un conjunto de pequeñas aplicaciones distribuidas y que se acoplan libremente para que los cambios realizados en cualquiera de estas, no afecte a todo el sistema" (RedHat, 2022b).
- "Los microservicios son un método arquitectónico para crear aplicaciones donde cada función o servicio principal se compila e implementa de forma independiente. La arquitectura de microservicios es distribuida y ligeramente acoplada, por lo que un error en un componente no interrumpe toda la aplicación. Los componentes independientes funcionan juntos y se comunican con contratos de API[5](*Applicattion Programming Interface*) bien definidos" (Microsoft, 2022b).
- "Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde el software está compuesto por pequeños servicios independientes que se comunican a través de API bien definidas. Los propietarios de estos servicios son equipos pequeños independientes. Las arquitecturas de microservicios hacen que las aplicaciones sean más fáciles de escalar y más rápidas de desarrollar. Esto permite la innovación y acelera el tiempo de comercialización de las nuevas características" (Amazon, 2022b).
- "Los grupos de pequeñas unidades modulares agrupadas según sus códigos son microservicios. Como no están tan interconectados como los monolitos, pueden desplegarse o ejecutarse de forma independiente. Si una unidad necesita ser modificada, las demás permanecen inalteradas. Además, cada unidad se centra sólo en un tema a la vez" (Nu Technology, 2021).

[5] La Interfaz de Programación de Aplicaciones es un conjunto de subrutinas, funciones y procedimientos que se ofrecen como una capa de abstracción para ser utilizada por otro software.

- “Un microservicio implementa un conjunto de funciones o características distintas. Cada microservicio es una mini-aplicación que tiene su propia arquitectura y lógica empresarial. Por ejemplo, algunos microservicios exponen una API que consumen otros microservicios o los clientes de la aplicación, como integraciones de terceros con puertas de enlace de pago y logística” (Google, 2021).

En los últimos años, la implementación de los microservicios y su arquitectura ha incrementado considerablemente en todo el mundo debido a la facil integración y desarrollo de los sistemas informaticos asi como la reducción de los problemas que el escalado de estos implica. De acuerdo a Google y las búsquedas a nivel global del tema se puede observar el incremento del interes en este tema.

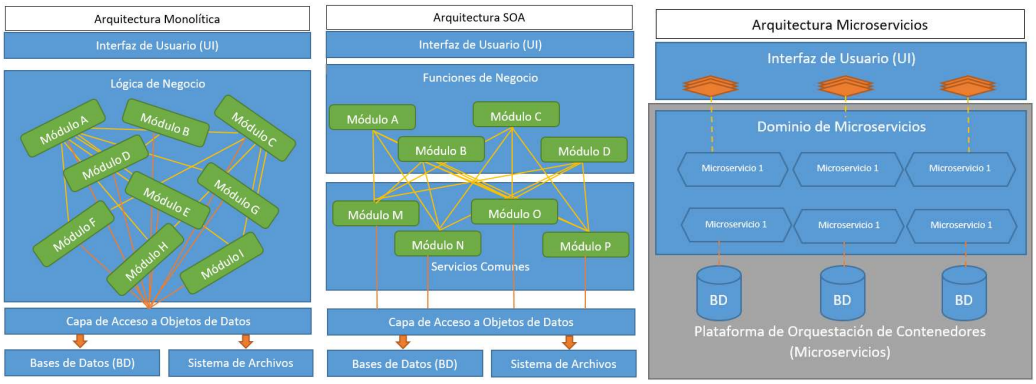
Fuente: ©2022 Google Trends (Google, 2022b)

Figura 2.1 Tendencia de los Microservicios en los últimos años

La arquitectura de microservicios ha mejorado de una manera efectiva el desarrollo de los sistemas computacionales, apoyado con *DevOps*[6] para la construcción y despliegue continuo.

Los microservicios se pueden visualizar como una evolución de SOA en la cual se guía para la creación de aplicaciones modulares, funcionales y autonomas, con una alta capacidad de reutilización de forma eficaz, en el cual, se despliega solo lo realmente necesario, en lugar de desplegar todo un sistema por completo sin necesidad.

[6] Son una serie de prácticas de desarrollo de software que combinan las actividades de desarrollo (Dev) y operaciones (Ops) para reducir el tiempo del ciclo de vida del desarrollo.



Fuente: Elaboración propia en base a (Sumerge, 2020)

Figura 2.2 Comparación Arquitecturas de Software

Los microservicios tienen dos características principales que los diferencian con respecto a los demás paradigmas:

- **Autónomos.** Cada componente en esta arquitectura se puede desarrollar, implementar, operar y escalar sin afectar el funcionamiento de otros servicios, es decir, los servicios no necesitan compartir código o implementaciones con otros servicios, y cualquier comunicación entre dichos componentes individuales ocurre a través del consumo de las API's.
- **Especializados.** Cada microservicio está diseñado para un conjunto de capacidades y se enfoca en resolver un problema específico. Para aquellos servicios que empiezan a escalar a través del tiempo por la cantidad de requerimientos solicitados, si el servicio comienza a ser complejo se puede dividir en servicios más pequeños.

Al implementar una arquitectura de microservicios en un sistema, se pueden identificar algunas ventajas (Amazon, 2022a), tales como:

- **Agilidad.** Los microservicios fomentan el agrupamiento de equipos pequeños e independientes que se apropian de las funcionalidades. Estos equipos actúan en un contexto pequeño y bien comprendido, y están facultados para trabajar de forma independiente y rápida, esto tiene como beneficio le reducción de tiempos de ciclo de desarrollo.
- **Escalamiento.** Los microservicios permiten que cada servicio se escale de forma independiente para satisfacer las necesidades del sistema, esto permite a los equipos adecuarse a las necesidades de la infraestructura, medir con precisión el costo de nuevas funcionalidades y mantener la disponibilidad de los servicios en cualquier momento sin importar la cantidad de peticiones realizadas a este.

- **Implementación sencilla.** Los microservicios permiten la integración y entrega continua, es decir, los cambios desarrollados y actualizaciones de software en los sistemas se deben realizar de manera ágil y fiable, facilitando la prueba de nuevas características y permite revertir cualquier cambio en caso de que estos nos funcionen. Esto trae como ventaja un bajo costo en los errores de desarrollo, los cuales permiten experimentar y agilizan la actualización de código, además de acelerar el tiempo de entrega de nuevos módulos.
- **Libertad tecnología.** Esta arquitectura no tiene un diseño único, es decir, cada uno de los equipos que integran nuevas funcionalidades tienen la libertad de elegir la mejor herramienta para resolver los problemas que surjan, esto les da la libertad de trabajar con diferentes lenguajes, bibliotecas, administradores de bases de datos, y otros *frameworks* que sean compatibles con la comunicación de las API's a consumir.
- **Reutilización de Código.** Otra ventaja que se tiene es debido a los módulos desarrollados de manera pequeña y bien definida, esto facilita a utilizar el mismo código para diferentes propósitos.
- **Resiliencia.** La resiliencia en los sistemas desarrollados con esta arquitectura es un punto clave, debido a que permite al sistema resistir mejor los errores en las aplicaciones. Por ejemplo, en un sistema monolítico, un error en un solo componente provoca que toda la aplicación falle, por otro lado, en los microservicios, si existe un error en algún servicio, las aplicaciones continúan trabajando y degradan la funcionalidad sin bloquear la aplicación completa.
- **Soporte.** En los últimos años, ha incrementado la popularidad de los microservicios, por lo tanto, ha provocado que el soporte a los mismos también se haya incrementado, dando la oportunidad de ser implementados y soportados por diferentes tecnológicas y proveedores en la nube.

Sin embargo, esta arquitectura también tiene algunos inconvenientes y desafíos para lograr una comunicación efectiva de la información, así como desarrollar e implementar sistemas eficaces que cumplan los requerimientos deseados (Google, 2021):

- **Complejidad.** El principal desafío es la gestión de los microservicios debido a la complejidad que se produce, ya que la arquitectura en si es un sistema distribuido. Es necesario implementar un mecanismo eficiente para la comunicación entre los servicios. Por otro lado, también es necesario que cada servicio pueda manejar fallas parciales y la falta de disponibilidad de otros servicios con los que se comuniquen.

- **Transacciones distribuidas.** Otro de los desafíos es la administración de las transacciones entre diferentes microservicios (transacción distribuida). Por lo general, en las operaciones comunes de un sistema monolítico, el sistema debe asegurarse de realizar de manera atómica todas las actualizaciones de información en la base de datos, en caso de que alguna de las actualizaciones falla, se cancela toda la operación y cambios ya realizados para regresar a un estado anterior, y así asegurar estabilidad de la información. Por otro lado, para los microservicios, las operaciones de actualización se hacen de manera distribuida, es decir, se tienen que actualizar varias bases de datos pertenecientes a un conjunto de servicios, y en caso de que se produzca un error, no es tan trivial realizar un seguimiento de la falla o éxito de las llamadas a los diferentes microservicios y revertirlos. En el peor de los casos se puede generar redundancia o datos incoherentes entre los servicios cuando se intenta regresar a un estado anterior.
- **Pruebas.** Las pruebas integrales de las aplicaciones basadas en la arquitectura de microservicios son aún más complejas que los sistemas monolíticos, debido a que, para realizar pruebas del flujo de algún proceso, es necesario interactuar con un conjunto de servicios que se comunican entre sí para completar cualquier operación, y eso eleva la complejidad dependiendo de la cantidad microservicios conectados.
- **Conectividad.** Por lo general, para la implementación de un sistema, es necesario el desarrollo de muchos microservicios, donde cada uno de estos tiene un conjunto de instancias de entorno de ejecución. Es necesario implementar un mecanismo para el descubrimiento de los servicios que permita a un microservicio ubicar y comunicarse con los demás.
- **Infraestructura.** Otra desventaja es que la arquitectura provoca una sobrecarga de operaciones, debido a que existe una gran cantidad de servicios a supervisar y generar alertas, además de que se pueden generar fallas debido al aumento de los puntos de comunicación entre servicios. Cada servicio puede ser compilado, probado, implementado y ejecutado con diferentes lenguajes y entornos, pero, deben agruparse en clústeres para la conmutación por error y resiliencia. Es por esta razón, que para tener una arquitectura de microservicios eficiente, es necesario de una infraestructura de alta calidad para la supervisión y operaciones que tiene que realizar.
- **Monitoreo e Historial de Registros.** Otro punto débil es la latencia, debido a que esta arquitectura permite que la aplicación realice más operaciones al mismo tiempo, y cada servicio se ejecuta de manera independiente. La latencia es medida a través del tiempo de

respuesta entre las llamadas en red de los servicios conectados que realizan una operación.

- **Versionamiento.** Debido a que cada microservicio es independiente, al desarrollar e implementar nuevas versiones, es necesario considerar la compatibilidad con versiones anteriores, debido a que, al realizar algún cambio no considerado con anterioridad a la comunicación con otros servicios, este puede provocar resultados inesperados, además, el no llevar un control adecuado en el versionamiento, ocasiona problemas con la administración y mantenimiento del sistema.
- **Depuración.** Un problema importante para la etapa de desarrollo es la depuración, en el caso de un sistema monolítico, al tener todos los módulos y componentes juntos, se puede realizar una depuración de todo el sistema sin problema, el detalle de los microservicios recae, en que no se puede depurar la funcionalidad de servicios externos al que se está desarrollando.

2.1 Comunicación entre microservicios

En una aplicación monolítica, los componentes se invocan directamente mediante llamadas a funciones; por otro lado, en una arquitectura de microservicios se encuentran un conjunto de servicios que interactúan entre sí a través de la red (Google, 2022c). Cuando se diseña la comunicación entre los servicios, antes que todo es necesario definir como serán las interacciones entre estos; existen dos opciones:

- **Interacción uno a uno.** Cada solicitud es procesada exactamente por uno de los microservicios en la red.
- **Interacción de uno a varios.** Cada solicitud es procesada por varios servicios a la vez.

Por otro lado, también es necesario considerar que tipo de interacción tendrá la comunicación:

- **Síncrona.** Cuando un servicio hace la petición a otro servicio en la red, este se bloquea hasta recibir la respuesta solicitada.
- **Asíncrona.** Cuando un servicio realiza una petición a cualquier otro servicio, este no necesita esperar la respuesta, y puede continuar realizando operaciones hasta que llegue la información solicitada.

Cada servicio suele utilizar una combinación entre las interacciones uno a uno o uno a varios, con sus respectivos tipos, del cual se pueden identificar cuatro operaciones principales:

- **Notificación:** operación en la cual un servicio solicita información a otro servicio, pero no se espera ni se envía ninguna respuesta.
- **Solicitud y respuesta síncronas:** un servicio realiza una petición, y un servicio responde de forma síncrona o asíncrona, para el primer caso se bloquea el servicio que solicitó la información, para el segundo caso, el servicio continuo sus operaciones normales, pero continúa esperando la respuesta de la petición.
- **Publicar y suscribirse:** utilizado para peticiones asíncronas de uno a varios, donde un servicio publica la información en la red, y uno o varios servicios consumen dicho mensaje.
- **Publicación y respuestas asíncronas:** este caso es muy similar al anterior, la diferencia radica en que, para este caso, el servicio que publicó la información, espera la respuesta de los servicios interesados en su mensaje.

Para implementar la comunicación entre servicios, se pueden elegir diferentes tecnologías de comunicación entre procesos. Algunos de estos pueden ser mecanismos síncronos basados en solicitudes y respuestas, como lo es REST basado en HTTP[7](*Hypertext Transfer Protocol*) (Fielding et al., 1999), gRPC[8](*Remote Procedure Call*) (gRPC Authors, 2022) o Thrift[9] (Apache Software Foundation, 2022). Por otro lado, también pueden utilizarse tecnologías para comunicación asíncrona basadas en mensajes, tales como AMQP[10](*Advanced Message Queuing Protocol*) (OASIS, 2022) o STOMP[11](*Streaming Text Oriented Messaging Protocol*) (Google, 2022d). Para el formato de transferencia de información pueden utilizarse varios formatos, algunos de estos pueden ser JSON[12](*JavaScript Object Notation*) (Ecma International, 2017), XML[13](*eXtensible Markup Language*) (World

[7] Protocolo de Transferencia de Hipertexto protocolo de comunicación que permite las transferencias de información a través de archivos en la *World Wide Web*.

[8] gRPC es un sistema de llamada a procedimiento remoto de alto rendimiento y de código abierto desarrollado por Google.

[9] Apache Thrift es un *framework* para la creación de servicios escalables e interoperables desarrollado por Facebook.

[10] Protocolo de Cola de Mensajes Avanzado estipula el comportamiento del como los mensajes son proveídos y consultados por diferentes entidades.

[11] Protocolo de Transferencia de Mensajes de Texto Simple proporciona un formato de conexión interoperable para comunicación entre un agente de mensajes con sus respectivos consumidores.

[12] Notación de Objetos de *Javascript* es un formato ligero de intercambio de datos, simple para escribirlo y leerlo por humanos.

13 Lenguaje de Marcado Extensible es un metalenguaje que permite definir lenguajes de marcas desarrollado por el World Wide Web Consortium utilizado para almacenar datos en forma legible.

Wide Web Consortium [W3C], 2020), y formatos binarios como Avro[14] (The Apache Software Foundation, 2021).

En la configuración de la comunicación de los microservicios, se identifican la mensajería y la comunicación basada en eventos, que están relacionadas a la interacción asíncrona: a) Mensajería. - Esta configuración quita la necesidad de que los servicios se comuniquen entre sí, y para resolver las peticiones, se implementa un agente de mensajes que administra y gestiona las peticiones y respuestas de todos los servicios; b) Basada en Eventos. – La comunicación entre los servicios se realiza a través de eventos que producen los servicios de manera individual, también se utiliza un agente de mensajes que registra los servicios y sus eventos asociados, posteriormente, cada servicio necesita registrarse en los eventos que quiere escuchar, y el agente responde a estos servicios cuando se ejecuta dicho evento.

Para el caso de la comunicación asíncrona tiene un conjunto de ventajas tales como:

- Reduce el acoplamiento entre servicios, debido a que divide la interacción de solicitud y respuesta en dos mensajes separados, con esto, el consumidor de un servicio inicia el mensaje de solicitud y espera la respuesta, por otro lado, el proveedor espera los mensajes de solicitud a los que responde con los mensajes respectivos. En ningún momento ni el consumidor ni el proveedor se bloquean para ejecutar las operaciones.
- Fallas aisladas. El remitente puede continuar enviando mensajes incluso si el consumidor falla. El consumidor recoge tareas pendientes cada vez que se recupera.
- Capacidad de respuesta. Un servicio ascendente responde más rápido si no espera respuestas de otros servicios, en el caso de cadenas de dependencias de servicios, la latencia incrementa de acuerdo a la espera de llamadas necesarias para la operación.
- Control de flujo. Una cola de mensajes actúa como un buffer, de modo que los receptores pueden procesar sus propios mensajes con la frecuencia que la necesiten.

Sin embargo, también existen algunos problemas que afectan el rendimiento de la comunicación asíncrona:

14 Avro es un marco de serialización de datos y llamadas de procedimiento remoto orientado a filas desarrollado por Apache en el proyecto Hadoop.

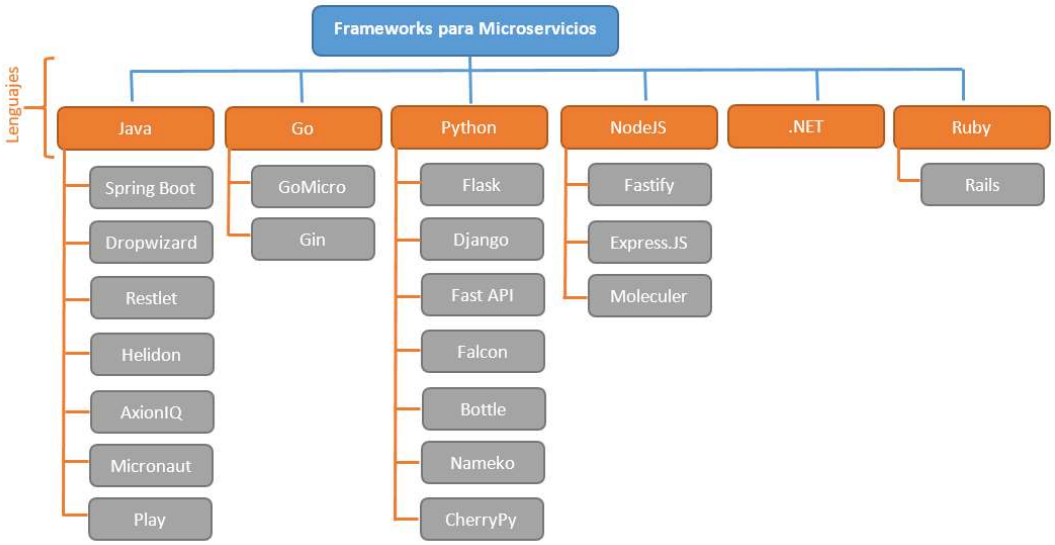
- Latencia. Si el agente de mensajes se convierte en un cuello de botella para el sistema, este comienza a afectar en la comunicación de todos los servicios.
- Sobrecarga en el desarrollo y pruebas. En la solicitud y respuesta de los mensajes podrían generarse problemas de duplicidad, y esto provocaría idempotencia[15] en las operaciones. Es difícil implementar y probar la semántica de las solicitudes y respuestas para todos los mensajes de todos los microservicios.
- Capacidad de procesamiento. Los mensajes asíncronos necesitan un gran procesamiento debido a la gran cantidad de información que deben analizar, y esto en la mayoría de los casos se convierte en un cuello de botella para el sistema.
- Manejo de errores. El control de los errores para un sistema asíncrono incrementa debido a que los emisores no conocen el estado de las solicitudes, esto dificulta la implementación lógica del sistema y los procesos.

2.2 Frameworks

Existe una gran cantidad de *frameworks*[16] que pueden ser utilizados para el desarrollo de microservicios, y en los últimos años han aparecido aún más, debido a la gran relevancia que han tomado los sistemas desarrollados con esta arquitectura; cada uno de estos *frameworks* se pueden clasificar de acuerdo a la tecnología o lenguaje de programación que tienen como base para su desarrollo, tal como se muestra en la Figura 2.3. Algunos de estos *frameworks*, ya existían desde antes que el término de microservicios apareciera, y a través del tiempo se fueron ajustando para adaptarse al paradigma.

15 El término idempotente se usa para describir una operación que produce los mismos resultados si se ejecuta una o varias veces.

16 Un framework es un conjunto estandarizado de conceptos, prácticas y criterios a seguir para el desarrollo de software, sirve como referencia con una estructura conceptual y tecnológica para modelar de manera organizada los sistemas.



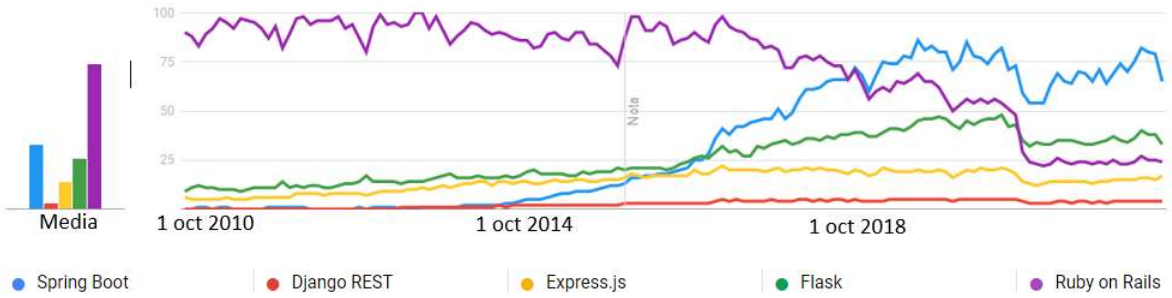
Fuente: Elaboración propia con base en (Kiran & Kumar, 2021)

Figura 2.3 Frameworks clasificados por lenguaje de programación

Cada uno de estos *frameworks* tiene sus propias características, ventajas y desventajas, pero eso no quiere decir que uno sea mejor que otro, al contrario, debido a las características de los microservicios, pueden trabajar en conjunto para compartir la información de diferentes proyectos (servicios), y cada uno de estos puede ser utilizado dependiendo de los requerimientos o funcionalidades que se deseen desarrollar.

La evolución del uso de estos *frameworks* se puede observar en la Figura 2.4, donde se identifican los *frameworks* que surgieron con los microservicios, y los que se adaptaron al paradigma. Debido a que esta investigación no es analizar cada uno de los *frameworks* existentes, sino, seleccionar alguno de estos para el desarrollo de los microservicios de los proyectos en el SiT Log Lab, solo se explicarán estos 5 *frameworks* que son los más importantes de acuerdo al ranking de GitHub[17].

[17] GitHub es un sitio "social coding". Te permite subir repositorios de código para almacenarlo en el sistema de control de versiones. En GitHub se maneja una métrica de “Estrellas” con las cuales se pueden identificar los repositorios con más seguidores o más gustados por la comunidad, entre más Estrellas es más reconocido el código a nivel mundial.



Fuente: ©2022 Google Trends (Google, 2022b)

Figura 2.4 Búsquedas de los *frameworks* en google

2.2.1 Spring Boot

Es el *framework* más popular del lenguaje Java para el desarrollo de microservicios (Gradle Enterprise, 2022; VMware, 2022b), está basado en el *framework* de Spring el cual proporciona un conjunto de proyectos para el desarrollo *full-stack*, y permite la construcción de sistemas a gran escala iniciando con una arquitectura simple a partir de los componentes que lo conforma. Puede ser fácilmente integrado con otros *frameworks* debido a la Inversión de Control. Spring Boot configura de manera ágil cualquiera de los módulos de Spring para cualquiera de las siguientes funcionalidades: gestión de datos, manejo de fallas, herramientas de integración, nube nativa, seguridad, gestión de configuración distribuida, descubrimiento de servicios, desempeño y pruebas. Actualmente cuenta con 61.6 mil estrellas en GitHub.

Fuente: ©2022 VMware (VMware, 2022a)

Figura 2.5 Logo del proyecto Spring

2.2.2 Rails

Es un *framework* que incluye todo lo necesario para crear aplicaciones web respaldadas por bases de datos de acuerdo al patrón *Model-View-Controller* (MVC) (Heinemeier Hansson, 2022; Rails Repository Contributors, 2022). Usar el patrón MVC es la clave para comprender como funciona internamente Rails, debido a que divide la aplicación en tres capas:

- La capa de Modelo donde se representan los modelos del dominio y encapsula la lógica de negocio específica para la aplicación, es decir, maneja la información de la base de datos, como son representados por objetos, y cuáles son los métodos que interactúan con estos para representar a funcionalidad del sistema.
- La capa de Vista que es la responsable de proveer las representaciones apropiadas de los recursos que serán visibles para el usuario final, la mayoría está basada en plantillas HTML obtenidas del Controlador.
- La capa de Controlador que se encarga de gestionar las peticiones HTTP y proveer las respuestas adecuadas, para páginas y aplicaciones web son las plantillas HTML y para el caso de los microservicios la información se genera en formatos XML y JSON.

Este proyecto cuenta con 50.9 mil estrellas en GitHub.

Fuente: ©2022 MIT License (David Heinemeier Hansson, 2022)

Figura 2.6 Logo del proyecto Rails

2.2.3 Flask

Es un *framework* de aplicación web ligero basado en el estándar de Interfaz de Puerta de Enlace del Servidor Web (Web Server Gateway Interface-WSGI)[18] (Flask Repository Contributors, 2022; Pallets, 2022). Está diseñado para que al empezar un proyecto sea fácil y rápido, pero también tiene la capacidad de escalar a aplicaciones complejas. Ofrece sugerencias, pero no impone dependencias ni diseño al desarrollar proyectos, es decir, depende de cada desarrollador elegir las herramientas y bibliotecas que desea utilizar. Es un proyecto con muchas extensiones proporcionadas por la comunidad que trabajan en este *framework*, dichas extensiones facilitan la adición de nuevas funcionalidades de manera sencilla para los sistemas desarrollados en Flask. Cuenta con 59.2 mil estrellas de GitHub.

[18] Es una especificación que describe como un servidor web se comunica con una aplicación web, y como las aplicaciones web pueden ser encadenadas en conjunto para procesar una petición.

Fuente: ©2010 Pallets (Pallets, 2022)

Figura 2.7 Logo del proyecto Flask

2.2.4 Express

Express es un *framework* flexible de Node.js que proporciona un conjunto sólido de funciones para aplicaciones web y móviles (Express Repository Contributors, 2022; StrongLoop & IBM, 2022). Cuenta con un conjunto de métodos HTTP y middleware para el desarrollo de API's robustas, tales como redireccionamiento y caching, y se enfoca en el alto rendimiento de los sistemas desarrollados. Actualmente tiene 57.3 mil estrellas de GitHub.

2.2.5 Django Rest

Django Rest es una herramienta flexible y poderosa para el desarrollo e implementación de API's Web (Django Repository Contributors, 2022; Encode OSS Ltd, 2022); cuenta con políticas de autenticación para OAuth1 y OAuth2, serialización para bases de datos relacionales y no relacionales, y tiene la peculiaridad que las características desarrolladas para este framework pueden ser seleccionadas de manera personalizable. Este *framework* cuenta con 23.5 mil estrellas en Github.

Fuente: ©2011 Encode OSS Ltd (Encode OSS Ltd, 2022)

Figura 2.8 Logo del proyecto Django Rest

2.2.6 Selección del *framework*

A partir de todas las características que tiene cada uno de los *frameworks*, es necesario seleccionar uno que se adecue a las funcionalidades que ya se encuentran implementadas en los sistemas actuales, que tengan la capacidad de ser fácilmente implementados y se adapten al escalado de los proyectos. Por ello, el *framework* utilizado para el desarrollo de este proyecto fue Spring Boot. Aunque no es tan relevante debido a que esta arquitectura de microservicios permite la interacción y trabajo

colaborativo de diferentes *frameworks*, siempre y cuando se normalicen las estrategias de comunicación. El cómo se codificará el sistema depende más de cada uno de los desarrolladores que del *framework* en sí. Para llegar a seleccionar Spring Boot, se tomaron en cuenta los siguientes criterios a consideración de los investigadores involucrados en este proyecto:

- **Popularidad.** Medida por la aceptación del *framework* en la industria en función de la cantidad de clientes que han trabajado con este en un estándar empresarial. Indicadores que también pueden ser incluidos aquí son la documentación y la cantidad de recursos calificados disponibles en el mercado.
- **Madurez de la comunidad.** La reputación del *framework* en sí, es decir, en términos de soporte que se le da al código y la frecuencia de los lanzamientos para solucionar problemas y agregar nuevas funciones y funcionalidades.
- **Facilidad de desarrollo.** Identificación de la sencillez al desarrollar aplicaciones con este *framework* y la productividad de los proyectos. También se evalúan los ambientes de desarrollo (*Interface Development Environment*-IDE) y las herramientas que admiten el *framework*, debido a que son un papel esencial cuando se habla del desarrollo rápido de aplicaciones.
- **Curva de aprendizaje.** La disponibilidad de la documentación en forma de tutoriales, mejores prácticas y soluciones para problemas típicos, es una característica importante que debe cubrir la curva de aprendizaje para mejorar la productividad general de los desarrolladores involucrados en el proyecto.
- **Compatibilidad de la arquitectura.** Los *frameworks* proporcionan módulos de código e interfaces con patrones de diseño incorporados que eliminan la complejidad de la codificación.
- **Soporte de automatización.** El soporte existente para que el *framework* tenga la capacidad de automatizar las tareas relacionadas con la creación y despliegue de los microservicios.
- **Implementaciones independientes.** El *framework* debe permitir aspectos de la implementación independientes, tales como la compatibilidad ascendente, compatibilidad descendente, reutilización y portabilidad.
- **Integración continua.** Cuando los desarrolladores integran el código en un repositorio compartido, con frecuencia este puede ser accedido por diversos usuarios en varias ocasiones por día. Es por eso que debe existir una integración verificable, es decir, debe existir una compilación automatizada y un marco de pruebas automatizado para verificar y brindar soporte a cualquier cambio en el sistema.

3. Migración de Aplicaciones

Para muchas empresas el desarrollo monolítico es la mejor opción debido a su simplicidad y facilidad de uso, pero esto solo es funcional cuando los proyectos son pequeños y sencillos, tienen muy corta duración o no necesitan actualizaciones o modificaciones frecuentes, sin embargo, con el tiempo, estos proyectos se vuelven muy complicados y desordenados a medida que van escalando, es por eso que la arquitectura de los microservicios viene a resolver este tipo de problemas, y para realizar los cambios necesarios para que los proyectos puedan controlarse, es necesario seguir estrategias o metodologías para la migración de sistemas monolíticos ya existentes, y tratar de verificar que las funcionalidades originales no se pierdan, y se ajusten adecuadamente a los requerimientos originales.

3.1 Estrategia de Migración

Para realizar una correcta migración, es necesario tener una estrategia que permita contemplar las mejoras desde diferentes aspectos, tales como recursos humanos y monetarios. Debido a que la modernización de las aplicaciones que se utilizan y se utilizarán, deben tener un objetivo claro, y no todos los sistemas son recomendables para actualizarse. Para eso es necesario contemplar tres aspectos (Espadas Pech, 2022a), a) un diagnostico tecnológico, b) la estrategia tecnológica y c) la estrategia de implementación.

En el diagnostico tecnológico se analizan los puntos importantes a considerar enfocados a la organización:

- **Visión de negocio.** Aquí se consideran cuáles son los objetivos a corto, mediano y largo plazo, de la institución que se desea modernizar Para algunas empresas los sistemas informáticos son un activo tecnológico y estratégico, los cuales deben ser prioridad para la migración, por otro lado, existen otros sistemas que son *BackOffice*[19] para los cuales se dejan como muy baja prioridad, y en muchos casos no son necesarios actualizar.

[19] Sistemas para uso interno de la institución, por ejemplo: sistemas administrativos o la Intranet.

- **Arquitectura tecnológica actual.** Otro punto importante es revisar la documentación técnica de los sistemas actuales, sus arquitecturas, infraestructura, tecnologías, bases de datos, API's, etc. Esto sirve como una radiografía que permite realizar un diagnóstico e identificar en qué estado se encuentra la institución desde un enfoque tecnológico.
- **Capacidad operativa actual.** De la revisión anterior, se puede identificar la capacidad instalada, tanto técnica como de soporte. Esto permite determinar si para realizar la modernización son suficientes los recursos actuales, o es necesario conseguir más, y en donde se encuentran los puntos débiles de los sistemas actuales.
- **Problemas actuales.** Este punto se puede evaluar a través de los KPI's[20] institucionales para identificar cuales es el nivel de servicio, si existen problemas de disponibilidad en sistemas críticos, el tiempo de espera para atender las solicitudes de negocio (todo desde un enfoque informático).
- **Presupuesto.** Cualquier tipo de cambio que se desee realizar cuesta, es por eso que se necesita evaluar el presupuesto para modernizar las soluciones, así como el tiempo de vida de los sistemas actuales, antes de ser modernizados, y hasta el retorno de inversión que los nuevos sistemas atraerán.

Del análisis anterior, se puede identificar la prioridad para las aplicaciones que deben ser modernizadas. Posteriormente, se identifica la Estrategia Tecnológica (Espadas Pech, 2022b), en la cual se definen las arquitecturas, tecnologías y patrones que se requieren para la migración. Pero antes, es necesario identificar cuáles son las aplicaciones que no deberían ser migradas, por ejemplo:

- Aplicaciones BackOffice que no requieran mantenimiento continuo.
- Sistemas modulares y estables que no tengan problemas de escalabilidad y disponibilidad.
- Aplicaciones que, aunque tengan problemas de escalabilidad, puedan ser solucionados con escalamiento vertical y horizontal.
- Sistemas estables con un proceso bien definido, y que no tengan un tiempo de espera que sobrepase los KPI's.
- Aplicaciones que no se tengan contempladas para crecer a mediano y largo plazo.
- Aplicaciones que sean instaladas y administradas por clientes finales.

[20] Son indicadores de calidad o indicadores clave del negocio que permite evaluar el rendimiento de un proceso

- Aplicaciones nuevas que no tengan problemas de escalamiento.

Y cuales son las aplicaciones que deben ser migradas:

- Sistemas que sean el núcleo del objetivo institucional, que vayan a crecer de manera exponencial y que tengan problemas de escalabilidad.
- Sistemas con tecnologías antiguas monolíticas con tiempos de mantenimiento muy largos y complicados.
- Aplicaciones que dependan de un escalamiento rápido del número de desarrolladores.
- Soluciones segmentadas por API's que no tengan un acoplamiento adecuado y que permitan segmentar a nivel de dominio los datos de cada servicio.

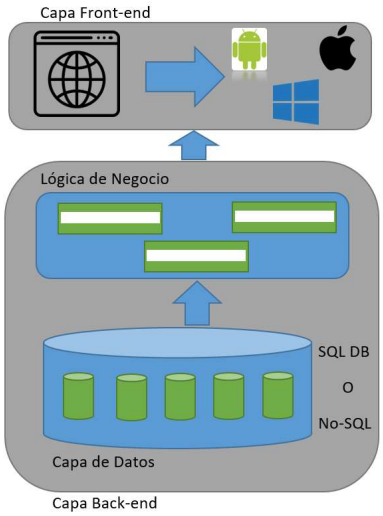
Cuando ya se tenga una lista de los proyectos que se deseen migrar, el siguiente paso es definir la arquitectura, aunque es recomendable reutilizar el conocimiento de las tecnologías anteriores para el desarrollo e implementación de las nuevas soluciones. Una ventaja de la arquitectura de microservicios, es que no se encuentra limitada a utilizar una tecnología en particular, es por eso que pueden coexistir sistemas con diferentes lenguajes de programación, tales como .NET Core, Node.js, Go y Java, sin problemas de comunicación. Además, es recomendable que dichas plataformas estén soportadas por tecnologías de contenedores y que se puedan ejecutar en servicios *serverless*[21].

Una consideración importante para migrar sistemas a microservicios es la arquitectura de los datos. Desde este enfoque, un microservicio debe ser totalmente independiente y desacoplado, esto incluye cualquier dependencia de fuentes de datos entre servicios o componentes. En el caso de las aplicaciones monolíticas, surge el problema de *bottleneck*[22], debido a que todo el sistema trata de acceder a la información de la base de datos a través de diferentes servicios, los cuales apuntan a una sola base de datos de gran tamaño y esta no tiene la capacidad tecnológica para responder a cada petición, como se aprecia en la **¡Error! No se encuentra el origen de la referencia.**. El manejar la información de esta manera tiene ventajas como la alta transaccionalidad y la gestión de toda la base de

[21] Es un modelo de ejecución en el cual existe un proveedor en la nube (AWS, Azure, Google Cloud) que es responsable de ejecutar un fragmento de código mediante la asignación dinámica de recursos, y cobrando solo por los recursos utilizados. Generalmente se ejecutan dentro de contenedores sin estado que se activan de acuerdo a un conjunto de eventos que incluyen HTTP, eventos de bases de datos, servicios de colas, alertas de monitoreo, carga de archivos, eventos programados, entre otros (Anomaly Innovations, 2022).

[22] El problema de *bottleneck* surge cuando la capacidad de una aplicación o un sistema está limitada por un solo recurso y esto provoca la ralentización del flujo en general.

datos en un solo lugar, sin embargo, esto conlleva un mantenimiento lento y complicado, interdependencia entre servicios, despliegue lento de nuevas funcionalidades, problemas generados por errores en servicios que afectan a todo el sistema, entre otros. Estas desventajas se resuelven al manejar la arquitectura de microservicios.



Fuente: Elaboración propia con base en (Espadas Pech, 2022a)

Figura 3.1 Aplicación monolítica multicapa

Otras consideraciones importantes a la hora de implementar la arquitectura de datos son:

- API *Gateway*. Un servicio que permite la administración de la creación, publicación, mantenimiento, monitoreo y protección de las API's a cualquier escala (Amazon, 2022a; Google, 2022a). Actúa como la puerta de entrada para las aplicaciones que acceden a los datos o a la funcionalidad de los servicios de backend.
- *Command and Query Responsability Segregation* (CQRS). Es un patrón que separa las operaciones de escritura y lectura del almacenamiento de datos (Martin Fowler, 2022). Es utilizado para maximizar la escalabilidad, rendimiento y seguridad.
- *Cold Data*. Se refiere a la información que no se accesa con frecuencia. Se utiliza para optimizar los costos de almacenamiento a través del guardado de esta información en sistemas con menos rendimiento (Komprise, 2022).
- Comunicación asíncrona. La independencia de la información entre microservicios es la clave de la arquitectura, y para esto, es necesario contar con un sistema de comunicación asíncrono, a través de colas o mensajes (gRPC y AMQP) (gRPC Authors, 2022; OASIS, 2020).

Finalmente, en la Estrategia de Implementación se definen los elementos necesarios para ir evaluando los cambios que se van realizando en la migración para evitar afectar la operatividad del negocio (Espadas Pech, 2022c). Dentro de estos podemos encontrar los patrones de migración y descomposición, los cuales establecen prácticas para la factorización de los proyectos monolíticos, su objetivo es minimizar el riesgo y aumentar el porcentaje de éxito, con la facilidad, de realizar un *rollback*[23] en cualquier momento. Existe una gran cantidad de patrones que pueden ser aplicados para la migración de los sistemas monolíticos a una arquitectura de microservicios, mismos que serán explicados más adelante.

Por otro lado, para tener una migración exitosa, se debe seguir la metodología de desarrollo correcta, algunos ejemplos claros que podemos encontrar son las metodologías agiles (que se enfocan en el alto desempeño de los recursos humanos) y los pasos a seguir para el desarrollo de un solo dominio de negocio. Es de bastante importancia contemplar esto, debido a que los procesos son la parte medular de los proyectos. El tener mecanismos de administración de riesgos, aseguramiento de la calidad, planes de trabajo agiles y *checkpoints* para evaluar los indicadores clave de la migración, aumentan las posibilidades de tener una migración exitosa. También el uso de procesos y herramientas de *DevOps* permiten la integración y liberación continua de los productos o sistemas de acuerdo a las funcionalidades que se vayan desarrollando, entre otras palabras, *DevOps* se enfoca en entregar el mayor valor en el menor tiempo posible.

3.2 Patrones de migración

Existe un gran número de pautas o patrones de migración para sistemas monolíticos a microservicios, cada uno tiene sus características, ventajas y desventajas para su implementación, es por eso que se describirán los más importantes y más implementados en la industria (Newman, 2015, 2019; Richardson, 2018).

3.2.1 Patrón *Big Bang*

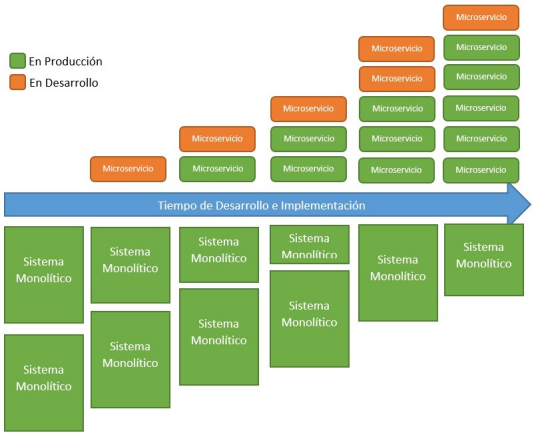
Este patrón conlleva la migración de todos los sistemas monolíticos a la vez, es decir, una reestructuración y reescritura de código desde cero, cubriendo todas las funcionalidades de los sistemas actuales. A pesar que tiene la ventaja de que se pueden rehacer y arreglar todos los defectos de los sistemas actuales (ya que son proyectos nuevos e independientes), el

[23] Es una operación que devuelve un sistema a un estado anterior, un punto de guardado antes de cualquier cambio.

gran problema radica en que supone la inversión de demasiado tiempo para el desarrollo del código nuevo, además de otros recursos necesarios. Debido a que son completamente independientes los sistemas, si existe algún cambio en el sistema actual, se tendrá que reescribir en los microservicios, y esto conlleva más tiempo de codificación, o en caso extremo, la detención del desarrollo de la nueva arquitectura.

3.2.2 Strangler Fig

Este patrón se enfoca en ir migrando el sistema monolítico en aquellos componentes con muy poca dependencia con los demás, la idea es ir desintegrando el sistema actual hasta su completa desaparición. El usar este patrón minimiza el riesgo de migrar todo el sistema y liberar todo de un momento a otro, permite repartir proporcionalmente el esfuerzo de desarrollo durante el tiempo de migración. El identificar los componentes más propensos a migrar es una de las actividades principales de este patrón, se debe realizar un análisis exhaustivo de la arquitectura monolítica, identificando componentes y separarlos por funcionalidades o por dominio. El modelar cada componente puede ayudar a especificar las dependencias y a mejorar los sub-proyectos, librerías o clases que se puedan encapsular en nuevos microservicios.



Fuente: Elaboración propia con base en (Espadas Pech, 2022c)

Figura 3.2 Patrón Strangler Fig

3.2.3 Parallel Run

Es un enfoque muy similar a *Strangler Fig*, en el cual se desarrollan evolutivamente los procesos a la par de que el sistema sigue funcionando. En el análisis se deben identificar los componentes que puedan separarse para implementarse y ejecutarse en paralelo con el sistema actual, y estos nuevos componentes cubren la funcionalidad completa de lo que sustituyen, sin dejar de comunicarse con el sistema a actualizar. Para este

patrón, se comienza la migración con componentes transversales de las aplicaciones, tales como inicio de sesiones, monitoreo, seguridad, API's monolíticas, integraciones de servicios a terceros, entre otros. Para verificar que la información que se esté trabajando en el microservicio y el sistema actual se pueden almacenar las respuestas de ambos sistemas y contrastarlas con un proceso *batch*, con el objetivo de eliminar errores o desperfectos identificados en la información del microservicio.

3.2.4 Branch By Abstraction

Este patrón permite sacar funcionalidades cuando el sistema se encuentra en producción, es perfecto para aquellos casos en los que la funcionalidad depende de otras funcionalidades o módulos. La idea es generar una capa de abstracción sobre la funcionalidad que se desea implementar, para eso se necesitan un conjunto de etapas:

- **Crear la capa de abstracción.** Se desarrolla una capa sobre el módulo que se va a sustituir.
- **Refactorización.** Se modifica el código para que los módulos alternos se comuniquen con esta nueva capa en lugar de la funcionalidad original.
- **Creación del microservicio.** Ya que se delimito claramente la funcionalidad a reemplazar, se crea el microservicio cubriendo las necesidades.
- **Migración.** Con el microservicio existente y validado de las operaciones de la cual es encargado, se conecta y sustituye el microservicio para responder a las peticiones de la funcionalidad deseada.

3.2.5 Patrones para Bases de Datos Compartidas

Dentro de los patrones de migración, también se debe considerar como se trabajará con la información, es decir, como se manejarán las bases de datos y que controles se llevarán para adaptarlos a los microservicios. Para este objetivo, existen algunos patrones a seguir, que agilizan y facilitan esta tarea.

3.2.5.1 Database View

Cuando se comienza la actualización entre tecnologías de un sistema monolítico a uno enfocado a microservicios, en muchos casos se desea continuar con la misma base de datos original antes de hacer la migración completa, compartiendo la información entre el sistema actual y el sistema en desarrollo. Por ello, existe este patrón, en el cual se generan vistas como servicios para acceder a los datos como si fuera un esquema.

Estas vistas permiten visualizar únicamente la información que pertenece a un servicio. Sirve como una aproximación para ir incorporando los microservicios con información real, pero aislando la base de datos original.

A través de vistas se puede controlar el acceso a los datos, ocultando la información que no se requiera para la funcionalidad que se esté desarrollando. Sin embargo, tiene algunas limitaciones:

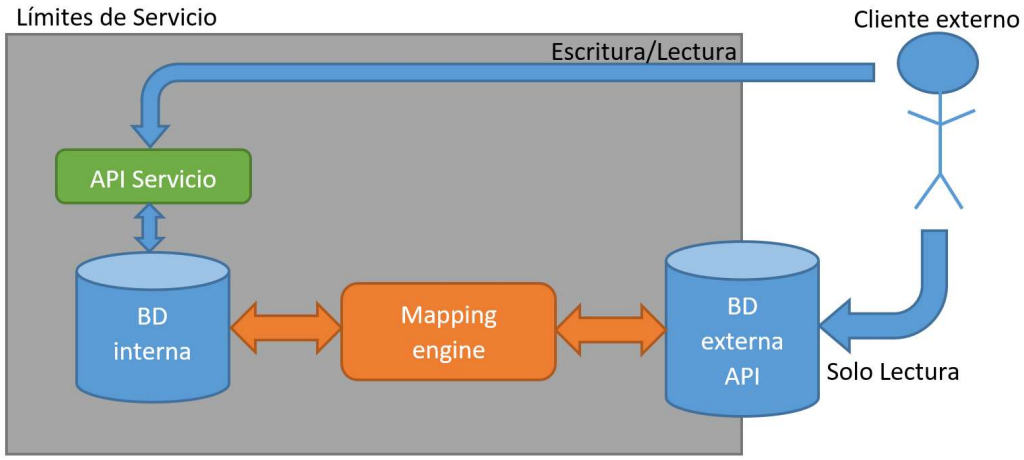
- Una vista es de solo lectura, lo que limita su utilidad,
- No es compatible con la mayoría de las bases de datos NoSQL,
- El microservicio sigue siendo dependiente de la base de datos,
- El escalado se tiene que realizar a toda la base de datos,
- Existe un único punto de fallo, es decir, si falla la base de datos, falla tanto el sistema monolítico como los microservicios.

3.2.5.2 Database Wrapping Service

Se utiliza cuando es complicado crear vistas de la información. Este patrón nos permite la creación de un servicio que envuelve la base de datos y nos devuelve la información específica que se necesita. Funciona como un servicio intermedio encargado de administrar la información de la base de datos, a través de API's específicas para los esquemas existentes, además, proporciona el tiempo suficiente para reajustar o dividir la base de datos en más esquemas o más bases de datos.

3.2.5.3 Database as a Service Interface

Hay ocasiones en las cuales nuestra arquitectura únicamente tiene una base de datos para realizar consultas, en estos casos, es necesario separar la base de datos que exponemos con la que tiene toda la información, es decir, se crea una base de datos dedicada para solo lectura de la información relevante y que pueda ser accedida fuera de los límites del servicio a través de una API dedicada. Para el proceso de actualización de la base de datos de solo lectura con la base de datos interna, se deberá desarrollar una función destinada a esta tarea llamada *mapping engine*.



Fuente: Elaboración propia con base en (Espadas Pech, 2022c)

Figura 3.3 Patrón Database as a Service Interface

Como ventajas de este patrón se puede identificar mayor flexibilidad a comparación del uso de vistas, además, es recomendable para casos que solo se necesite leer información. Por otro lado, se aíslan las lecturas de las escrituras, y esto provoca mayor complejidad en las bases de datos compartidas y un mayor esfuerzo de inversión para implementar este patrón.

3.2.5.4 Trace Writer

Finalmente, uno de los patrones más utilizados, es *Trace Writer*, el cual define una convivencia entre dos bases de datos, lo que origina que estará trabajando con el sistema actual, y una copia de este que será modificada de manera incremental. Esta última puede ser representada por una o más bases de datos dependiendo de cómo se hayan definido los microservicios. La idea es liberar versiones estables de la base de datos de los microservicios de manera secuencial, de acuerdo a las funcionalidades que se vayan separando del sistema original, dichas versiones deben estar validadas para evitar problemas de inconsistencia, y poco a poco ir utilizando las nuevas bases de datos hasta el punto de eliminar la conexión con la original. Algunas desventajas de este patrón, es la duplicación de información, lo cual implica mayor cantidad de recursos para el mantenimiento de los datos.

3.3 Recomendaciones y buenas prácticas

Cuando se cambia el diseño del software, se modifica la arquitectura completa del mismo, esto incluyen el patrón de trabajo de la empresa. Para realizar una migración exitosa es necesario considerar los cambios desde dos enfoques, un enfoque organizacional y un enfoque técnico, y

para cada uno de estos es necesario tomar en cuenta lo siguiente (Espadas Pech, 2022c):

- **Construir metas y objetivos.** Es imprescindible establecer los objetivos específicos, explícitos y comprensibles, en los cuales se considere como afectaran los cambios en el trabajo de equipo actual, y si es necesario crear nuevos flujos de trabajo para igualar las actividades. La planificación debe corresponder a que todos los miembros estén de acuerdo. También debe considerarse un sistema de respaldo para realizar un *rollback* en cualquier momento.
- **Supervisar y medir los KPI's constantemente.** El uso de los indicadores clave de rendimiento para gestionar los microservicios que se van desarrollando, garantizará una transferencia exitosa entre los diferentes tipos de arquitectura. Calcular como aumenta la frecuencia de la implementación, y anotar cualquier desperfecto o cambio no considerado en la planeación del proyecto, servirá para agilizar el proceso de implementación y su resiliencia.
- **Empezar con pocos módulos.** Los microservicios incluyen una gama amplia de bases de código y módulos, el intentar conseguir y perfeccionar todo a la vez, es algo muy complicado de llevar a cabo e incontrolable, al medir el rendimiento del nuevo sistema. Es por eso, que se necesita controlar el proceso de codificación, y enfocar el desarrollo en los factores críticos que afecten a todo el sistema monolítico.
- **Establecer equipos multifuncionales.** Identificar y definir un equipo capaz de desarrollar, diseñar y construir operaciones sin necesidad de asistencia, agilizará la implementación.
- **Olvidar la estructura monolítica original.** Las bases de código de cada microservicio deben ser independientes, es decir, no deben depender de las funcionalidades originales del sistema, aunque el sistema monolítico se encuentre activo y mientras el desarrollo de la nueva arquitectura este en proceso, es necesario dejar a un lado la arquitectura del sistema original y cómo funciona, y enfocarse en las funcionalidades independientes que se van desarrollando.
- **Agilizar la construcción.** Entender de manera eficiente como trabaja internamente el sistema es una clave principal para una migración exitosa, al identificar y eliminar todas las funciones no imprescindibles o innecesarias se acorta el tiempo de codificación. Además, el entendimiento agiliza las estrategias para eliminar las dependencias entre diferentes módulos del sistema monolítico.
- **Uso de herramientas.** Es necesario utilizar software especializado de desarrollo de aplicaciones y monitorización, para identificar, verificar y validar las comunicaciones existentes entre microservicios

y bases de datos. Estas herramientas suelen ofrecer métricas para comparar el rendimiento de los diferentes módulos.

Observación. La comprobación de las tasas de error y sus respectivas causas, es una de las tareas más importantes a través de herramientas que puedan recolectar toda la información del uso de los microservicios. las peticiones que estos reciben y los resultados que estos responden, ayuda a identificar problemas de comunicación para su posterior resolución. También es necesario revisar cómo funciona internamente cada microservicio para validar que las actividades para las que fueron creados están siendo aplicadas de manera correcta.

4. Arquitectura de Microservicios

La arquitectura de microservicios de esta investigación, fue desarrollada en Spring Boot, basada en el *framework* Spring, del cual se tomaron varios módulos para separar de manera correcta cada uno de los servicios. Además, la comunicación al exterior de cada microservicio se basó en el desarrollo de API's RESTful.

En este capítulo se explica de manera detallada la parte teórica de la codificación realizada de la arquitectura, y los componentes finales que se desarrollaron para el sistema final. Aunque no se incluye la codificación, se explica a detalle una abstracción de la infraestructura final, y se anexa un manual que explica cómo está conformada la API de un proyecto del Laboratorio y como puede ser utilizada. Dentro de este manual se incluyen las referencias correspondientes para solicitar acceso al código implementado y subido a GitHub[24].

4.1 Spring *framework*

Antes de explicar y adentrarse a la arquitectura de microservicios propuesta en esta investigación, es necesario conocer la infraestructura de Spring, ya que es la base con la que está desarrollado el *framework* de Spring Boot, el cual fue utilizado para la migración de los sistemas y proyectos del Laboratorio.

El *framework* de Spring (SpringSource, 2017) es una plataforma desarrollada en Java *OpenSource*[25] que provee toda la infraestructura de soporte para el desarrollo de aplicaciones Java, desde aplicaciones integradas, aplicaciones embebidas, hasta aplicaciones empresariales del lado servidor con n niveles de complejidad. Spring codifica patrones de diseño formales como objetos de primera clase para que puedan ser integrados en la elaboración de nuevas aplicaciones (Johnson & Foote,

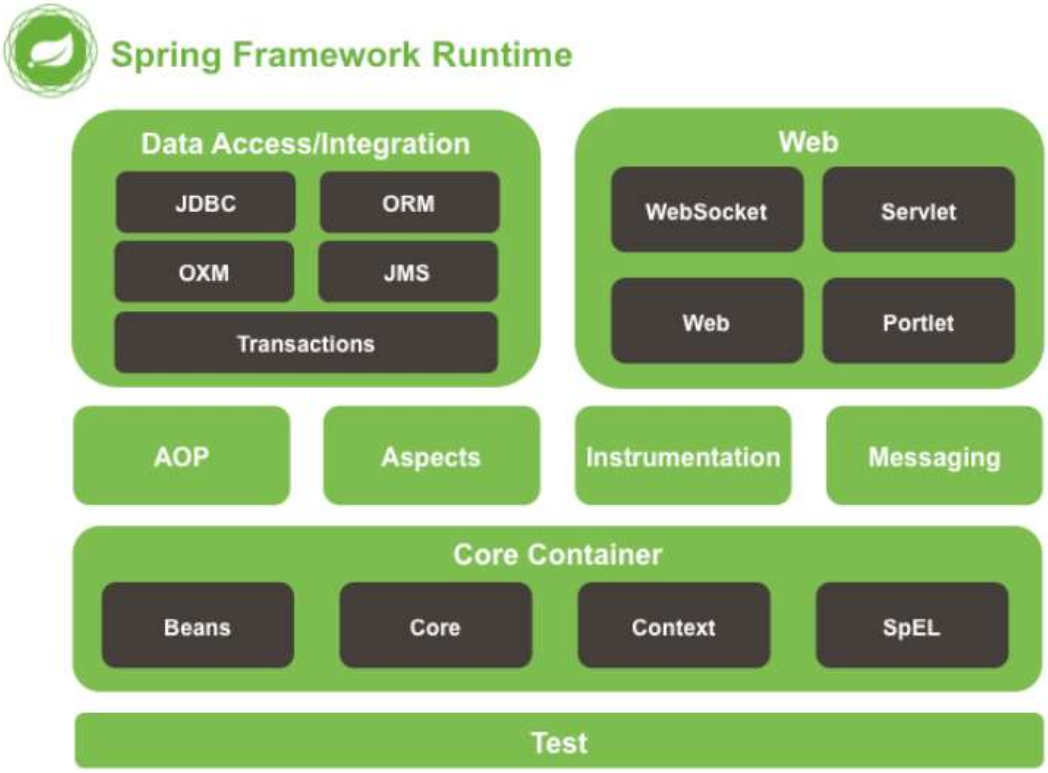
[24] El dominio utilizado para acceder a la API o al código de GitHub puede ser cambiado en cualquier momento debido a actualización de nuevas versiones, se recomienda contactar a los desarrolladores para cualquier aclaración.

[25] Software de código abierto cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de código abierto o forman parte del dominio público.

1988; Mattsson, 1999). Cuenta con dos características (patrones) esenciales que lo diferencian de los demás*:*

- Inversión de Control (*Inversion of Control*-IoC). Es un principio de diseño de software donde el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales. En los métodos tradicionales la interacción se expresa de forma imperativa haciendo llamadas a procedimientos o funciones. En IoC se especifican respuestas deseadas a solicitudes de datos concretos, dejando el control a la arquitectura para llevar las acciones necesarias, en el orden deseado y para el conjunto de eventos que vayan a ocurrir. IoC puede ser implementado a través de eventos o Inyección de Dependencias (Fowler, 2004).
- Inyección de Dependencias (*Dependency Injection*-DI). Es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos. Esos objetos cumplen contratos que necesitan las clases para poder funcionar, estos objetos son suministrados a través de una clase contenedora que inyectará la implementación directamente al contrato. En otras palabras, se encarga de extraer la responsabilidad de creación de instancias de un componente para delegarla a otro (CDI Repository Contributors, 2022).

Spring consiste en funciones organizadas en cerca de 20 módulos, de los cuales son agrupados de acuerdo a sus funcionalidades en Contenedor Central, Acceso e Integración de Datos, Web, Programación Orientada a Aspectos (*Aspect Oriented Programming*-AOP), Instrumentación, Mensajería, y Pruebas, tal como se muestra en la Figura 4.1.



Fuente: Obtenido de Overview of *Spring Framework* (SpringSource, 2017)

Figura 4.1 Módulos agrupados de Spring

4.1.1 Contenedor Central

Este módulo provee la funcionalidad esencial del *framework*, está compuesto de los siguientes módulos: **spring-core**, **spring-beans**, **spring-context**, **spring-context-support** y **spring-expression**.

- **Spring-Core/Spring-Beans**. Estos módulos proveen las partes fundamentales, las cuales incluyen IoC y DI. Su núcleo principal es el uso de *BeanFactory*, componente que define la implementación del patrón *Factory* (Gamma et al., 1994), y que se encarga de manejar las relaciones de los objetos a través de *Beans*. Se pueden soportar dos tipos de objetos.
    - *Singleton*. Existe únicamente una instancia compartida de un objeto con un nombre en particular, que puede ser llamado cada vez que se necesite. Es el más utilizado y se basa en un patrón del mismo nombre.
    - *Prototype*. Este método define como se crea un objeto independiente cada vez que se realiza una llamada, también es conocido como *non-singleton.*

- **Spring-Context/Spring-Context-Support**. El primer módulo se encarga de definir el cómo se accesarán a los objetos dentro del framework, el cual es muy similar al registro JNDI[26](*Java Naming and Directory Interface*) (Oracle, 2022). Este módulo hereda las características del módulo de *Beans* y soporta la internacionalización, propagación de eventos, carga de recursos, y una creación transparente de contextos para contenedores de *servlets*. También proporciona soporte para EJB (*Enterprise Java Bean*), JMX (*Java Management Extensions*), y control remoto básico. **Spring-Context-Support** es una integración del primer módulo, pero este permite la integración de bibliotecas de terceros, particularmente para1 administración de caché y planificación.
- **Spring-Expression**. Provee un lenguaje de expresión para la consulta y manipulado de un gráfico de objeto en tiempo de ejecución. Se basa en el lenguaje de expresión unificado (Chung et al., 2006). Soporta el establecimiento y obtención de valores de propiedades, asignación de propiedades, invocación de métodos, acceso al contenido de matrices, colecciones e indexadores, operadores lógicos y aritméticos, variables con nombre, recuperación de objetos a través de su nombre definido por IoC, proyección y selección de listas, así como agregación a listas comunes.

4.1.2 AOP e Instrumentación

El módulo **spring-aop** provee la implementación de la programación orientada a aspectos (Clarke, 2009), lo que permite definir interceptores de métodos y puntos de corte para desacoplar el código que implementa diferentes funcionalidades. Las actividades principales de las que se encarga este módulo son: persistencia, manejo de transacciones, seguridad, *logging*, *debugging*. Funciona tanto para servidores web como contenedores EJB.

También se puede identificar el módulo **spring-aspects** que provee la integración con AspectJ[27] (Eclipse Foundation, 2021). Por otro lado, **spring-instrument** es un módulo que facilita la implementación con ciertos servidores de aplicaciones a través de utilidades para la carga de clases,

[26] Interfaz de Nombrado y Directorio Java es una API que permite descubrir y buscar objetos a través de nombre normalizado.

[27] Lenguaje de programación orientada a aspectos construido como una extensión del lenguaje Java creado por Xerox PARC.

por ejemplo, el utilizado como agente para Tomcat[28] es **spring-instrument-tomcat**.

4.1.3 Integración/Acceso a Datos

Es la capa o grupo que se encarga de gestionar la comunicación con la información de las bases de datos y manipular de manera correcta el mapeo de la información relacional o no relacional con los objetos respectivos que los representan. Consiste en la conectividad con JDBC (*Java Database Connectivity*) (IBM, 2021), ORM (*Object-Relational Mapping*) (RedHat, 2022a), OXM (*Object/XML or O/X Mapping*) (The Eclipse Foundation, 2017), JMS (*Java Message Service*) (H. Mahmoud, 2004) y las operaciones de transacción respectivas. Algunas de las operaciones que se pueden realizar a través de los módulos de este grupo son: el manejo de sesiones, manejo de recursos, gestión de transacciones integradas, tratamiento de excepciones, facilidad de migración entre ORM's, y la facilidad de pruebas aisladas en diferentes fuentes de datos.

- **Spring-JDBC**. Módulo encargado de proporcionar una capa de abstracción para evitar la necesidad de realizar codificación de bases de datos y el análisis de los errores específicos del DBMS (*DataBase Management System*).
- **Spring-TX**. Admite la gestión de transacciones programáticas y declarativas para clases que implementan interfaces especiales y sus respectivos POJO's (*Plain Old Java Objects*).
- **Spring-ORM**. Integra una capa para el mapeo objeto-relacional de las API's con algunos *frameworks* existentes tales como JPA y *Hibernate*. Proporciona la configuración de estos *frameworks* con otras herramientas de Spring, además de una gestión sencilla para transacciones declarativas.
- **Spring-OXM**. Permite configurar como una capa abstracta que realice operaciones con implementaciones de otros *frameworks* como JAXB, Castor, JiBX, y XStream, los cuales se enfocan en el mapeo de información XML a objetos.
- **Spring-JMS**. Este módulo se encarga de administrar las características principales para la producción y consumo de mensajes a través de la integración con otro módulo llamado

[28] Tomcat, también llamado Apache Tomcat o Jakarta Tomcat es un contenedor open source de servlets para la implementación de Java Servlet, JavaServer Pages (JSP), Java Expression Language y Java WebSocket.

spring-messaging, el cual es un gestor de sistemas de mensajería asíncrona.

4.1.4 Mensajería

Este módulo fue incorporado en la versión 4 de Spring, **spring-messaging**, que se encarga de las abstracciones de los sistemas de mensajería (tal como se mencionó anteriormente). Incluye algunos proyectos como *Message*, *MessageChannel*, *MessageHandler*, que sirven como base para estos sistemas con un conjunto de anotaciones para mapear mensajes a métodos.

4.1.5 Web

El módulo **spring-web** se encuentra en la parte superior del grupo de contexto, y se encarga de gestionar e integrar las aplicaciones web y el framework de *Struts* del proyecto Yakarta (The Apache Software Foundation, 2022a). Algunas de las actividades que realiza son: la gestión de las peticiones *multipart* que pueden ocurrir al realizar cargas de archivos, y la relación de los parámetros de estas peticiones con los objetos correspondientes, también llamados objetos de dominio y objetos de negocio. También contiene un cliente HTTP y las partes relacionadas del sistema web para soporte remoto.

Por otro lado, el módulo **spring-webmvc**, también conocido como *web-servlet*, contiene la configuración necesaria para el desarrollo de aplicaciones web con una estructura MVC (*Model-View-Controller*) (Microsoft, 2020) o Servicios Web REST. Define una separación clara entre el dominio, los modelos y los formularios web.

4.1.6 Pruebas

El módulo **spring-test** se encarga de proporcionar las funcionalidades de Pruebas Integradas y Pruebas Unitarias (Pittet, 2022) a las aplicaciones desarrolladas en Spring. Provee una carga consistente a través de *AplicationContext* y caché de los componentes integrados en la aplicación. También proporciona objetos *Mock* (Meszaros, 2008) que pueden utilizarse para pruebas de código de manera aislada.

4.1.7 Spring Boot

Spring Boot es un proyecto para simplificar y optimizar el proceso de configuración inicial y preparación de las aplicaciones para producción desarrolladas con Spring (Pahino, 2020). Este *framework* utiliza dos mecanismos principales para realizar esta tarea:

- **Contenedor de Aplicaciones Integrado.** Permite compilar aplicaciones Web como archivos ***.jar*** las cuales pueden ser ejecutadas como aplicaciones normales de Java, siendo una alternativa a los archivos ***.war*** que son aquellas que se despliegan en los servidores de aplicaciones tales como Tomcat (The Apache Software Foundation, 2022b). Este mecanismo incorpora un servidor de aplicaciones propio que se levanta automáticamente al ejecutar la aplicación, ahorrando el tiempo de configuración de un servidor externo a la aplicación; esta característica es muy útil en la arquitectura de microservicios ya que permite distribuir las aplicaciones como imágenes para contenedores que se pueden escalar horizontalmente.
- ***Starters*.** Spring Boot proporciona un conjunto de dependencias llamadas *Starters* que pueden ser añadidas a cualquier proyecto de manera sencilla considerando las necesidades de la aplicación. Este mecanismo tiene configuraciones por defecto que minimizan la necesidad de configuración al desarrollar nuevos sistemas. Al igual que el mecanismo anterior, no impone las dependencias y configuraciones, es decir, se pueden realizar cambios de acuerdo a las necesidades, desde el puerto en que la aplicación escuchará las peticiones, hasta la información que se mostrará al arrancar inicialmente la aplicación.

Por la facilidad de desarrollo de nuevos sistemas que ofrece Spring Boot, y la compatibilidad que tiene para la implementación de microservicios, es la razón principal por la cual se seleccionó este *framework* para la migración de los proyectos del Laboratorio Nacional en Sistemas de Transporte y Logística.

4.2 RESTful API's

Para el desarrollo del sistema en esta investigación se utilizó la comunicación entre microservicios a través de API's RESTful. Como primera fase se decidió este método de comunicación, debido a que no existe una separación clara entre las funcionalidades de cada proyecto del Laboratorio, y antes de implementar comunicación asíncrona, tal como se mencionó en el Capítulo 2.1, primero fue necesario enfocarse en la lógica de negocio de todo el sistema y en el desarrollo de los microservicios.

La selección del estándar RESTful API (también conocido como REST API) fue debido a la facilidad de implementación, y a que es uno de los protocolos más utilizados y populares entre los desarrolladores, se basa en el uso de comandos HTTP para realizar la mayoría de sus operaciones, sus características principales son: la definición del modelo REST para la

definición de las API's, utiliza encriptación SSL[29](*Secure Sockets Layer*)(The OpenSSL Project Authors, 2021), el lenguaje de comunicación es independiente al lenguaje de desarrollo de los microservicios, este tipo de API's permiten crear una aplicación web con operaciones CRUD (*Create, Retrieve, Update, Delete*).

Para entender mejor como se trabaja con este tipo de API's es necesario conocer tres términos importantes:

- **Recursos.** Son objetos que representan algo en el sistema, los cuales en la mayoría de los casos están relacionados con información de la base de datos, los cuales pueden ser manipulados a través de métodos que operan con estos.
- **Colecciones.** Representan un conjunto de objetos, por ejemplo, si en la base de datos se maneja información de diferentes personas, el objeto "Usuario" representaría la información de una sola persona, y la colección sería "Usuarios" representando un conjunto de personas.
- **URL (*Uniform Resource Locator*).** Representa una ruta por el cual se puede ubicar a los recursos, además, también pueden indicar las operaciones a realizar con dichos recursos.

El formato de las API's desarrolladas sigue el mismo patrón y formato de las URL (Internet Engineering Task Force [IETF], 2022b):

Tabla 4.1 Formato de las API's desarrolladas

Donde: *esquema://host.dominio:puerto/directorio/recurso*		
esquema	Es el protocolo utilizado para la solicitud y recepción de datos entre los microservicios	HTTP HTTPS
host	Es donde se alberga el recurso a solicitar	WWW
dominio	El nombre del lugar donde se encuentra el contenido	Dominio-IMT
puerto	Número de puerto que será usado como punto de conexión	8001, 8002, 9000,...
directorio	Define que en parte del dominio se encuentra el material solicitado	/Proyecto/Subdirectorio
recurso	Identifica el recurso al que se desea acceder	/Recurso

[29] La Capa de Sockets Seguros es un protocolo para navegadores web y servidores que permite la autenticación y encriptación de datos enviados a través de Internet.

4.2.1 API *Endpoints*

Los *endpoints* son los puntos de acceso o de comunicación con las API's de cada uno de los microservicios. Cada endpoint sirve como un canal de comunicación y define la ubicación desde la cual la API puede acceder a los recursos solicitados o realizar las funciones necesarias. Los *endpoints* desarrollados para los microservicios de esta investigación están conformados por un conjunto de URL's que definen los recursos y operaciones que se pueden realizar en cada servicio. Las operaciones implementadas en el sistema se basaron en los comandos básicos utilizados para HTTP y HTTPS: PUT, POST, DELETE, GET y PATCH. Aunque para la arquitectura de los proyectos solo fueron utilizados GET y POST; para este último se realizaron algunas modificaciones para que pudieran realizar las operaciones de Inserción, Actualización y Borrado a la vez.

Existen diferentes tipos de parámetros (Internet Engineering Task Force [IETF], 2014) que se pueden enviar a los endpoints: ***Path***, ***Header-Form***, ***Body*** y ***Query***.

- Los parámetros ***Path*** se encuentran en el URI tal como si fuera una dirección web, y en la mayoría de los casos son utilizados para reducir el alcance de la petición a un recurso individual, o para definir una operación especifica dentro del servicio. Un ejemplo claro se puede observar como ***.../{a}/{b}*** donde ***a*** define el nombre del recurso que se desea acceder y ***b*** es el parámetro que se envía en la petición, tal como el nombre de un usuario o un identificador en la información.
- El parámetro ***Body*** indica cuando se crea un conjunto de datos dentro de la petición, pero no se muestra directamente en la URI, también se le denomina como carga útil. Esta información debe tener un formato especifico tal como JSON o XML, aunque existen algunas API's que también manejan YAML, información binaria o hexadecimal. Para el caso de esta investigación se utilizó el formato JSON.
- Los parámetros ***Query*** son un caso especial de parámetros que permiten cambiar el alcance de las peticiones realizadas a un subconjunto del total de la información solicitada. Por ejemplo, al solicitar la información de usuarios en el sistema y limitar la respuesta a usuarios con un nombre o apellido en específico. Estos parámetros se encuentran al final de la URI y se identifican como todas las variables que siguen después de un símbolo ***?***, en el siguiente caso ***.../{a}/{b}?c=x*** la ***c*** indica el nombre del parámetro y la ***x*** indica el valor de dicho parámetro. Estos parámetros siempre se identificarán con un formato de ***clave-valor.***

- Los parámetros ***Header-Form*** son aquellos datos que describen la carga útil de la petición, el tipo de contenido que se manejará en la solicitud, el lenguaje, información de autenticación y autorización, entre otros datos. También son definidos como ***clave-valor***, pero estos no pueden observarse directamente en la URI de la solicitud.

En la Tabla 4. se puede observar la estructura base de las peticiones a la arquitectura de microservicios desarrollada. Debido a que son varios servicios por proyecto, se generalizan las operaciones y no se explica a detalle ninguno de estos.

Tabla 4.2 Operaciones de los Endpoints del Sistema

Operación	Método	URI	Parámetros	Resultado	
Consulta	***GET***	.../{recurso}	*Query*	Colección del tipo de Recurso solicitado	JSON
		.../{recurso}/{id}	*Query*	Recurso con el identificador solicitado	JSON
Creación	***POST***	.../{recurso}	*Body*	Creación del recurso con la información enviada en el Body	Código 201
Actualización		.../{recurso}/{id}	*Body*	Cuando el recurso exista, sobreescribirá la información enviada en el Body	Código 200
Eliminación		.../{recurso}/{id}	*Path*	Eliminación del recurso	Código 200

Fuente: Elaboración propia con base en el estándar HTTP Semantics and Content (Internet Engineering Task Force [IETF], 2014)

4.2.2 Códigos de Estado HTTP

Para el desarrollo de las API's de cada microservicio, es necesario seguir algunas normativas dependiendo del estado de respuesta de las peticiones. En este sentido, el Grupo de Trabajo de Ingeniería de Internet (*Internet Engineering Task Force* IETF)(Internet Engineering Task Force [IETF], 2022a), organismo internacional encargado de desarrollar los estándares de Internet, definieron un conjunto de códigos que explican la respuesta que los sistemas web deberían devolver en caso de ciertas situaciones. Estas se pueden observar en la Tabla 4.:

Tabla 4.3 Clasificación Códigos de Estado peticiones HTTP

Código del Estado	Tipo	Explicación
1xx	Respuesta Informativa	Indican que la solicitud fue recibida y se encuentra un proceso en curso.
2xx	Respuesta Satisfactoria	Indican que la acción fue recibida, entendida y aceptada con éxito.
3xx	Redirecciones	Indica que la solicitud no fue realizada con éxito y deben tomarse medidas adicionales para completar dicha solicitud (cambios en la URI).
4xx	Errores Cliente	La solicitud contiene errores de sintaxis o no se pueden realizar las operaciones correspondientes, estos errores son provocados de lado del usuario que realiza la petición.
5xx	Errores Servidor	El servidor no puede responder correctamente con la solicitud, aunque esta sea válida, estos errores son provocados en el servidor que responde las peticiones.

Fuente: Elaboración propia con base en el estándar HTTP Status Code Registry(Internet Engineering Task Force [IETF], 2022a)

Dentro de cada clasificación existe un gran número de códigos, pero inicialmente para esta investigación, solo fueron implementados algunos de estos, y así, como se vayan desarrollando nuevas versiones del sistema, se irán implementando los códigos restantes. Los códigos seleccionados a partir del estándar "*RFC9110 HTTP Semantics*" fueron:

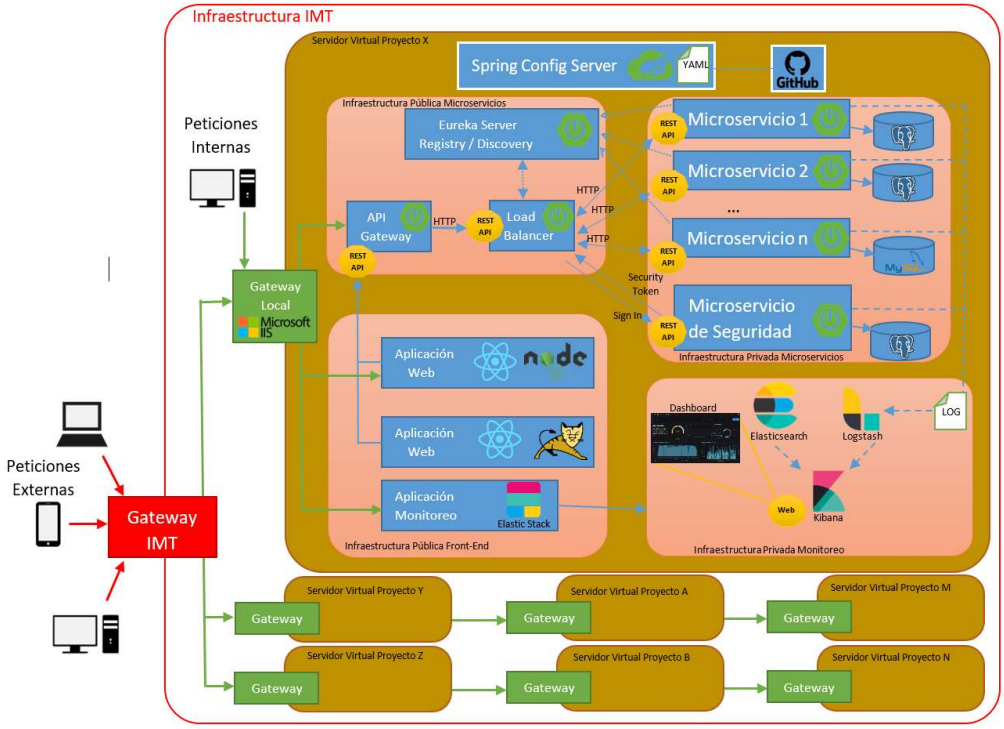
Tabla 4.4 Códigos de Estado utilizados

Código del Estado	Descripción	Explicación	Referencia
200	*Ok*	La solicitud se realizó con éxito.	Sección 15.3.1
201	*Created*	La solicitud se realizó con éxito y se ha creado un nuevo recurso.	Sección 15.3.2
400	*Bad Request*	El servidor no puede interpretar la solicitud.	Sección 15.5.1
401	*Unauthorized*	Es necesaria la autenticación para obtener la respuesta de la solicitud.	Sección 15.5.2
403	*Forbidden*	El cliente no posee los permisos necesarios para el contenido de esta petición.	Sección 15.5.4
404	*Not Found*	El servidor no puede encontrar el recurso solicitado.	Sección 15.5.5
500	*Internal Server Error*	El servidor ha encontrado un error que no puede solucionar.	Sección 15.6.1
503	*Service Unavailable*	El servidor no está listo para realizar dicha petición, en la mayoría de los casos es porque el servidor está caído o se encuentra en mantenimiento.	Sección 15.6.4

Fuente: Elaboración propia con base en el estándar RFC 9110 *HTTP Semantics* (Fielding et al., 2022)

4.3 Arquitectura Propuesta

La arquitectura final propuesta y desarrollada en esta investigación fue elaborada a través de la identificación de las funcionalidades de los proyectos del Laboratorio. Debido a que cada proyecto cuenta con muchas secciones, se elaboró un diagrama abstracto que muestra la interacción entre los componentes principales de la arquitectura, el cual se puede observar en la **¡Error! No se encuentra el origen de la referencia.**:



Fuente: Elaboración propia con información de tecnologías complementarias a Spring para microservicios

Figura 4.2 Abstracción Arquitectura Microservicios

Actualmente, los microservicios son desarrollados con Spring Boot, pero esto no evita que para proyectos futuros sea necesario utilizar este *framework*, al contrario, pueden ser desarrollados con cualquier plataforma y lenguaje de programación siempre y cuando cumplan los protocolos de comunicación REST, al menos para esta versión de la arquitectura.

Por otro lado, cada proyecto se encuentra configurado en un servidor virtual, y aunque todas cuentan con sistema operativo de Microsoft, las configuraciones técnicas de cada servidor varían.

4.3.1 Tecnologías

Para la definición y desarrollo de la arquitectura completa, fue necesario utilizar distintas tecnologías para la comunicación, monitoreo, redireccionamiento, bases de datos, gestión de cargas, entre otras tareas necesarias para la arquitectura. Estas tecnologías se describen a continuación:

- **Tomcat.** Servidor que funciona como contenedor de *servlets* desarrollado bajo el proyecto Jakarta en la Fundación de Apache Software (The Apache Software Foundation, 2022b).
- **Node.JS.** Entorno en tiempo de ejecución multiplataforma, *OpenSource*, para la capa de servidor basado en el lenguaje de programación JavaScript, asíncrono, con una arquitectura orientada a eventos y basado en el motor V8 de Google (OpenJS Foundation, 2022).
- **React.** Es una biblioteca de Javascript para la creación de interfaces de usuario interactivas de forma sencilla, con la capacidad de actualizar y renderizar de manera eficiente los componentes (Meta Platforms, 2022).
- **Microsoft IIS.** Servidor web que contiene un conjunto de servicios para los sistemas operativos de Microsoft (Microsoft, 2022a).
- **Spring Cloud Config Server.** Proporciona el soporte para la configuración externalizada de un sistema distribuido del lado servidor y del lado cliente(VMware, 2022c).
- **Yaml.** Es un formato de serialización de datos legible por humanos, inspirado en XML, C, Python, Perl y otros formatos de correos electrónicos (RedHat, 2021).
- **Maven.** Herramienta para la gestión y construcción de proyectos, se basa en el concepto de un modelo de objetos de proyecto (Project Object Model POM), y gestiona la construcción, los informes y la documentación de un proyecto desde una pieza central de información (The Apache Software Foundation, 2022c).
- **GitHub.** Es una plataforma de integración de código para desarrolladores que utiliza el sistema de control de versiones Git (GitHub, 2022).
- **Spring Gateway.** Es un proyecto dentro de Spring que se encarga de proporcionar una biblioteca que define el punto de entrada al ecosistema de microservicios, propiciando enrutamiento dinámico, seguridad y monitorización de las llamadas que se realicen(VMware, 2022d).
- ***Spring Load Balancer*.** Biblioteca dentro de Spring que ofrece la capacidad de distribuir el tráfico de los procesos en un conjunto de instancias diferentes de la misma aplicación (VMware, 2022a).

- **Spring Eureka Server.** Es un servicio REST, ofrecido por Spring, que se comporta como un servidor que registra y localiza microservicios en un ecosistema. Proporciona información de su localización, su estado y datos relevantes de cada uno de estos servicios (Tarnum Java SRL, 2022).
- **PostgreSQL.** Es un sistema de gestión de bases de datos relacional orientado a objetos y de código abierto (The PostgreSQL Global Development Group, 2022).
- **Elastic Stack.** Plataforma de búsqueda y análisis compuesto por un conjunto de tres proyectos *OpenSource* (Elasticsearch B.V., 2022):
    - **Logstash.** Es un motor de búsqueda y analítica distribuido, gratuito y abierto para todos los tipos de datos: textuales, numéricos, geoespaciales, estructurados y no estructurados.
    - **Elasticsearch.** Es un pipeline de procesamiento de datos *OpenSource* y del lado del servidor que te permite ingresar datos de múltiples fuentes simultáneamente y transformarlos.
    - **Kibana.** Es una herramienta de visualización y gestión de datos para Elasticsearch que brinda histogramas en tiempo real, gráficos circulares y mapas. Kibana también incluye aplicaciones avanzadas, como *Canvas*, que permite a los usuarios crear infografías dinámicas personalizadas con base en sus datos, y *Elastic Maps* para visualizar los datos geoespaciales.

4.3.2 Gateway

Existen tres puertas de enlace diferentes a lo largo de la infraestructura, y cada una se encarga de gestionar la conexión entre componentes de diferente nivel.

- **Gateway IMT.** Esta puerta de enlace se encuentra configurada por el área de sistemas del IMT, y no tiene una relación directa con la arquitectura, sin embargo, debe ser considerada ya que aquí se definen las direcciones y conexiones a los servidores virtuales donde se encuentran los proyectos.
- **Gateway Local.** Esta puerta de enlace se encarga del redireccionamiento dentro del servidor virtual, dentro de cada servidor virtual existen “N” servidores de aplicaciones con diferentes funciones. Algunos ejemplos de los servidores que se pueden encontrar son Tomcat y Node.js, los cuales principalmente son utilizados para las aplicaciones web que el usuario final utilizará en Internet.

- **API Gateway.** Esta puerta de enlace se encarga de administrar y gestionar las peticiones a las API's de cada uno de los microservicios, además, se encarga de validar la información de autenticación de los usuarios que realizan las peticiones.

4.3.3 Ejemplo de implementación

Dentro del Laboratorio Nacional en Sistemas de Transporte y Logística se han desarrollado e implementado una gran cantidad proyectos de investigación y desarrollo con la industria regional, nacional y a nivel América del Norte(Instituto Mexicano del Transporte [IMT], 2021b), de los cuales, se pueden identificar tres proyectos principales:

- LogistiX Lab: Laboratorio de Logística Urbana
- EraclituX: Mapa Digital de Autotransporte
- IMT-X: Vehículos Aéreos Autónomos no Tripulados

Dentro de cada uno de estos proyectos (Instituto Mexicano del Transporte [IMT], 2021a), se han elaborado un conjunto de sub-proyectos de los cuales se han obtenido artículos científicos publicados en conferencias y revistas internacionales, y en publicaciones técnicas del Instituto Mexicano del Transporte. La idea de esta investigación fue, la definición e implementación de una arquitectura que sirviera como base para la migración de cada uno de los proyectos y sub-proyectos dentro del laboratorio, con el objetivo de desarrollar un sistema que permitiera la compatibilidad para comunicarse entre estos, que fuera escalable, resiliente, seguro, y ofreciera una implementación sencilla para los nuevos proyectos a desarrollar.

Uno de los proyectos es la Arquitectura Tecnológica Integrada para Internet de las Cosas encontrada en la Publicación Técnica No. 664 (Hernández Sánchez et al., 2021) del IMT y las fases que continúan en desarrollo actualmente; este proyecto define una arquitectura funcional de IoT para la gestión efectiva de bahías de carga y descarga de mercancías, y se encuentra ligado al Primer Laboratorio de Logística Urbana en Tiempo Real de América Latina LogistiX-Lab.

La investigación realizada en este proyecto apoya la definición del sistema *backend* del proyecto descrito anteriormente, del cual, se generó un manual de usuario para desarrolladores, donde se describe de manera precisa, el uso de la API para este proyecto, las condiciones de uso, el versionamiento y la información detallada de la aplicación de un microservicio desarrollado con esta arquitectura.

Actualmente, para el uso del endpoint de este microservicio, es necesario solicitar acceso al proyecto, debido a que se encuentra en versión Beta, aún no se encuentra disponible de manera pública, ni el código ni el dominio. Para conocer la liberación de los productos, se puede dar seguimiento al Laboratorio Nacional en Sistemas de Transporte y Logística en su cuenta de GitHub: https://github.com/SiTLogLab.

Conclusiones

Actualmente, la implementación de soluciones digitales para las operaciones ha incrementado debido a los avances tecnológicos, y al auge y aplicación de nuevas tecnologías. Por esta razón, la arquitectura monolítica ha comenzado a ser obsoleta para cubrir las necesidades de la actualización de los sistemas y aplicaciones utilizadas día con día. La arquitectura de microservicios ofrece una solución novedosa para la actualización de sistemas heredados, ofreciendo mayor flexibilidad en el desarrollo de aplicaciones, sistemas escalables y con costos operativos más bajos, además de reducir la complejidad del proceso de desarrollo de software. El Laboratorio Nacional en Sistemas de Transporte y Logística, como parte de los Laboratorios Nacionales CONACyT que ofrecen infraestructura interinstitucional distribuida a nivel nacional, también tiene la necesidad de mantener su infraestructura tecnológica actualizada, para desarrollar proyectos nacionales e internacionales de manera eficiente.

Dentro del SiTLog Lab se han desarrollado proyectos representativos, los cuales han sido desarrollados e implementados con una arquitectura monolítica, lo que ha impedido el ágil desarrollo para las nuevas funcionalidades solicitadas de nuevos proyectos. Es por esta razón, que se desarrolló en esta investigación una nueva arquitectura que cubriera estos huecos y problemas, y optimizara las capacidades de cada aplicación dentro de los proyectos.

La arquitectura propuesta e implementada ha sido favorable para cubrir las necesidades antes mencionadas., Actualmente ya se comenzó a utilizar en otros proyectos, como una alternativa eficaz para su planificación, brindando la oportunidad de comunicación con sistemas ya existentes que ofrezcan capacidades que aún no se tengan en el sistema actual, y con la ventaja de evitar nuevamente el desarrollo de estas funciones, por ejemplo, los sistemas de ruteo de INEGI o de Google Maps.

Por otro lado, los microservicios que se han desarrollado en esta aplicación, se encuentran en desarrollo continuo sin la necesidad de que el sistema este deshabilitado, debido a la independencia de las funcionalidades. Es por eso que a pesar de ya encontrarse la API disponible para acceso dentro del IMT, se continúan desarrollando microservicios con nuevas funciones.

Finalmente, este proyecto continuará en desarrollo, para la implementación de otros protocolos de comunicación, o la migración de los microservicios a un paradigma de contenedores y clústeres. Por esta razón, se recomienda mantenerse actualizado en la página de GitHub del Laboratorio, así como en contacto con los desarrolladores que conocen el versionamiento de las diferentes aplicaciones.

Bibliografía

Amazon. (2022a). *Amazon API Gateway*. https://aws.amazon.com/es/api-gateway/

Amazon. (2022b). *What are Microservices?* https://aws.amazon.com/microservices/?nc1=h_ls

Anomaly Innovations. (2022). *Serverless Stack Guide*. https://serverless-stack.com/guide.html#table-of-contents

Apache Software Foundation. (2022). *Apache Thrift*. https://thrift.apache.org/

CDI Repository Contributors. (2022). *What is CDI?* https://www.cdi-spec.org/

Chung, K.-M., Delisle, P., & Roth, M. (2006). *Expression Language Specification Version 2.1*.

Clarke, J. (2009). Platform-Level Defenses. In *SQL Injection Attacks and Defense* (pp. 377–413). Elsevier. https://doi.org/10.1016/B978-1-59749-424-3.00009-8

Cockcroft, A. (2013, April 9). *Architectural Patterns for High Availability*. https://www.infoq.com/presentations/Netflix-Architecture/

Diario Oficial de la Federación. (2019). *PLAN Nacional de Desarrollo 2019-2024*.

Divante. (2021). *eCommerce Trends Report 2021*. https://www.divante.com/ecommerce-trends?hsCtaTracking=00067105-7419-4437-865d-067aa8ccfcf1%7Cc934c52c-fea4-43b9-9d48-92ed5c3a7c13

Django Repository Contributors. (2022). *Django-rest-framework Repositorio GitHub*. https://github.com/encode/django-rest-framework

Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and

Tomorrow. In Mazzara Manuel & B. and Meyer (Eds.), *Present and Ulterior Software Engineering* (pp. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12

Eclipse Foundation. (2021). *AspectJ*. https://www.eclipse.org/aspectj/

Ecma International. (2017). *The JSON data interchange syntax*. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

Elasticsearch B.V. (2022). *What is Elasticsearch?* https://www.elastic.co/what-is/elasticsearch

Encode OSS Ltd. (2022). *Django Rest Framework*. https://www.django-rest-framework.org/#

Espadas Pech, J. M. (2022a). *Roadmap: De la Arquitectura Monolítica a Microservicios (Parte 1)*. https://www.linkedin.com/pulse/roadmap-de-la-arquitectura-monolítica-microservicios-1-espadas-pech/

Espadas Pech, J. M. (2022b). *Roadmap: De la Arquitectura Monolítica a Microservicios (Parte 2)*. https://www.linkedin.com/pulse/roadmap-de-la-arquitectura-monolítica-microservicios-2-espadas-pech/

Espadas Pech, J. M. (2022c). *Roadmap: De la Arquitectura Monolítica a Microservicios (Parte 3)*. https://www.linkedin.com/pulse/roadmap-de-la-arquitectura-monolítica-microservicios-3-espadas-pech?trk=organization-update-content_share-article

Express Repository Contributors. (2022). *Express Repositorio GitHub*. https://github.com/expressjs/express

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext Transfer Protocol -- HTTP/1.1*. https://doi.org/10.17487/rfc2616

Fielding, R., Nottingham, M., & Reschke, J. (2022). *RFC 9110 HTTP Semantics*. https://www.rfc-editor.org/rfc/rfc9110.html

Fielding, R., & Taylor, R. N. (2000). *Architectural Styles and the Design of Network-Based Software Architectures* [University of California, Irvine]. https://dl.acm.org/doi/book/10.5555/932295

Flask Repository Contributors. (2022). *Flask Repositorio GitHub*. https://github.com/pallets/flask

Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection pattern*. https://www.martinfowler.com/articles/injection.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional.

GitHub, I. (2022). *Hello World*. https://docs.github.com/en/get-started/quickstart/hello-world

Google. (2021, June 24). *Introduction to microservices*. https://cloud.google.com/architecture/microservices-architecture-introduction

Google. (2022a). *API Gateway*.

Google. (2022b). *Google Trends*. https://trends.google.es/trends/explore?date=2010-09-05 2022-06-09&q=Spring Boot,Django REST,%2Fm%2F0_v2szx,%2Fm%2F0dgs72v,%2Fm%2F0505cl

Google. (2022c). *Refactoring a monolith into microservices*. https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths#design_interservice_communication

Google. (2022d). *STOMP The Simple Text Oriented Messaging Protocol*. https://stomp.github.io/

Gradle Enterprise. (2022). *Spring Boot Repositorio GitHub*. https://github.com/spring-projects/spring-boot

gRPC Authors. (2022). *gRPC*. https://grpc.io/

H. Mahmoud, Q. (2004). *Getting Started with Java Message Service (JMS)*. https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html

Heinemeier Hansson, D. (2022). *Ruby on Rails*. https://rubyonrails.org/

Hernández Sánchez, B., Barrón, M., & Cedillo, M. G. (2021). *IoTX: Arquitectura tecnológica integrada (Fase 1)*. https://imt.mx/archivos/Publicaciones/PublicacionTecnica/pt664.pdf

IBM. (2021). *What is JDBC?* https://www.ibm.com/docs/es/developer-for-zos/9.5.1?topic=support-what-is-jdbc

Instituto Mexicano del Transporte [IMT]. (2021a). *Folleto LABORATORIO NACIONAL EN SISTEMAS DE TRANSPORTE Y LOGÍSTICA (SiT-LOG)*. https://imt.mx/images/files/GRAL/documentos/Folleto-SiT-LogLab.pdf

Instituto Mexicano del Transporte [IMT]. (2021b). *Laboratorio Nacional. Sistemas de Transporte y Logística*. https://lab-nacional-logistica.imt.mx/

Internet Engineering Task Force [IETF]. (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. https://datatracker.ietf.org/doc/html/rfc7231

Internet Engineering Task Force [IETF]. (2022a). *Standards process*. https://www.ietf.org/standards/process/

Internet Engineering Task Force [IETF]. (2022b). *Uniform Resource Identifier (URI): Generic Syntax*. https://datatracker.ietf.org/doc/html/rfc3986

Johnson, R. E., & Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming* , *1*(2), 22–35.

Kiran, R., & Kumar, K. (2021, May 29). *Top Microservices Frameworks*. https://dzone.com/articles/top-microservices-frameworks

Komprise. (2022). *Cold Data Storage*. https://www.komprise.com/glossary_terms/cold-data/

Lewis, J., & Fowler, M. (2014, March 25). *Microservices*. https://martinfowler.com/articles/microservices.html

MarketsAndMarkets. (2018). *Cloud Microservices Market by Component (Platform and Services), Deployment Mode (Public Cloud, Private Cloud, and Hybrid Cloud), Organization Size (Large Enterprises and SMEs), Vertical, and Region - Global Forecast to 2023*. https://www.marketsandmarkets.com/Market-Reports/cloud-microservices-market-60685450.html

Martin Fowler. (2022). *CQRS*. https://martinfowler.com/bliki/CQRS.html

Mattsson, M. (1999). *Object-Oriented Frameworks*.

Meszaros, G. (2008). *Mocks, Fakes, Stubs and Dummies*. http://xunitpatterns.com/Mocks, Fakes, Stubs and Dummies.html

Meta Platforms, I. (2022). *React*. https://es.reactjs.org/

Microsoft. (2014). *Chapter 1: Service Oriented Architecture (SOA)*. https://web.archive.org/web/20140321085658/http://msdn.microsoft.com/en-us/library/bb833022.aspx

Microsoft. (2020). *ASP.NET MVC Overview*. https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/overview/asp-net-mvc-overview

Microsoft. (2021, December 16). *What Is Windows Communication Foundation*. https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf

Microsoft. (2022a). *IIS*. https://www.iis.net/overview

Microsoft. (2022b). *Microservices on Azure*. https://azure.microsoft.com/es-mx/solutions/microservice-applications/#overview

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems: Vol. ISBN: 1491*. Oreilly & Associates Inc.

Newman, S. (2019). *Monolith to Microservices Evolutionary Patterns to Transform Your Monolith: Vol. ISBN:97814*. O'Reilly Media, Inc.

Nu Technology. (2021, July 26). *Migrar de una arquitectura monolítica a una arquitectura de microservicios: Mejores prácticas*. https://nutech.lat/migrar-de-una-arquitectura-monolitica-a-una-arquitectura-de-microservicios-mejores-practicas/

OASIS. (2020). *OASIS*. https://www.oasis-open.org/

OASIS. (2022). *AMQP Advanced Message Queuing Protocol*. https://www.amqp.org/

OpenJS Foundation. (2022). *Node JS*. https://nodejs.org/en/

Oracle. (2022). *Lesson: Overview of JNDI*. https://docs.oracle.com/javase/tutorial/jndi/overview/index.html

Pahino, R. (2020). *¿Qué son Spring framework y Spring Boot? Tu primer programa Java con este framework*. https://www.campusmvp.es/recursos/post/que-son-spring-framework-y-spring-boot-tu-primer-programa-java-con-este-framework.aspx

Pallets. (2022). *Flask*. https://flask.palletsprojects.com/en/2.1.x/

Pittet, S. (2022). *The different types of software testing*. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

Rails Repository Contributors. (2022). *Rails Repositorio GitHub*. https://github.com/rails/rails

Ramírez, D. (2019). Industria 4.0 en México tiene terreno por desarrollar. *T21*. http://t21.com.mx/logistica/2019/09/17/industria-40-mexico-tiene-terreno-desarrollar

RedHat. (2021). *What is YAML?* https://www.redhat.com/en/topics/automation/what-is-yaml

RedHat. (2022a). *Hibernate ORM*. https://hibernate.org/orm/what-is-an-orm/

RedHat. (2022b, May 11). *What are microservices?* https://www.redhat.com/en/topics/microservices/what-are-microservices

Richardson, C. (2018). *Microservices Patterns: With Examples in Java* (1st ed.). Manning Publications.

Rodgers, P. (2005). *Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity*. https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883

Secretaria de Comunicaciones y Transportes. (2020). *Programa Sectorial de Comunicaciones y Transportes 2020-2024*.

SpringSource. (2017). *Overview of Spring Framework Version 5.0.0.RC2*. https://docs.spring.io/spring-framework/docs/5.0.0.RC2/spring-framework-reference/overview.html

StrongLoop, & IBM. (2022). *Express*. https://expressjs.com/

Sumerge. (2020, November 9). *What is Microservices Architecture?* https://www.sumerge.com/what-is-microservices-architecture/

Tarnum Java SRL. (2022). *Introduction to Spring Cloud Netflix – Eureka*. https://www.baeldung.com/spring-cloud-netflix-eureka

The Apache Software Foundation. (2021). *Apache Avro*. https://avro.apache.org/

The Apache Software Foundation. (2022a). *Apache Struts*. https://struts.apache.org/

The Apache Software Foundation. (2022b). *Apache Tomcat*. https://tomcat.apache.org/

The Apache Software Foundation. (2022c). *Welcome to Apache Maven*. https://maven.apache.org/

The Eclipse Foundation. (2017). *About Object-XML Mapping*. https://www.eclipse.org/eclipselink/documentation/2.4/concepts/blocks002.htm

The OpenSSL Project Authors. (2021). *Welcome to OpenSSL!* https://www.openssl.org/

The PostgreSQL Global Development Group. (2022). *PostgreSQL: The World's Most Advanced Open Source Relational Database*. https://www.postgresql.org/

Transporte. (2018). El mapa interactivo de logística para transporte de carga. *Transporte.Mx*. https://transporte.mx/el-mapa-interactivo-de-logistica-para-transporte-de-carga/

VMware, I. (2022a). *Client-Side Load-Balancing with Spring Cloud LoadBalancer*. https://spring.io/guides/gs/spring-cloud-loadbalancer/

VMware, I. (2022b). *Spring Boot*. https://spring.io/projects/spring-boot

VMware, I. (2022c). *Spring Cloud Config*. https://cloud.spring.io/spring-cloud-config/reference/html/

VMware, I. (2022d). *Spring Cloud Gateway*. https://spring.io/projects/spring-cloud-gateway

World Wide Web Consortium [W3C]. (2007, April 27). *SOAP Version 1.2 Part 0: Primer (Second Edition)*. https://www.w3.org/TR/2007/REC-soap12-part0-20070427/

World Wide Web Consortium [W3C]. (2020). *XML*. https://www.w3.org/XML/

COMUNICACIONES

SECRETARÍA DE INFRAESTRUCTURA, COMUNICACIONES Y TRANSPORTES

Km 12+000 Carretera Estatal 431 “El Colorado-Galindo”
San Fandila, Pedro Escobedo
C.P. 76703
Querétaro, México
Tel: +52 442 216 97 77 ext. 2610

publicaciones@imt.mx

http://www.imt.mx/