UNIVERSIDAD TÉCNICA ESTATAL DE QUEVEDO

Facultad de ciencias de la ingeniería

ESTUDIANTES:

Anderson Jaime

John Vera

William Vera

CARRERA:

Ingeniería de software

CURSO:

7to “A”

ASIGNATURA:

Aplicaciones distribuidas

DOCENTE:

Ing. Gleiston Guerrero

TEMA:

Las arquitecturas distribuidas

Índice

Tabla de ilustraciones

1. Introducción

Las arquitecturas distribuidas son una solución eficiente para abordar problemas y proyectos que requieren procesamientos complejos de manera más eficiente. Según Raj y Sadam [1], un sistema distribuido puede describirse como una colección de aplicaciones informáticas que utilizan recursos computacionales distribuidos en múltiples nodos informáticos con el objetivo de lograr un objetivo común.

Estos sistemas, que también se denominan "computación distribuida" o "bases de datos distribuidas", emplean diferentes nodos para establecer comunicación y sincronización a través de una red compartida [2]. Estos nodos pueden abarcar varios dispositivos de hardware físico, diversos procesos de software e incluso otros sistemas que están encapsulados dentro de ellos. El propósito subyacente de los sistemas distribuidos es erradicar cualquier posible cuello de botella o punto centralizado de falla que pueda existir dentro de un sistema [3].

Este trabajo aborda de manera integral el estudio y análisis de diversas arquitecturas utilizadas en el desarrollo de aplicaciones distribuidas. Se profundiza en la planificación, diseño y estructuración de estas arquitecturas, destacando su importancia en la experiencia del usuario y el rendimiento general de los sistemas. Se explora el desarrollo de diagramas de arquitecturas de aplicaciones web, los cuales sirven como herramienta de comunicación y documentación, permitiendo comprender fácilmente la disposición y la interacción de los diversos elementos que componen la aplicación.

Además, se aborda el desarrollo práctico de una aplicación web usando la arquitectura RestFull, denominada "Gestión Eficiente de Tareas", que busca abordar de manera integral la gestión de tareas, facilitando la colaboración en entornos grupales. Este enfoque práctico permite la aplicación y puesta en práctica de las diferentes arquitecturas distribuidas estudiadas, brindando una visión concreta de su implementación y funcionamiento en un contexto real.

2. Arquitecturas distribuidas

El monitoreo distribuido es una técnica empleada para examinar o controlar los resultados de una acción ejecutada en un sistema distribuido. La tarea de monitorear sistemas distribuidos puede ser bastante compleja ya que cada nodo individual produce su propia disposición de registros y mediciones. Para obtener una perspectiva precisa de un sistema distribuido, resulta imperativo fusionar estas diversas medidas de nodos en una perspectiva que lo abarque todo [1].

Cuando se realizan solicitudes a sistemas distribuidos, no se necesita a acceder a todos los nodos del sistema. En cambio, normalmente pasamos por un grupo específico de nodos o seguimos una ruta particular. El monitoreo distribuido nos ayuda a identificar estas rutas que se usan comúnmente en un sistema distribuido. El seguimiento permite examinar de cerca y vigilar estos caminos. Para lograr esto, cada nodo del sistema está equipado con capacidades de monitoreo distribuido. [2].

2.1. Definición de arquitecturas distribuidas

Un sistema distribuido puede describirse como una colección de aplicaciones informáticas que utilizan recursos computacionales distribuidos en múltiples nodos informáticos con el objetivo de lograr un objetivo común. Estos sistemas, que también se denominan "computación distribuida" o "bases de datos distribuidas", emplean diferentes nodos para establecer comunicación y sincronización a través de una red compartida. Estos nodos pueden abarcar varios dispositivos de hardware físico, diversos procesos de software e incluso otros sistemas que están encapsulados dentro de ellos. El propósito subyacente de los sistemas distribuidos es erradicar cualquier posible cuello de botella o punto centralizado de falla que pueda existir dentro de un sistema[3].

2.2. Características de los sistemas distribuidos.

Compatibilidad, se refiere a la capacidad de los dispositivos para operar con diversos sistemas operativos no afecta su capacidad para proporcionar servicios uniformes a los usuarios. En consecuencia, todos los dispositivos interconectados son mutuamente compatibles. Tolerancia a fallos, al tratarse de una red única con múltiples computadoras, la eventual falla de uno de sus componentes no obstaculiza la realización efectiva de las funciones por parte de los demás,

evitando rápidamente posibles errores. Middleware y API, los procesadores emplean un middleware de distribución para facilitar el intercambio de recursos y capacidades, proporcionando a los usuarios una red coherente e integrada. Este middleware también brinda servicios a las aplicaciones, como seguridad y recuperación de fallos [4].

La escalabilidad en sistemas distribuidos se destaca por su modularidad, proporcionando flexibilidad y facilitando la expansión del sistema sin incrementar su complejidad ni afectar su rendimiento. La capacidad de crecimiento se define como la capacidad del sistema para integrar servicios sin limitaciones inherentes, asegurando al mismo tiempo un rendimiento eficiente en el acceso a los recursos a medida que el sistema evoluciona. La Transparencia en Sistemas Distribuidos es primordial, ya que radica en ofrecer a los usuarios y aplicaciones una percepción de los recursos del sistema, tratados como si fueran gestionados por una única máquina virtual, logrando así la transparencia en la distribución física de los recursos. Diversos aspectos de esta transparencia pueden ser delineados[5].

Se hace más énfasis en las interfaces de programación de aplicaciones (API), que actúan como puntos de acceso para la comunicación entre aplicaciones. Las aplicaciones no requieren conocimiento alguno sobre otras aplicaciones, salvo su API, los sistemas distribuidos generan confianza al trabajar con ellos, ya que es poco común que el sistema falle por completo, dado que las tareas no están centralizadas en un único dispositivo, sino distribuidas en diferentes equipos [5].

2.3. Ventajas de las arquitecturas distribuidas

La utilización de un conjunto de computadoras independientes para realizar procesos o almacenar datos como si fueran un solo equipo ofrece beneficios sustanciales en términos de eficacia y fiabilidad. Los sistemas distribuidos permiten abordar problemas y proyectos que requieren procesamientos complejos de manera más eficiente y rentable. La distribución de tareas entre múltiples nodos optimiza el rendimiento al realizar procesos de manera eficaz en diversos puntos de la red. La arquitectura distribuida exhibe una mayor tolerancia a fallos, ya que la información permanece disponible en otros nodos en caso de que uno falle, lo que asegura una robustez superior en comparación con sistemas centralizados.

Además, al distribuir la carga de trabajo en múltiples nodos, la velocidad de procesamiento se ve notablemente mejorada, brindando respuestas más rápidas, como en consultas a bases de datos. La flexibilidad y escalabilidad se destacan al permitir que el sistema distribuido se amplíe horizontalmente, agregando nuevos nodos para satisfacer las demandas crecientes de recursos como procesamiento, almacenamiento o memoria RAM, evitando la necesidad de aumentar verticalmente la capacidad de los equipos[6].

2.4. Desventajas de las arquitecturas distribuidas

A pesar de las notables ventajas de los sistemas distribuidos, se presentan desafíos que están relacionados con la complejidad y la seguridad en este tipo de arquitecturas. En comparación con los sistemas centralizados, los distribuidos muestran un mayor nivel de complejidad en su diseño, configuración y gestión eficiente. La seguridad se convierte en un punto crítico, dado que estos sistemas conectan numerosos nodos a través de la red, aumentando el riesgo de comprometer la integridad y privacidad de los datos y las comunicaciones. Para abordar este riesgo, se requieren medidas de seguridad adicionales que puedan compensar posibles ataques o, en caso de producirse, mitigar sus efectos. La gestión de un sistema distribuido implica un esfuerzo más considerable por parte de los administradores, ya que puede incluir máquinas con sistemas operativos diversos o versiones distintas, complicando la tarea de hacer funcionar eficientemente toda la arquitectura. [7].

3. Arquitectura de aplicaciones

La arquitectura de aplicaciones sirve como guía y conjunto de prácticas recomendadas para el diseño y desarrollo efectivo de aplicaciones. En una arquitectura de aplicaciones, se distinguen servicios de frontend, relacionados con la experiencia del usuario, y backend, encargados de facilitar el acceso a datos y servicios. Aunque la arquitectura proporciona un mapa para el diseño, las decisiones de implementación, como la elección de un lenguaje de programación, quedan fuera de su alcance. Diversos lenguajes, como JavaScript, Ruby, Python, y Swift, se emplean según el tipo de aplicación y los requisitos. Las arquitecturas modernas priorizan el bajo acoplamiento, adoptando microservicios y APIs para conectar servicios, fundamentales en el desarrollo de aplicaciones en la nube [8].

3.1. Arquitectura de Monolíticas

La arquitectura monolítica, un método convencional en el desarrollo de aplicaciones, implica crear e implementar todos los componentes y funcionalidades como una entidad unificada. Bajo este marco, los servicios y la funcionalidad están estrechamente conectados y dependen de una base de código común, lo que plantea desafíos a la escalabilidad y adaptabilidad de la aplicación a medida que se expande. A pesar de sus inconvenientes en términos de agilidad y escalabilidad, las arquitecturas monolíticas encuentran uso persistentemente en numerosos sistemas heredados y aplicaciones de pequeña y mediana escala[9].

La arquitectura monolítica es un método convencional en el desarrollo de aplicaciones, implica la integración de todas las funcionalidades en una sola unidad. Si bien puede ser apropiado para aplicaciones más simples, puede imponer restricciones a la escalabilidad y la mantenibilidad a medida que las aplicaciones crecen tanto en tamaño como en complejidad. La comunicación interna se realiza mediante llamadas directas de funciones, y las actualizaciones se implementan como unidades completas. Aunque este modelo simplifica el desarrollo inicial y la implementación, puede presentar desafíos a medida que la aplicación crece en complejidad, enfrentando obstáculos en escalabilidad y mantenimiento. La elección entre arquitecturas, como microservicios o basadas en servicios, dependerá de factores como la naturaleza del proyecto, los requisitos de escalabilidad y las preferencias tecnológicas [10].

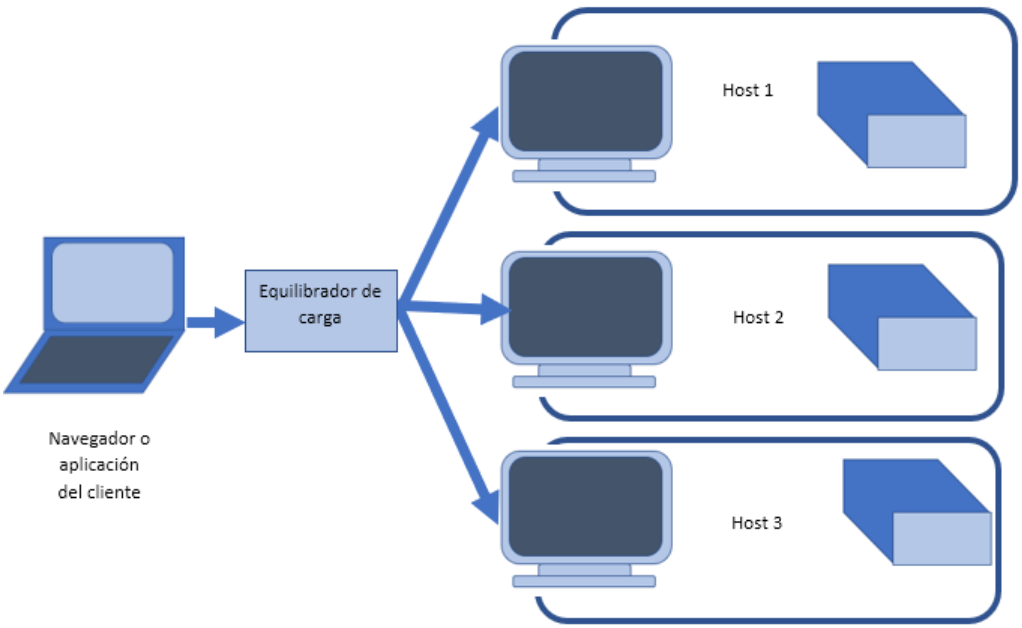

Ilustración 1 Extraído de: The Monolith Strikes Back: Why Istio Migrated from Microservices to a Monolithic Architecture [10]

3.2. Arquitectura de microservicios

Los microservicios son una arquitectura de software que se basa en el desarrollo y despliegue de aplicaciones como un conjunto de servicios pequeños e independientes, cada uno ejecutándose en su propio proceso y comunicándose a través de mecanismos ligeros. Cada microservicio se enfoca en una tarea específica y puede ser desarrollado, desplegado, y escalado de forma independiente. Esta arquitectura fomenta el modularidad, la flexibilidad y la capacidad de escalar componentes individuales, lo que permite a las organizaciones desarrollar, desplegar y mantener aplicaciones de manera más ágil y eficiente[11].

Representan una evolución en la arquitectura de software, permitiendo a las organizaciones desarrollar aplicaciones altamente escalables y flexibles. divide una aplicación en una serie de servicios implementables de forma independiente que se comunican a través de API. Este enfoque permite implementar y escalar cada servicio individual de forma independiente, así como la entrega rápida y frecuente de aplicaciones grandes y complejas. Esta arquitectura fomenta modularidad, la independencia y la capacidad de escalar componentes individuales, lo que resulta en un desarrollo y despliegue más ágil. Sin embargo, la adopción de microservicios también conlleva desafíos en términos de gestión de la complejidad y coordinación entre servicios [12].

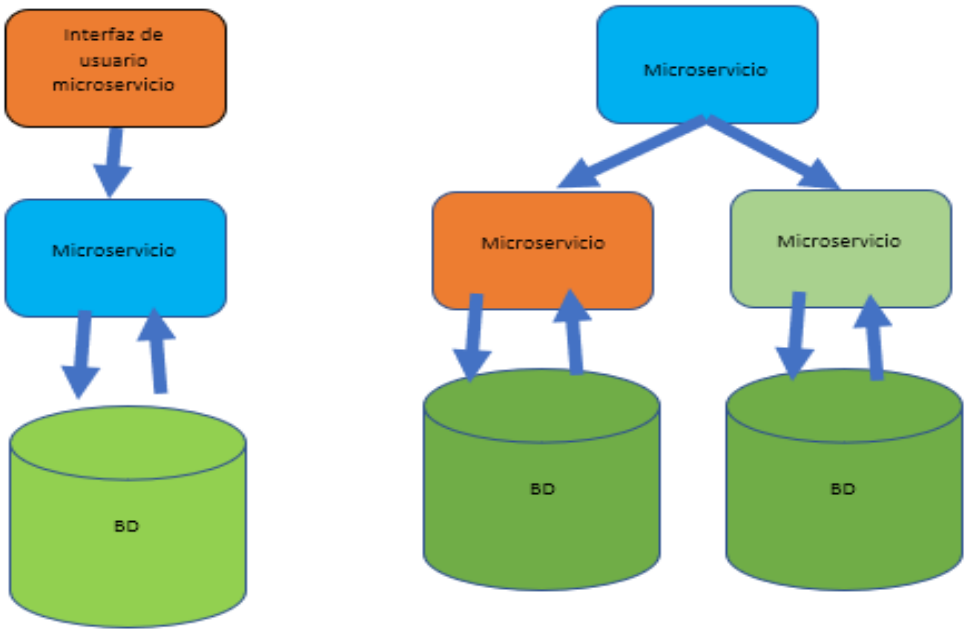

Ilustración 2 Extraído de: A metrics framework for evaluating microservices architecture designs [12].

3.3. Arquitectura de componentes y módulos

La arquitectura de software de componentes y módulos es un enfoque de diseño de sistemas informáticos que organiza y estructura un programa o aplicación mediante la separación de sus funcionalidades en componentes y módulos independientes y reutilizables. La arquitectura de componentes y módulos de software tiene sus ventajas: reutilización de código, mantenimiento más fácil, escalabilidad y la habilidad para trabajar en paralelo en el desarrollo. Promueve la "separación de preocupaciones", en donde cada componente se centra en una labor específica [13].

En el entorno de Angular, la base de una aplicación reside en sus componentes, considerados como los elementos fundamentales. Cada componente, autónomo en la interfaz de usuario, integra plantillas HTML, estilos CSS y lógica en TypeScript. La construcción de la aplicación implica la agrupación de estos componentes para establecer su estructura. Los módulos cumplen un papel esencial en la organización y encapsulamiento de componentes afines y sus recursos en Angular. Estos módulos, conceptualizados como contenedores, definen contextos que permiten la colaboración efectiva entre componentes y servicios [14].

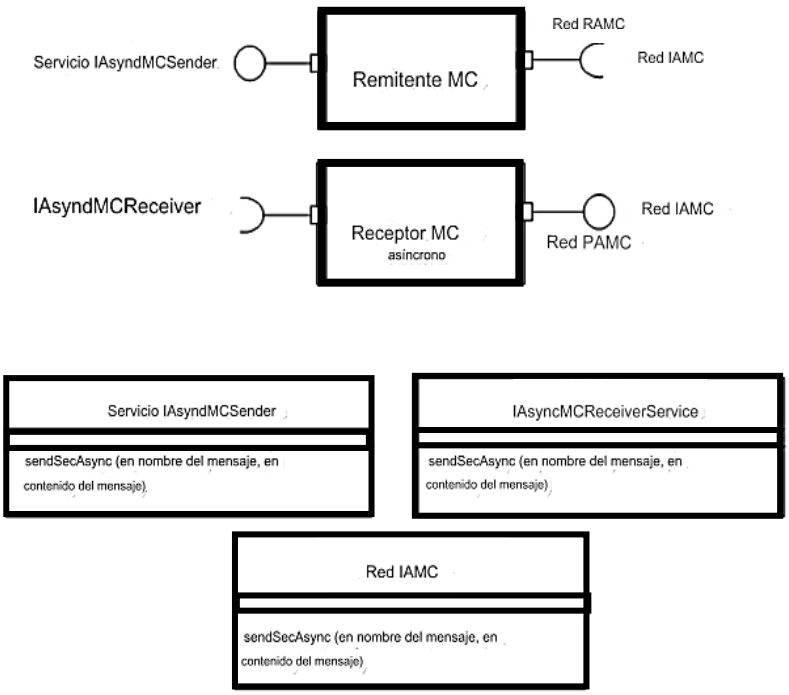

Ilustración 3 Extraído de: Software Architecture Design of a Serverless System [14]

4. Arquitecturas web

La arquitectura web aborda la planificación, diseño y estructuración de las páginas de un sitio, agrupando contenido en categorías y organizándolo según su relevancia. Este enfoque facilita la navegación de los usuarios, mejora la usabilidad y proporciona pautas para motores de búsqueda. La definición de la estructura del sitio involucra elementos como categorización, estructura URL, migas de pan, menús de navegación y enlazado interno [15].

La importancia de una arquitectura web efectiva radica en su impacto tanto en la experiencia del usuario como en la visibilidad en motores de búsqueda. Para los usuarios, una estructura bien pensada facilita la localización de información, mientras que una navegación intuitiva aumenta la probabilidad de conversión. En términos de motores de búsqueda, una arquitectura clara facilita la rastreabilidad del contenido, influyendo directamente en la visibilidad y clasificación del sitio. En resumen, una arquitectura web sólida es esencial para el éxito y rendimiento de cualquier sitio web, independientemente de su tamaño[16].

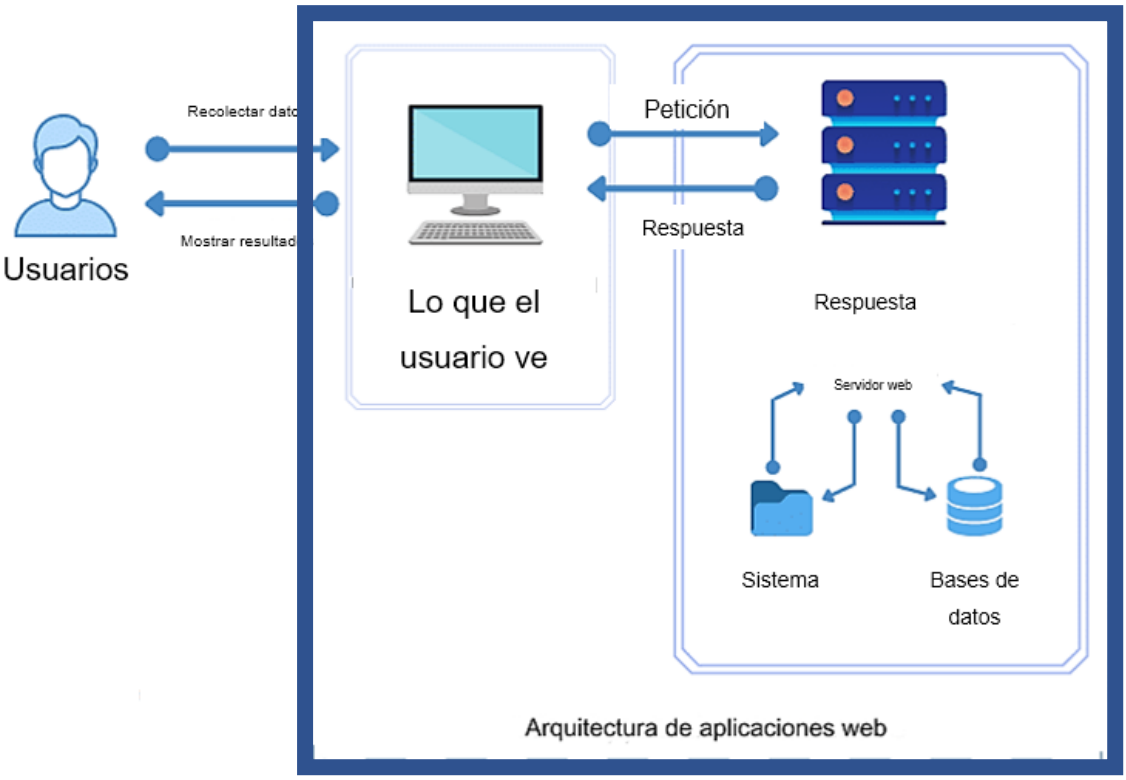

Ilustración 4 Extarído de: Software Architecture Design of a Serverless System[16].

4.1. Arquitectura Cliente-Servidor

La arquitectura cliente-servidor es un modelo informático en el cual el servidor alberga, proporciona y gestiona la mayoría de los recursos y servicios destinados a ser utilizados por el cliente. Este modelo implica que uno o más computadoras cliente están conectadas a un servidor central a través de una conexión de red o internet. También se denomina modelo de computación en red o red cliente-servidor, ya que todas las solicitudes y servicios se realizan a través de una red. La arquitectura cliente-servidor es un modelo común en sistemas distribuidos en el que se divide la funcionalidad en dos partes distintas: el cliente y el servidor. El cliente realiza peticiones y el servidor proporciona los recursos o servicios solicitados, facilita la escalabilidad y gestión centralizada, ya que los clientes pueden estar distribuidos, pero interactúan con un servidor central [17].

El cliente es la interfaz de usuario y la entidad que solicita servicios o recursos al servidor. Puede ser una aplicación, un navegador web u otro dispositivo que interactúa con el usuario. El servidor responde a las solicitudes del cliente proporcionando recursos, servicios o datos solicitados. Maneja las operaciones de procesamiento y gestiona los recursos centrales del sistema. La comunicación entre el cliente y el servidor se realiza a través de protocolos específicos, como HTTP para aplicaciones web o TCP/IP para aplicaciones más genéricas. Modelo de Petición-Respuesta: El cliente envía solicitudes al servidor, y este responde con los resultados correspondientes. La comunicación sigue un modelo de petición-respuesta [18].

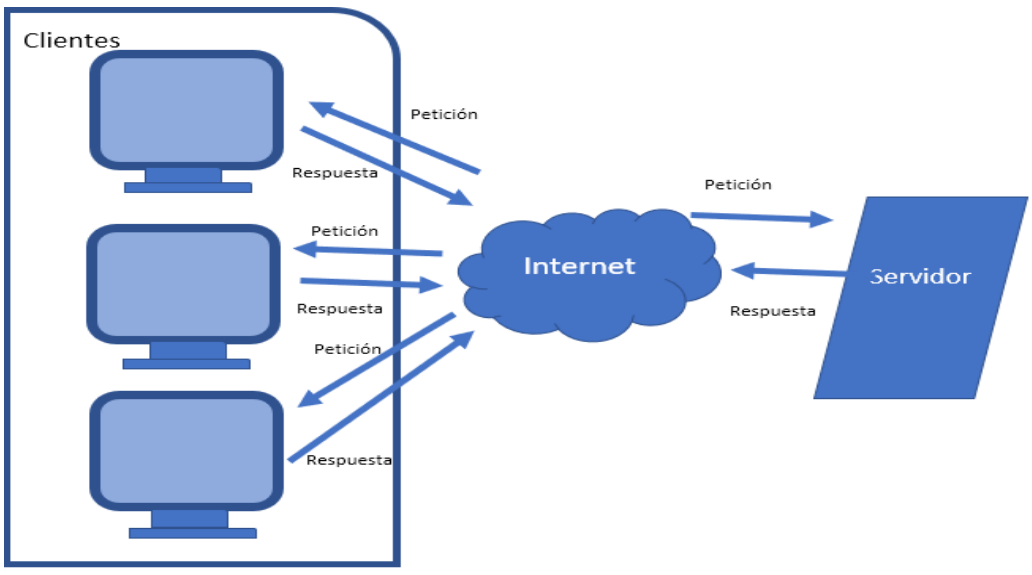

Ilustración 5 Extraído de: An Introduction to Client Server Computing[18]

4.2. Arquitecturas de tres capas

La arquitectura de tres niveles constituye un sólido modelo para el desarrollo de aplicaciones de software, estructurando estas en niveles lógicos y físicos: la interfaz de usuario, el nivel de aplicación y el nivel de datos. Un beneficio clave es la capacidad de desarrollo simultáneo por equipos distintos para cada nivel, permitiendo actualizaciones o escalabilidad sin interferencias. La migración a la nube y el uso de tecnologías nativas de la nube como contenedores y microservicios son comunes, divide la aplicación en tres componentes: la Capa de Presentación, que interactúa directamente con el usuario; la Capa de Lógica de Negocio, el cerebro que procesa datos y aplica reglas; y la Capa de Acceso a Datos, que maneja la interacción con la base de datos [19].

En el ámbito del desarrollo web, los niveles se denominan de manera diferente, aunque desempeñan funciones similares. El servidor web, que actúa como el nivel de presentación, suministra la interfaz de usuario, típicamente en páginas o sitios web como un portal de comercio electrónico. El contenido, ya sea estático o dinámico, se construye comúnmente con HTML, CSS y Javascript. El servidor de aplicación corresponde al nivel intermedio y alberga la lógica empresarial para procesar las entradas del usuario. Su desarrollo suele involucrar lenguajes como Python, Ruby o PHP, ejecutándose en infraestructuras como Django, Rails, Symphony o ASP.NET. El servidor de base de datos representa el nivel de datos o backend de una aplicación web, funcionando en un software de gestión de base de datos como MySQL, Oracle, DB2 o PostgreSQL [20].

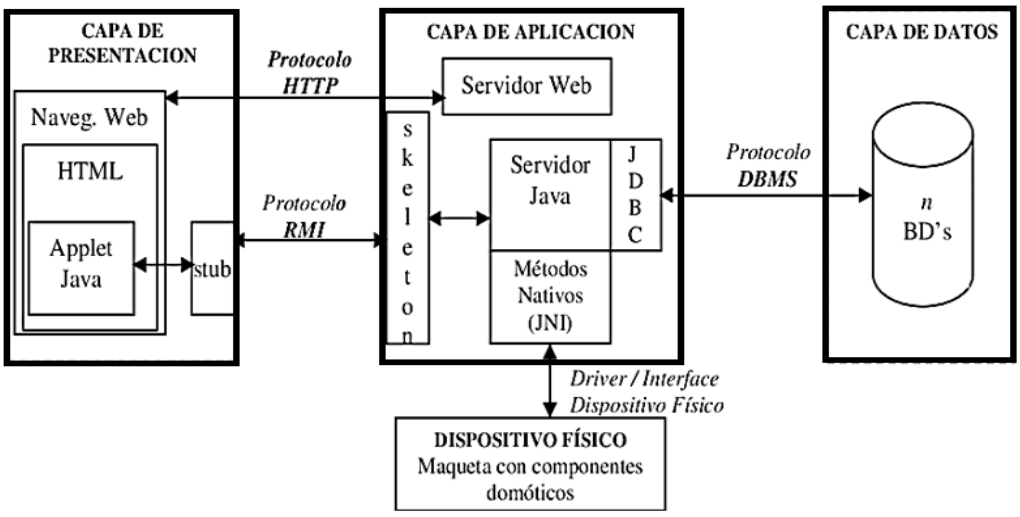

Ilustración 6 Extraído de: Delay optimization strategy for service cache and task offloading in three-tier architecture mobile edge computing system [20].

4.3. Arquitectura RESTfull

La arquitectura RESTful es un conjunto de principios que se utilizan para diseñar servicios web que sean escalables, flexibles y fáciles de mantener. Estos principios incluyen el uso de URI únicas para cada recurso, la utilización explícita de los métodos HTTP, la ausencia de estado en el servidor y la utilización de formatos de intercambio de datos ligeros como JSON, se basa en la idea de que cada recurso debe ser accesible a través de una URL única y que los clientes deben ser capaces de interactuar con estos recursos utilizando los métodos HTTP estándar, como GET, POST, PUT y DELETE. Esta arquitectura utiliza de formatos de intercambio de datos ligeros como JSON, que permiten una transferencia de datos más rápida y eficiente. [21].

La arquitectura posee un enfoque que permite la interoperabilidad entre sistemas informáticos en Internet, destaca la simplicidad, escalabilidad, efectividad, seguridad y confiabilidad. Los recursos en RESTful son identificados por URI (Universal Resource Identifier) y las operaciones se realizan, mantiene el lado del servidor sin estado entre múltiples interacciones. Son a destacar los aspectos clave de la implementación en Laravel, incluyendo la autenticación JWT (JSON Web Tokens) y la configuración de guardias en el archivo "config/auth.php". Se considera el problema de las solicitudes entre dominios, los cuales explican el modelo de seguridad de mismo origen y la implementación de CORS (Cross-Origin Resource Sharing) como mecanismo para permitir recursos restringidos en una página web[22].

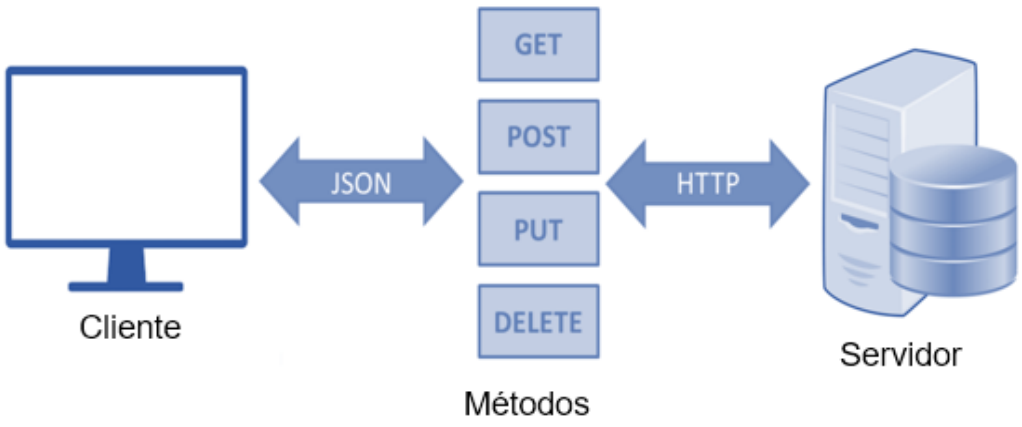

Ilustración 7 Extraído de: Type-directed program synthesis for RESTful APIs [22].

4.4. Arquitectura SOAP

La arquitectura SOAP utiliza XML para encapsular mensajes y HTTP o SMTP para su transporte. Se usa para extender servicios existentes y se integra bien con arquitecturas e interfaces de middleware convencionales. Se lo considera como un protocolo simple para transferir datos entre plataformas de middleware. Aunque SOAP tiene ventajas significativas en la interoperabilidad entre plataformas de middleware heterogéneas, también tiene limitaciones. Estas incluyen su adhesión al modelo cliente-servidor, intercambios de datos como parámetros en invocaciones de métodos, patrones de interacción rígidos y su simplicidad, es un protocolo que proporciona una forma estándar de intercambiar información entre aplicaciones a través de la web, utilizando XML [23].

Las cuatro áreas principales cubiertas por la arquitectura son: Formato de mensaje: describe cómo un mensaje puede ser empaquetado en un documento XML. Transporte: especifica cómo un mensaje puede ser transportado utilizando "HyperText Transfer Protocol" (para interacción basada en la web) o SMTP (para interacción basada en correo electrónico). Reglas de procesamiento: define las reglas que deben seguirse al procesar un mensaje, incluyendo la clasificación de las entidades involucradas en el procesamiento del mensaje. Estilo de interacción del procedimiento de llamadas remota: proporciona convenciones sobre cómo convertir una llamada RPC en un mensaje SOAP y viceversa, tiene la capacidad de encapsular mensajes en documentos, mapear con mensajes en solicitudes y transformar llamadas de procedimientos remotos en mensajes [24].

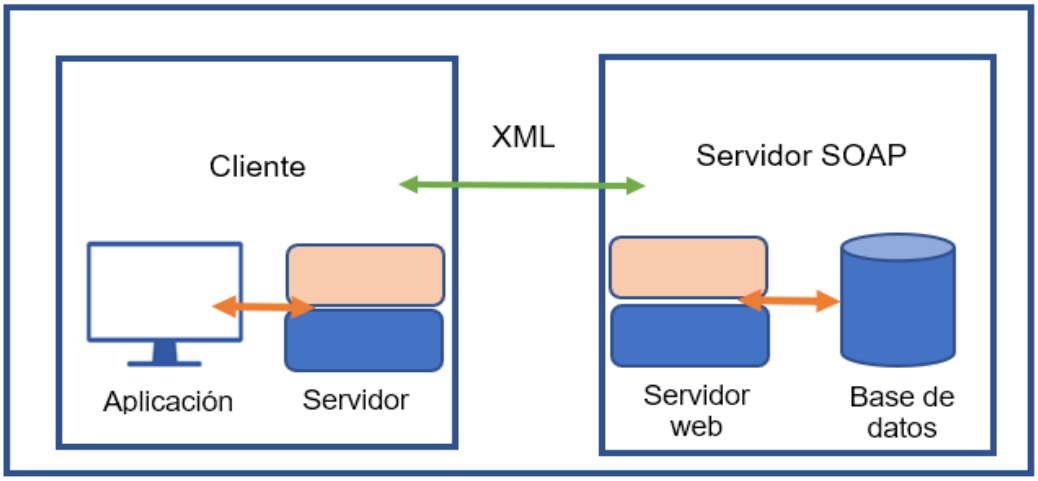

Ilustración 8 Extraído de: Arquitectura orientada a servicios, un enfoque basado en proyectos [24].

5. Planteamiento de la práctica

En el desarrollo de la práctica de arquitectura de aplicaciones distribuidas, se crea el planteamiento de un problema escalable que se desea solucionar, planteando y creando la estructura de solución a través de las arquitecturas distribuidas, a continuación, se muestra el planteamiento del problema, la solución, arquitectura y sus correspondientes versiones utilizando las diferentes arquitecturas explicadas en el documento.

5.1. Planteamiento del problema

Aplicación Web: Gestión Eficiente de Tareas

En la dinámica diaria de proyectos, metas o trabajos, la organización y gestión de tareas juega un papel crucial. La falta de una herramienta eficiente puede resultar en una pérdida de productividad y dificultades en la colaboración. Por lo tanto, surge la necesidad de una aplicación web que aborde de manera integral la gestión de tareas, permitiendo a los usuarios organizar sus actividades de manera efectiva y facilitar la colaboración en entornos grupales.

5.2. Objetivo General de la Aplicación

El propósito fundamental de la aplicación es proporcionar a los usuarios una herramienta eficiente para organizar y gestionar tareas. Además, busca fomentar la colaboración efectiva en grupos, proyectos o trabajos, ofreciendo una plataforma centralizada y fácil de usar.

5.3. Funcionalidades importantes

1. Creación de Grupos: Permite a los usuarios organizar tareas relacionadas en grupos, facilitando la gestión de proyectos o actividades con múltiples componentes.
2. Añadir Tareas: Proporciona la capacidad de agregar tareas con detalles específicos, como título, descripción, fecha de creación y prioridad. Esto asegura una captura detallada de la información esencial para cada tarea.
3. Modificación de Tareas: Ofrece funciones de edición para ajustar la información de las tareas existentes, incluyendo la descripción y otros detalles

relevantes. Esto permite una adaptabilidad continua a medida que evolucionan los proyectos.

4. Marcado de Tareas como Completadas: Permite a los usuarios indicar la finalización de tareas, proporcionando una visión clara del progreso y logros alcanzados en el desarrollo de proyectos o trabajos.
5. Selección de Tareas Importantes: Introduce la capacidad de destacar tareas importantes, ofreciendo una manera efectiva de priorizar y resaltar elementos cruciales en medio de otras actividades.

6. Diagramas de arquitecturas de aplicaciones web.

Los diagramas que se mostrarán a continuación sirven como herramienta de comunicación y documentación, permitiendo a comprender fácilmente la disposición y la interacción de los diversos elementos que componen la aplicación, para distribuir el desarrollo de la aplicación de manera equitativa.

6.1. Diagrama arquitectura cliente-servidor

A continuación, se muestra el desarrollo de aplicación web de Gestión Eficiente de Tareas a nivel de arquitectura cliente-servidor, se han considerado tambien los Framework que adaptaría según nuestras necesidades.

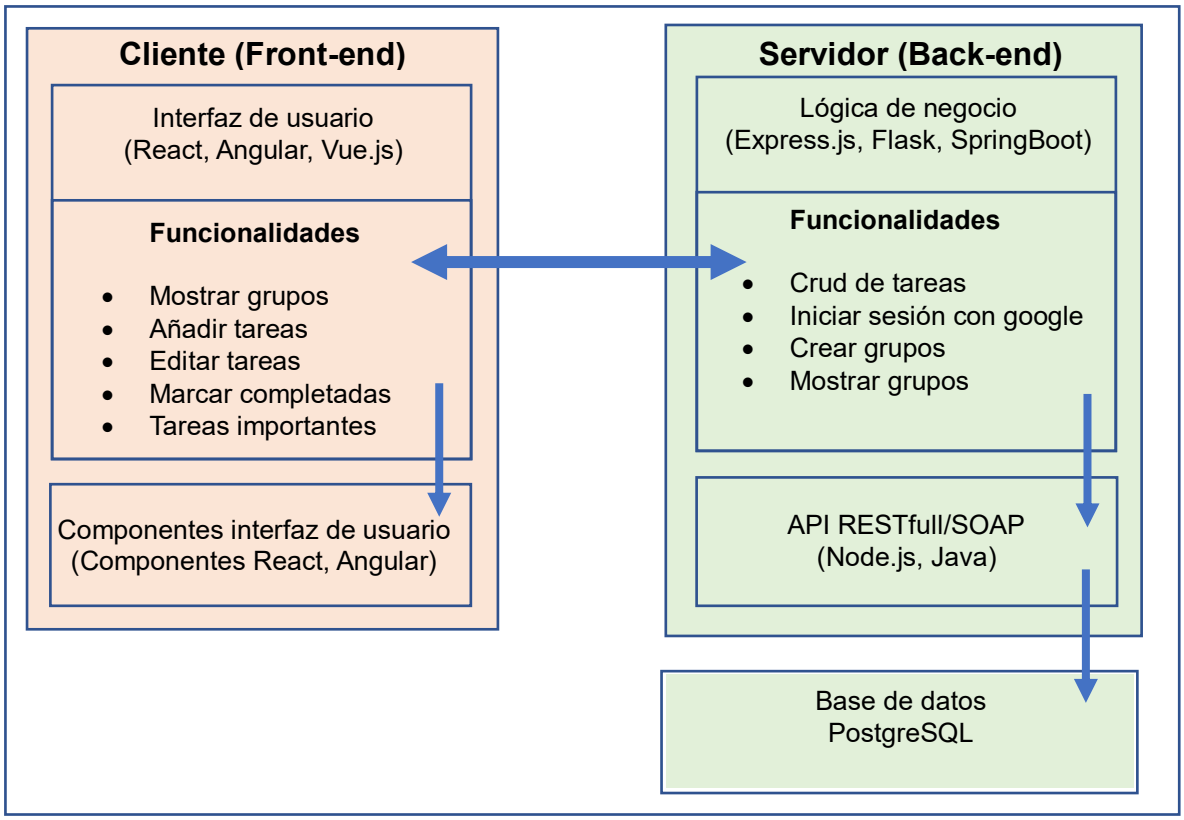

Ilustración 9 Diagrama de arquitectura Cliente-Servidor

6.2. Diagrama arquitectura tres capas

Ahora se mostrará la arquitectura de la aplicación si se desarrollara a través de una estructura de tres capas, basándonos en la conceptualización mostrada anteriormente, se han considerado Frameworks que se ha ajustan a nuestras necesidades funcionales.

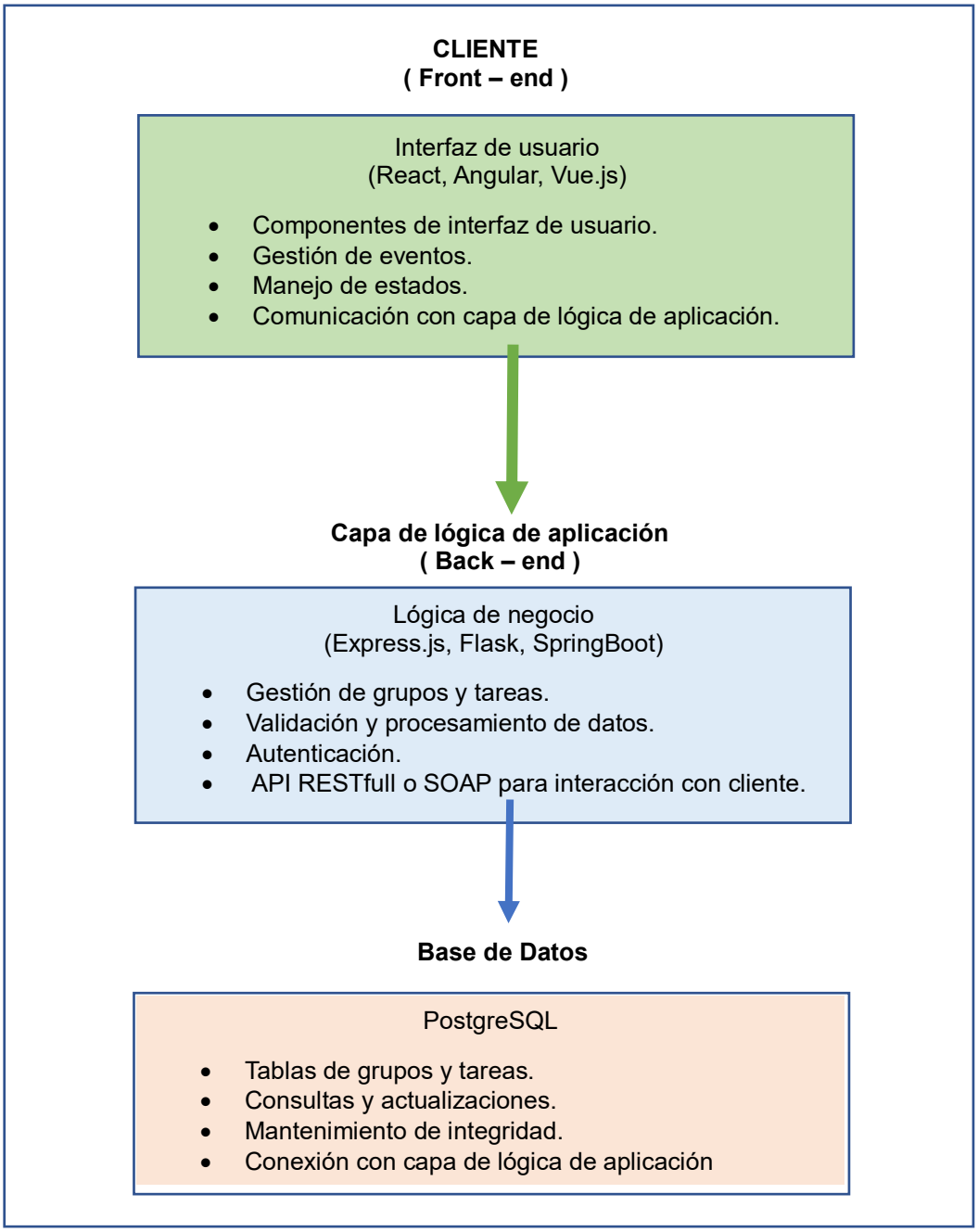

Ilustración 10 Diagrama arquitectura de tres capas

6.3. Diagrama arquitectura RESTfull

Para el planteamiento de la solución hemos considerado la arquitectura RESTfull, ya que posee atributos que facilitan la implementación de funciones que se adaptan a nuestras necesidades, a continuación, se muestra el diseño de la arquitectura.

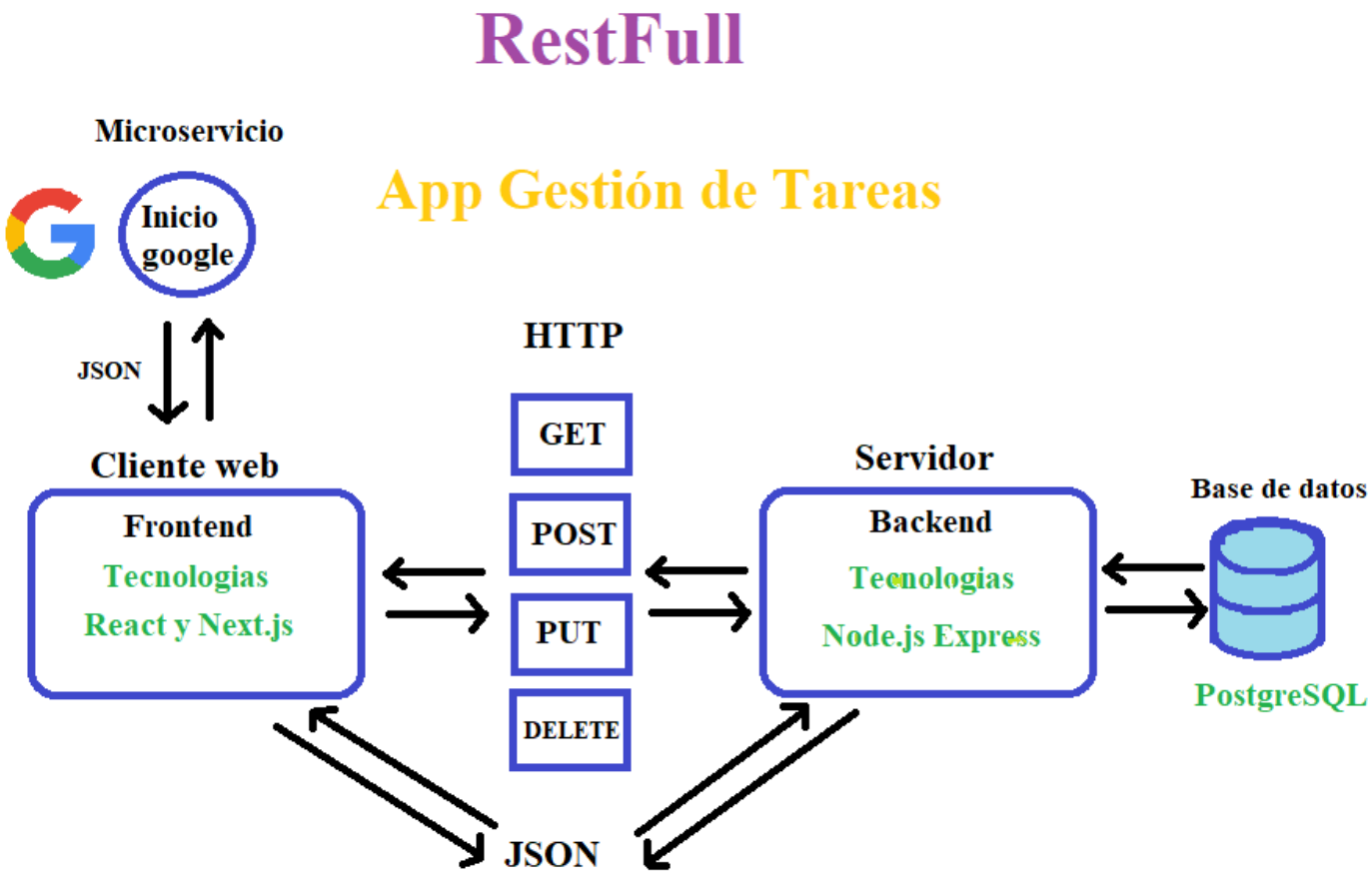

Ilustración 11 Diagrama de arquitectura Restful

6.4. Explicación de solución

Como ya se mencionó previamente se está haciendo uso de la arquitectura RestFull para la creación de la APP de gestión de tareas, a continuación se detalla todo el proceso, tecnologías, que se utilizó a nivel de codificación u diseño con respecto a la arquitectura.

6.5. Tecnologías usadas

La aplicación está dividida en cliente, servidor y base de datos, a continuación de detalla las tecnologías usadas en cada una de las partes:

- **El cliente**

El cliente web (Frontend), el cual realiza las peticiones a la Api, haciendo uso del formato JSON para las peticiones, se realizó con ayuda de Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto. También, se hizo uso del framework nextjs el cual es un marco web de desarrollo front-end y a su vez el uso de la biblioteca de React con ayuda del framework Tailwind CSS, de código abierto para el diseño de páginas web.

- **El Servidor (API)**

La API, la cual recibe las peticiones del cliente y devuelve una respuesta al cliente en formato JSON, se realizó con ayuda de Node.js el cual es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. También, se utilizó el framework Express, este proporciona mecanismos para: Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).

- **Base de datos**

Para la gestión y almacenamiento de los datos se hizo uso de la base de datos PostgreSQL 15, en donde se creó la base de datos con sus respectivas tablas, procedimiento y funciones, que requería la API.

6.6. Estructura de la API Rest (Servidor)

La API, la cual recibe las peticiones del cliente y devuelve una respuesta al cliente en formato JSON, se realizó con ayuda de Node.js y se utilizó el framework Express, este

proporciona mecanismos para: Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).

La estructura de la Api es la siguiente:

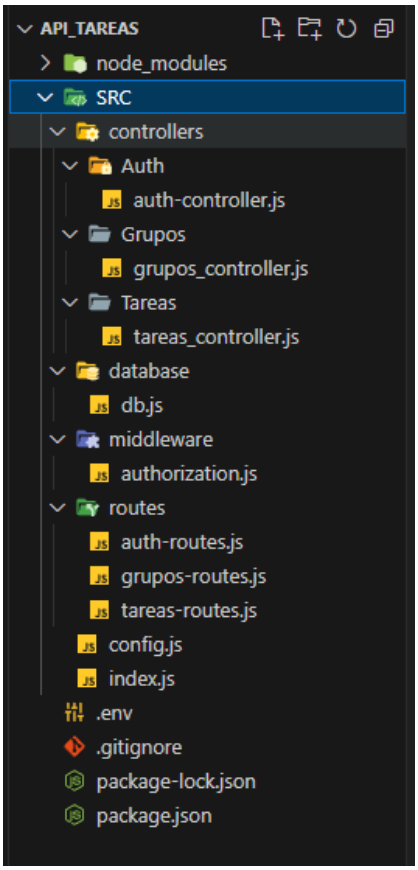

Ilustración 12. Estructura de la Api.

Como se observa en la imagen de arriba tenemos la estructura de la API, en donde tenemos principalmente la caparte SRC la cual es la fuente del conjunto de carpeta que compone el proyecto. Dentro de esta tenemos las siguientes carpetas:

- **Controllers:** En esta carpeta van los métodos encargados de procesar la información enviada por el cliente, y realizan la gestión necesaria de los datos con la base de datos si es requerido.
- **Database:** Aquí se establece la configuración necesaria para establecer la conexión con la base datos.
- **Middleware:** Se establece la gestión pertinente de los token para el inicio de sesión, esta sería la capa de protección de los datos mediante token que se almacenan en la cookie.
- **Routes:** En esta carpeta se establecen las rutas a las cuales con las que el cliente va a establecer las peticiones.

- **Archivo index:** Se establece toda la lógica y configuración de ejecución de la API, como el puerto, rutas principales, cord, entre otros.
- **Archivo packege.json:** Almacena todas las dependencias y librerías que usamos en el proyecto.

6.7. Estructura del Cliente Web

El cliente web (Frontend), el cual realiza las peticiones a la Api, haciendo uso del formato JSON para las peticiones, se realizó con ayuda de Node.js. También, se hizo uso del framework nextjs y a su vez el uso de la biblioteca de React con ayuda del framework Tailwind CSS.

La estructura del cliente web es la siguiente:

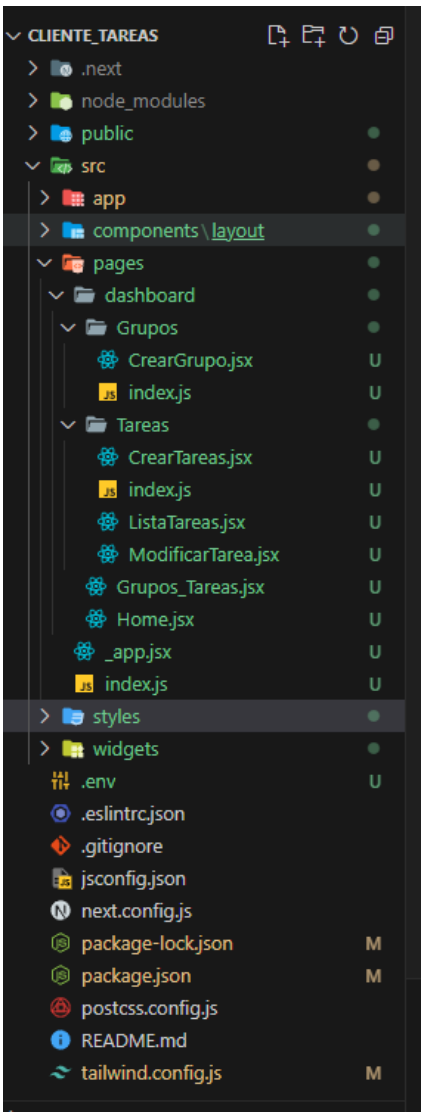

Ilustración 13. Estructura del cliente Web.

Como se observa en la imagen de arriba tenemos la estructura del cliente web, en donde tenemos principalmente la caparte SRC la cual es la fuente del conjunto de carpeta que compone el proyecto. Dentro de esta tenemos las siguientes carpetas:

- **Components:** En esta carpeta van componentes de React que deseamos reutilizar como el Footer y la barra de navegación.
- **Pages:** Aquí se establece la paginas a las que vamos acceder en la aplicación, dentro de ella tenemos divida por carpetas pertinentes para una mejor organización.
- **Styles:** Se establece un estilo de CSS global para la aplicación.
- **Dahboard:** En esta carpeta se establecen las páginas principales de la aplicación para una mayor organización.
- **Archivo Home:** Esta sería la página principal de la aplicación donde están todas las demás paginas.
- **Archivo index:** Esta sería la página de inicio que aparece al ingresar a la aplicación, es la página para iniciar sesión con Google.
- **Archivo packege.json:** Almacena todas las dependencias y librerías que usamos en el proyecto.

6.8. Estructura de la base de datos:

A continuación se muestra la estructura de la base de datos usada de APP tareas.

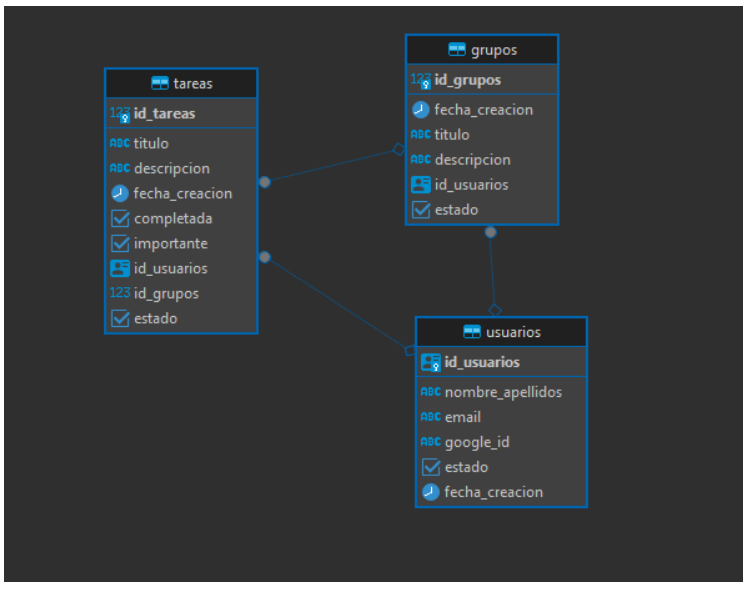

Ilustración 14. Estructura de la base de datos.

6.9. Uso del servició de Google.

La app también hizo uso del servicio de inicio de sesión que nos ofrece Google para ingresar a la aplicación, esto funciona realizando una petición al servidor de Google mediante un correo originario de este, así si el inicio fue correcto, devolviendo un archivo JSON con los respectivos datos del usuarios de ese correo. Como vemos a continuación:

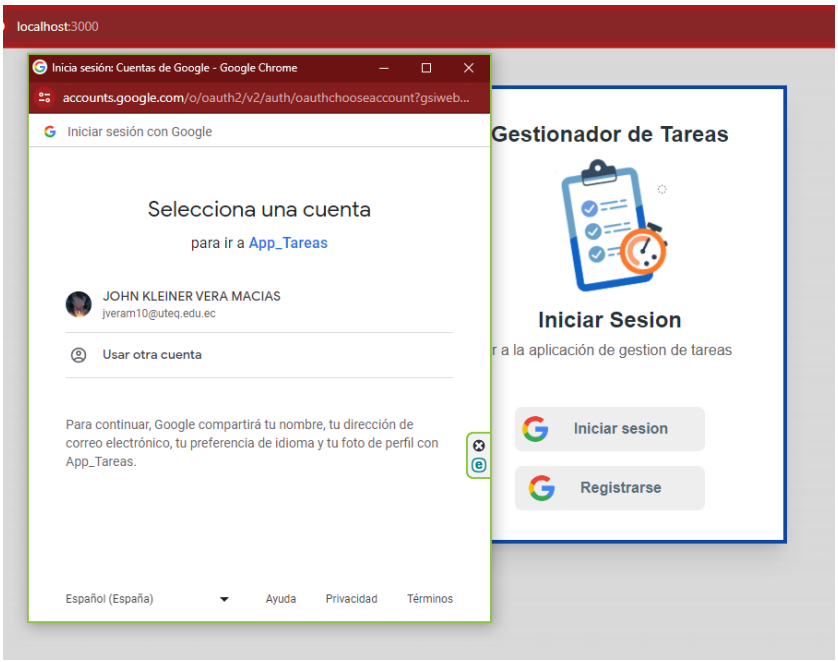

Ilustración 15. Interfaz para el uso del inicio con google.

A continuación se muestra los datos que nos devuelve Google:

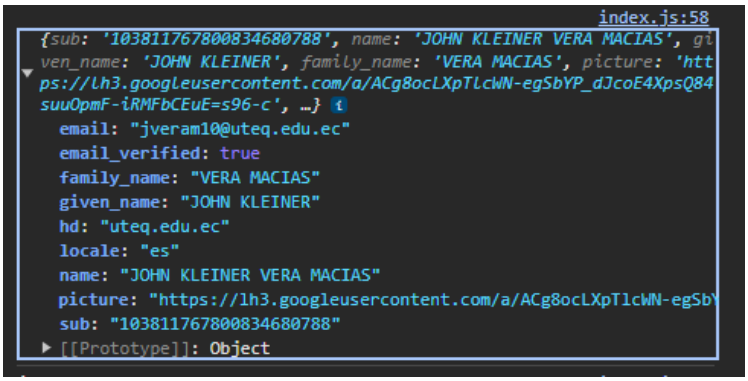

Ilustración 16. Datos en formato JSON devueltos por Google.

Con dichos datos se los procede a enviar a la Api para su respectivo procesamiento, esto se hace mediante el método GET como se ve a continuación:

```
const GoogleLogin = async (p_email) => {
    try {

        console.log(p_email);
        const result = await axios.get(
            process.env.NEXT_PUBLIC_ACCESLINK + "authgoogle/LoginGoogle/"+ p_email,
            {
                withCredentials: true,
            }
        );
```

Ilustración 17. Envió de los datos de Google desde el cliente a la API mediante el método Get.

```
.env
1   NEXT_PUBLIC_ACCESLINK = http://localhost:4099/
```

Ilustración 18. Ruta y puerto de la API.

Como vemos en la ilustración 17, se observa él envió de los datos de google a la Api en donde se procesaran para verificar el inicio de sesión, también, como se observar la petición se realiza a una ruta en específico de la API mediante el método GET.

```
1  //Rutas publicas
2  app.use("/authgoogle", authRoutes);
```

```
1  const {iniciarUserGoogle } = require('../controllers/Auth/auth-controller');
2
3  router.get('/LoginGoogle/:p_email',iniciarUserGoogle);
4
5  module.exports = router;
```

Ilustración 19. Ruta donde recibe la petición del cliente la API.

En la ilustración 18 se observa la ruta establecida en la API (servidor) mediante el método GET, en esta ruta se recibe la petición del cliente para iniciar sesión con google. En donde lo datos recibidos los enviar al método establecido en el controlador que se encarga de procesar la petición y verificar si el usuario se encuentra en la base de datos, así concediendo el permiso y la generación del token para almacenar en las cookie del navegador. Como se observa en la siguiente imagen:

```
7   const iniciarUserGoogle = async (req, res, next) => {
8       try {
9
10          const { p_email } = req.params;
11          const users = await pool.query('select * from verification_google($1)', [p_email]);
12          let verification = users.rows[0];
13          //Extraer el resultado del bool para saber si el login es correcto
14          let result = verification.verification;
15          //console.log('The result is:' + result);
16          //Si el Login fallo es decir es diferente del estado 1
17          if (result != 1) return res.status(401).json({ error: verification.mensaje });
18          //Si no entonces se le otorga un token xd
19          const token = jwt.sign({
20              exp: Math.floor(Date.now() / 1000) + 60 * 60 * 24 * 30,
21              p_email: p_email,
22          }, 'SECRET') //el secret deberia estan en el .env
23
24          const serialized = serialize('myTokenName', token, {
25              httpOnly: true,
26              secure: process.env.NODE_ENV === 'production',
27              sameSite: 'none',
28              maxAge: 1000 * 60 * 2, //dos minutos para hacer las pruebas
29              path: '/'
30          })
```

Ilustración 20. Procesamiento del inicio se sesión con los datos de Google.

6.10. Crud de tareas, aplicación de la arquitectura RestFull

A continuación se demuestra el funcionamiento del crud de las tareas aplicando la arquitectura RestFull. Primero, tenemos la parte de la interfaz de usuario para la gestión de tareas del lado del cliente:

Ilustración 21. Interfaz principal para la gestión de las tareas (Visualizar las tareas).

Ilustración 22. Interfaz para modificar o eliminar una tarea.

Ilustración 23. Interfaz para crear una tarea.

A continuación se muestra el manejo mediante código.

6.10.1. Listar Tareas (Uso GET)

Como se muestra en la imagen de abajo, aquí se envía la solicitud a la ruta establecida por la Api mediante el método Get, el cual se usa para solicitar datos.

Cliente Web

```
const Obtener_Tareas_Usuario = async () => {
    try {
        const response = await fetch(
            process.env.NEXT_PUBLIC_ACCESLINK +
            "tareas/ListaTareas/" +
            cookies.get("id_user") + "/"
            + r_id_grupo,
            {
                method: "GET",
                headers: { "Content-Type": "application/json" },
                credentials: "include",
            }
        );
```

Ilustración 24. Función para mandar la petición a la Api.

Como se muestra en la imagen de abajo, aquí se recibe la solicitud del cliente y la manda al método para ser procesada, así devolver una repuesta si es necesario.

API Rest (Servidor)

```
//Obtener recursos
router.get('/ListaTareas/:id/:id_grupo', listar_tareas_usuario);
```

Ilustración 25. Ruta establecida para procesar la petición.

6.10.2. Crear Tarea (Uso POST)

Como se muestra en la imagen de abajo, aquí se envía la solicitud a la ruta establecida por la Api mediante el método Get, el cual se usa para solicitar datos.

Cliente Web

```
const Crear_Tarea = async () => {
    try {
        const result = await axios.post(
            process.env.NEXT_PUBLIC_ACCESLINK + "tareas/CrearTarea",
            tarea,
            {
                withCredentials: true,
            }
        );
```

Ilustración 26. Función para mandar la petición a la Api.

Como se muestra en la imagen de abajo, aquí se recibe la solicitud del cliente y la manda al método para ser procesada, así devolver una repuesta si es necesario.

API Rest (Servidor)

```
//Crear recursos
router.post('/CrearTarea', crear_tareas_usuario);
```

Ilustración 27. Ruta establecida para procesar la petición.

6.10.3. Modificar Tarea (Uso PUT)

Como se muestra en la imagen de abajo, aquí se envía la solicitud a la ruta establecida por la Api mediante el método Get, el cual se usa para solicitar datos.

Cliente Web

```
const Modificar_Tarea = async () => {
    try {
        const result = await axios.put(
            process.env.NEXT_PUBLIC_ACCESLINK + "tareas/ModificarTarea",
            tarea,
            {
                withCredentials: true,
            }
        );
```

Ilustración 28. Función para mandar la petición a la Api.

Como se muestra en la imagen de abajo, aquí se recibe la solicitud del cliente y la manda al método para ser procesada, así devolver una repuesta si es necesario.

API Rest (Servidor)

```
//Editar recursos
router.put('/ModificarTarea',modificar_tareas_usuario);
```

Ilustración 29. Ruta establecida para procesar la petición.

6.10.4. Eliminar Tarea (Uso DELETE)

Como se muestra en la imagen de abajo, aquí se envía la solicitud a la ruta establecida por la Api mediante el método Get, el cual se usa para solicitar datos.

Cliente Web

```
const Eliminar_Tarea = async () => {
    try {
        const result = await axios.delete(
            process.env.NEXT_PUBLIC_ACCESLINK + "tareas/EliminarTarea/" + tarea.p_id_tareas,
            {
                withCredentials: true,
            }
        );
```

Ilustración 30. Método para mandar la petición a la Api.

Como se muestra en la imagen de abajo, aquí se recibe la solicitud del cliente y la manda al método para ser procesada, así devolver una repuesta si es necesario.

API Rest (Servidor)

```
//Eliminar recursos
router.delete('/EliminarTarea/:id',eliminar_tareas_usuario);
```

Ilustración 31. Ruta establecida para procesar la petición.

7. Código de la práctica:

Link: [Codigo de la practica en git hub]

8. Conclusión

En conclusión, en esta investigación se llevó a cabo una revisión exhaustiva de diversas arquitecturas empleadas en el desarrollo de aplicaciones distribuidas, centrándose especialmente en la planificación, diseño y estructuración de estas complejas configuraciones. Se ha destacado la importancia crucial de estas estructuras en la experiencia del usuario y el rendimiento general de los sistemas, con una exploración detallada mediante la creación de diagramas de arquitecturas distribuidas.

Con el objetivo de proporcionar una comprensión más concisa de las distintas arquitecturas, se han presentado dos diagramas: uno basado en la arquitectura cliente-servidor y otro en una estructura de tres capas. Ambos esquemas ofrecen una representación clara de los componentes y tecnologías utilizadas en cada capa de la aplicación.

Además, como parte integral de este estudio, se desarrolló de manera práctica la aplicación web 'Gestión Eficiente de Tareas', diseñada para facilitar la colaboración en entornos grupales. Este enfoque práctico permitió la aplicación concreta de las diferentes arquitecturas distribuidas estudiadas, brindando una visión detallada de su implementación y funcionamiento en un contexto real.

La solución al problema práctico se planteó utilizando la arquitectura RESTful, basándose en sus características y atributos, los cuales se adaptaron de manera adecuada a las necesidades del proyecto. En general, este estudio proporciona una comprensión profunda de las arquitecturas distribuidas y su aplicación en el mundo real, resaltando la importancia de una planificación y diseño adecuados para garantizar un sistema distribuido de calidad.

9. Bibliografía

[1] V. Raj y R. Sadam, «Patterns for migration of soa based applications to microservices architecture», *Journal of Web Engineering*, vol. 20, n.º 5, pp. 1229-1246, jul. 2021, doi: 10.13052/jwe1540-9589.2051.

[2] L. Turchet y M. De Cet, «A web-based distributed system for integrating mobile music in choral performance», *Pers Ubiquitous Comput*, vol. 27, n.º 5, pp. 1829-1842, oct. 2023, doi: 10.1007/s00779-023-01709-0.

[3] H. M. Heyn, E. Knauss, y P. Pelliccione, «A compositional approach to creating architecture frameworks with an application to distributed AI systems», *Journal of Systems and Software*, vol. 198, abr. 2023, doi: 10.1016/j.jss.2022.111604.

[4] Y. Zhang *et al.*, «Understanding and Detecting Software Upgrade Failures in Distributed Systems», en *SOSP 2021 - Proceedings of the 28th ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, Inc, oct. 2021, pp. 116-131. doi: 10.1145/3477132.3483577.

[5] A. Lafuente, «Introducción a los sistemas distribuidos Introducción a los Sistemas Distribuidos 1.2 Contenido», Medillin, oct. 2022. doi: 10.1109/ACCESS.2022.5479246.

[6] H. Noguchi, T. Demizu, M. Kataoka, y Y. Yamato, «Distributed Search Architecture for Object Tracking in the Internet of Things», *IEEE Access*, vol. 6, pp. 60152-60159, 2018, doi: 10.1109/ACCESS.2018.2875734.

[7] C. S. Mummidi y S. Kundu, «ACTION: Adaptive Cache Block Migration in Distributed Cache Architectures», *ACM Transactions on Architecture and Code Optimization*, vol. 20, n.º 2, mar. 2023, doi: 10.1145/3572911.

[8] H. Mora, J. Peral, A. Ferrandez, D. Gil, y J. Szymanski, «Distributed Architectures for Intensive Urban Computing: A Case Study on Smart Lighting for Sustainable Cities», *IEEE Access*, vol. 7, pp. 58449-58465, 2019, doi: 10.1109/ACCESS.2019.2914613.

[9] G. Blinowski, A. Ojdowska, y A. Przybylek, «Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation», *IEEE Access*, vol. 10, pp. 20357-20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[10] N. C. Mendonca, C. Box, C. Manolache, y L. Ryan, «The Monolith Strikes Back: Why Istio Migrated from Microservices to a Monolithic Architecture», *IEEE Softw*, vol. 38, n.º 5, pp. 17-22, sep. 2021, doi: 10.1109/MS.2021.3080335.

[11] V. Velepucha y P. Flores, «A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges», *IEEE Access*, vol. 11, pp. 88339-88358, 2023, doi: 10.1109/ACCESS.2023.3305687.

[12] O. Al-Debagy y P. Martinek, «A metrics framework for evaluating microservices architecture designs», *Journal of Web Engineering*, vol. 19, n.º 3-4, pp. 341-369, jun. 2020, doi: 10.13052/jwe1540-9589.19341.

[13] M. Shin, T. Kang, y H. Gomaa, «Design of Secure Connectors for Complex Message Communications in Software Architecture», en *ACM International Conference Proceeding Series*, Association for Computing Machinery, nov. 2021, pp. 21-28. doi: 10.1145/3501774.3501778.

[14] M. Hamza, «Software Architecture Design of a Serverless System», en *ACM International Conference Proceeding Series*, Association for Computing Machinery, jun. 2023, pp. 304-306. doi: 10.1145/3593434.3593471.

[15] C. S. Oh, S. Lee, C. Qian, H. Koo, y W. Lee, «DeView: Confining Progressive Web Applications by Debloating Web APIs», en *ACM International Conference Proceeding Series*, Association for Computing Machinery, dic. 2022, pp. 881-895. doi: 10.1145/3564625.3567987.

[16] M. Hamza, «Software Architecture Design of a Serverless System», en *ACM International Conference Proceeding Series*, Association for Computing Machinery, jun. 2023, pp. 304-306. doi: 10.1145/3593434.3593471.

[17] S. Ali, R. Alauldeen, y R. A. Khamees, «What is Client-Server System: Architecture, Issues and Challenge of Client-Server System (Review)»,

IEEE Access, vol. 4º, n.º 23, pp. 251-658, 20219, doi: 10.5281/zenodo.3673071.

[18] S. Chandra, S. Kumar, y S. kumar Singh, «An Introduction to Client Server Computing», San Fierro, ene. 2019. doi: 10.9790/0661-16195771.

[19] A. Tolba y A. Altameem, «A three-tier architecture for securing IOV communications using vehicular dependencies», *IEEE Access*, vol. 7, pp. 61331-61341, 2019, doi: 10.1109/ACCESS.2019.2903597.

[20] L. Li y H. Zhang, «Delay optimization strategy for service cache and task offloading in three-tier architecture mobile edge computing system», *IEEE Access*, vol. 8, pp. 170211-170224, 2020, doi: 10.1109/ACCESS.2020.3023771.

[21] X. Chen, Z. Ji, Y. Fan, y Y. Zhan, «Restful API Architecture Based on Laravel Framework», en *Journal of Physics: Conference Series*, Institute of Physics Publishing, nov. 2017. doi: 10.1088/1742-6596/910/1/012016.

[22] Z. Guo, D. Cao, D. Tjong, J. Yang, C. Schlesinger, y N. Polikarpova, «Type-directed program synthesis for RESTful APIs», en *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Association for Computing Machinery, jun. 2022, pp. 122-136. doi: 10.1145/3519939.3523450.

[23] A. Soni y V. Ranga, «API features individualizing of web services: REST and SOAP», *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, n.º 9 Special Issue, pp. 664-671, jul. 2019, doi: 10.35940/ijitee.I1107.0789S19.

[24] M. Castro-León, «Arquitectura orientada a servicios, un enfoque basado en proyectos», *Enseñanza y Aprendizaje de Ingeniería de Computadores*, dic. 2020, doi: 10.30827/digibug.32208.