UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas

Carrera de Ingeniería en Sistemas Computacionales

DESARROLLO DE UNA ARQUITECTURA EFICIENTE ORIENTADA A MICROSERVICIOS CON API REST UTILIZANDO LA CALIDAD EXTERNA DE LAS NORMAS ISO/IEC 25023

Trabajo de grado presentado ante la ilustre Universidad Técnica del Norte previo a la obtención del título de Ingeniero en Sistemas Computacionales

Autor:

Jhony Marcelo Martínez Chugá

Director:

MSc. José Antonio Quiña Mera

Ibarra – Ecuador 2021

UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

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

1. IDENTIFICACIÓN DE LA OBRA

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

DATOS DE CONTACTO			
CÉDULA DE IDENTIDAD:	1004383731		
APELLIDOS Y NOMBRES:	Martínez Chugá Jhony Marcelo		
DIRECCIÓN:	Zaruma y Av. 13 de abril, Ibarra		
EMAIL:	jmmartinezc1@utn.edu.ec		
TELÉFONO FIJO:	062546706	**TELÉFONO MÓVIL:**	0979532425

DATOS DE LA OBRA	
TÍTULO:	DESARROLLO DE UNA ARQUITECTURA EFICIENTE ORIENTADA A MICROSERVICIOS CON API REST UTILIZANDO LA CALIDAD EXTERNA DE LAS NORMAS ISO/IEC 25023
AUTOR (ES):	Martínez Chugá Jhony Marcelo
FECHA: DD/MM/AAAA	10/08/2021
SOLO PARA TRABAJOS DE GRADO	
PROGRAMA:	■ **PREGRADO** □ **POSGRADO**
TITULO POR EL QUE OPTA:	**INGENIERÍA EN SISTEMAS COMPUTACIONALES**
ASESOR /DIRECTOR:	**MSc. José Antonio Quiña Mera**

2. CONSTANCIAS

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

Ibarra, a los 10 días del mes de agosto de 2021

EL AUTOR:

(Firma).....
Nombre: Marcelo Martínez

CERTIFICACIÓN DIRECTOR

En mi calidad de tutor de Trabajo de Grado presentado por el egresado **JHONY MARCELO MARTINEZ CHUGA** para obtener Título de Ingeniería en Sistemas Computacionales cuyo tema es: **Desarrollo de una arquitectura eficiente orientada a microservicios con Api Rest utilizando la calidad externa de las normas ISO/IEC 25023.** Considero que el presente trabajo reúne los requisitos y méritos suficientes para ser sometido a la presentación pública y evaluación por parte del tribunal examinador que se designe.

En la ciudad de Ibarra, a los 10 días del mes de agosto del 2021

JOSE ANTONIO QUIÑA MERA

Firmado digitalmente por JOSE ANTONIO QUIÑA MERA
Fecha: 2021.08.12 13:05:25 -05'00'

Msc. Antonio Quiña

DIRECTOR DE TRABAJO DE GRADO

DEDICATORIA

Quiero dedicar este trabajo a mi madre Nancy Chugá, por su sacrificio y esfuerzo durante todos estos años, por creer en mis capacidades, por enseñarme que los sueños se consiguen con dedicación y esfuerzo, por brindarme su comprensión, cariño, amor y sobre todo porque sin ella nada de esto sería posible.

A Mishell mi hermana y Mario su padre, por su apoyo incondicional y paciencia porque solo ustedes y mi madre saben en realidad cuanto ha costado este logro.

A mis abuelos Vicente y Clara, por enseñarme valores los cuales han sido fundamentales para forjar mi carácter y gran parte de la persona que me he convertido.

A Oliver, Ervin, Jerson, Leonardo y Joselyn, quienes son y serán futuros colegas profesionales, por todas las vivencias dentro y fuera del ámbito universitario, por estar siempre dispuestos a brindar apoyo.

AGRADECIMIENTO

A todos los docentes de la carrera de Ingeniería en Sistemas Computacionales, por compartir sus conocimientos, por la motivación y por las guías brindadas para un correcto desarrollo profesional y personal.

Al director de este trabajo de titulación, Msc. Antonio Quiña, por el apoyo ofrecido, por los consejos brindados y por el alto nivel de profesionalismo demostrado a lo largo de la investigación.

A mis asesores Msc. Diego Trejo y Msc. Mauricio Rea por aportar con su conocimiento para llevar de la mejor manera posible este trabajo de titulación.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

La presente investigación se centra en el desarrollo de una Arquitectura eficiente orientada a microservicios con API-REST, utilizando la calidad externa de la norma ISO/IEC 25023. Para esto, la pertinente investigación documental permitió establecer un marco teórico y tecnológico que abarca los tópicos de las arquitecturas orientadas a microservicios, lenguajes de programación para desarrollar aplicaciones REST y en definitiva sobre la experimentación en la ingeniería de software.

Para poder desarrollar una arquitectura eficiente se realizó un experimento de software que permitió definir el alcance, la planificación, la parte operativa y analizar e interpretar los resultados de este. Dicho experimento se basó en comparar tecnologías tanto para el Backend como para el Frontend que permitan desarrollar aplicaciones usando el patrón arquitectónico REST y, teniendo en cuenta las métricas de tiempo medio de espera y tiempo medio de rendimiento de la norma ISO/IEC 25023.

Y finalmente se presenta los resultados de la investigación representados en tablas de valores y graficas que le brinden al lector un mayor entendimiento de la comparación de las diferentes arquitecturas de desarrollo planteadas y de cómo se procedió a seleccionar la más eficiente.

ABSTRACT

This research focuses on the development of an efficient architecture oriented to microservices with API-REST, using the external quality of the ISO / IEC 25023 standard. For this, the pertinent documentary research allowed to establish a theoretical and technological framework that covers the topics of architectures oriented to microservices, programming languages to develop REST applications and ultimately about experimentation in software engineering.

In order to develop an efficient architecture, a software experiment was carried out to define the scope, the planning, the operational part and analyze and interpret the results. This experiment was based on comparing technologies for both the Backend and Frontend that allow the development of applications using the REST architectural pattern and, based on the metrics of mean waiting time and mean time of performance of the ISO / IEC 25023 standard.

The results of the investigation are presented represented in tables of values and graphs that give the reader a better understanding of the comparison of the different development architectures proposed and how the most efficient one was selected.

INTRODUCCIÓN

Problema

Antecedentes.

Las Aplicaciones Web son tendencias por sus potentes funcionalidades, adaptabilidad y buena aceptación por parte de las empresas que desarrollan Software, siendo estas Aplicaciones, en su mayoría diseñadas con una Arquitectura Monolítica (Ramchandra Desai, 2016).

En las últimas décadas, debido al desarrollo de tecnologías relacionadas con Internet, las arquitecturas de software de servicios web de Representational State Transfer (REST) se volvieron más populares debido a su simplicidad, interoperabilidad y escalabilidad (Pautasso, Zimmermann, & Leymann, 2008). Las aplicaciones de software que siguen este estilo de arquitectura de software comprenden un conjunto de servicios web REST que utilizan el Protocolo de transferencia de hipertexto (HTTP) (Mumbaikar, & Padiya, 2013).

Situación actual.

Intercambiar información entre las aplicaciones de forma estándar es el objetivo principal de los servicios web. Con la implementación de la arquitectura de microservicios el principio SOAP presenta inconvenientes respecto al tráfico de red, mayor latencia y retrasos en el procesamiento. Para superar estas limitaciones actualmente se utiliza la arquitectura REST aprovechando las características de simplicidad, interoperabilidad y escalabilidad (Ramchandra Desai, 2016).

Prospectiva.

Esta investigación consiste en realizar una búsqueda de las tecnologías en tendencia para backend y frontend con el fin de desarrollar una arquitectura eficiente utilizando el diseño arquitectónico REST mediante experimentos de laboratorio utilizando las tecnologías

seleccionadas y realizar una validación con las métricas establecidas en el estándar ISO/IEC 25023 enfocados en el tiempo medio de respuesta y el tiempo medio de rendimiento.

Planteamiento del problema.

La arquitectura de software juega un papel indispensable en el éxito o el fracaso de cualquier sistema de software, ya que se ocupa de la estructura base, los subsistemas y las interacciones entre estos subsistemas (Farshidi, Jansen, & Werf, 2020).

La implementación de arquitecturas monolíticas presenta inconvenientes respecto a escalabilidad de aplicaciones, tráfico de red, mayor latencia y retrasos en el procesamiento además de un alto costo de mantenimiento.

Para poder definir el diagrama de Planteamiento del Problema se utilizó el instrumento de investigación de identificación y clasificación de problemas (Matriz Vester).

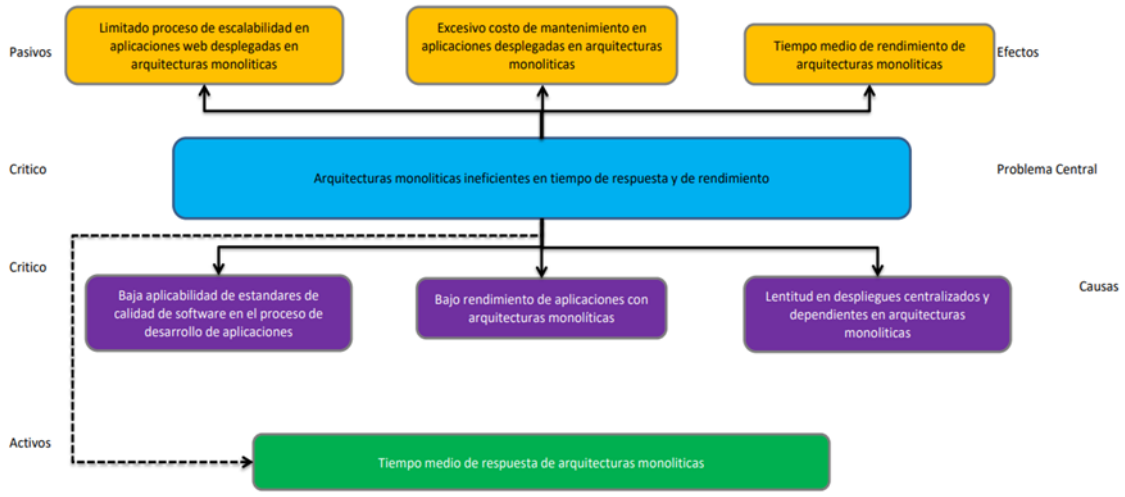


Figura 1. Árbol de problemas

Objetivos

Objetivo general.

Desarrollar una arquitectura eficiente orientada a microservicios utilizando el diseño arquitectónico REST utilizando la calidad externa de las normas ISO/IEC 25023.

Objetivos específicos.

- Elaborar un marco conceptual y tecnológico para desarrollar una arquitectura orientada a microservicios.
- Comparar tecnologías de backend y frontend utilizando el diseño arquitectónico REST mediante experimentos de laboratorio.
- Desarrollar una arquitectura eficiente y validar el funcionamiento de la arquitectura establecida mediante una prueba de concepto.

Alcance

Mediante la siguiente investigación se desarrollará una arquitectura orientada a microservicios eficiente, utilizando el diseño arquitectónico REST. Validando con el estándar ISO/IEC 25023.

El estudio consiste en realizar una búsqueda de tecnologías en tendencia para backend y frontend mediante una revisión bibliográfica y principales portales tecnológicos como GitHub, con el fin de desarrollar la arquitectura orientada a microservicios que se utilizará para el experimento.

Una vez establecidas se procederá a comparar dichas tecnologías mediante casos de uso con experimentos de laboratorio. Una vez establecida la arquitectura se procederá a validar el funcionamiento, mediante pruebas de concepto con referencia al estándar ISO/IEC 25023 para comprobar la eficiencia de comportamiento de la arquitectura específicamente en el tiempo medio de respuesta y tiempo medio de rendimiento.

De esta manera se podrá presentar una arquitectura orientada a microservicios con sus tecnologías ya establecidas, además de presentar su eficiencia validada por el estándar ISO/IEC 25023.

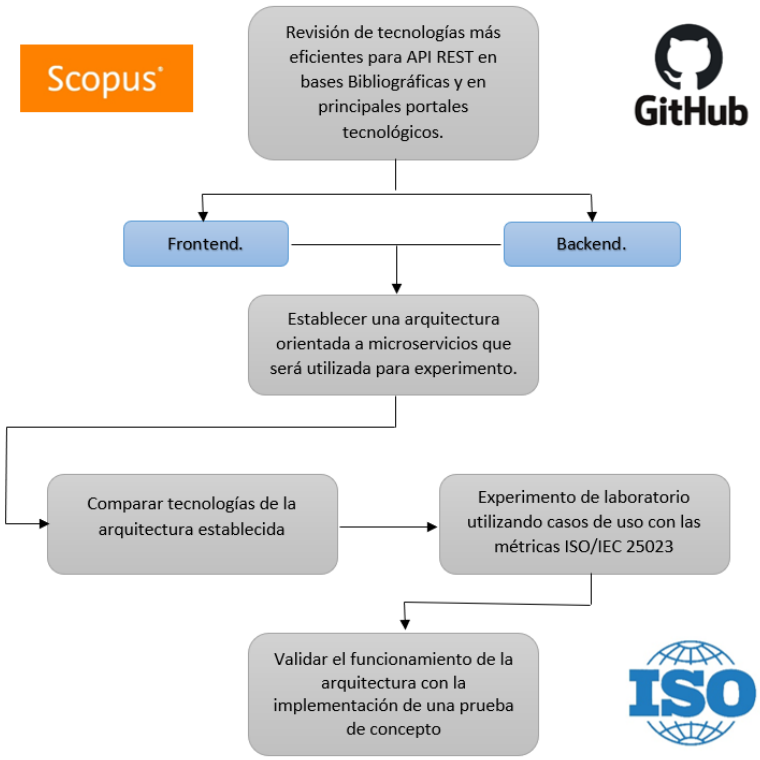


Figura 2. Mentefacto

Justificación

ODS.

El desarrollo validado de una arquitectura eficiente orientada a microservicios con API REST apoyará al fortalecimiento de los Objetivos de Desarrollo Sostenible (ODS), en especial el #9 “Industria, Innovación e Infraestructura” específicamente la meta #9.b que hace referencia a apoyar el desarrollo de tecnologías, la investigación y la innovación nacionales en los países en desarrollo, incluso garantizando un entorno normativo propicio a la diversificación industrial y la adición de valor a los productos básicos, entre otras cosas (Naciones Unidas, 2015).

Justificación tecnológica.

Los microservicios se desarrollan, implementan y mantienen por separado, esto permite que los equipos sean autónomos, los mismos que pueden decidir qué tecnología utilizar y adaptarse a las necesidades actuales del comportamiento empresarial. El objetivo de los

microservicios es dividir el comportamiento empresarial en pequeños servicios que pueden funcionar de forma independiente entre sí (Bottero, & Datola, 2017).

Justificación metodológica.

ISO/IEC 25023 - Medición de la calidad del producto del sistema y del software: proporciona medidas que incluyen funciones de medición asociadas a las características de calidad en el modelo de calidad del producto (Organización Internacional de Normalización [ISO], 2016).

Contexto

Tabla 1: Contexto

AUTOR	TEMA	DIFERENCIA
Saransig Alexis	Análisis de rendimiento entre una arquitectura monolítica y una arquitectura de microservicios tecnología basada en contenedores	Somete cada ambiente a pruebas de estrés y se analiza posteriormente los datos en bruto almacenados en archivos de logs acerca manejo eficiente de los recursos y la eficiencia de la producción de Software (Saransig, 2018).
López Hinojosa, José Daniel	Arquitectura de software basada en microservicios para desarrollo de aplicaciones web de la Asamblea Nacional	Propone una arquitectura de software para el desarrollo de aplicaciones web en la Coordinación General de Tecnologías de la Información y Comunicación de la Asamblea Nacional del Ecuador (López, 2017).
Flores Landeta, Jefferson David	Estudio de una arquitectura de microservicios mediante Spring Cloud para el desarrollo del módulo de registro y seguimiento médico de los deportistas en la Federación deportiva de Imbabura	Estudia una arquitectura de microservicios mediante Spring Cloud para el desarrollo del módulo de registro y seguimiento médico de los deportistas en la Federación Deportiva de Imbabura (Flores, 2020).
Chulca Quilachamín, Cristhian Andrés	Migración hacia una arquitectura basada en microservicios del sistema de gestión centralizada de laboratorios de la DGIP	Plantea una propuesta metodológica de migración de arquitecturas monolíticas hacia microservicios (Chulca, & Molina, 2020).

Molina López, Raúl Patricio		
Chicaiza Ríos, Diego Fernando	Diseño de un prototipo de una arquitectura basada en microservicios para la integración de aplicaciones Web altamente transaccionales. Caso: Entidades financieras.	Tiene su enfoque en las entidades financieras que demandan la estabilidad, la confiabilidad, un alto grado de tolerancia a fallas y una escala simple (Chicaiza, 2020).
Yaguachi Pereira, Lady Elizabeth	Aplicación de un modelo para evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios	Desarrolla un modelo para evaluar el rendimiento en el proceso de migración de una aplicación monolítica hacia una orientada a microservicios (Yaguachi, 2017).

CAPÍTULO I

Marco Teórico

1.1. Arquitecturas de Software

1.1.1. Evolución de las arquitecturas de software.

En la Ingeniería de Software, el cambio, es una constante que no se puede controlar, constante que ha permitido que las arquitecturas de software evolucionen. De tal manera, que la evolución de una arquitectura de software está relacionada con la ejecución metodológica de cambios arquitectónicos que tienen que ver con la modificación, adición y eliminación de los elementos que la componen, a fin de transformar el modelo arquitectónico como tal (Leone, Roldán , & Gonnet, 2015).

Entre las causas de los cambios de las arquitecturas de software tenemos:

- Redefinición de requerimientos existentes o aparición de nuevos requerimientos.
- Necesidad de cambios en la infraestructura o tecnología.
- Desgaste en el diseño.
- Nueva iteración en el proceso de desarrollo del sistema.
- Aparición de errores, debidos a malas decisiones de diseño arquitectónico.

1.1.2. Conceptos.

Existen un sin número de definiciones referentes a la arquitectura de software moderna, pues no existe una definición universal. A continuación, se citan algunas:

“El conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos de software, las relaciones entre ellos, y propiedades de ambos” (Bachmann et al., 2010).

“La arquitectura de software de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema. Comprende elementos de software, relaciones entre ellos, y propiedades de ambos” (Bass, Clements, & Kazman, 2012).

“La arquitectura se define por la práctica recomendada como la organización fundamental de un sistema, incorporada en sus componentes, sus relaciones entre sí y con el medio ambiente, y los principios que rigen su diseño y evolución” (Institute of Electrical and Electronics Engineers [IEEE], 2000).

En un contexto general, la arquitectura de software es el conjunto de componentes de software que integran un sistema, la forma en que estos se comportan y se relacionan entre sí para poder alcanzar la meta del sistema.

1.1.3. Arquitectura orientada a microservicios.

El término “microservicio” fue discutido por primera vez en un taller por un grupo de arquitectos de software en 2011, y en mayo del siguiente año, el mismo grupo decidió que “microservicios” es el término más apropiado. Hoy en día existen varias definiciones del concepto de microservicios, pero la que más ayuda a comprender qué son, es la que Newman (2015), describe como: “Servicios pequeños e independientes que trabajan juntos”.

Ahora bien, una arquitectura orientada a microservicios, que se ilustra en la *Figura 3*, se enfoca en desarrollar una única aplicación que engloba un conjunto de pequeños servicios, en donde cada uno se ejecuta independientemente en su propio proceso y comunicándose con mecanismos ligeros, a menudo un recurso de una Interfaz de Programación de Aplicaciones (API) de recursos de Protocolo de Transferencia de Hipertexto (HTTP). Dichos servicios están montados alrededor de las capacidades del negocio y con autonomía de despliegue e implementación cien por ciento automatizada. La gestión centralizada de estos servicios es mínima, dichos servicios pueden estar codificados en lenguajes de programación diferentes y usar distintas tecnologías para almacenar datos (Lewis & Fowler, 2014).

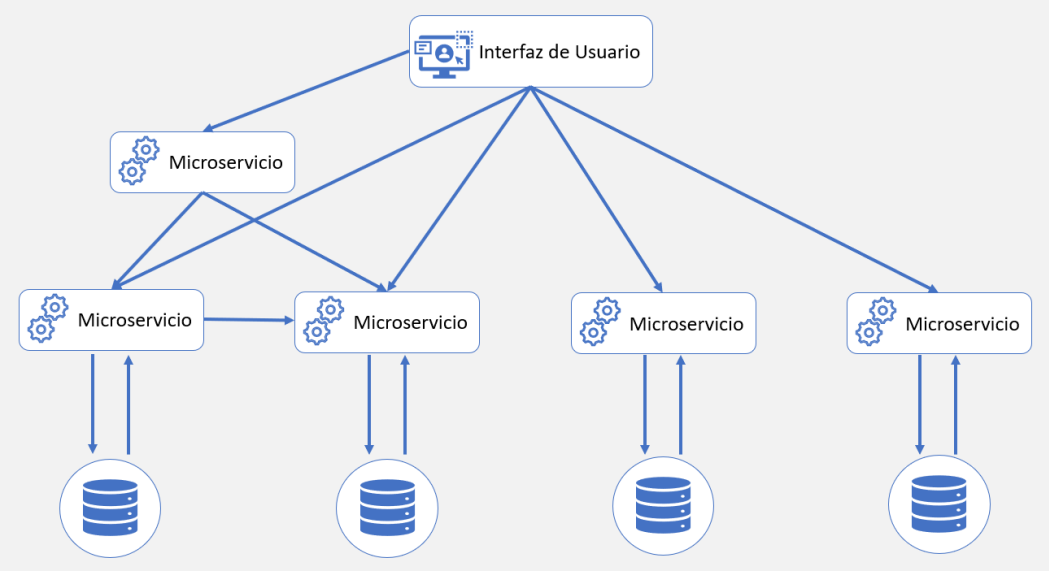


Figura 3. Arquitectura basada en microservicios

Una de las ventajas de utilizar microservicios es la capacidad de publicar una aplicación robusta como un conjunto de aplicaciones pequeñas (microservicios) que se pueden programar de manera independiente. Los microservicios permiten que las empresas puedan gestionar robustas aplicaciones de base de código, manejando una metodología más práctica, donde se dan mejoras incrementales e implementaciones independientes gracias a que estas son ejecutadas por equipos pequeños en bases de código (Villamizar et al., 2015).

Entre algunas de las características que posee esta arquitectura, son las que nos plantea (Dragoni et al., 2017):

- Implementa menos funcionalidades, de ahí que su base de código sea pequeña y que el alcance de errores sea limitado.
- Posibilita la migración progresiva a nuevas versiones de un microservicio.
- Al cambiar un módulo sólo se requiere el reinicio de los microservicios del respectivo módulo más no del sistema completo.
- Los microservicios facilitan la implementación de Contenedores, lo que permite a los desarrolladores configurar un ambiente de despliegue que se adapte a sus necesidades.

- En la escalabilidad de esta arquitectura no implica la duplicidad de sus componentes, permitiendo a los desarrolladores desplegar instancias de servicios dependiendo su carga.
- La comunicación en una red de microservicios interoperativos se restringe a la tecnología que se utilice (medios de comunicación, protocolos, codificaciones de datos). A parte de eso, los desarrolladores son libres de elegir el lenguaje y el Framework para la implementación de cada servicio.

1.2. API REST

Primeramente, antes de describir el concepto de API-REST, es preciso definir qué es una API (Interfaz de Programación de Aplicaciones). Una API es un conjunto de reglas o contratos que establece cómo los consumidores deben definir las entradas y salidas al interactuar con los servicios (Reynders, 2018).

REST es un acrónimo de REpresentational State Transfer, que según Reynders (2018) define como: “un estilo arquitectónico basado en un conjunto de principios predefinidos que describen cómo se manipulan los recursos en red”.

La tipología basada en API-REST es útil para sitios web que presentan servicios pequeños e independientes a través de algún tipo de API (Interfaz de Programación de Aplicaciones). Dicha tipología, que se ilustra en la *Figura 4*, consta de componentes de servicio (microservicios) que a su vez están conformados por uno o dos módulos que realizan funciones específicas independientes de tipo comercial. En este tipo de topología, se suele acceder a estos microservicios mediante una interfaz basada en REST (Transferencia de Estado Representacional) implementada a través de una API basada en Web desplegada por separado (Richards, 2015).

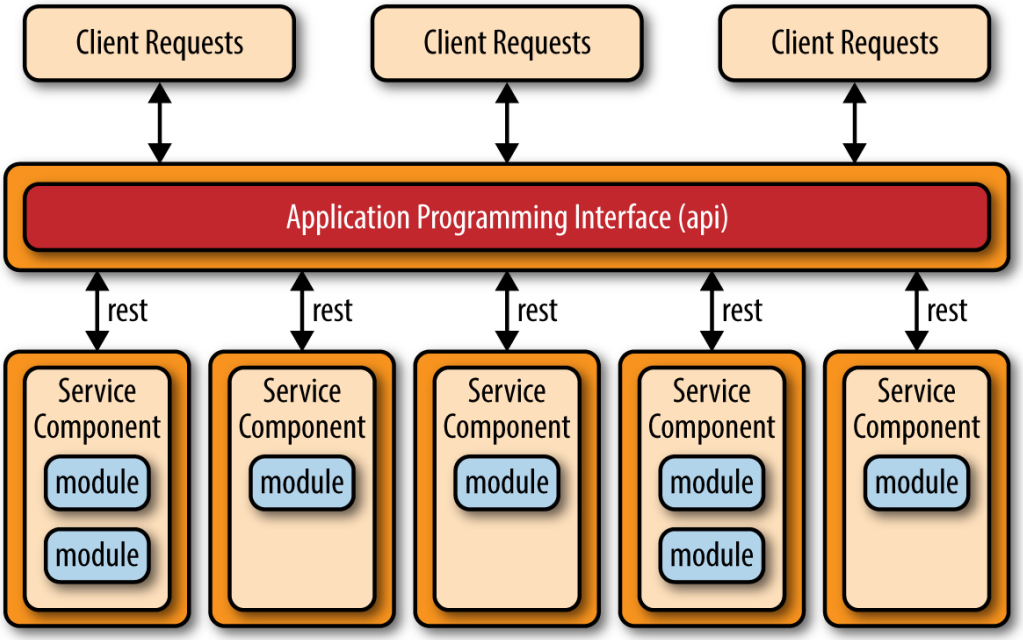


Figura 4. Tipología basada en API-REST
Fuente: *(Richards, 2015)*

Una API-REST se basa en el protocolo HTTP, y cuando la aplicación realiza una solicitud dicha solicitud suele ser la combinación del método HTTP y el Identificador Uniforme de Recursos (URI), donde el verbo HTTP describe los métodos y el URI identifica los recursos de la API-REST. Entre los métodos más frecuentes tenemos POST, GET, PUT, DELETE, todos los cuales implementan para enviar, solicitar, modificar, y eliminar un recurso, respectivamente. En resumen, una vez que la aplicación obtiene la dirección IP del controlador y el URI de la API-REST, esta última puede ser llamada usando cualquier método HTTP (Hu et al., 2021).

1.3. Principales lenguajes de programación utilizados con API-REST

Hoy en día, construir una API REST es muy fácil, ya que la mayoría de los lenguajes de programación permiten su desarrollo, mediante la implementación de frameworks o librerías.

A continuación, se describen los lenguajes más usados por los desarrolladores según unas de las principales plataformas de desarrollo colaborativo como lo es GitHub, puesto que permite la comunidad de programadores valoren y destaquen los proyectos con los lenguajes de programación utilizados para construir aplicaciones con el patrón arquitectónico REST y en el Capítulo 2 se analizan los frameworks que complementan la creación de estas aplicaciones.

1.3.1. PHP.

Según W3Techs (2020), PHP se posiciona como uno de los lenguajes de programación más populares, puesto que el 79,1% de los sitios web con lenguaje de programación del lado del servidor usan PHP.

1.3.2. C Sharp (C#).

En la documentación de Microsoft (2020), "C# es un lenguaje orientado a objetos elegante y con seguridad de tipos que permite a los desarrolladores crear muchos tipos de aplicaciones seguras y sólidas que se ejecutan en el ecosistema de .NET". W3Techs (2020), afirma que el 9,3% de los sitios web con lenguaje de programación del lado del servidor usan ASP.NET.

1.3.3. JavaScript (JS).

JS es un lenguaje de programación ligero y de compilado justo a tiempo (just-in-time) que tiene funciones de primera clase. También es conocido como un lenguaje scripting (secuencias de comando) para páginas web y cabe resaltar que es usado en otros entornos como Node.js (MDN, 2020). W3Techs (2020), menciona que el 1,1% de los sitios web con lenguaje de programación del lado del servidor usan JavaScript.

1.3.4. Revisión de bases de datos bibliográficas.

Para el desarrollo del marco teórico de este trabajo de titulación, se realizó una revisión bibliográfica que permitió consultar diferentes fuentes de información con el fin de ampliar el conocimiento acerca del problema que se abordó en esta investigación. Para lo cual, se realizó

una búsqueda en sitios oficiales de tecnología y bases de datos bibliográficas como Scopus, Springer y Taylor & Francis, para encontrar publicaciones, artículos e información relevante de los temas de arquitectura de software, microservicios y API-REST.

La revisión bibliográfica se llevó a cabo en las siguientes herramientas de investigación en línea: SpringerLink, Taylor & Francis Online, ScienceDirect. Estas permiten el acceso a una gran colección de libros, publicaciones, obras de referencia y artículos científicos; que contribuyeron en gran medida a la redacción de esta investigación.

1.3.5. Revisión en plataformas tecnológicas en la industria.

Como gestor bibliográfico se usó Mendeley, que es un software de gestión bibliográfica gratuito, que permite recopilar la información de manera que las referencias bibliográficas se generan, organizan y citan automáticamente (López, 2014).

1.4. Experimentos de Software

La experimentación del software nace con la falta de evidencias de las tecnologías utilizadas en el desarrollo de software sobre su adecuación, límites, cualidades, costos y riesgos. Es decir, no existe alguna evidencia que justifique las creencias sobre las que se basa la construcción del software. Es por eso por lo que los experimentos de software contribuyen a disentir las creencias y las opiniones para convertirlas en hechos (Mon et al., 2012).

Mon et al., (2012) manifiesta que: “el fin de la experimentación es identificar las causas por las que se producen determinados resultados. Un experimento modela en el laboratorio, en condiciones controladas, las principales características de una realidad lo que permite estudiarla y comprenderla mejor”.

Un experimento de software se concibe de la Experimentación de la Ingeniería de Software (ISE), posibilita el comprender e identificar las variables que se introducen en el juego de la construcción de software y las conexiones entre ellas. Experimentar con la construcción

de software permitirá aumentar la comprensión de lo que hace que el software sea bueno y como consiguiente, cómo fabricar software correctamente (Mon et al., 2012).

Los experimentos de software se llevan a cabo cuando se quiere controlar la situación y manipular el comportamiento directa, precisa y sistemáticamente. Además, los experimentos van más allá de un tratamiento para comparar los resultados. Por ejemplo, si es posible controlar quién está usando un método y quién está usando otro método, y cuándo y dónde se usan, es posible realizar un experimento (Wohlin et al., 2012). Según (Wohlin et al., 2012), la realización de un experimento implica una serie de pasos, estos son:

1. Alcance.
2. Planificación.
3. Operación.
4. Análisis e interpretación.

1.5. ISO/IEC 25023

1.5.1. Introducción.

Esta Norma Internacional (NI) define algunas medidas para la evaluación de la calidad de sistemas y productos de software en términos de características y subcaracterísticas que están determinadas en la Norma ISO/IEC 25010. También, la normativa ISO/IEC 25023:2016 se puede utilizar en conjunto con las Normas de las divisiones ISO/IEC 2503n e ISO/IEC 2504n o para satisfacer las necesidades de los usuarios con respecto a la calidad de los sistemas y productos de software (ISO, 2016).

El contenido de esta normativa abarca dos aspectos importantes: primero, “un conjunto básico de medidas de la calidad para cada una de las características y subcaracterísticas”, y segundo, “una explicación sobre cómo aplicar las medidas de la calidad de sistemas y productos de software” (ISO, 2016).

Cabe resaltar que, “la ISO/IEC 25023:2016 no asigna rangos de valores de las medidas a niveles nominales o grados de cumplimiento porque estos valores se definen con base en la naturaleza del sistema, producto o parte del producto, y dependiendo de factores como categoría del software, nivel de integridad y necesidades de los usuarios" (ISO, 2016).

Las medidas de calidad presentadas en la normativa están para ser utilizadas durante o después del ciclo de vida desarrollo, con el fin de asegurar la calidad y la mejora del sistema, y los productos de software. Además, sus principales usuarios son aquellas personas que realizan actividades de evaluación y especificación de requisitos de calidad como parte de: el desarrollo (análisis de requisitos, diseño, codificación y pruebas), la gestión de la calidad (garantía, control y certificación), el suministro (contrato con el adquiriente), la adquisición (selección de productos y pruebas de aceptación), y el mantenimiento (basado en la medición de la calidad); de un sistema, producto o servicio de software (ISO, 2016).

1.5.2. Medidas de la calidad del sistema y producto de software.

Eficiencia del desempeño.

Esta característica se utiliza para evaluar el desempeño relativo a la cantidad de recursos utilizados bajo condiciones determinadas, estos recursos pueden ser otros productos de software, la configuración de software y hardware del sistema, y materiales como, por ejemplo: papel de impresión, medios de almacenamiento, entre otros (INTE/ISO/IEC, 2020).

Comportamiento del tiempo.

Esta subcaracterística se utiliza para evaluar los tiempos de respuesta y procesamiento, y las tasas de rendimiento de las funciones realizadas de un producto o sistema cuando cumplen los requisitos (INTE/ISO/IEC, 2020).

Las métricas para evaluar la calidad externa se detallan en la siguiente tabla:

Tabla 2: Métricas de la calidad externa de la norma ISO/IEC 25023

Característica	Subcaracterística	Métrica	Fase del ciclo de vida de calidad del producto	Propósito de la métrica de calidad	Método de aplicación	Fórmula	Valor deseado	Tipo de medida	Recursos utilizados
Eficiencia del desempeño	Comportamiento del tiempo	Tiempo de respuesta	Interna/Externa	¿Cuál es el tiempo estimado para completar una tarea?	Tomar el tiempo desde que se envía la petición hasta obtener la respuesta	X = B - A A = Tiempo de envío de petición B = Tiempo en recibir la primera respuesta	0<=X<=1 El más cercano a 0 es el mejor. Donde el peor caso es >=15t.	X= Tiempo /Tiempo A=Tiempo B=Tiempo	Especificación de requerimientos, Código fuente, Desarrollado r, Tester
		Tiempo de espera	Interna/Externa	¿Cuál es el tiempo desde que se envía una instrucción, para que inicie un trabajo, hasta que lo completa?	Tomar el tiempo cuando se inicia un trabajo y el tiempo en completar el trabajo	X = B - A A= Tiempo cuando se inicia un trabajo B = Tiempo en completar el trabajo	0<=X<=1 El más cercano a 0 es el mejor. Donde el peor caso es >=15t.	X= Tiempo /Tiempo A= Tiempo B=Tiempo	Especificación de requerimientos, Código fuente, Desarrollado r, Tester
		Rendimiento	Interna/Externa	¿Cuántas tareas pueden ser procesadas por unidad de tiempo?	Contar el número de tareas completadas en un intervalo de tiempo	X = A/T A= Número de tareas completadas T = Intervalo de tiempo Dónde: T > 0	X = A/T El más lejano a 0/t es el mejor. Donde el mejor caso es >=10/t	X= Contable/ Tiempo A= Contable T= Tiempo	Especificación de requerimientos, Código fuente, Desarrollado r, Tester

Fuente: Elaboración propia según la norma ISO/IEC 25023

CAPÍTULO II

Desarrollar arquitectura eficiente con API-REST

1.1. Análisis de Tecnologías con API-REST

En esta sección se describe los Frameworks que permiten el desarrollo de aplicaciones con el patrón arquitectónico REST, como lo son: Laravel, .Net y NodeJS

Laravel Framework.

Laravel es un marco PHP muy flexible que facilita la ejecución de tareas altamente complicadas. Este marco de referencia provee de herramientas poderosas para la creación de aplicaciones grandes y robustas (Sinha, 2019).

Una de las ventajas que destaca a Laravel como framework en el desarrollo de una API-REST, es que sus controladores de recursos están estructurados en torno a verbos y patrones REST. Por esta razón, la vida de los desarrolladores Laravel es mucho más cómoda (Stauffer, 2019).

.NET Framework.

Esta tecnología permite a los desarrolladores tener varias alternativas sencillas y eficientes para crear una API. Por ejemplo, Microsoft pone a disposición ASP.NET Web API, un framework para construir servicios HTTP especialmente pensados para REST (Arsys, 2016).

NodeJS.

Node.js es un entorno basado en JavaScript que se orienta a eventos asíncronos, está diseñado para crear aplicaciones escalables y dado que el protocolo HTTP es un elemento destacado de esta tecnología, esto hace que Node.js sea muy adecuado para construir una API-REST (OpenJS Foundation, 2020).

Node.js permite desarrollar una API-REST de tal manera que se puede construir tanto el backend como el frontend con el mismo lenguaje, evitando el tedioso trabajo de migrar de un lenguaje a otro cuando se desarrollan ambas capas de una aplicación (Arsys, 2016).

1.2. Laboratorio Experimental Utilizando Métricas de la Norma ISO/IEC 25023

Para este laboratorio experimental, así como en investigaciones previas propuestas por otros autores, que han perseguido el mismo fin que es el de demostrar cuan eficiente puede ser una arquitectura de desarrollo al hacer uso de las tecnologías emergentes, en este caso implementando API-REST y microservicios.

Las métricas que se van a evaluar implementando la norma ISO/IEC 25023 son el tiempo de espera y el rendimiento, que pertenecen a la subcaracterística del comportamiento del tiempo y esta a su vez corresponde a la característica de eficiencia del desempeño, las cuales se abordan en la sección 1.5. del Capítulo I.

Así mismo, para la ejecución de dicho experimento de laboratorio se plantea comparar la eficiencia de tres API-REST desarrolladas en los lenguajes de programación analizados en este estudio: Node.js con JavaScript, .Net Core con C# y Laravel con PHP.

1.2.1. Alcance

Según (Wohlin et al., 2012), en esta fase se define la base del experimento, que se ve reflejada en la *Figura 5* y cabe recalcar que el alcance del experimento se logra al definir los objetivos de este.

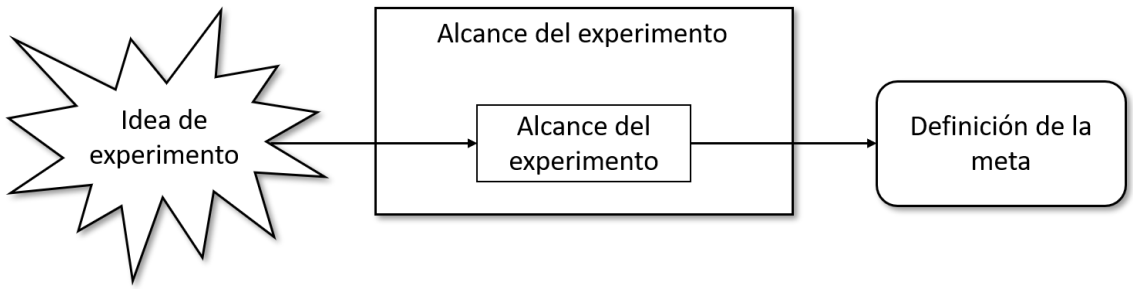


Figura 5. Descripción general de la fase de determinación del alcance
Fuente: Elaboración propia según (Wohlin et al., 2012)

A continuación, se detalla el proceso para la determinación del alcance en función de la meta del laboratorio experimental y sus respectivos pasos.

Definición de la meta.

En este caso, el objetivo empírico de estudio es comparar tecnologías de Backend y Frontend utilizando las métricas de tiempo de espera y rendimiento de la norma ISO/IEC 25023.

El experimento está motivado por la necesidad de saber qué tecnologías de Backend y Frontend son las más eficientes con base en la norma ISO/IEC 25023. Es bien sabido que existen un sin número de herramientas y tecnologías de desarrollo, y resulta importante conocer cuáles son las más adecuadas para desarrollar una arquitectura eficiente con el patrón arquitectónico REST.

Objeto de estudio.

El objeto de estudio son las tecnologías de Backend y Frontend y su capacidad en términos de eficiencia en función de sus características como herramientas de desarrollo.

Propósito.

El propósito del experimento es comparar tecnologías de Backend y Frontend en base a las métricas de tiempo de espera y rendimiento contempladas en la norma ISO/IEC 25023. El experimento proporciona información sobre lo que se puede esperar en términos de eficiencia al comparar estas tecnologías de desarrollo usando el diseño arquitectónico REST.

Perspectiva.

La perspectiva es desde el punto de vista del investigador, es decir, al investigador le gustaría saber que tecnologías son las más eficientes tanto en Backend como en Frontend para así establecer una arquitectura eficiente adecuada.

Enfoque de calidad.

El principal efecto estudiado en el experimento es la eficiencia de las tecnologías de Backend y Frontend. Así que, como enfoque de calidad, se enfatizan dos métricas de la norma ISO/IEC 25023:

- Tiempo de espera, que responde a la interrogante: ¿Cuál es el tiempo desde que se envía una instrucción, para que inicie un trabajo, hasta que lo completa?
- Rendimiento, que responde a la interrogante: ¿Cuántas tareas pueden ser procesadas por unidad de tiempo?

Contexto.

El contexto del experimento se realizó bajo un ambiente controlado, en este caso limitado a un solo computador, en el cual se instalaron todas las herramientas necesarias para la ejecución del experimento, que permitió la combinación de 9 arquitecturas de desarrollo distintas. Además, se contó con una base de datos para la ejecución de las consultas y las inserciones contempladas en los cados de uso.

Resumen del alcance.

El alcance del laboratorio experimental es analizar las tecnologías de Backend y Frontend, con el propósito de comparar la eficiencia de varias arquitecturas orientadas a microservicios usando el patrón arquitectónico REST, con respecto a las métricas de tiempo medio de espera y tiempo medio de rendimiento contempladas en la norma ISO/IEC 25023; desde el punto de vista del investigador, en el contexto de un ambiente controlado.

1.2.2. Planificación

Una vez llevada a cabo la fase del alcance del experimento, se determina la planificación. La planificación tiene que ver sobre cómo se realiza el experimento, ya que como en toda actividad de ingeniería, la planificación del experimento debe realizarse y los

procedimientos deben seguirse, para garantizar un educado control sobre el proceso experimental.

Selección de contexto.

El experimento fue montado dentro un ambiente controlado limitado a una sola computadora portátil cuyas especificaciones de hardware se muestran en la Tabla 3. Cabe recalcar que el computador cumple con los requerimientos necesarios, pues permitió exitosamente la instalación del software preciso para el desarrollo de las 9 arquitecturas "eficientes" sometidas al laboratorio experimental con el propósito de compararlas.

Tabla 3: Especificaciones del hardware

ESPECIFICACIÓN	DETALLE
Nombre del equipo	Asus Rog Strix Hero Edition
Sistema Operativo	Windows 10 Pro
CPU	Intel Core i7 8th Generation
GPU	NVIDIA GeForce GTX 1050 Ti
RAM	32GB DDR4
Disco	128GB SSD – 1TB HDD

También, para la ejecución del experimento se necesitó una base de datos, en este caso se usó PostgreSQL para almacenar la información de usuarios, cabe mencionar que dicha información es hipotética, usada para efectos de la investigación. En la *Figura 6* se muestra el diseño del modelo lógico de la base datos que consta de 3 tablas:

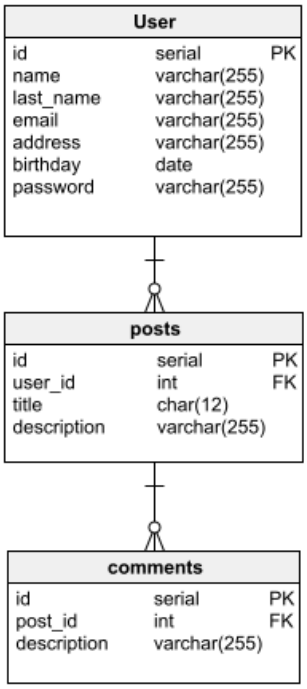


Figura 6. Modelo lógico de la base de datos

Formulación de la hipótesis.

Para la correcta definición del experimento, en la fase de planificación es necesario formalizar la hipótesis. Por ende, se formulan dos hipótesis:

Hipótesis nula (H_0): La eficiencia de todas las arquitecturas orientadas a microservicios que utilizan el patrón arquitectónico REST, es la misma.

Hipótesis alternativa (H_1): Es posible desarrollar una arquitectura orientada a microservicios con API-REST, que sea mayormente eficiente que otras arquitecturas.

Selección de variables.

Para poder diseñar los experimentos de laboratorio es necesario definir las variables dependientes e independientes, estas son:

Variables independientes:

- Las arquitecturas de software orientadas a microservicios en el desarrollo Backend.
- Las arquitecturas de software orientadas a microservicios en el desarrollo Frontend.

Variables dependientes:

- Tiempo medio de respuesta de arquitecturas orientadas a microservicios.
- Tiempo medio de rendimiento de arquitecturas orientadas a microservicios.

Selección de herramientas

Las herramientas se seleccionaron en función de la conveniencia, es decir, las herramientas que son apropiadas para desarrollar aplicaciones REST, basando dicha conveniencia en una de las principales plataformas de desarrollo colaborativo como lo es GitHub que permite alojar proyectos utilizando el sistema de control de versiones Git. Partiendo de que GitHub deja que los usuarios destaquen los repositorios con las herramientas más usadas y con el mejor desempeño, se pudo obtener una lista con las tecnologías más aceptadas por la comunidad de programadores de aplicaciones REST, tal y como se muestra en la Tabla 4.

Tabla 4: Selección de herramientas para el experimento

Tecnología	Valoración en GitHub	Último Commit
Laravel	65.7k[a]	2 de marzo de 2021[a]
Node.js	80k[b]	5 de marzo de 2021[b]
.Net Core	15.9k[c]	5 de marzo de 2021[c]
Angular	74.1k[d]	5 de marzo de 2021[d]
Entity Framework Core	10.3k[e]	5 de marzo de 2021[e]
npm	4.5k[f]	4 de marzo de 2021[f]
Express	53.5k[g]	28 de enero de 2021[g]
Sequelize	24.5k[h]	4 de marzo de 2021[h]
PostgreSQL	8.5k[i]	5 de marzo de 2021[i]

Nota. La valoración de cada herramienta tecnológica se muestra en miles de usuarios. [a]GitHub Inc. (s.f.-a). [b]GitHub Inc. (s.f.-b). [c]GitHub Inc. (s.f.-c). [d]GitHub Inc. (s.f.-d). [e]GitHub Inc. (s.f.-e). [f]GitHub Inc. (s.f.-f). [g]GitHub Inc. (s.f.-g). [h]GitHub Inc. (s.f.-h). [i]GitHub Inc. (s.f.-i). Fecha de revisión: 5 de marzo de 2021.

El software utilizado para el desarrollo del experimento es de código abierto y se lista a continuación:

- **NodeJS:** (OpenJS Foundation, 2020), en su sitio oficial describe a esta tecnología como un ambiente de ejecución para JavaScript montado con el motor de JavaScript V8 de Chrome. Se usa esta herramienta en los experimentos para el desarrollo del Backend, en su versión 14.16.0.
- **NPM:** es el gestor de paquetes de JavaScript, y se usa en este experimento para instalar las dependencias y agregar las librerías necesarias para la implementación de la API-REST en NodeJS, en su versión 6.14.11.
- **Express:** según el sitio oficial de OpenJS Foundation (2021), es un framework diseñado para crear aplicaciones web y API para Node.js. En los experimentos se usa esta tecnología para la construcción de la API-REST con NodeJS, en su versión 4.17.1.
- **Sequelize:** según (García, 2018), es un Object-Relational Mapping (ORM) de Node.js que se basa en promesas para PostgresSQL, MySQL, MariaDB, SQLite y Microsoft SQL Server. Sequelize se utiliza en estos experimentos con el fin de mapear las estructuras de la base de datos en el backend de NodeJS, en su versión 6.6.2.
- **Laravel:** (Baquero García, 2015)**,** lo describe como un framework para crear aplicaciones y servicios web con PHP utilizando una sintaxis refinada y expresiva, evitando el código espagueti. Esta herramienta se usa en los experimentos tanto para la construcción del backend y la API-REST con el lenguaje de programación de PHP, como para el desarrollo del Frontend, en su versión 8.0.7.
- **Eloquent:** según (Laravel LLC, 2011)**,** es el mapeador de objetos relacional (ORM) de Laravel y se utiliza en estos experimentos para que el backend de Laravel interactúe con la base de datos, en su versión 8.0.7.
- **.NET Core:** (Microsoft, 2021)**,** en su sitio oficial describe a esta tecnología como una herramienta multiplataforma de desarrollo para crear aplicaciones web, móviles y de escritorio. En los experimentos se utiliza esta tecnología para construir el backend y la

API-REST con el lenguaje de programación C#, así como para desarrollar el Frontend, en su versión .NET Core 3.1.

- **IIS Express:** (Microsoft, 2017) Internet Information Services (IIS) Express es una versión de IIS optimizada para desarrollar y probar sitios web. Esta tecnología se usa en los experimentos con el fin de servir como servidor de aplicaciones, en su versión 7.5.
- **Entity Framework Core:** (Microsoft, 2021), es el mapeador de bases de datos de .NET y se utiliza en los experimentos para mapear las estructuras de la base de datos en el backend de .NET, en su versión 3.1.15.
- **PostgreSQL:** (The PostgreSQL Global Development Group, 2021), lo describe como un poderoso sistema de base de datos relacional que usa el lenguaje SQL para almacenar grandes cargas de datos. Se usa en estos experimentos para almacenar la información de los usuarios, en su versión 12.7.
- **Angular:** según (Quality Devs, 2019), es un framework de desarrollo para JavaScript creado por Google para la creación y programación de aplicaciones web de una sola página (web SPA). Angular se utiliza en este experimento para la construcción del frontend de NodeJS, en su versión 11.2.7.

Diseño de experimentos.

Una vez definidas las variables dependientes e independientes, fue posible diseñar el experimento.

Para este laboratorio experimental, se plantearon 9 arquitecturas eficientes posibles, que resultaron de la combinación de los clientes (Angular, Laravel y .Net Core) con los proveedores API-REST (Node.js con JavaScript, Laravel con PHP y .Net Core con C#).

En la *Figura 7* se muestra de forma gráfica el experimento de laboratorio, para un mayor entendimiento, que evalúa el tiempo medio de espera para los primero 3 casos de uso y el

tiempo medio de rendimiento para el caso de uso 4, como métricas de la calidad externa de la norma ISO/IEC 25023.

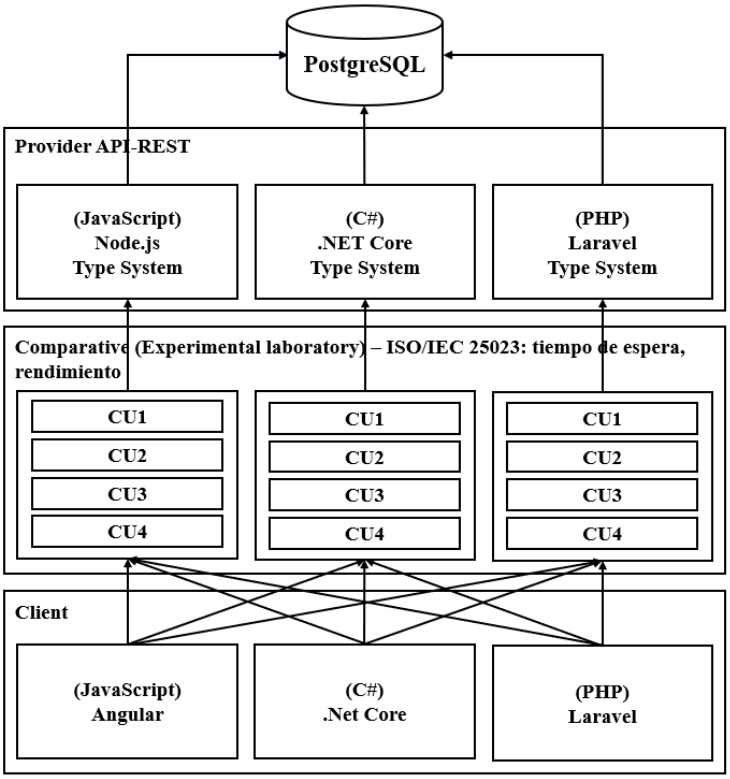


Figura 7. Diseño del experimento de laboratorio

Cabe recalcar que los primero tres casos de uso fueron diseñados en función de las consultas a la base de datos, es decir, el primer caso de uso accede a una sola tabla, el segundo a dos tablas y tercer caso de uso accede a las tres tablas de la base de datos, mientras que el cuarto caso de uso se diseñó en función de las inserciones a la base de datos. Para la ejecución de los casos de uso, se definió un único actor en el rol de INVESTIGADOR, quien fue el encargado de interactuar con la aplicación al realizar las consultas y las inserciones, con el fin de obtener los tiempos de respuesta y los tiempos de rendimiento de las diferentes arquitecturas propuestas.

A continuación, se detallan los casos de uso diseñados para este laboratorio experimental:

Caso de uso 1 (CU1): consulta a 1 tabla de la base de datos.

Al seleccionar este caso de uso, el INVESTIGADOR recolectará los tiempos de espera en los que dependiendo de la arquitectura que se esté evaluando, la consulta accederá a una sola tabla de la base de datos. El detalle de la consulta se muestra en la Tabla 5:

Tabla 5: *Detalle de la consulta del caso de uso 1*

Caso de uso 1	
Autor	Marcelo Martínez.
Fecha	22 de marzo de 2021.
Descripción	Se realiza una consulta a la tabla “users” de la base de datos.
Actor	INVESTIGADOR.
Precondiciones	La base de datos cuenta con datos de usuarios precargados.
Flujo normal	1. El INVESTIGADOR selecciona el número de registros o usuarios a consultar (1, 1000, 25000, 50000 o 100000). 2. El INVESTIGADOR selecciona el tipo de caso, para esta consulta el caso de uso 1. 3. El INVESTIGADOR ejecuta la consulta. 4. La aplicación comprueba la validez de la consulta. 5. La aplicación realizará la consulta del número de usuarios seleccionados en la tabla “users” de la base de datos. 6. La aplicación devuelve el tiempo de espera de la consulta ejecutada.
Flujo alternativo	En el caso de que la consulta no pueda ser procesada, la aplicación devuelve un mensaje al actor, permitiéndole repetir la prueba.

Caso de uso 2 (CU2): consulta a 2 tablas de la base de datos.

Al seleccionar este caso de uso, el INVESTIGADOR recolectará los tiempos de espera en los que dependiendo de la arquitectura que se esté evaluando, la consulta accederá a dos tablas de la base de datos. El detalle de la consulta se muestra en la Tabla 6:

Tabla 6: Detalle de la consulta del caso de uso 2

Caso de uso 2

Autor	Marcelo Martínez.
Fecha	22 de marzo de 2021.
Descripción	Se realiza una consulta a las tablas “users” y “posts” de la base de datos.
Actor	INVESTIGADOR.
Precondiciones	La base de datos cuenta con datos de usuarios precargados.
Flujo normal	1. El INVESTIGADOR selecciona el número de registros o usuarios a consultar (1, 1000, 25000, 50000 o 100000). 2. El INVESTIGADOR selecciona el tipo de caso, para esta consulta el caso de uso 2. 3. El INVESTIGADOR ejecuta la consulta. 4. La aplicación comprueba la validez de la consulta. 5. La aplicación realizará la consulta del número de usuarios seleccionados en las tablas “users” y “posts” de la base de datos. 6. La aplicación devuelve el tiempo de espera de la consulta ejecutada.
Flujo alternativo	En el caso de que la consulta no pueda ser procesada, la aplicación devuelve un mensaje al actor, permitiéndole repetir la prueba.

Caso de uso 3 (CU3): consulta a 3 tablas de la base de datos.

Al seleccionar este caso de uso, el INVESTIGADOR recolectará los tiempos de espera en los que dependiendo de la arquitectura que se esté evaluando, la consulta accederá a tres tablas de la base de datos. La tabla 7 muestra el detalle de la consulta.

Tabla 7: Detalle de la consulta del caso de uso 3

Caso de uso 3	
Autor	Marcelo Martínez.
Fecha	22 de marzo de 2021.
Descripción	Se realiza una consulta a las tablas “users”, “posts” y “comments” de la base de datos.
Actor	INVESTIGADOR.
Precondiciones	La base de datos cuenta con datos de usuarios precargados.

Flujo normal	1. El INVESTIGADOR selecciona el número de registros o usuarios a consultar (1, 1000, 25000, 50000 o 100000). 2. El INVESTIGADOR selecciona el tipo de caso, para esta consulta el caso de uso 3. 3. El INVESTIGADOR ejecuta la consulta. 4. La aplicación comprueba la validez de la consulta. 5. La aplicación realizará la consulta del número de usuarios seleccionados en las tablas “users” y “posts” de la base de datos. 6. La aplicación devuelve el tiempo de espera de la consulta ejecutada.
Flujo alternativo	En el caso de que la consulta no pueda ser procesada, la aplicación devuelve un mensaje al actor, permitiéndole repetir la prueba.

Caso de uso 4 (CU4): inserción de usuarios en la base de datos.

Al seleccionar este caso de uso, el INVESTIGADOR recolectará los tiempos de rendimiento al insertar usuarios en la tabla “users”. La tabla 8 muestra el detalle de la consulta.

Tabla 8: Detalle de las inserciones del caso de uso 4

Caso de uso 4	
Autor	Marcelo Martínez.
Fecha	22 de marzo de 2021.
Descripción	Se realiza la inserción de usuarios en la tabla “users” de la base de datos.
Actor	INVESTIGADOR.
Precondiciones	La base de datos cuenta con datos de usuarios precargados.
Flujo normal	1. El INVESTIGADOR selecciona el número de registros o usuarios a insertar (1, 100, 500 o 1000). 2. El INVESTIGADOR ejecuta la inserción. 3. La aplicación comprueba la validez de la inserción. 4. La aplicación realizará la inserción del número de usuarios seleccionados en la tabla “users” de la base de datos.

	5. La aplicación devuelve el tiempo de rendimiento de la inserción ejecutada.
Flujo alternativo	En el caso de que la inserción no pueda ser procesada, la aplicación devuelve un mensaje al actor, permitiéndole repetir la prueba.

Instrumentación.

Como instrumento de medición, se utiliza la norma ISO/IEC 25023, específicamente las métricas relacionadas a la característica de la eficiencia del desempeño y a la subcaracterística del comportamiento del tiempo. Las métricas son el tiempo medio de espera y el tiempo medio de rendimiento. Ver sección 1.5.

Evaluación de validez.

La evaluación de validez se garantizó gracias a que todo el proceso del laboratorio experimental se mantuvo bajo control, monitoreando las posibles amenazas. Un importante aspecto que influyo en la validez del experimento fue el contar con una arquitectura computacional de baja complejidad asegurando que las versiones de las herramientas utilizadas para su desarrollo fueran compatibles entre sí, lo que permitió un constante control y seguimiento a cada prueba realizada en los 4 casos de uso. Cabe mencionar, que cada una de las pruebas se realizaron una a la vez, lo que favoreció mayor control de la ejecución del experimento.

En cuanto al proceso de observación, lo llevo a cabo únicamente el investigador quien se aseguró de mantener los mismos criterios durante el desarrollo del experimento. Para la medición de los tiempos de cada métrica, se hizo uso del reloj de la arquitectura computacional, puesto que las métricas de calidad dependían del tiempo asegurando que fuse la misma fuente de datos para ambos casos.

1.2.3. Operación

Una vez diseñado y planificado el experimento, se procede a realizar la parte operativa de este, es decir, el cómo se llevó a cabo. Esta fase se divide en 3 partes, preparación, ejecución y validación de los datos:

Preparación.

En este parte se preparó todo lo necesario para la ejecución del laboratorio experimental, que abarco las siguientes tareas:

- Configuración del ambiente de desarrollo, que tiene que ver con la descarga, instalación y configuración de las herramientas de desarrollo seleccionadas.
- Programación de los clientes y los proveedores API-REST, para la ejecución de las pruebas relacionadas a los casos de uso.

Ejecución.

La ejecución del experimento se dividió en dos partes, la primera relacionada a la evaluación de la métrica del tiempo medio de espera que abarca los casos de uso 1, 2, y 3; y la segunda que corresponde a la evaluación del tiempo medio de rendimiento que concierne al caso de uso 4.

En la primera parte se ejecutaron 405 consultas en total, es decir; se obtuvieron 135 tiempos por cada caso de uso, registrando solamente 45 tiempos de espera como resultado de un promedio entre las 3 iteraciones realizadas por cada número de usuarios consultados a la base de datos. En la *Figura 8* se muestra el detalle de cómo se procedió con la ejecución de las consultas, poniendo de ejemplo el CU1, cuyo procedimiento es el mismo para los CU2 y CU3.

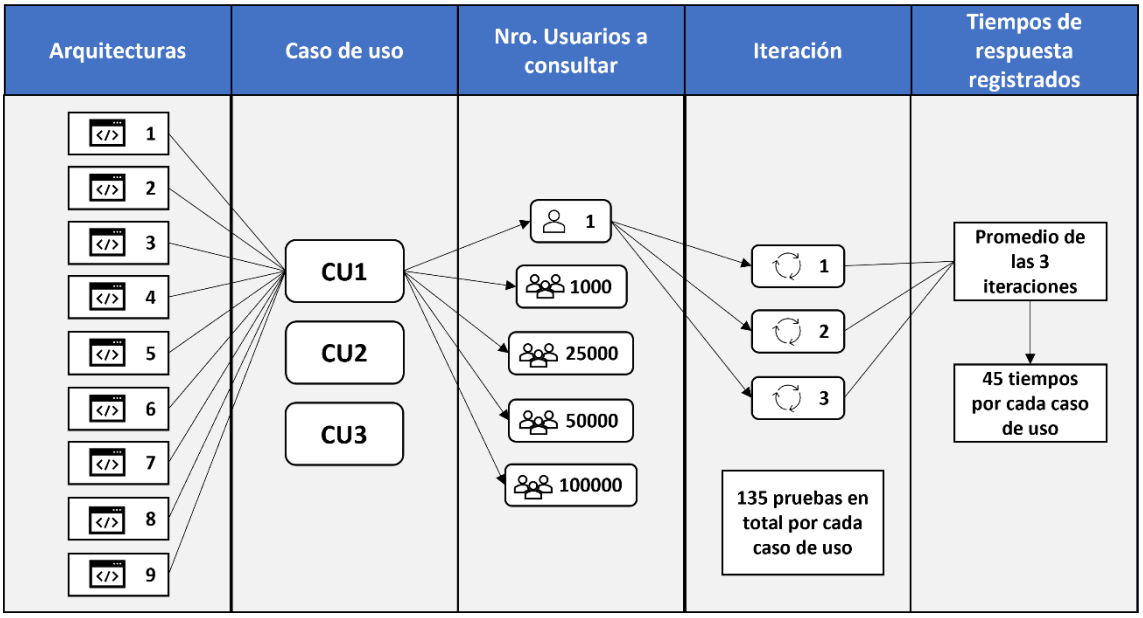


Figura 8. Esquema de la ejecución del experimento para los CU1, CU2 y CU3

Con respecto a la segunda parte del experimento, se ejecutaron 108 inserciones, registrando solamente 36 tiempos de rendimiento como resultado de un promedio entre las 3 iteraciones realizadas por cada número de usuarios ingresados en la base de datos. En la *Figura 9* se muestra en detalle cómo se procedió con la ejecución de las inserciones para el CU4.

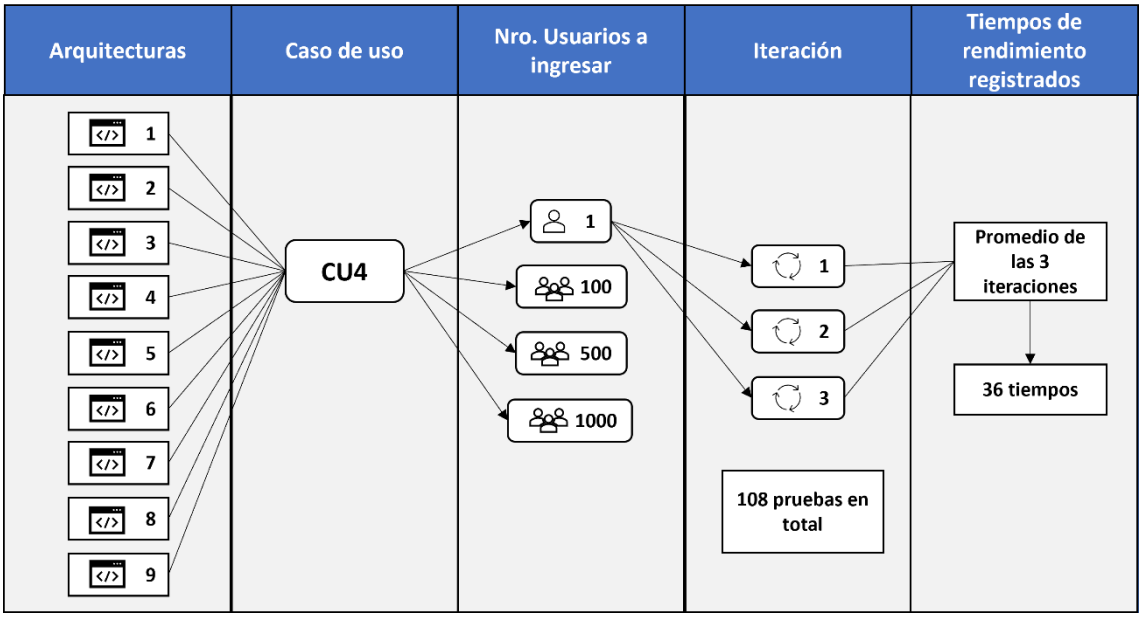


Figura 9. Esquema de la ejecución del experimento para el CU4

Validación de datos.

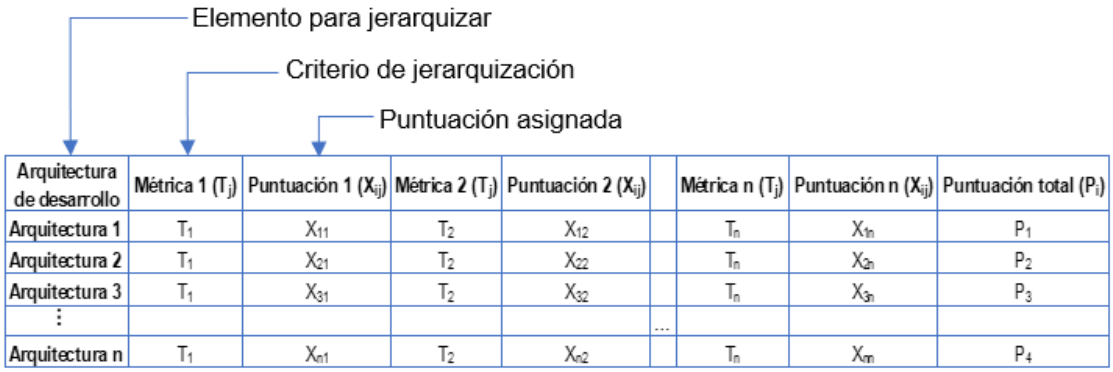
Después de la ejecución de las consultas y las inserciones se llevó a cabo la validación de los datos provenientes de cada uno de los caos de uso. Para ambos experimentos, todas las

solicitudes fueron procesadas exitosamente en cada uno de los casos de uso, dejando así los siguientes datos para su posterior análisis e interpretación de los resultados.

1.2.4. Análisis e interpretación

Tomando en cuenta el objetivo del experimento el cual estuvo encaminado en la elección de una arquitectura de desarrollo eficiente orientada a microservicios usando APIREST, se pudo medir la eficiencia de las 9 arquitecturas propuestas al compararlas tomando en cuenta el tiempo medio de espera y el tiempo medio de rendimiento, métricas de la norma ISO/IEC 25023.

Para poder seleccionar la arquitectura más eficiente se usó un método de comparación para poder jerarquizar o priorizar las arquitecturas. Es así como se empleó el modelo de puntuación, que permitió comparar las arquitecturas en una matriz de puntuaciones, tal y como se muestra en la *Figura 10*.



Arquitectura de desarrollo	Métrica 1 (T_j)	Puntuación 1 (X_{ij})	Métrica 2 (T_j)	Puntuación 2 (X_{ij})		Métrica n (T_j)	Puntuación n (X_{ij})	Puntuación total (P_i)
Arquitectura 1	T_1	X_{11}	T_2	X_{12}		T_n	X_{1n}	P_1
Arquitectura 2	T_1	X_{21}	T_2	X_{22}		T_n	X_{2n}	P_2
Arquitectura 3	T_1	X_{31}	T_2	X_{32}		T_n	X_{3n}	P_3
⋮					...			
Arquitectura n	T_1	X_{n1}	T_2	X_{n2}		T_n	X_{nn}	P_4

Figura 10. Matriz de puntuaciones
Fuente: Elaboración propia, basado en el modelo de puntuación según (Pacheco & Contreras, 2008)

Dicho modelo consiste en asignar una puntuación a cada elemento a jerarquizar (en este caso las 9 arquitecturas propuestas), de acuerdo con el criterio de jerarquización (en este caso las métricas de la norma ISO/IEC 25023). Empleando dichas ponderaciones se determina un puntaje único para cada arquitectura. Para ello se pueden emplear modelos aditivos, multiplicativos u otras funciones matemáticas con el fin de determinar la puntuación final de las arquitecturas (Pacheco & Contreras, 2008).

Con base en la matriz de puntuaciones se jerarquizo las arquitecturas frente a los criterios de jerarquización de los casos de uso. Se asignó 9 puntos a la arquitectura con el mejor tiempo, luego 8 puntos a la segunda y así sucesivamente hasta llegar a la arquitectura que tuvo el tiempo menos eficiente, a la cual se le asigno 1 punto; esto procedimiento se realizó por cada caso de uso. Cabe resaltar que, el mejor tiempo para los CU1, CU2, y CU3 que miden la eficiencia de acuerdo con la métrica del tiempo medio de espera, es el más cercano a 0, mientras que el mejor tiempo para el CU4 que mide la eficiencia de acuerdo con la métrica del tiempo medio de rendimiento, es el más lejano a 0, tal y como explica la norma ISO/IEC 25023, el valor deseado para cada métrica (ver Tabla 2 de la sección 1.5 del Capítulo I). Luego se usó un modelo aditivo para calcular el puntaje final de cada arquitectura utilizando la siguiente formula:

$$P_i = \sum_{j=1}^{n} X_{ij}$$

Donde: P_i = puntaje total de la arquitectura i, y X_{ij} = puntuación de la arquitectura i frente al criterio j.

Gracias al modelo de puntuaciones, se pudo seleccionar que las tecnologías más eficientes son Angular como framework de desarrollo para el Frontend y Node.js para el Backend, con un total de 36 puntos, tal y como se muestra en la *Figura 11*. El uso de las tecnologías anteriormente mencionadas es comprobado en la sección 2.4 de este capítulo, mediante una prueba de concepto.

RESULTADOS DEL LABORATORIO EXPERIMENTAL									
Arquitectura de Desarrollo	CASO DE USO 1		CASO DE USO 2		CASO DE USO 3		CASO DE USO 4		Puntuación Final
	Tiempo de espera (ms)	Puntuación	Tiempo de espera (ms)	Puntuación	Tiempo de espera (ms)	Puntuación	Tiempo de rendimiento (t/s)	Puntuación	
Angular & Laravel (PHP)	39895,4	3	246827,73	3	1510761,33	3	0,00298	3	12
Angular & .Net Core (C#)	937,67	6	4788,67	6	16030,27	6	0,04052	7	25
Angular & Node.js (JS)	372,53	9	2482,87	9	10611,8	9	0,10394	9	36
Laravel & Laravel (PHP)	42295,53	2	258446,4	2	1517152,07	2	0,00242	2	8
Laravel & .Net Core (C#)	1690,67	5	4979,73	5	16541,13	5	0,03134	6	21
Laravel & Node.js (JS)	527,93	7	2664,27	8	12314,93	7	0,04361	8	30
.Net Core & Laravel (PHP)	42896,8	1	263489,33	1	1529588,07	1	0,00042	1	4
.Net Core & .Net Core (C#)	1847,8	4	8095,67	4	25131,67	4	0,017	4	16
.Net Core & Node.js (JS)	391,6	8	2716,13	7	11351,53	8	0,02478	5	28

Arquitectura seleccionada, conformada por Angular y NodeJS

Figura 11. Jerarquización de las arquitecturas en base a la matriz de puntuaciones

Cabe mencionar que, las arquitecturas conformadas por Node,js como Backend obtuvieron las primeras posiciones en la jerarquización final, mientras que las arquitecturas que estuvieron conformadas por el Backed de Laravel con el lenguaje de programación PHP, obtuvieron los últimos puestos, tal y como se muestra en la Tabla 9:

Tabla 9: Jerarquización final de las arquitecturas

Arquitectura		Puntuación Total	Posición en la Jerarquización Final
Frontend	**Backend**		
Angular	Node.js (JS)	36	1
Laravel	Node.js (JS)	30	2
.Net Core	Node.js (JS)	28	3
Angular	.Net Core (C#)	25	4
Laravel	.Net Core (C#)	21	5
.Net Core	.Net Core (C#)	16	6
Angular	Laravel (PHP)	12	7
Laravel	Laravel (PHP)	8	8
.Net Core	Laravel (PHP)	4	9

1.3. Desarrollar Arquitectura eficiente con API-REST

En base al laboratorio experimental realizado, al análisis, comparación y jerarquización de las diferentes arquitecturas propuestas, se pudo desarrollar una arquitectura eficiente de acuerdo con las métricas de la norma ISO/IEC 25023, dicha arquitectura esta conforma de las tecnologías que se muestran en la *Figura 12*:

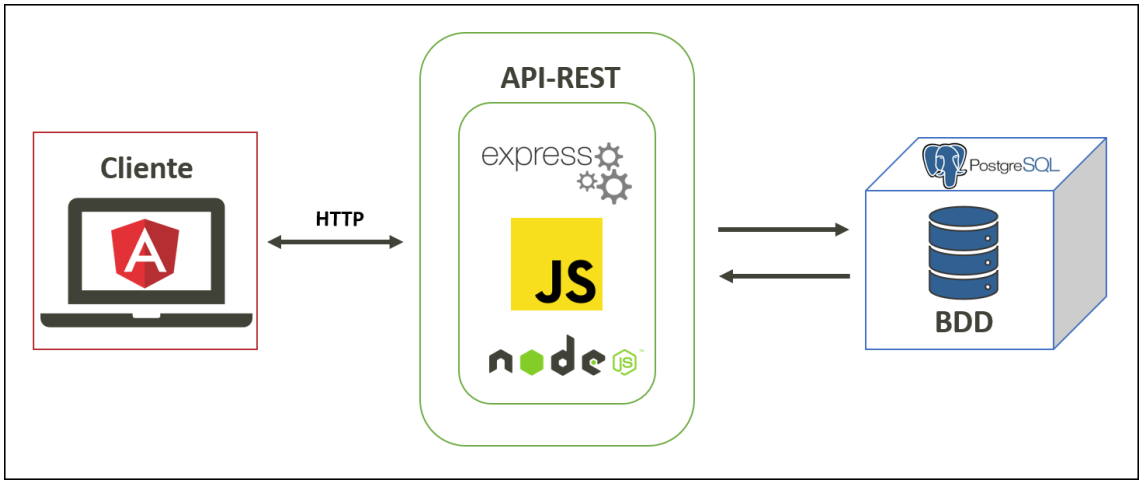


Figura 12. Arquitectura eficiente con APIREST

Primeramente, para esta arquitectura eficiente se tiene a Angular como Framework de desarrollo de parte del cliente, Node.js y Express para la construcción de la APIREST con el lenguaje de programación de JavaScript y PostgreSQL como base de datos para almacenar la información necesaria para la ejecución de los experimentos.

Cabe recalcar que la arquitectura eficiente conformada por las tecnologías antes mencionadas se validará con una prueba de concepto. Ver la sección 2.4.

1.4. Validar Arquitectura con una prueba de concepto

En esta sección, se describe el desarrollo de la prueba de concepto pertinente para la validación de la Arquitectura de desarrollo eficiente, como resultado del laboratorio experimental planteado en la sección 2.2.

1.4.1. Desarrollo de la prueba de concepto.

Para la prueba de concepto, primeramente, se procedió levantando los requisitos de software plasmados en historias de usuario, el diseño de los mockups de las vistas, el desarrollo de la prueba de concepto en donde se definen los roles, los tiempos y las pantallas funcionales de la arquitectura desarrollo eficiente, y por último se plantean las pruebas de aceptación pertinentes al desarrollo de la prueba de concepto.

Requisitos.

Los requisitos de la prueba de concepto se plasman en historias de usuario (HU) que, en este caso se definen 2, una HU para los CU1, CU2 y CU y otra para el CU4.

En la Tabla 10 se muestra la HU1 relacionada con la métrica del tiempo de espera que permitió realizar consultas a la base de datos:

Tabla 10: Historia de usuario 1 (HU1)

VALIDACIÓN DEL TIEMPO DE ESPERA			
Número	1	**Usuario**	Investigador
Prioridad	Alta	**Puntos Estimados**	1
Riesgo	Bajo	**Iteración**	1
Descripción: El investigador podrá validar la arquitectura de desarrollo eficiente resultante del laboratorio experimental conformada por Angular y Node.js, al realizar consultas a la base de datos, esto con el fin de registrar el tiempo de espera de la consulta en cuestión. Para ello, el investigador debe: 1) Seleccionar la pestaña "Query" en la esquina superior derecha de la barra de navegación de la vista; 2) Seleccionar el número de registros o usuarios a consultar (se podrá elegir 1, 1000, 25000, 50000 o 100000); 3) Seleccionar el caso de uso (se podrá elegir entre el CU1, CU2, CU3); 4) Presionar el botón "Execute Query" para ejecutar la consulta.			
Criterios de aceptación: Para considerar la prueba como válida, no debe haber evidencia alguna de que hubo algún error mientras se ejecutaba la consulta, para esto el tiempo de espera de la consulta debe mostrarse en la tabla de tiempos de la vista al finalizar la consulta.			

En la Tabla 11 se muestra la HU2 relacionada con la métrica del tiempo de rendimiento que permitió realizar inserciones en la base de datos:

Tabla 11: Historia de usuario 2 (HU2)

VALIDACIÓN DEL TIEMPO DE RENDIMIENTO			
Número	2	**Rol:**	Investigador
Prioridad	Alta	**Puntos Estimados**	1
Riesgo	Bajo	**Iteración**	1
Descripción: El investigador podrá validar la Arquitectura de desarrollo eficiente resultante del laboratorio experimental conformada por Angular y Node.js, al realizar inserciones en la base de datos, esto con el fin de registrar el tiempo de rendimiento de la inserción en cuestión. Para ello, el investigador debe: 1) Seleccionar la pestaña "Insert" en			

la esquina superior derecha de la barra de navegación de la vista; 2) Seleccionar el número de registros o usuarios a consultar (se podrá elegir 1, 100, 500 o 1000); 3) Presionar el botón “Execute Insert” para ejecutar la inserción.
Criterios de aceptación: Para considerar la prueba como válida, no debe haber evidencia alguna de que hubo algún error mientras se ejecutaba la inserción, para esto el tiempo de rendimiento de la inserción debe mostrarse en la tabla de tiempos de la vista al finalizar la consulta.

Diseño.

A continuación, se muestran las maquetas de cómo quedarían las vistas de la aplicación una vez desarrolladas en base a la arquitectura de desarrollo eficiente conformada por Angular & Node.js:

Mockup de la vista para los CU1, CU2 y CU3, para realizar consultas.

La vista contiene los siguientes elementos:

- **# Records:** es un combo box que permite seleccionar el número de usuario o registros que se desea consultar a la base de datos, en este caso las opciones a escoger son: 1, 1000, 25000, 50000 y 100000.
- **Use Case:** es un combo box que permite seleccionar el caso de uso para la consulta.
- **Execute Query:** es un botón que permite ejecutar la consulta y procesar la información para posteriormente mostrarla.
- **Tabla de datos:** aquí es donde se podrá visualizar la información del caso de uso seleccionado, el número de registros procesados y el tiempo de espera que tardó en completar la solicitud.

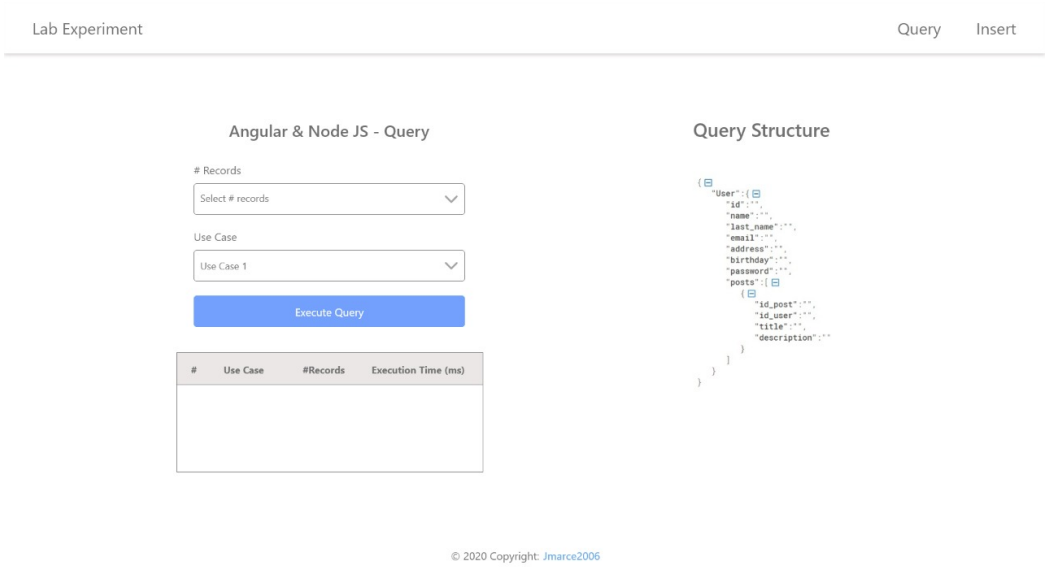


Figura 13. Mockup de la vista requerida para ejecutar consultas

Mockup de la vista para el CU4, para realizar inserciones.

La vista contiene los siguientes elementos:

- **# Records:** permite seleccionar el número de usuarios o registros que se desea ingresar en la base de datos, en este caso los registros a escoger son: 1, 100, 500, y 1000.
- **Execute Query:** es un botón que permite ejecutar la inserción y procesar la información para posteriormente mostrarla.
- **Tabla de datos:** aquí es donde se podrá visualizar la información del número de usuarios insertados y el tiempo de rendimiento de la inserción.

Figura 14. Mockup de la vista requerida para ejecutar inserciones

Roles.

Para esta prueba de concepto se definió un solo rol, en este caso el investigador, cuyo trabajo es ejecutar la prueba de concepto, es decir, es la persona que lleva a cabo las historias de usuario.

Pantallas.

En esta sección se presentas las vistas funcionales para la validación de la Arquitectura de desarrollo eficiente conformada por Angular & NodeJS.

En la *Figura 15* se puede apreciar la vista correspondiente a los CU1, CU2 y CU3, que permite al investigador realizar las siguientes acciones para poder validar la arquitectura con respecto a la métrica del tiempo de espera:

- Seleccionar el número de registros o usuarios a consultar, en este caso: 1, 1000, 25000, 50000 o 100000 usuarios.
- Seleccionar el caso de uso, el investigador podrá elegir entre el CU1, CU2 o CU3.
- Presionar el botón para ejecutar la consulta de usuarios.

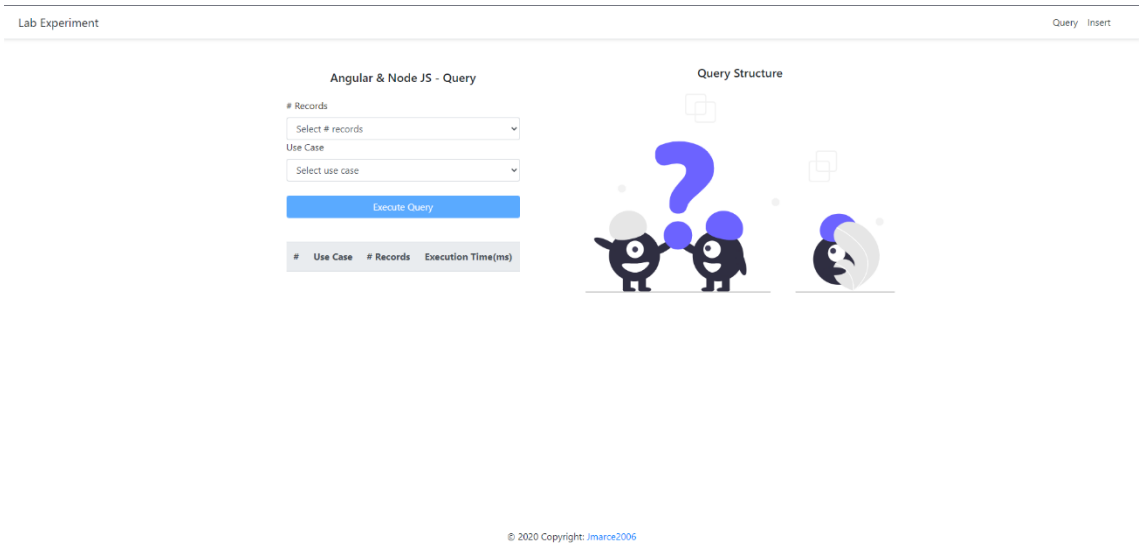


Figura 15. Vista funcional para la ejecución de consultas

En la *Figura 16 s*e puede apreciar la vista correspondiente al CU4, que permite al investigador realizar las siguientes acciones para validar la arquitectura con respecto a la métrica del tiempo de rendimiento:

- Seleccionar el número de registros o usuarios a insertar, en este caso: 1, 100, 500, o 1000 usuarios.
- Presionar el botón para ejecutar la inserción de usuarios.

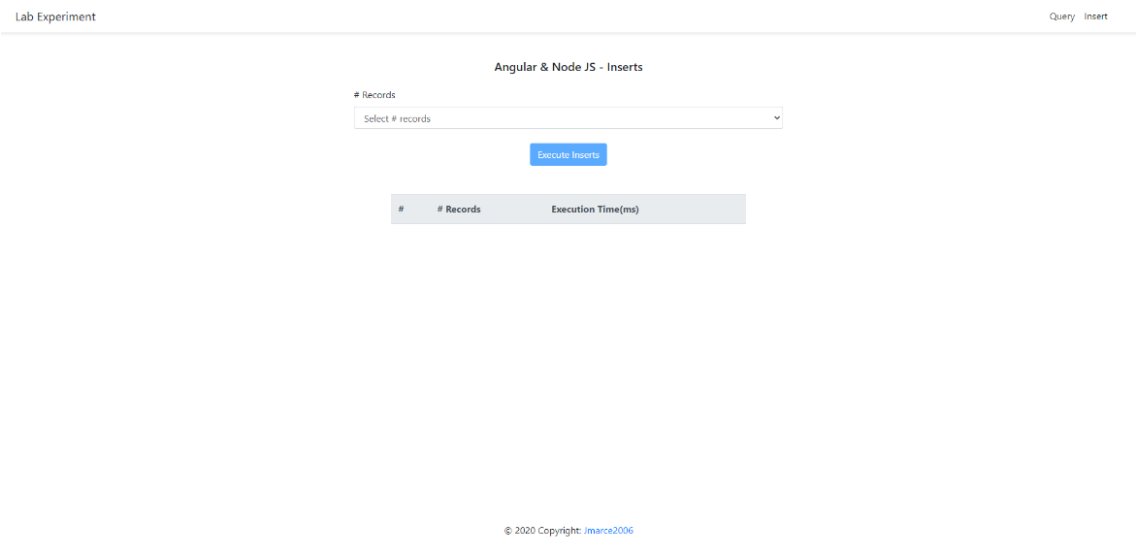


Figura 16. Vista funcional para la ejecución de inserciones

Criterios de aceptación (pruebas de aceptación de las historias de usuario).

Con la finalidad de comprobar que la arquitectura cumple con los requerimientos planteados en las historias de usuario, no hubo error alguno durante la ejecución de las consultas y las inserciones, es decir, todas las pruebas realizadas debieron ser cumplidas satisfactoriamente y así validar que la Arquitectura de desarrollo conformada por Angular & NodeJS es eficiente.

A continuación, se muestran las pruebas de aceptación de la HU1 que esta familiarizada con la métrica del tiempo de espera:

En la *Figura 17*, se muestra la prueba de aceptación del caso de uso 1, en donde se puede verificar que las consultas han sido ejecutas satisfactoriamente sin error alguno. Los 5 tiempos de espera que se pueden visualizar, pertenecen uno por cada número de registros o usuarios que la vista permite seleccionar, para este caso se tiene:

- 64 ms al consultar 1 usuario.
- 60 ms al consultar 1000 usuarios.

- 402 ms al consultar 25000 usuarios.
- 523 ms al consultar 500000 usuarios.
- 1738 ms al consultar 100000 usuarios.

También, en el gráfico de la derecha de la vista, se puede apreciar la estructura que se está consultando al acceder a 1 tabla de la base de datos, en este caso a la tabla “users”.

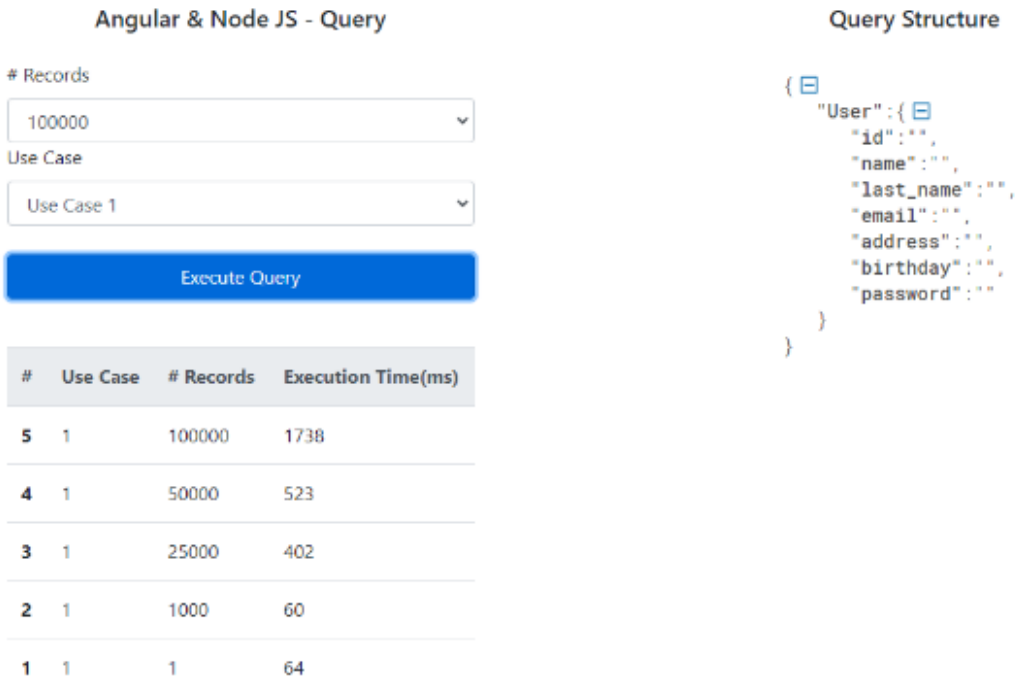


Figura 17. Prueba de aceptación para el Caso de uso 1

En la *Figura 18*, se muestra la prueba de aceptación del caso de uso 2, en donde se puede verificar que las consultas han sido ejecutas satisfactoriamente sin error alguno. Los 5 tiempos de espera que se pueden visualizar, pertenecen uno por cada número de registros o usuarios que la vista permite seleccionar, para este caso se tiene:

- 160 ms al consultar 1 usuario.
- 286 ms al consultar 1000 usuarios.
- 1966 ms al consultar 25000 usuarios.
- 3839 ms al consultar 500000 usuarios.
- 4739 ms al consultar 100000 usuarios.

También, en el gráfico de la derecha de la vista, se puede apreciar la estructura que se está consultando al acceder a 2 tablas de la base de datos, en este caso a las tablas “users” y “post”.

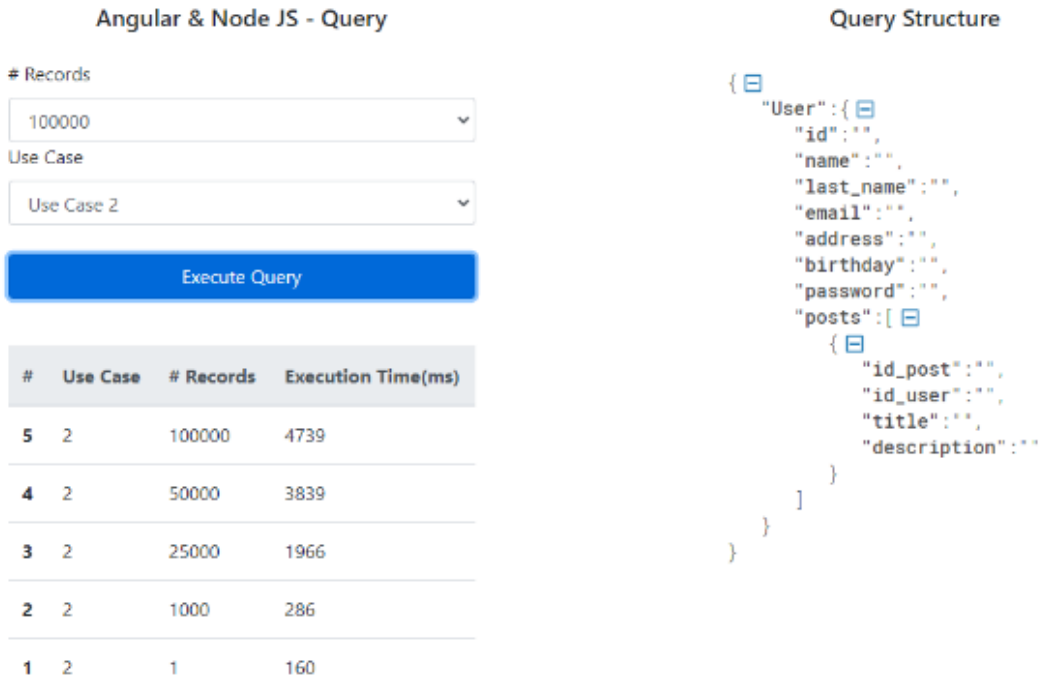


Figura 18. Prueba de aceptación para el Caso de uso 2

En la *Figura 19*, se muestra la prueba de aceptación del caso de uso 3, en donde se puede verificar que las consultas han sido ejecutas satisfactoriamente sin error alguno. Los 5 tiempos de espera que se pueden visualizar, pertenecen uno por cada número de registros o usuarios que la vista permite seleccionar, para este caso se tiene:

- 520 ms al consultar 1 usuario.
- 631 ms al consultar 1000 usuarios.
- 2676 ms al consultar 25000 usuarios.
- 16040 ms al consultar 500000 usuarios.
- 32403 ms al consultar 100000 usuarios.

También, en el gráfico de la derecha de la vista, se puede apreciar la estructura que se está consultando al acceder a 3 tablas de la base de datos, en este caso a las tablas “users”, “post” y “comments”.

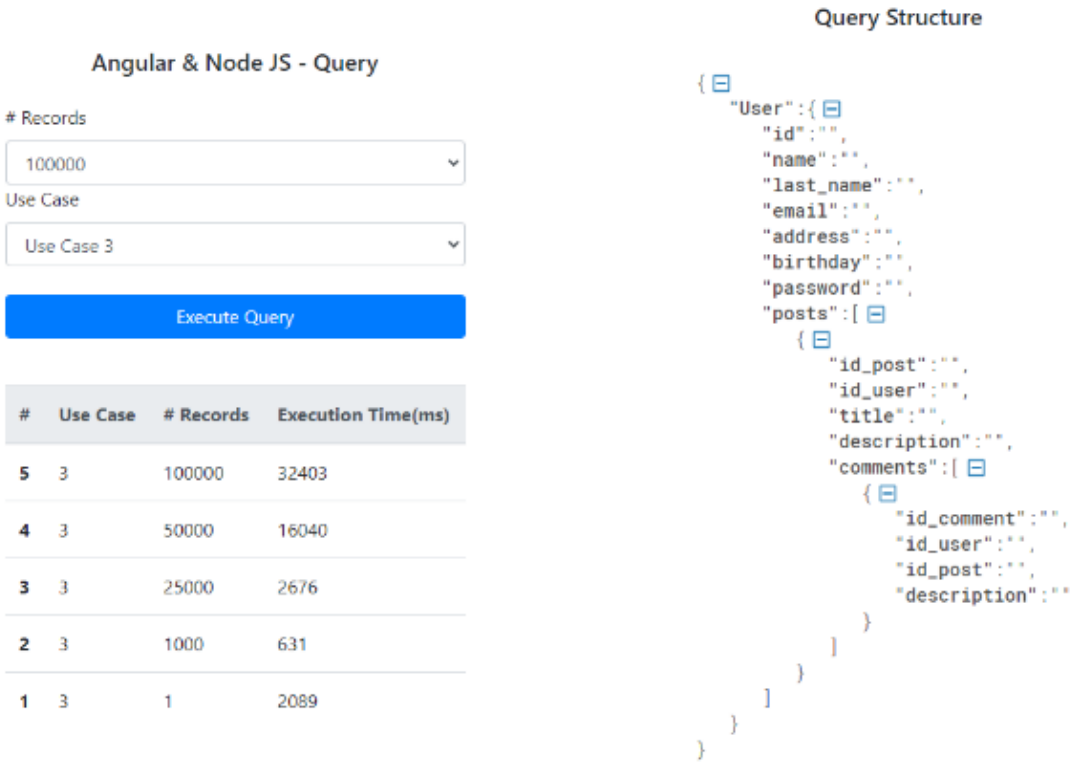


Figura 19. Prueba de aceptación para el Caso de uso 3

En la *Figura 20*, se muestra la prueba de aceptación del caso de uso 3, en donde se puede verificar que, las consultas han sido ejecutas satisfactoriamente sin error alguno. Los 4 tiempos de rendimiento que se pueden visualizar, pertenecen uno por cada número de registros o usuarios que la vista permite seleccionar, en este caso se tiene:

- 13 ms al ingresar 1 usuario.
- 311 ms al ingresar 100 usuarios.
- 1300 ms al ingresar 500 usuarios.
- 2836 ms al ingresar 1000 usuarios.

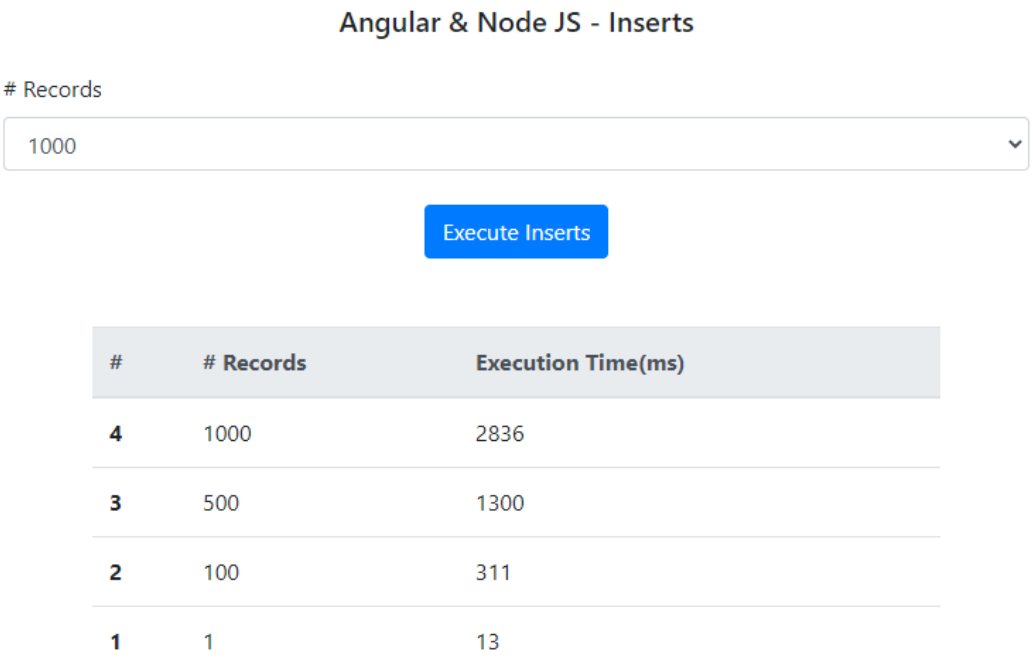


Figura 20. Prueba de aceptación para el Caso de uso 4

1.4.2. Conclusión de la validación.

Con el desarrollo de la prueba de concepto, se pudo validar el uso de las tecnologías de la Arquitectura de desarrollo conformada por Angular & NodeJS, al cumplir con las pruebas de aceptación de las historias de usuario, ya que no hubo error alguno al ejecutar dichas pruebas que consideraban las variables del tiempo de espera y el rendimiento de la arquitectura en cuestión.

CAPÍTULO III

Resultados

3.1. Análisis y evaluación de resultados

En este capítulo se analiza a profundidad los datos obtenidos de la del laboratorio experimental realizado en el capítulo 2 de esta investigación. Se evalúa e interpreta los tiempos derivados de la ejecución de las pruebas de las métricas de la calidad externa de la norma ISO/IEC 25023, específicamente las métricas del tiempo de espera y el tiempo de rendimiento. Se hace un análisis de las tecnologías más eficientes tanto para el Backend como para el Frontend y, por último, se comprueba si las hipótesis definidas para efecto del experimento resultaron ser correctas o no.

3.1.1. Resultados de Tiempo de espera

El tiempo de espera permitió recolectar la información de los tiempos de espera de los recursos de las Arquitecturas de desarrollo que se plantearon, es decir, el tiempo desde que se envía la solicitud, para que se ejecute la consulta, hasta que se completa. El valor deseado, según la métrica de la norma ISO/IEC 25023, el más cercano a cero es el mejor, es decir, la arquitectura que registre el menor tiempo es la más eficiente.

Esta primera parte del experimento se divide en 3 casos de uso y en cada uno de ellos se realizan peticiones GET vía HTTP a los *end points* de cada servicio para consultar la información de los usuarios que se encuentra almacenada en la Bade de Datos de PostgreSQL. Se han planteado 5 escenarios para los casos de uso 1, 2, y 3 con diferente número de registros a consultar tal y como se muestra en la Tabla 12, con el fin de poder recolectar la información suficiente para analizarla e interpretarla. Es de mencionar que el experimento en general, se lo realizo con el fin de verificar qué Arquitectura de desarrollo es más eficiente, ya que el rendimiento de los recursos se ve afectado dependiendo la cantidad de datos procesados y eso

es posible verificar comparando con los resultados de los 3 casos de uso planteados en esta primera parte del experimento.

Tabla 12: Detalle de los escenarios de los CU1, CU2, y CU3

Tipo	Escenario	Nro. Registros
Query	Escenario 1	1
	Escenario 2	1000
	Escenario 3	25000
	Escenario 4	50000
	Escenario 5	100000

Caso de uso 1.

Para este caso de uso, se consulta la información de los usuarios con cada uno de los escenarios planteados tal y como se muestra en la *Figura 21*.

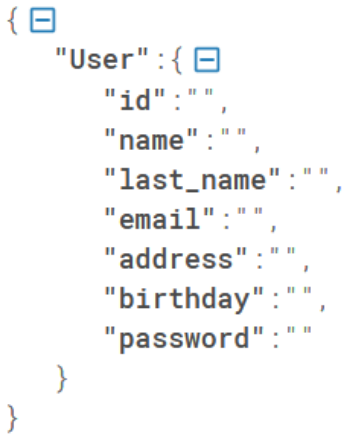


Figura 21. Detalle de la consulta para el Caso de uso 1

Escenario 1.

En este escenario, como se muestra en la Tabla 13, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 1 registro y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.

- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de respuesta son tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 22*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 32 milisegundos al ejecutar una petición GET de 1 usuario.

Tabla 13: Análisis comparativo del tiempo de espera – Escenario 1 – Caso de uso 1

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	263 ms	86 ms	30 ms	451 ms	57 ms	49 ms	520 ms	85 ms	54 ms
Prueba 2	268 ms	79 ms	35 ms	430 ms	69 ms	64 ms	505 ms	91 ms	55 ms
Prueba 3	325 ms	62 ms	31 ms	516 ms	71 ms	55 ms	516 ms	81 ms	58 ms
Promedio	285,33 ms	75,67 ms	32 ms	465,67 ms	65,67 ms	56 ms	513,67 ms	85,67 ms	55,67 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

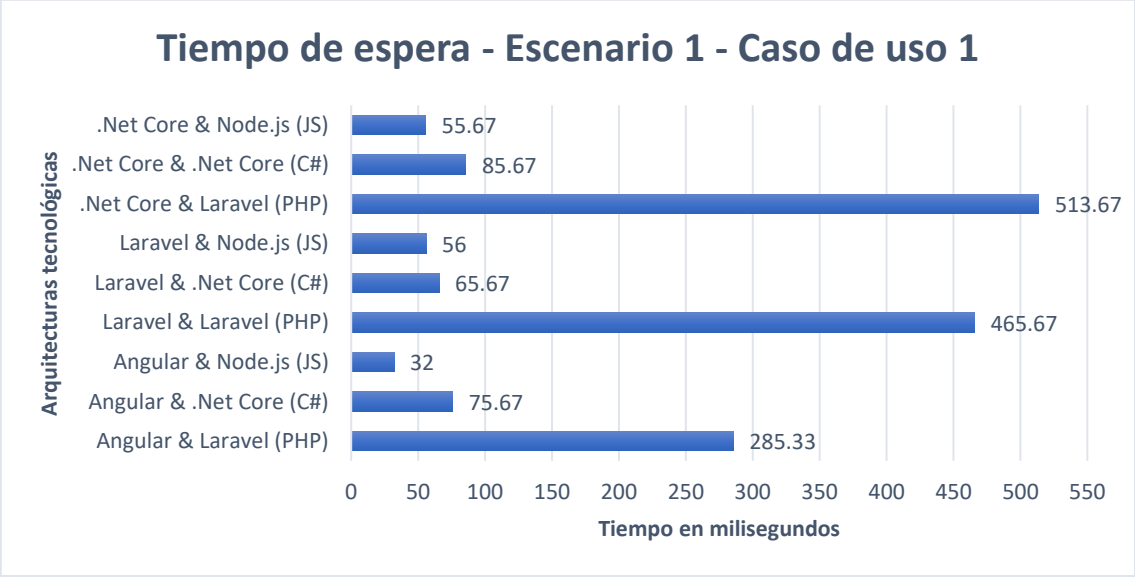


Figura 22. Tiempo de espera de la ejecución de las consultas – Escenario 1 – Caso de uso 1

Escenario 2.

En este escenario, como se muestra en la Tabla 14, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 1000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 23*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 68,67 milisegundos al ejecutar una petición GET de 1000 usuarios.

Tabla 14: Análisis comparativo del tiempo de espera – Escenario 2 – Caso de uso 1

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	1783 ms	213 ms	67 ms	1608 ms	138 ms	156 ms	1689 ms	173 ms	69 ms
Prueba 2	1524 ms	187 ms	69 ms	1661 ms	113 ms	156 ms	1630 ms	117 ms	71 ms
Prueba 3	1743 ms	178 ms	70 ms	1605 ms	107 ms	155 ms	1657 ms	135 ms	73 ms
Promedio	1683,33 ms	192,67 ms	68,67 ms	1624,67 ms	119,33 ms	155,67 ms	1658,67 ms	141,67 ms	71 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

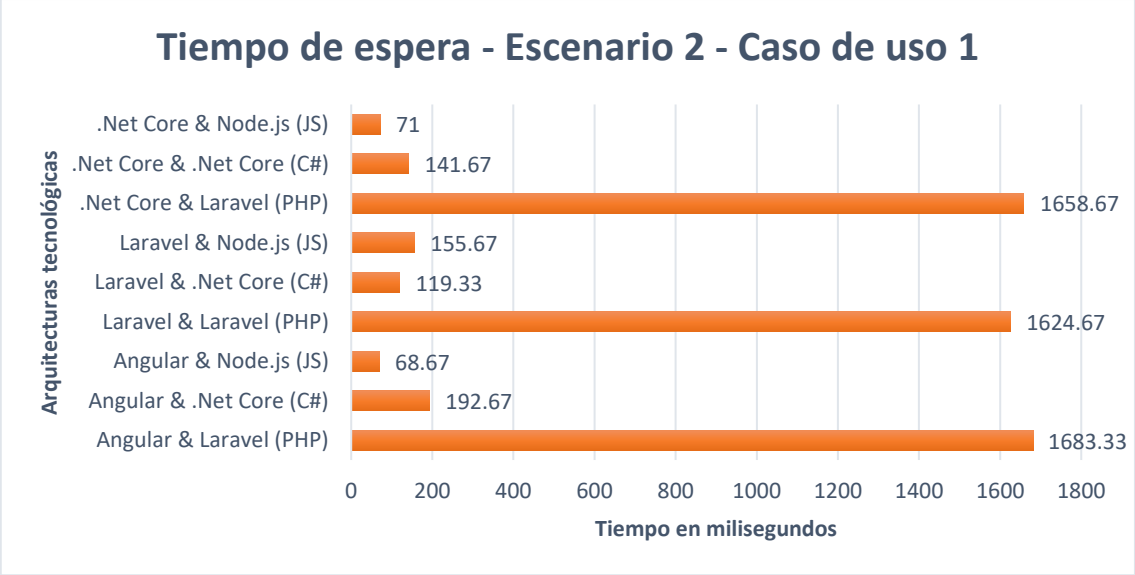


Figura 23. Tiempo de espera de la ejecución de las consultas – Escenario 2 – Caso de uso 1

Escenario 3.

En este escenario, como se muestra en la Tabla 15, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 25000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 24*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 283 milisegundos al ejecutar una petición GET de 25000 usuarios.

Tabla 15: Análisis comparativo del tiempo de espera – Escenario 3 – Caso de uso 1

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	27788 ms	728 ms	285 ms	29053 ms	1065 ms	369 ms	30189 ms	1313 ms	360 ms
Prueba 2	27663 ms	723 ms	269 ms	31064 ms	1033 ms	430 ms	31512 ms	1384 ms	343 ms
Prueba 3	27798 ms	732 ms	295 ms	31196 ms	1254 ms	372 ms	30361 ms	1320 ms	314 ms
Promedio	27749,67 ms	727,67 ms	283 ms	30437,67 ms	1117,33 ms	390,33 ms	30687,33 ms	1339 ms	339 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

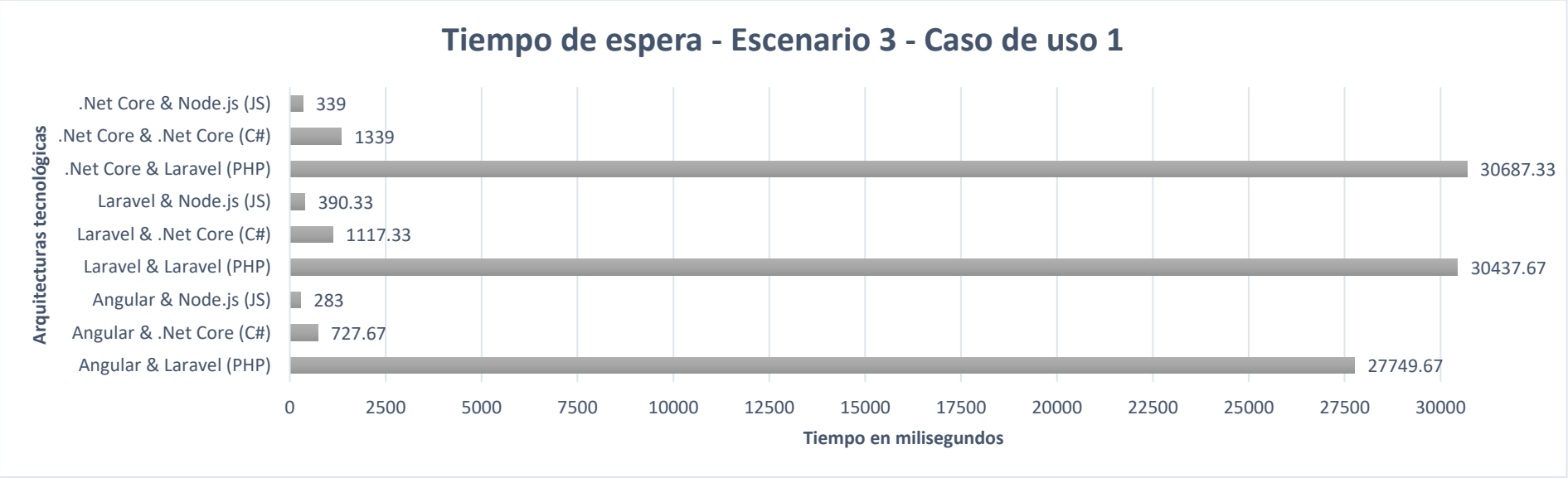


Figura 24. Tiempo de espera de la ejecución de las consultas – Escenario 3 – Caso de uso 1

Escenario 4.

En este escenario, como se muestra en la Tabla 16, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 50000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 25*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 429,67 milisegundos al ejecutar una petición GET de 50000 usuarios.

Tabla 16: Análisis comparativo del tiempo de espera – Escenario 4 – Caso de uso 1

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	56880 ms	1337 ms	420 ms	60192 ms	2528 ms	761 ms	59452 ms	2559 ms	526 ms
Prueba 2	55960 ms	1402 ms	414 ms	57939 ms	2460 ms	766 ms	61123 ms	2562 ms	502 ms
Prueba 3	56500 ms	1312 ms	455 ms	58833 ms	2376 ms	724 ms	60975 ms	2595 ms	487 ms
Promedio	56446,67 ms	1350,33 ms	429,67 ms	58988 ms	2454,67 ms	750,33 ms	60516,67 ms	2572 ms	505 ms

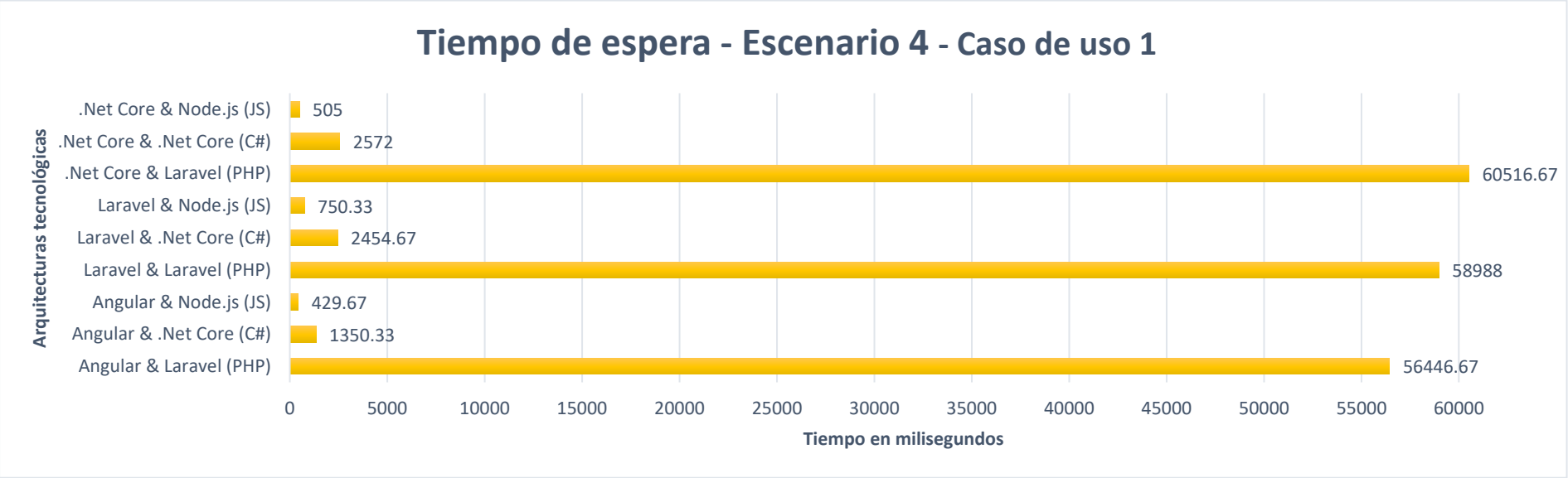


Figura 25. Tiempo de espera de la ejecución de las consultas – Escenario 4 – Caso de uso 1

Escenario 5.

En este escenario, como se muestra en la Tabla 10, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 100000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 25*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 694,67 milisegundos al ejecutar una petición GET de 100000 usuarios.

Tabla 17: Análisis comparativo del tiempo de espera – Escenario 5 – Caso de uso 1

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	110182 ms	2357 ms	999 ms	119318 ms	4697 ms	1288 ms	120879 ms	5078 ms	1041 ms
Prueba 2	119185 ms	2313 ms	941 ms	118763 ms	4753 ms	1299 ms	121453 ms	4914 ms	948 ms
Prueba 3	110569 ms	2356 ms	954 ms	121804 ms	4639 ms	1275 ms	120991 ms	5310 ms	973 ms
Promedio	113312 ms	2342 ms	964,67 ms	119961,67 ms	4696,33 ms	1287,33 ms	121107,67 ms	5100,67 ms	987,33 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

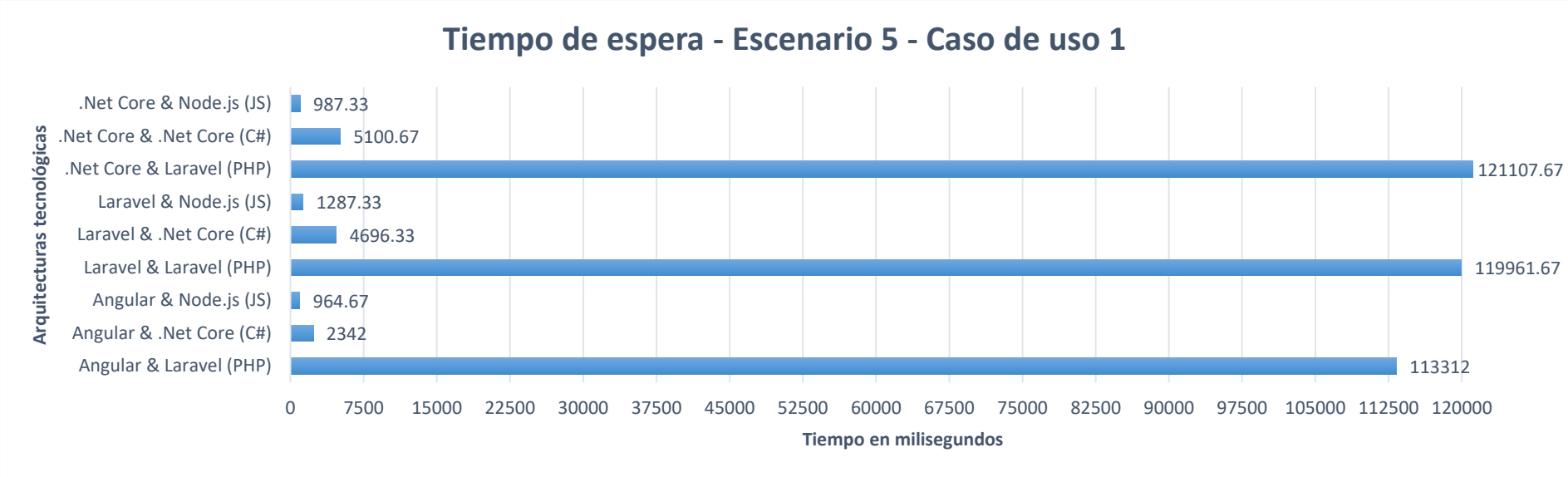


Figura 26. Tiempo de espera de la ejecución de las consultas – Escenario 5 – Caso de uso 1

Caso de uso 2.

Para este caso de uso, se ejecuta la misma consulta del primer caso de uso con cada uno de los escenarios planteados, con la diferencia de que se agregan *posts* y estos con sus respectivos parámetros tal y como se muestra en la *Figura 27*.

Figura 27. Detalle de la consulta para el Caso de uso 2
Fuente: Elaboración propia

Escenario 1.

En este escenario, como se muestra en la Tabla 18, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 1 registro y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.

- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser tan eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 28*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 167 milisegundos al ejecutar una petición GET de 1 usuario.

Tabla 18: Análisis comparativo del tiempo de espera – Escenario 1 – Caso de uso 2

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	423 ms	189 ms	165 ms	531 ms	242 ms	223 ms	542 ms	221 ms	199 ms
Prueba 2	428 ms	183 ms	167 ms	526 ms	235 ms	220 ms	513 ms	239 ms	205 ms
Prueba 3	459 ms	180 ms	169 ms	491 ms	237 ms	219 ms	498 ms	234 ms	200 ms
Promedio	436,67 ms	184 ms	167 ms	516 ms	238 ms	220,67 ms	517,67 ms	231,33 ms	201,33 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

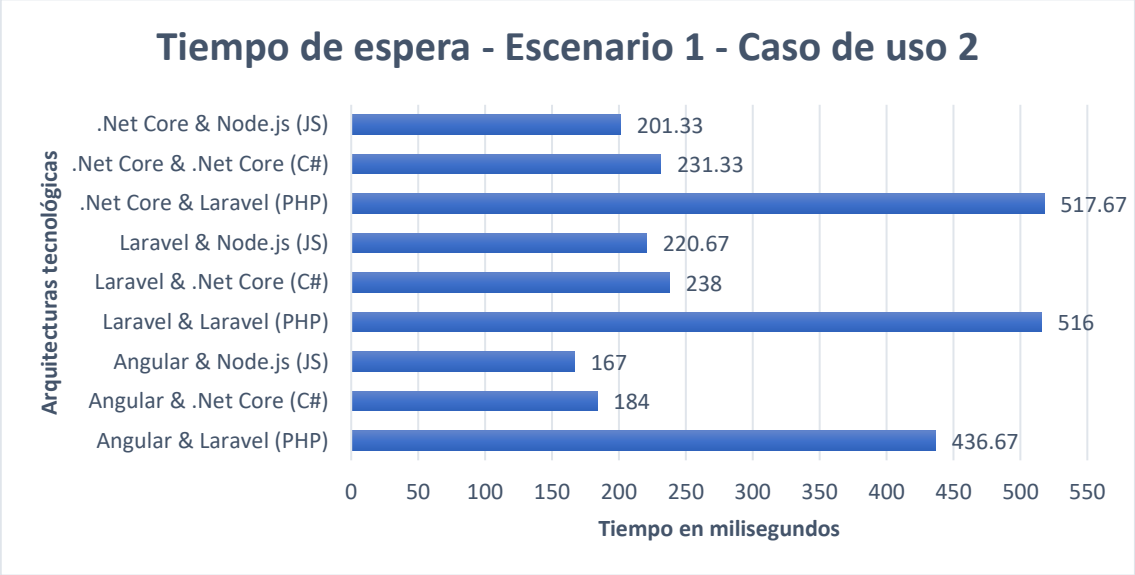


Figura 28. Tiempo de espera de la ejecución de las consultas – Escenario 1 – Caso de uso 2

Escenario 2.

En este escenario, como se muestra en la Tabla 19, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 1000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 29*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 274,67 milisegundos al ejecutar una petición GET de 1000 usuarios.

Tabla 19: Análisis comparativo del tiempo de espera – Escenario 2 – Caso de uso 2

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	7925 ms	305 ms	279 ms	8071 ms	474 ms	300 ms	8210 ms	488 ms	317 ms
Prueba 2	8077 ms	300 ms	275 ms	7852 ms	490 ms	327 ms	8099 ms	452 ms	299 ms
Prueba 3	9098 ms	302 ms	270 ms	8294 ms	485 ms	315 ms	8150 ms	428 ms	297 ms
Promedio	8366,67 ms	302,33 ms	274,67 ms	8072,33 ms	483 ms	314 ms	8153 ms	456 ms	304,33 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

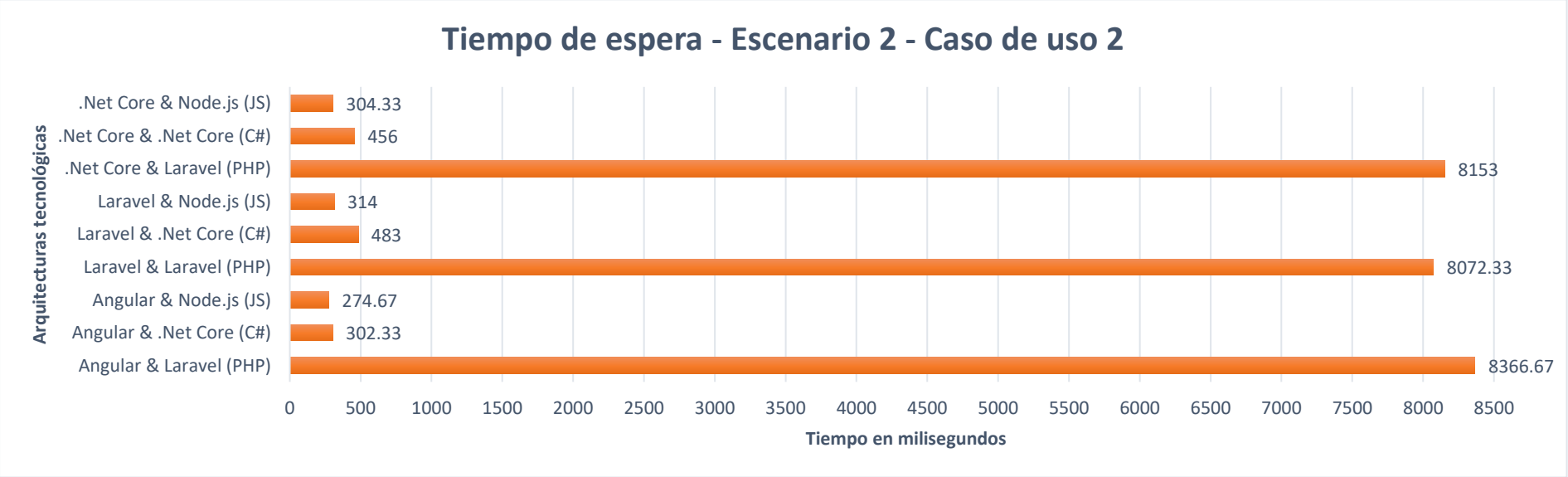


Figura 29. Tiempo de espera de la ejecución de las consultas – Escenario 2 – Caso de uso 2

Escenario 3.

En este escenario, como se muestra en la Tabla 20, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 25000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 30*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 1778,33 milisegundos al ejecutar una petición GET de 25000 usuarios.

Tabla 20: Análisis comparativo del tiempo de espera – Escenario 3 – Caso de uso 2

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	176389 ms	3706 ms	1842 ms	186113 ms	3661 ms	2165 ms	189845 ms	5573 ms	1948 ms
Prueba 2	179319 ms	3429 ms	1759 ms	178535 ms	3417 ms	1998 ms	187653 ms	5974 ms	1933 ms
Prueba 3	178330 ms	3690 ms	1734 ms	182324 ms	3791 ms	2001 ms	189101 ms	5804 ms	1984 ms
Promedio	178012,67 ms	3608,33 ms	1778,33 ms	182324 ms	3623 ms	2054,67 ms	188866,33 ms	5783,67 ms	1955 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

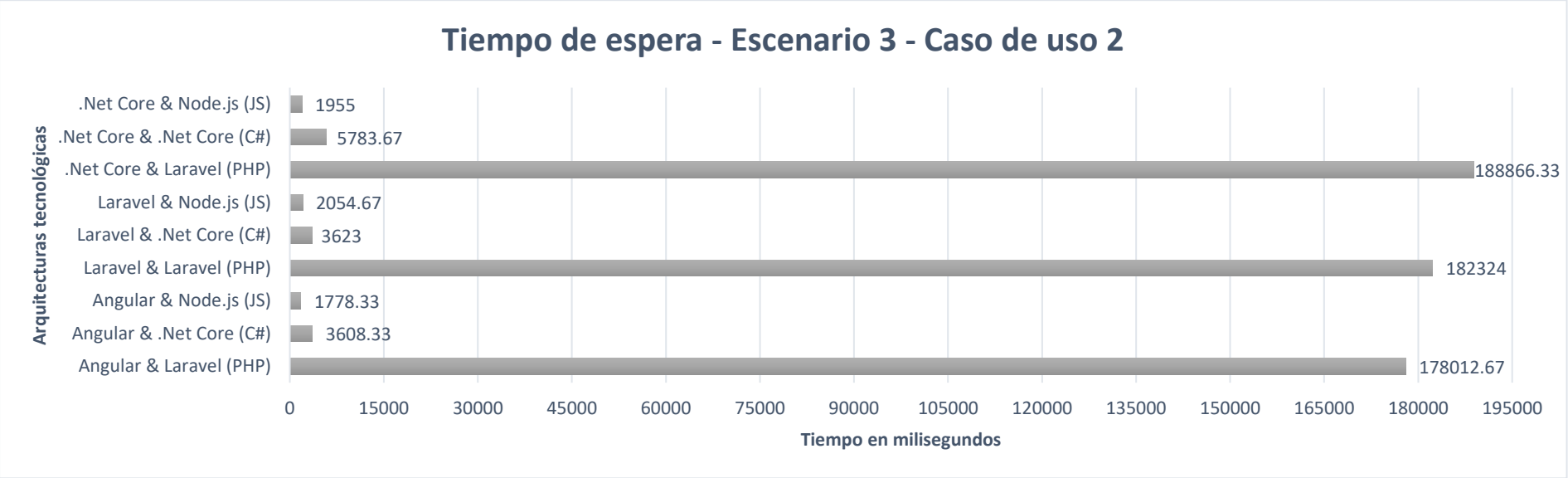


Figura 30. Tiempo de espera de la ejecución de las consultas – Escenario 3 – Caso de uso 2

Escenario 4.

En este escenario, como se muestra en la Tabla 21, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 50000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 31*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 3460 milisegundos al ejecutar una petición GET de 50000 usuarios.

Tabla 21: Análisis comparativo del tiempo de espera – Escenario 4 – Caso de uso 2

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	354603 ms	6686 ms	3409 ms	382109 ms	6897 ms	3701 ms	391659 ms	10961 ms	3706 ms
Prueba 2	366096 ms	6591 ms	3413 ms	379130 ms	6947 ms	3773 ms	389542 ms	11504 ms	3487 ms
Prueba 3	360349 ms	7257 ms	3558 ms	380630 ms	6964 ms	3585 ms	390197 ms	11364 ms	3687 ms
Promedio	360349,33 ms	6844,67 ms	3460 ms	380623 ms	6936 ms	3686,33 ms	390466 ms	11276,33 ms	3626,67 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

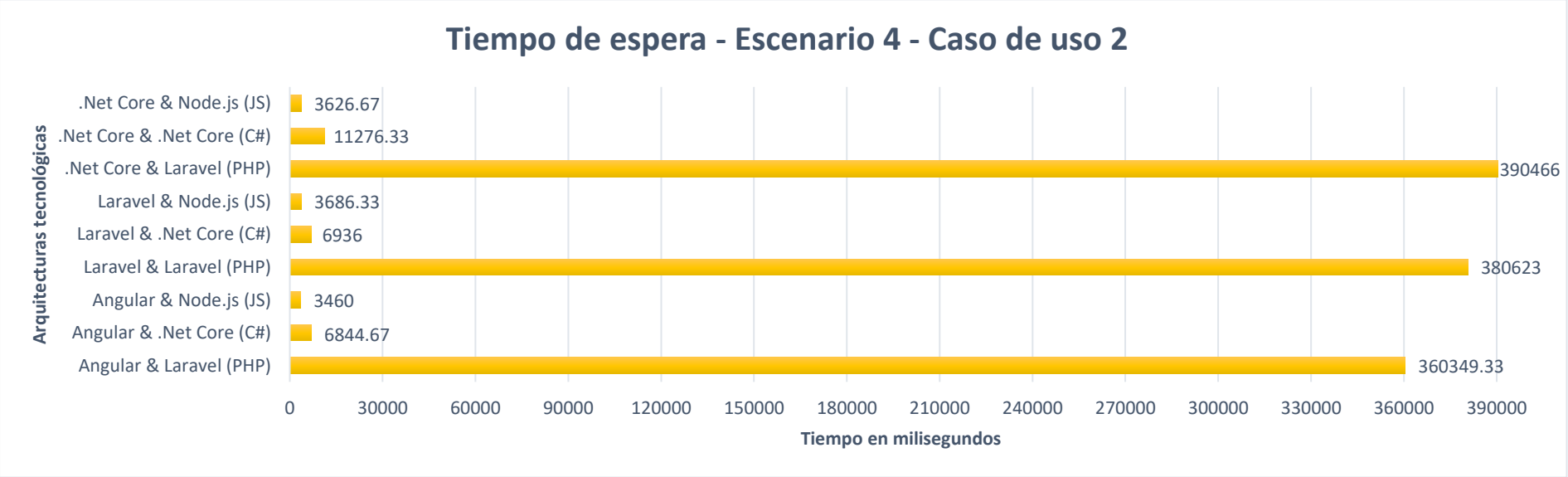


Figura 31. Tiempo de espera de la ejecución de las consultas – Escenario 4 – Caso de uso 2

Escenario 5.

En este escenario, como se muestra en la Tabla 22, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 100000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 32*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 6731,33 milisegundos al ejecutar una petición GET de 100000 usuarios.

Tabla 22: Análisis comparativo del tiempo de espera – Escenario 5 – Caso de uso 2

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	683718 ms	12525 ms	6668 ms	723268 ms	13699 ms	6991 ms	729101 ms	22661 ms	8330 ms
Prueba 2	690247 ms	13196 ms	6812 ms	718111 ms	13511 ms	7002 ms	730999 ms	23278 ms	6832 ms
Prueba 3	686955 ms	13291 ms	6714 ms	720711 ms	13646 ms	7144 ms	728231 ms	22254 ms	7318 ms
Promedio	686973,33 ms	13004 ms	6731,33 ms	720696,67 ms	13618,67 ms	7045,67 ms	729443,67 ms	22731 ms	7493,33 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

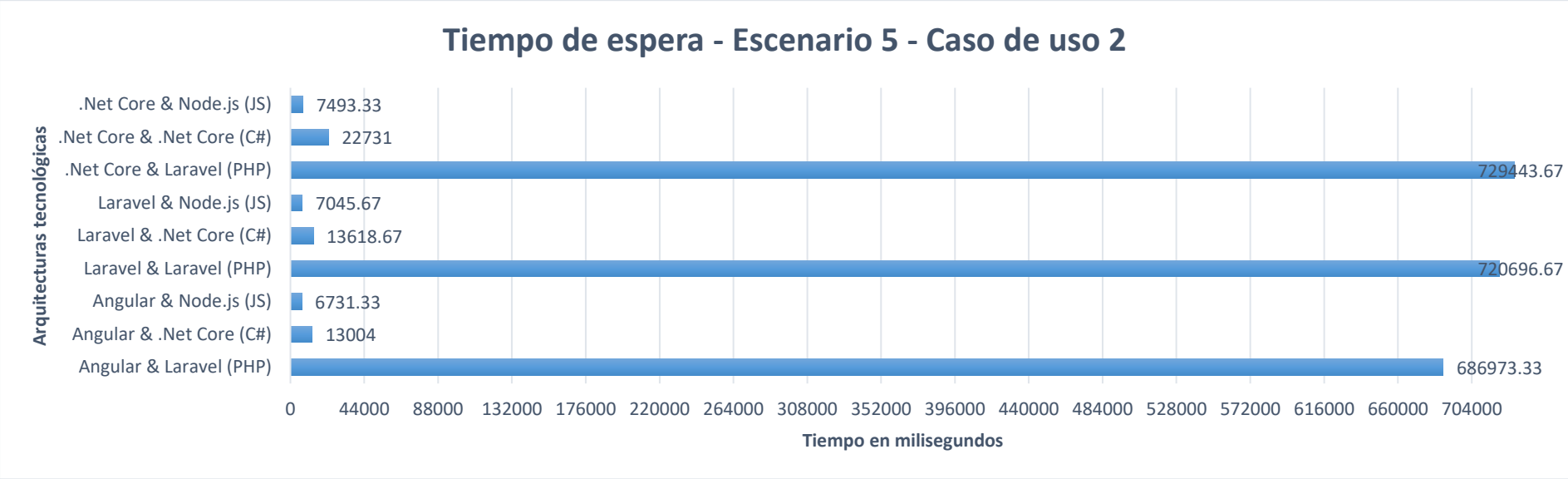


Figura 32. Tiempo de espera de la ejecución de las consultas – Escenario 5 – Caso de uso 2

Caso de uso 3.

Para este caso de uso, se ejecuta la misma consulta del segundo caso de uso con cada uno de los escenarios planteados, y con la diferencia de que se agregan *comments* y estos con sus respectivos parámetros tal y como se muestra en la *Figura 33*.

Figura 33. Detalle de la consulta para el Caso de uso 3

Escenario 1.

En este escenario, como se muestra en la Tabla 23, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 1 registro y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.

- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript, seguido de las Arquitecturas que se conforman del Backend de .Net Core con el lenguaje C#, y por último aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 34*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 504,67 milisegundos al ejecutar una petición GET de 1 usuario.

Tabla 23: Análisis comparativo del tiempo de espera – Escenario 1 – Caso de uso 3

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	766 ms	519 ms	507 ms	820 ms	699 ms	659 ms	819 ms	637 ms	630 ms
Prueba 2	963 ms	521 ms	504 ms	850 ms	730 ms	620 ms	801 ms	625 ms	571 ms
Prueba 3	809 ms	531 ms	503 ms	833 ms	739 ms	625 ms	825 ms	644 ms	614 ms
Promedio	846 ms	523,67 ms	504,67 ms	834,33 ms	722,67 ms	634,67 ms	815 ms	635,33 ms	605 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

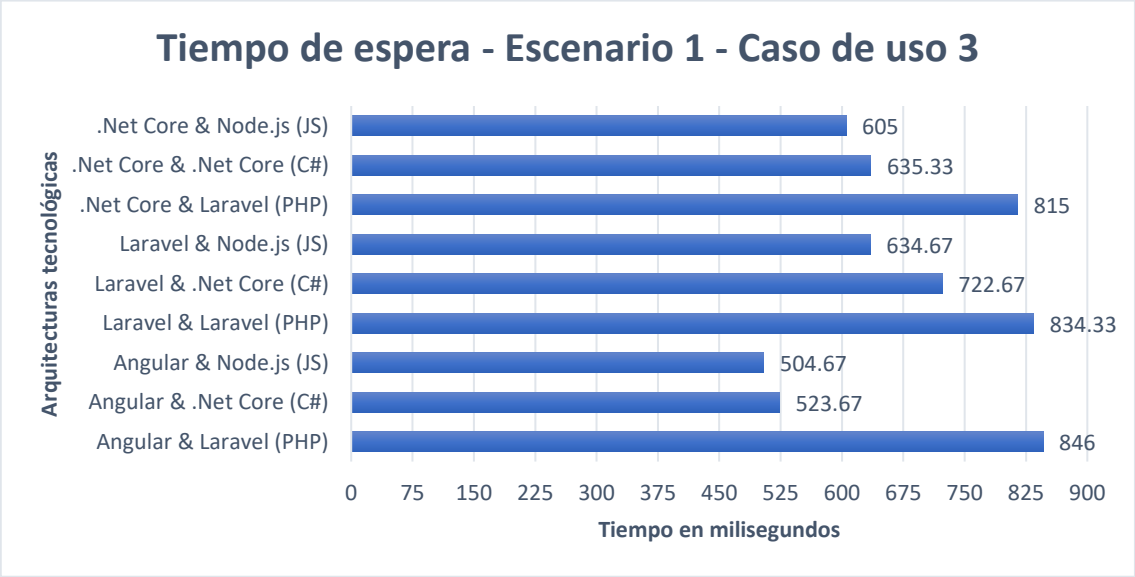


Figura 34. Tiempo de espera de la ejecución de las consultas – Escenario 1 – Caso de uso 3

Escenario 2.

En este escenario, como se muestra en la Tabla 24, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 1000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 35*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 872 milisegundos al ejecutar una petición GET de 1000 usuarios.

Tabla 24: Análisis comparativo del tiempo de espera – Escenario 2 – Caso de uso 3

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	49280 ms	940 ms	867 ms	52327 ms	1405 ms	1155 ms	53149 ms	1299 ms	920 ms
Prueba 2	52287 ms	955 ms	872 ms	53206 ms	1287 ms	1119 ms	54097 ms	1247 ms	915 ms
Prueba 3	52324 ms	961 ms	877 ms	52851 ms	1332 ms	1034 ms	53342 ms	1248 ms	913 ms
Promedio	51297 ms	952 ms	872 ms	52794,67 ms	1341,33 ms	1102,67 ms	53529,33 ms	1264,67 ms	916 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

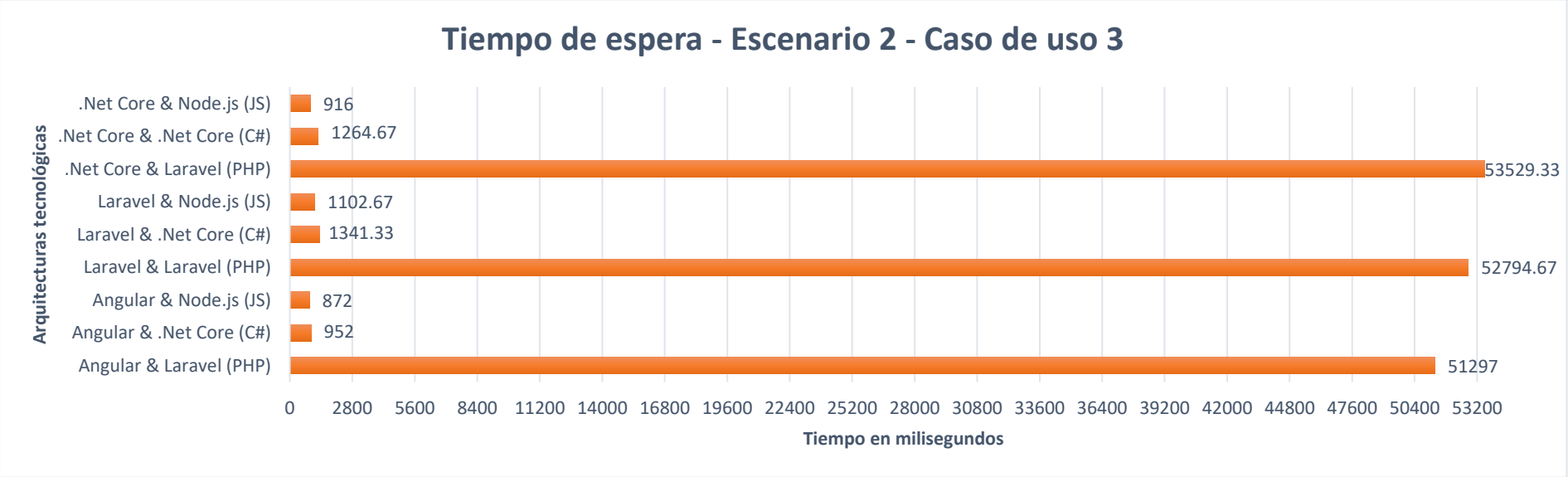


Figura 35. Tiempo de espera de la ejecución de las consultas – Escenario 2 – Caso de uso 3

Escenario 3.

En este escenario, como se muestra en la Tabla 25, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 25000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 36*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 7198 milisegundos al ejecutar una petición GET de 25000 usuarios.

Tabla 25: Análisis comparativo del tiempo de espera – Escenario 3 – Caso de uso 3

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	1216144 ms	12109 ms	7204 ms	1220109 ms	12615 ms	7564 ms	1239467 ms	18434 ms	7529 ms
Prueba 2	1219530 ms	12080 ms	7208 ms	1219900 ms	14770 ms	7701 ms	1238730 ms	17690 ms	7848 ms
Prueba 3	1217589 ms	11720 ms	7182 ms	1218651 ms	12844 ms	7317 ms	1229755 ms	17299 ms	8122 ms
Promedio	1217754,33 ms	11969,67 ms	7198 ms	1219553,33 ms	13409,67 ms	7527,33 ms	1235984 ms	17807,67 ms	7833 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

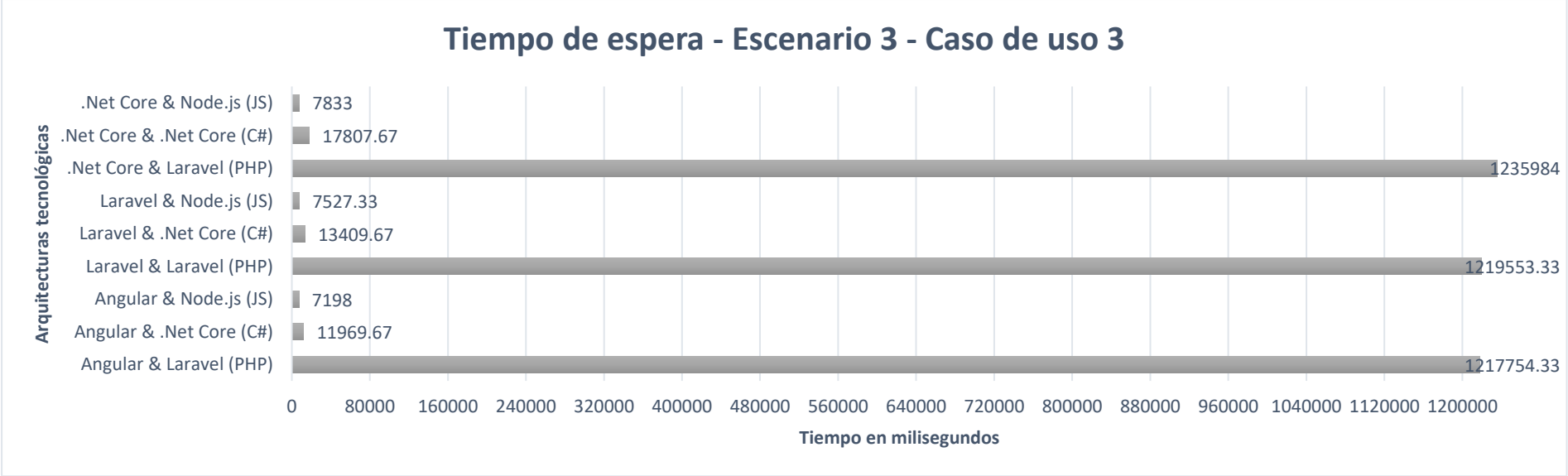


Figura 36. Tiempo de espera de la ejecución de las consultas – Escenario 3 – Caso de uso 3

Escenario 4.

En este escenario, como se muestra en la Tabla 26, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 50000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 37*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 14619 milisegundos al ejecutar una petición GET de 50000 usuarios.

Tabla 26: Análisis comparativo del tiempo de espera – Escenario 4 – Caso de uso 3

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	2645244 ms	22564 ms	14512 ms	2654838 ms	23411 ms	14983 ms	2657223 ms	35677 ms	15744 ms
Prueba 2	2656111 ms	22692 ms	14659 ms	2669478 ms	22762 ms	14897 ms	2620898 ms	35310 ms	15843 ms
Prueba 3	2661319 ms	23173 ms	14686 ms	2678871 ms	22569 ms	14934 ms	2699109 ms	34940 ms	15257 ms
Promedio	2654224,67 ms	22809,67 ms	14619 ms	2667729 ms	22914 ms	14938 ms	2659076,67 ms	35309 ms	15614,67 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

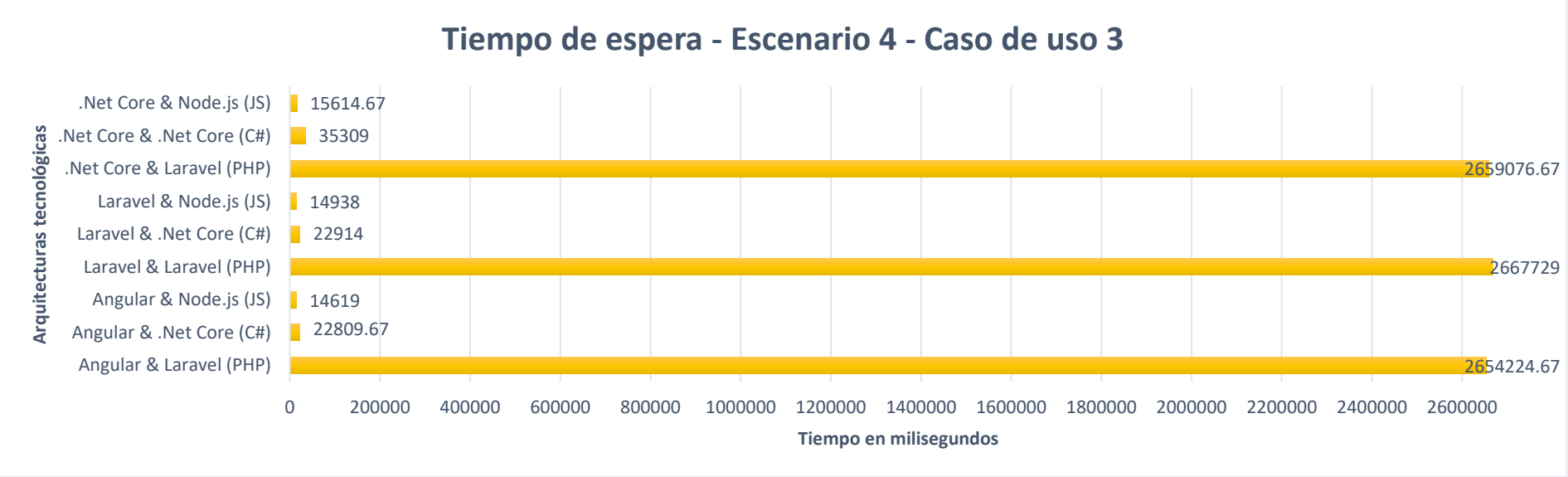


Figura 37. Tiempo de espera de la ejecución de las consultas – Escenario 4 – Caso de uso 3

Escenario 5.

En este escenario, como se muestra en la Tabla 27, se analiza los resultados de las solicitudes ejecutadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en procesar 100000 registros y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las solicitudes han sido procesadas exitosamente en cada una de las pruebas.
- El tiempo de espera se consigue al restar el tiempo que se tardó en completar la solicitud menos el tiempo en cuando se inició, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las más eficientes muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript y .Net Core con el lenguaje C#, mientras que no resultan ser eficientes aquellas arquitecturas conformadas por el Backend de Laravel con el lenguaje de programación PHP, puesto que los tiempos de espera son mucho más tardíos en comparación con las demás Arquitecturas. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 38*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un tiempo de espera de 29865,33 milisegundos al ejecutar una petición GET de 100000 usuarios.

Tabla 27: Análisis comparativo del tiempo de espera – Escenario 5 – Caso de uso 3

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	3629432 ms	42466 ms	28355 ms	3648614 ms	44730 ms	37018 ms	3687542 ms	77141 ms	31357 ms
Prueba 2	3625945 ms	44237 ms	29693 ms	3638417 ms	44042 ms	38095 ms	3709208 ms	66149 ms	31176 ms
Prueba 3	3633677 ms	44986 ms	31548 ms	3647516 ms	44182 ms	37003 ms	3698856 ms	68635 ms	32834 ms
Promedio	3629684,67 ms	43896,33 ms	29865,33 ms	3644849 ms	44318 ms	37372 ms	3698535,33 ms	70641,67 ms	31789 ms

De la tabla anterior se analizan los valores de los tiempos de las Arquitecturas tecnológicas propuestas, el tiempo de espera promedio:

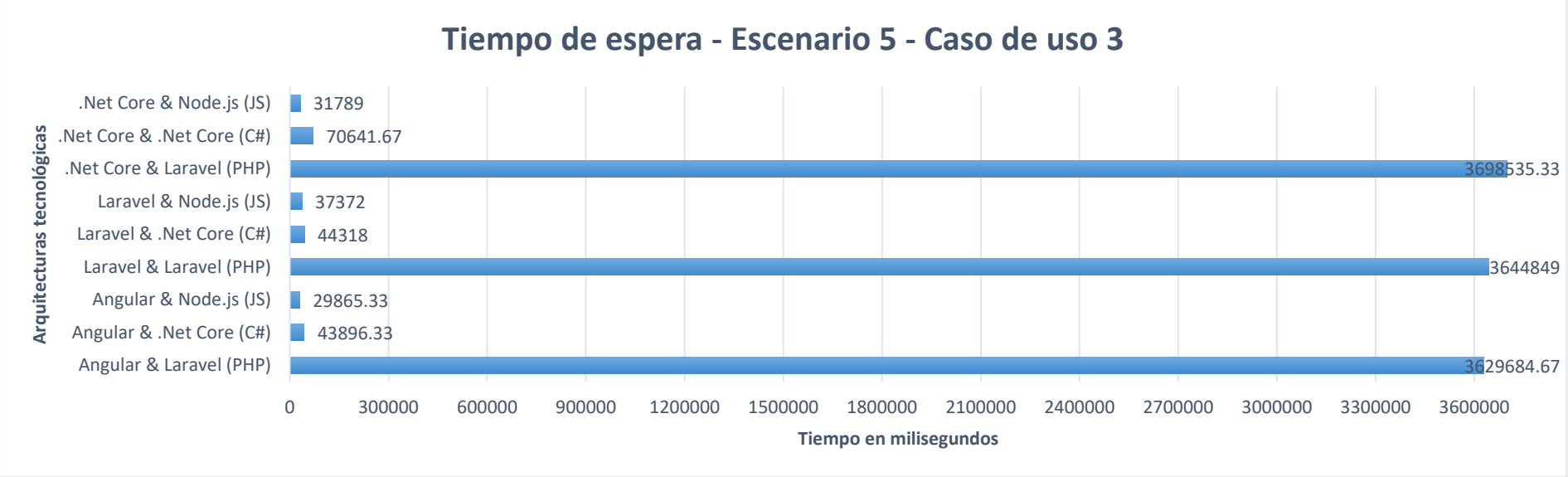


Figura 38. Tiempo de espera de la ejecución de las consultas – Escenario 5 – Caso de uso 3

3.1.2. Resultados de Tiempo de rendimiento

El tiempo de rendimiento permitió recolectar la información correspondiente al rendimiento de las Arquitecturas de desarrollo que se plantearon, es decir, cuántas tareas pueden ser procesadas por unidad de tiempo. El valor deseado, según la métrica de la norma ISO/IEC 25023, el más lejano a 0/t es el mejor, es decir, la arquitectura que registre el mayor tiempo es la que tiene mejor rendimiento.

Caso de uso 4.

En esta segunda parte del experimento se presenta un caso de uso en el que se realizan peticiones POST vía HTTP a los endpoints de cada servicio para insertar la información relacionada con los usuarios, en la Bade de Datos de PostgreSQL. Se han planteado 4 escenarios con diferente número de registros a insertar tal y como se muestra en la Tabla 28, con el fin de poder recolectar la información suficiente para analizarla e interpretarla.

Tabla 28: Detalle de los escenarios de cada Insert

Tipo	Escenario	Nro. Registros
Insert	Escenario 1	1
	Escenario 2	100
	Escenario 3	500
	Escenario 4	1000

Escenario 1.

En este escenario, como se muestra en la Tabla 29, se analiza los resultados del número de inserciones realizadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en insertar 1 usuario y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las inserciones han sido procesadas exitosamente en cada una de las pruebas.

- El rendimiento se consigue al dividir el número de registros insertados, en este caso 1, para el tiempo que se tarda en completar la inserción, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las que tienen un mejor rendimiento muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 39*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Laravel y Node.js con el lenguaje de programación JavaScript, con un rendimiento de 0,02282 tareas/milisegundos al ejecutar una petición POST de 1 usuario.

Tabla 29: Análisis comparativo del rendimiento – Escenario 1 – Caso de uso 4

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	0,00143	0,00568	0,01235	0,00245	0,00980	0,03448	0,00043	0,01205	0,01887
Prueba 2	0,00143	0,00568	0,01587	0,00246	0,01316	0,01613	0,00043	0,01020	0,01852
Prueba 3	0,00145	0,00610	0,01639	0,0024	0,01408	0,01786	0,00043	0,01235	0,01408
Promedio	0,00144	0,00582	0,01487	0,00245	0,01235	0,02282	0,00043	0,01153	0,01716

De la tabla anterior se analiza el rendimiento promedio de las Arquitecturas tecnológicas propuestas:

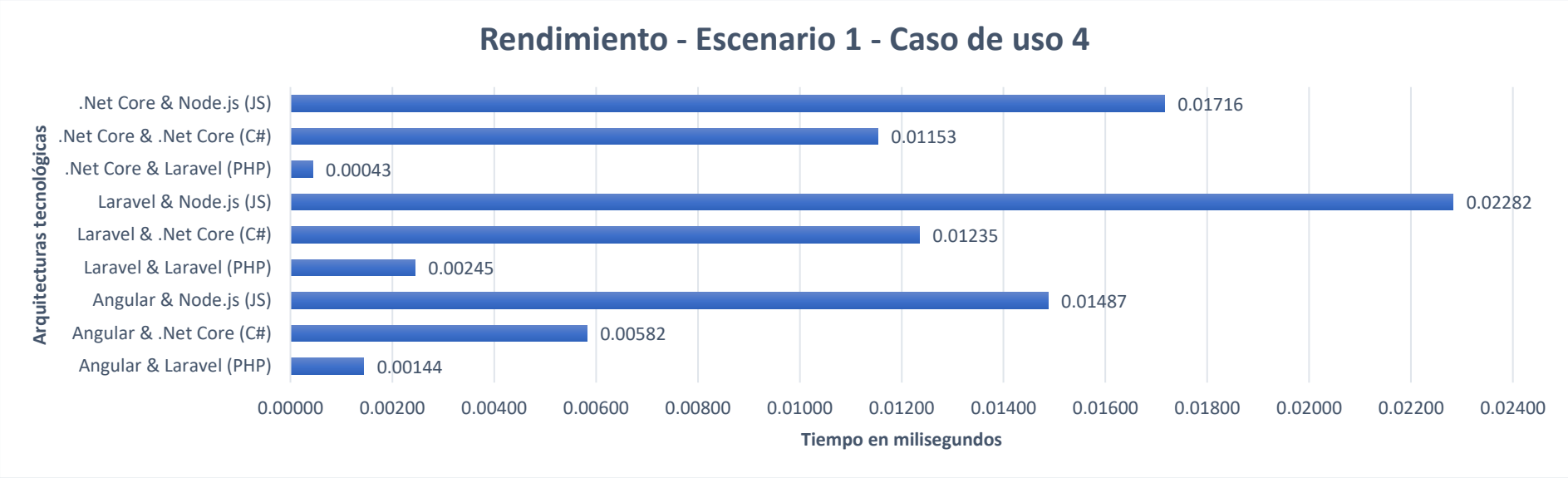


Figura 39. Tiempo de rendimiento de las inserciones – Escenario 1 – Caso de uso 4

Escenario 2.

En este escenario, como se muestra en la Tabla 30, se analiza los resultados del número de inserciones realizadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en insertar 100 usuarios y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las inserciones han sido procesadas exitosamente en cada una de las pruebas.
- El rendimiento se consigue al dividir el número de registros insertados, en este caso 100, para el tiempo que se tarda en completar la inserción, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las que tienen un mejor rendimiento muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 40*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un rendimiento de 0,10435 tareas/milisegundos al ejecutar una petición POST de 100 usuarios.

Tabla 30: Análisis comparativo del rendimiento – Escenario 2 – Caso de uso 4

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	0,00367	0,04978	0,10549	0,00234	0,03738	0,04566	0,00043	0,01873	0,02530
Prueba 2	0,00340	0,05144	0,09381	0,00239	0,03449	0,04108	0,00044	0,01856	0,02672
Prueba 3	0,00344	0,05277	0,11377	0,00242	0,03792	0,04482	0,00043	0,01903	0,02701
Promedio	0,00350	0,05133	0,10435	0,00238	0,03660	0,04386	0,00043	0,01877	0,02634

De la tabla anterior se analiza el rendimiento promedio de las Arquitecturas tecnológicas propuestas:

Figura 40. Tiempo de rendimiento de las inserciones – Escenario 2 – Caso de uso 4

Escenario 3.

En este escenario, como se muestra en la Tabla 31, se analiza los resultados del número de inserciones realizadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en insertar 500 usuarios y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las inserciones han sido procesadas exitosamente en cada una de las pruebas.
- El rendimiento se consigue al dividir el número de registros insertados, en este caso 500, para el tiempo que se tarda en completar la inserción, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las que tienen un mejor rendimiento muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 41*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un rendimiento de 0,13693 tareas/milisegundos al ejecutar una petición POST de 500 usuarios.

Tabla 31: Análisis comparativo del rendimiento – Escenario 3 – Caso de uso 4

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	0,00341	0,05384	0,13151	0,00244	0,03838	0,05320	0,00044	0,01899	0,02743
Prueba 2	0,00350	0,05067	0,13358	0,00238	0,03659	0,05337	0,00044	0,01890	0,02702
Prueba 3	0,00346	0,05195	0,14569	0,00241	0,03809	0,05371	0,00044	0,01887	0,02780
Promedio	0,00345	0,05216	0,13693	0,00241	0,03769	0,05343	0,00044	0,01892	0,02741

De la tabla anterior se analiza el rendimiento promedio de las Arquitecturas tecnológicas propuestas:

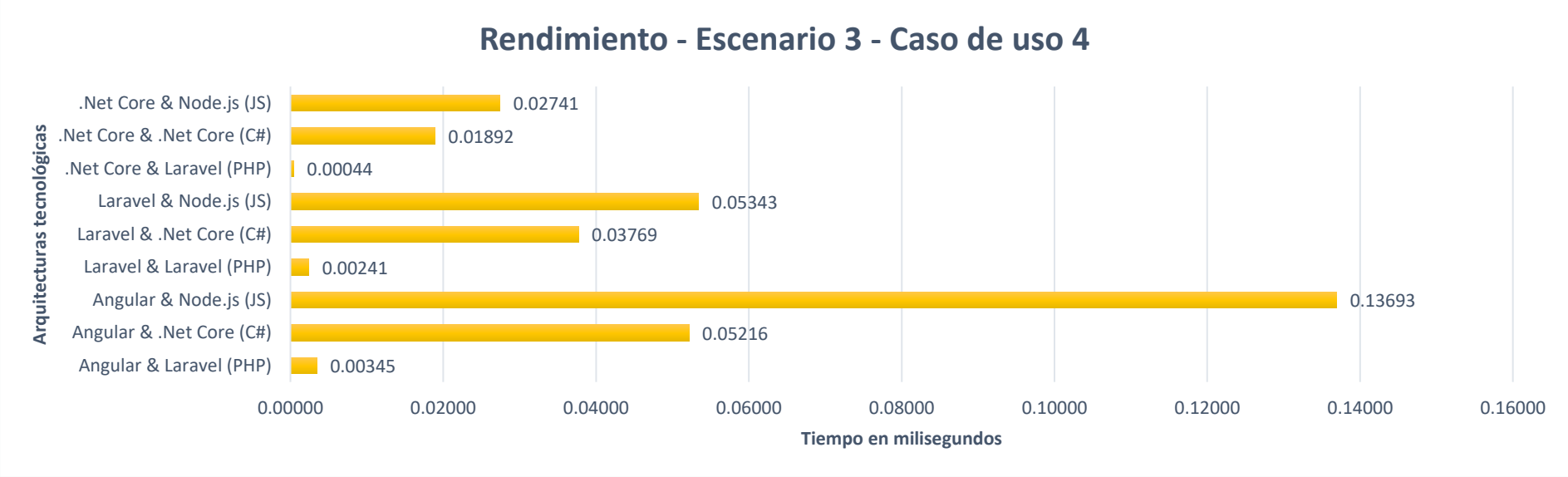


Figura 41. Tiempo de rendimiento de las inserciones – Escenario 3 – Caso de uso 4

Escenario 4.

En este escenario, como se muestra en la Tabla 32, se analiza los resultados del número de inserciones realizadas desde el Frontend de las Arquitecturas tecnológicas, el tiempo que tarda en insertar 500 usuarios y si terminaron correctamente con respuesta del Backend.

Se obtiene que:

- Para cada arquitectura las inserciones han sido procesadas exitosamente en cada una de las pruebas.
- El rendimiento se consigue al dividir el número de registros insertados, en este caso 500, para el tiempo que se tarda en completar la inserción, todo esto en milisegundos.
- Al comparar cada una de las Arquitecturas tecnológicas, las que tienen un mejor rendimiento muestran ser aquellas que están conformadas por el Backend de NodeJS con el lenguaje de programación JavaScript. Con respecto al Frontend, los 3 Frameworks resultan ser eficientes: Angular, .Net Core y Laravel. Ver *Figura 42*.
- En este escenario, la Arquitectura tecnológica más eficiente es la conformada por Angular y Node.js con el lenguaje de programación JavaScript, con un rendimiento de 0,15963 tareas/milisegundos al ejecutar una petición POST de 500 usuarios.

Tabla 32: Análisis comparativo del rendimiento – Escenario 4 – Caso de uso 4

	Angular & Laravel	Angular & .Net Core	Angular & NodeJS	Laravel & Laravel	Laravel & .Net Core	Laravel & NodeJS	.Net Core & Laravel	.Net Core & .Net Core	.Net Core & NodeJS
Prueba 1	0,00348	0,05188	0,15463	0,00244	0,03760	0,05456	0,00036	0,01828	0,02838
Prueba 2	0,00356	0,05226	0,16184	0,00246	0,03793	0,05569	0,00036	0,01900	0,02885
Prueba 3	0,00350	0,05421	0,16242	0,00245	0,04067	0,05273	0,00036	0,01908	0,02736
Promedio	0,00351	0,05278	0,15963	0,00245	0,03873	0,05433	0,00036	0,01879	0,02820

De la tabla anterior se analiza el rendimiento promedio de las Arquitecturas tecnológicas propuestas:

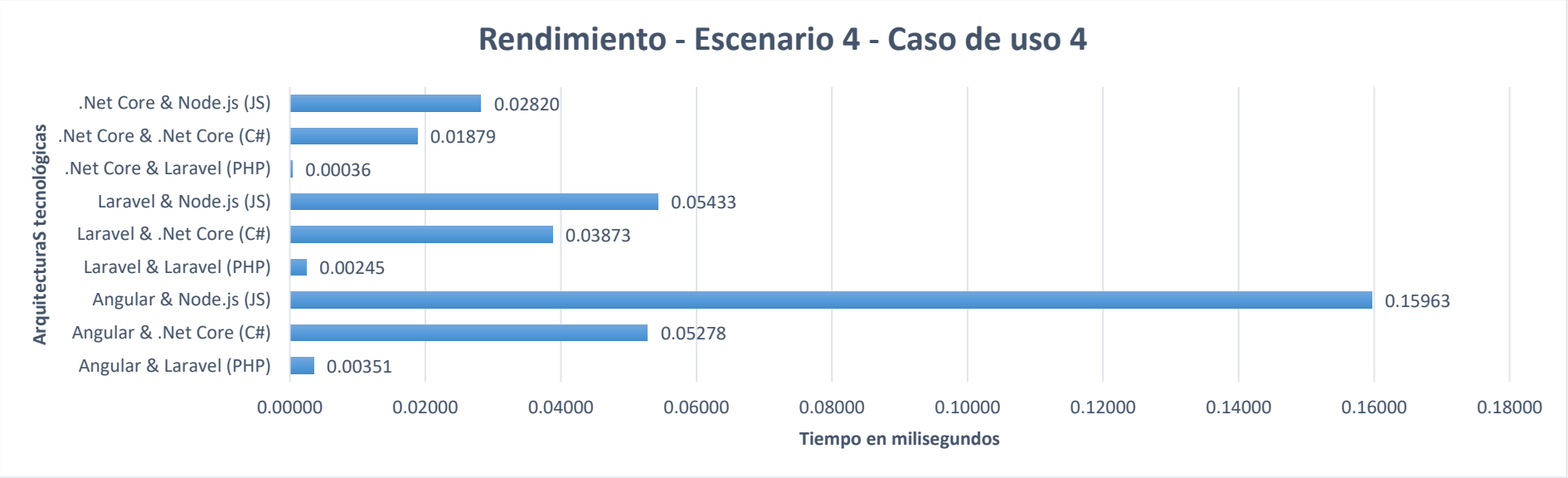


Figura 42. Tiempo de rendimiento de las inserciones – Escenario 4 – Caso de uso 4

3.2. Análisis de resultados y verificación de hipótesis

Para la comprobación de las dos hipótesis planteadas se comparó los promedios de las variables de eficiencia de cada arquitectura de desarrollo por cada caso de uso, teniendo así:

Para la hipótesis nula (H_0): La eficiencia de todas las arquitecturas orientadas a microservicios que utilizan el patrón arquitectónico REST, es la misma.

Se considera válida esta hipótesis si todos los promedios de las arquitecturas planteadas son iguales, es decir, si H_0: $\mu_{ij} = \mu_{ij}$; donde i es el caso de uso y j la arquitectura evaluada. Tomando como punto de partida el enunciado anterior y los resultados de los casos de uso, en la *Figura 43* se evidencia que la hipótesis en cuestión no se cumple, de hecho, el hallazgo es que $\mu_{ij} \neq \mu_{ij}$, para toda i y j. Por lo tanto, se rechaza la hipótesis H_0.

Para la hipótesis alternativa (H_1): Es posible desarrollar una arquitectura orientada a microservicios con APIREST, que sea mayormente eficiente que otras arquitecturas.

Se considera válida esta hipótesis si el promedio de una de las arquitecturas planteadas es mayor que el resto, es decir, si H_1: $\mu_{ij} > \mu_{ij}$; donde i es el caso de uso y j la arquitectura evaluada. Tomando como punto de partida el enunciado anterior y los resultados de los casos de uso, en la *Figura 43* es posible identificar la arquitectura más eficiente al comparar los promedios obtenidos, teniendo a la Arquitectura conformada por Angular y NodeJS como la más eficiente en tiempo de respuesta y rendimiento. Por lo tanto, se acepta la hipótesis H_1.

CASO DE USO (CU)	ARQUITECTURA DE DESARROLLO								
	Angular & Laravel (PHP)	Angular & .Net Core (C#)	Angular & Node.js (JS)	Laravel & Laravel (PHP)	Laravel & .Net Core (C#)	Laravel & Node.js (JS)	.Net Core & Laravel (PHP)	.Net Core & .Net Core (C#)	.Net Core & Node.js (JS)
TIEMPO DE ESPERA (ms)									
CU1	μ_{11}	μ_{12}	μ_{13}	μ_{14}	μ_{15}	μ_{16}	μ_{17}	μ_{18}	μ_{19}
	39895,4	937,67	372,53	42295,53	1690,67	527,93	42896,8	1847,8	391,6
CU2	μ_{21}	μ_{22}	μ_{23}	μ_{24}	μ_{25}	μ_{26}	μ_{27}	μ_{28}	μ_{29}
	246827,73	4788,67	2482,87	258446,4	4979,73	2664,27	263489,33	8095,67	2716,13
CU3	μ_{31}	μ_{32}	μ_{33}	μ_{34}	μ_{35}	μ_{36}	μ_{37}	μ_{38}	μ_{39}
	1510761,33	16030,27	10611,8	1517152,07	16541,13	12314,93	1529588,07	25131,67	11351,53
RENDIMIENTO (t/s)									
CU4	μ_{41}	μ_{42}	μ_{43}	μ_{44}	μ_{45}	μ_{46}	μ_{47}	μ_{48}	μ_{49}
	0,00298	0,04052	0,10394	0,00242	0,03134	0,04361	0,00042	0,017	0,02478
μ_{ij} donde i es el caso de uso y j la arquitectura de desarrollo									

Figura 43. Comparación de las variables de eficiencia para la comprobación de hipótesis

CONCLUSIONES

- Con la investigación documental se elaboró un marco conceptual y tecnológico que permitió establecer una arquitectura de desarrollo eficiente orientada a microservicios con APIREST, como resultado de los experimentos y poniendo en consideración la norma ISO/IEC 25023 para evaluar la calidad externa del producto de software.
- Mediante la ejecución del laboratorio experimental y a los casos de uso planteados, se pudo realizar un análisis comparativo de las Arquitecturas de desarrollo propuesta, en donde, dicho análisis y al hacer uso del modelo de puntuaciones para jerarquizar y priorizar las arquitecturas, permitió determinar la más eficiente al utilizar el patrón arquitectónico REST.
- De acuerdo con los resultados del laboratorio experimental, la Arquitectura de desarrollo más eficiente está conformada por el Frontend de Angular, el Backend de NodeJS con Java Script y PostgreSQL como base de datos, puesto que los tiempos de espera y rendimiento que se evaluaron en base a la norma ISO/IEC25023 resultaron ser menores, es decir; eficientes en comparación con las otras Arquitecturas de desarrollo planteadas.
- Una vez identificada la arquitectura más eficiente, se comprobó el uso de las mediante el desarrollo de una prueba de concepto con tecnologías Angular y NodeJS. Dicha Arquitectura eficiente pudo ser validad gracias a una prueba de concepto en donde se comprobó que su funcionamiento es el adecuado y sobre todo eficiente.

RECOMENDACIONES

- Poniendo en consideración el alcance de la presente investigación, es recomendable, para trabajos futuros, evaluar las demás métricas de calidad definidas en la norma ISO/IEC 25023 que en esta investigación no son medidas.
- Se recomienda a los desarrolladores de software usar REST como patrón arquitectónico, puesto que es unas de las tecnologías más usadas y en internet se encuentra una cantidad considerable de documentación para poder implementarlo en los proyectos de software.
- Se recomienda usar el framework de Angular para el Frontend y NodeJS con JavaScript para el Backend como arquitectura de desarrollo, ya que estas herramientas optimizan eficientemente el procesamiento de grandes cantidades de datos.
- Es recomendable usar guías de experimentación en las investigaciones, ya que esto agrega valor a la realización del trabajo.

REFERENCIAS

Arsys. (Agosto de 2016). *Desarrollo basado en API REST, la mejor manera de proyectar un backend*. Obtenido de https://www.arsys.es/blog/programacion/proyectar-un-backend-hoy/

Bachmann, F., Bass, L., Clements, P. C., Garlan, D., Ivers, J., Little, R., . . . Stafford, J. A. (2010). *Documenting Software Architectures: Views and Beyond* (Segunda ed.). Addison-Wesley Professional.

Baquero García, J. (2015). *Arsys*. Obtenido de https://www.arsys.es/blog/programacion/que-es-laravel/

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in Practice* (Tercera ed.). Pearson Education.

Bottero, Marta, and Giulia Datola. 2017. "Computational Science and Its Applications – ICCSA 2017." 10409(October):338–53.

Chicaiza Rios, D. F. (2020). DISEÑO DE UN PROTOTIPO DE UNA ARQUITECTURA BASADA EN MICROSERVICIOS PARA LA INTEGRACIÓN DE APLICACIONES WEB ALTAMENTE TRANSACCIONALES. CASO: ENTIDADES FINANCIERAS.

Chulca Quilachamín, C. A., & Molina Lopez, R. P. (2020). MIGRACIÓN HACIA UNA ARQUITECTURA BASADA EN MICROSERVICIOS DEL SISTEMA DE GESTIÓN CENTRALIZADA DE LABORATORIOS DE LA DGIP.

Doglio, F. (2015). *Pro REST API Development with Node.js.* doi:10.1007/978-1-4842-0917-2

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In Present and Ulterior Software Engineering (pp. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12

Farshidi, Siamak, Slinger Jansen, and Jan Martijn Van Der Werf. 2020. "Jou Rna LP." The Journal of Systems & Software 110714.

Flores Jeferson. (2020). ESTUDIO DE UNA ARQUITECTURA DE MICROSERVICIOS MEDIANTE SPRING CLOUD PARA EL DESARROLLO DEL MÓDULO DE REGISTRO Y SEGUIMIENTO MÉDICO DE LOS DEPORTISTAS EN LA FEDERACIÓN DEPORTIVA DE IMBABURA. Universidad Técnica Del Norte, 1–27.

García, A. (2018). *EDteam*. Obtenido de https://ed.team/blog/agiliza-tu-desarrollo-en-nodejs-con-el-orm-sequelize

GitHub Inc. (s.f.-a). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/laravel/laravel

GitHub Inc. (s.f.-b). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/nodejs/node

GitHub Inc. (s.f.-c). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/dotnet/core

GitHub Inc. (s.f.-d). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/angular/angular

GitHub Inc. (s.f.-e). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/dotnet/efcore

GitHub Inc. (s.f.-f). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/npm/cli

GitHub Inc. (s.f.-g). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/expressjs/express

GitHub Inc. (s.f.-h). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/sequelize/sequelize

GitHub Inc. (s.f.-i). *GitHub*. Recuperado el 5 de Marzo de 2021, de https://github.com/postgres/postgres

Hu, T., Zhang, Z., Yi, P., Liang, D., Li, Z., Ren, Q., Hu, Y., & Lan, J. (2021). SEAPP: A secure application management framework based on REST API access control in SDN-enabled cloud environment. Journal of Parallel and Distributed Computing, 147, 108–123. https://doi.org/10.1016/j.jpdc.2020.09.006

IEEE. (2000). Recommended Practice for Architectural Description for Software-Intensive Systems (IEEE Std. 1471-2000). 1-23. doi:10.1109/IEEESTD.2000.91944

INTE/ISO/IEC. (2020). Ingeniería de sistemas y de software - Requisitos y evaluación de la calidad de sistemas y del software (SQuaRE) – Medición de la calidad de sistemas y productos de software (INTE/ISO/IEC 25023:2020)., (pág. 60). Costa Rica. Obtenido de https://www.inteco.org/shop/product/inte-iso-iec-25023-2020-ingenieria-de-sistemas-y-de-software-requisitos-y-evaluacion-de-la-calidad-de-sistemas-y-del-software-square-medicion-de-la-calidad-de-sistemas-y-productos-de-software-5949

ISO. (2016). ISO/IEC 25023: Systems and Software Engineering -- Systems and Software Quality Requirements and Evaluation (SQuaRE) -- Measurement of System and Software Product Quality. https://www.iso.org/obp/ui/es/#iso:std:iso-iec:25023:ed-1:v1:en

José López. (2017). ARQUITECTURA DE SOFTWARE BASADA EN MICROSERVICIOS PARA DESARROLLO DE APLICACIONES WEB DE LA ASAMBLEA NACIONAL. Universidad Técnica Del Norte.

Laravel LLC. (2011). *Laravel Docs*. Obtenido de https://laravel.com/docs/8.x/eloquent

Leone, H. P., Roldán , M. L., & Gonnet, S. M. (Noviembre de 2015). Representación de la Evolución y Refactoring de Arquitecturas de Software mediante la Aplicación y Captura de Operaciones Arquitectónicas. *Revista Tecnología y Ciencia, 13*(27), 197-213.

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

López, M. (2014). Zotero , Docear y Mendeley : características y prestaciones . Cuadernos de Gestión de Información, 4, 51–66. https://revistas.um.es/gesinfo/article/view/219511

MDN. (23 de Noviembre de 2020). *MDN Web Docs*. Obtenido de https://developer.mozilla.org/es/docs/Web/JavaScript

Microsoft. (2017). *Internet Information Services (IIS) 10.0 Express*. Obtenido de https://www.microsoft.com/es-es/download/details.aspx?id=48264

Microsoft. (2020). *REST APIs with .NET and C#*. Obtenido de https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/

Microsoft. (2021). *Entity Framework documentation*. Obtenido de https://docs.microsoft.com/en-us/ef/

Microsoft. (2021). *Microsoft dotnet*. Obtenido de https://dotnet.microsoft.com/download

Mon, A., Vinjoy, M., Estayno, M., Serra, D., & S, G. D. I. D. S. G. I. (2012). Experimentación en Ingeniería de Software - Análisis de la influencia de la personalidad en los equipos en el desarrollo de software. XIV Workshop de Investigadores En Ciencias de La Computación, 646–650.

Mumbaikar, Snehal, and Puja Padiya. 2013. "Web Services Based On SOAP and REST Principles." International Journal of Scientific and Research Publications 3(5):3–6.

Naciones Unidas. 2015. "ODS #9." Retrieved September 4, 2020 (https://www.un.org/sustainabledevelopment/es/infrastructure/).

Newman, S. (2015). *Building Microservices* (Primera ed.). Estados Unidos: O'Reilly Media.

OpenJS Foundation. (2020). *Node.js*. Recuperado el 3 de Diciembre de 2020, de https://nodejs.org/es/

OpenJS Foundation. (2021). *Express.js*. Recuperado el 10 de Enero de 2021, de https://expressjs.com/es/

Pacheco, J. F., & Contreras, E. (2008). Manual para la evaluación multicriterio para programas y proyectos. In Instituto Latinoamericano y del Caribe de Planificación Económica y Social (ILPES). https://doi.org/978-92-1-323231-6

Pautasso, Cesare, Olaf Zimmermann, and Frank Leymann. 2008. "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision." Proceeding of the 17th International Conference on World Wide Web 2008, WWW'08 805–14.

Quality Devs. (16 de Septiembre de 2019). *¿Qué es Angular y para qué sirve?* Obtenido de https://www.qualitydevs.com/2019/09/16/que-es-angular-y-para-que-sirve/

Ramchandra Desai, Prashant. 2016. "A Survey of Performance Comparison between Virtual Machines and Containers." International Journal of Computer Sciences and Engineering International Journal of Computer Sciences and Engineering

International Journal of Computer Sciences and Engineering International Journal of Computer Sciences and Engineering Open (7):55–59.

Reynders, F. (2018). *Modern API Design with ASP.NET Core 2: Building Cross-Platform Back-End Systems.* Apress, Berkeley, CA. doi:https://doi.org/10.1007/978-1-4842-3519-5

Richards, M. (2015). Microservices Architecture Pattern. In *Software Architecture Patterns* (Primera ed., pp. 27-35). Estados Unidos: O'Reilly Media, Inc. Retrieved from https://learning.oreilly.com/library/view/software-architecture-patterns/9781491971437/

Saransig, A. (2018). ANÁLISIS DE RENDIMIENTO ENTRE UNA ARQUITECTURA MONOLÍTICA Y UNA ARQUITECTURA DE MICROSERVICIOS – TECNOLOGÍA BASADA EN CONTENEDORES". Universidad Técnica Del Norte.

Sinha, S. (2019). Working with APIs. En *Beginning Laravel* (págs. 367-398). Apress, Berkeley, CA. doi:https://doi.org/10.1007/978-1-4842-4991-8_12

Stauffer, M. (2019). Writing APIs. In *Laravel Up & Running* (Segunda ed., pp. 337-376). Estados Unidos: O'Reilly Media, Inc. Retrieved from https://learning.oreilly.com/library/view/laravel-up-and/9781491936078/

The PostgreSQL Global Development Group. (2021). *PostgreSQL: The World's Most Advanced Open Source Relational Database*. Obtenido de https://www.postgresql.org/about/

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., & Casallas, R. (2015). Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. *10th Computing Colombian Conference (10CCC)*, (págs. 583-590). Bogotá. doi:10.1109/columbiancc.2015.7333476

W3Techs. (Diciembre de 2020). *Most popular server-side programming languages*. Obtenido de https://w3techs.com/

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in Software Engineering. Springer, Berlin, Heidelberg. https://doi.org/10.1109/TSE.1986.6312975

Yaguachi Pereira, L. E. (2017). APLICACION DE UN MODELO PARA EVALUAR EL RENDIMIENTO EN EL PROCESO DE MIGRACIÓN DE UNA APLICACION MONOLÍTICA HACIA UNA ORIENTADA A MICROSERVICIOS.