Migración de serverbox hacia un sistema de microservicios aplicando arquitectura vertical

Migration of serverbox to a microservices system using vertical architecture

José Cruz Montes Baez
Facultad de Ciencias Básicas, Ingeniería y Tecnología, Universidad Autónoma de Tlaxcala, Apizaco, Tlaxcala, México
jcmontes711@gmail.com

Alberto Portilla Flores
Facultad de Ciencias Básicas, Ingeniería y Tecnología, Universidad Autónoma de Tlaxcala, Apizaco, Tlaxcala, México
alberto.portilla@gmail.com

doi: https://doi.org/10.36825/RITI.12.27.006

Recibido: Junio 22, 2024
Aceptado: Agosto 17, 2024

Resumen: La gestión eficiente de datos es esencial en la administración pública para el funcionamiento efectivo de los sistemas catastrales municipales. En 2010, se implementó el Programa de Modernización Catastral Multifinalitario, una alianza entre el gobierno del municipio de Puebla y Sistemas de Información Geográfica S.A. de C.V. (SIGSA), que incluía la plataforma *ServerBox*. Con el tiempo, la arquitectura monolítica de *ServerBox* presentó limitaciones en mantenimiento, actualización y despliegue de cambios. Este proyecto plantea la hipótesis de que migrar *ServerBox* a una arquitectura de microservicios distribuidos, aplicando una arquitectura vertical, puede superar estos desafíos y mejorar áreas operativas del sistema. El diseño propuesto establece una estructura modular y escalable que facilita el mantenimiento y la evolución del sistema. La implementación incluye el desarrollo de *TagHelpers* en ASP.NET Core para simplificar la creación de elementos HTML y el uso de *snippets* para agilizar la codificación. Además, se define un archivo de configuración JSON para gestionar múltiples entornos de despliegue. Este enfoque estratégico para la migración del sistema *ServerBox* a una arquitectura distribuida y vertical ofrece ventajas significativas en términos de escalabilidad, mantenimiento y eficiencia en el desarrollo de software, y puede servir de guía para proyectos similares en la administración pública.

Palabras clave: *Sistema Distribuido, Arquitectura Vertical, Microservicio, TagHelper, Migración.*

Abstract: Efficient data management is essential in public administration for the effective operation of municipal cadastral systems. In 2010, the Multifinality Cadastral Modernization Program was implemented, a partnership between the Puebla municipality government and Sistemas de Información Geográfica S.A. de C.V. (SIGSA), which included the ServerBox platform. Over time, the monolithic architecture of ServerBox showed limitations in maintenance, updates, and change deployment. This project hypothesizes that migrating ServerBox to a distributed microservices architecture, using a vertical approach, can overcome these challenges and improve operational areas of the system. The proposed design establishes a modular and scalable structure that facilitates system maintenance and evolution. The implementation includes the development of TagHelpers in ASP.NET Core to simplify HTML element creation and the use of snippets to streamline coding. Additionally, a JSON

configuration file is defined to manage multiple deployment environments. This strategic approach to migrating ServerBox to a distributed and vertical architecture offers significant advantages in terms of scalability, maintenance, and software development efficiency, and can serve as a guide for similar projects in public administration.

Keywords: *Distributed System, Vertical Architecture, Microservice, TagHelper, Migration.*

1. Introducción

En el ámbito de la administración pública, la gestión eficiente y precisa de datos es esencial para el funcionamiento efectivo de cualquier sistema catastral municipal. Sin embargo, muchos municipios se enfrentan a desafíos significativos en este aspecto, como lo evidencia el caso del sistema de catastro del municipio de Puebla en el año 2010. En ese año, el sistema enfrentaba retos considerables debido a la desvinculación entre su base de datos cartográfica y tabular, así como a la obsolescencia de sus sistemas y la falta de coordinación entre los distintos procesos.

Para abordar estos desafíos, se lanzó el Programa de Modernización Catastral Multifinalitario, una iniciativa conjunta entre el Gobierno Municipal y la empresa. Parte integral de esta modernización fue la implementación de la plataforma *ServerBox* [1]. Sin embargo, con el tiempo, la arquitectura monolítica de *ServerBox* comenzó a presentar limitaciones en términos de mantenimiento, actualización y despliegue de cambios.

Ante esta problemática, surge la hipótesis de que migrar la arquitectura de *ServerBox* hacia un sistema distribuido de microservicios, aplicando una arquitectura vertical, podría resolver estos desafíos y mejorar significativamente diversas áreas de operación del sistema.

Para alcanzar este objetivo, se han trazado objetivos específicos que abordan aspectos cruciales del desarrollo, tales como el diseño e implementación de una nueva versión del sistema. Estas metas específicas incluyen la mejora del tiempo de despliegue en producción, la optimización de las tareas de mantenimiento en las funcionalidades existentes, la garantía de una escalabilidad óptima del sistema, la reducción de la curva de aprendizaje para nuevos programadores, la minimización de la probabilidad de errores que puedan afectar la integridad del sistema y el establecimiento de pruebas unitarias para las funciones de las APIs.

Estos objetivos específicos se encuentran alineados con la misión general de migrar hacia una arquitectura de microservicios distribuidos, y una arquitectura vertical, representando así un enfoque profesional y estratégico hacia la evolución del sistema *ServerBox*.

En la sección 2 se presenta una revisión de la literatura relacionada, abordando las ventajas y desafíos de ambas arquitecturas. En la sección 3 se presentan los conceptos claves relacionados a este trabajo. En la sección 4 se muestra el diseño propuesto para el sistema distribuido. En la sección 5 se presenta el desarrollo de software y los elementos para la implementación de la propuesta de solución. En la sección 6 se presenta las pruebas y resultados obtenidos. En la sección 7 se presentan las conclusiones. En la sección 8 se presenta una breve discusión de lo aprendizaje obtenido con este trabajo, finalmente tenemos el trabajo futuro y la bibliografía.

2. Estado del arte

El trabajo de Montiel Luna [2] se centra en la definición de una arquitectura de software para aplicaciones empresariales multiplataforma, con su implementación en el módulo de registro de aspirantes del SIIA de la UATx. Este enfoque, basado en patrones como Cliente-Servidor, Microservicios y Capas, demuestra ser fácil de implementar, sin costos adicionales significativos y con beneficios notables en términos de mantenibilidad, escalabilidad y rendimiento, gracias a la aplicación de buenas prácticas del marco SCRUM.

En [3] se aborda la migración de un módulo de software a un microservicio en un contexto industrial, utilizando un caso de estudio de la empresa EDICOM. El proyecto demostró que la migración a microservicios tiene impactos positivos en el mantenimiento, eficiencia, escalabilidad y tolerancia a fallos del sistema, contribuyendo así a mejorar la calidad del servicio empresarial.

Por otro lado, en [4] se enfoca en la evaluación de las ventajas e inconvenientes de una arquitectura basada en microservicios frente a una arquitectura monolítica. El caso de estudio, que implica el desarrollo de una aplicación móvil para el comercio electrónico, revela que mientras las actividades de despliegue y pruebas pueden

ser más desafiantes en una solución basada en microservicios, las de mantenimiento e implementación son más manejables debido a la menor y más flexible base de código.

Estos trabajos muestran la diversidad de enfoques en la implementación de arquitecturas de software, desde la definición de estructuras para sistemas multiplataforma hasta la migración estratégica a microservicios en contextos industriales, y la evaluación exhaustiva de las ventajas e inconvenientes asociados.

Estos estudios contribuyen significativamente al entendimiento y aplicación práctica de las arquitecturas de software en distintos contextos, proporcionando valiosas lecciones y recomendaciones para futuras investigaciones y desarrollos en este campo.

3. Conceptos fundamentales

- Arquitectura monolítica: Es aquel que se concibe como un único elemento funcional donde sus prestaciones se ofrecen a través de una sola pieza de código que resuelve la presentación al usuario (*interface*), el acceso a los datos y la lógica algorítmica del problema que aborda [5].
- Sistema distribuido: Son aquellos en el que dos o más máquinas colaboran para la obtención de un resultado y están basados en las características de transparencia, eficiencia, flexibilidad, escalabilidad y fiabilidad [6].
- Microservicios: Es un enfoque al desarrollo de una única aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos, a menudo una API de recursos HTTP [7].
- Arquitectura vertical: Es un enfoque para la construcción de software que se centra en entregar funcionalidad completa y utilizable en cada iteración del ciclo de desarrollo [8].

4. Diseño

En la Figura 1 se establecen las etapas previas del proceso de migración.

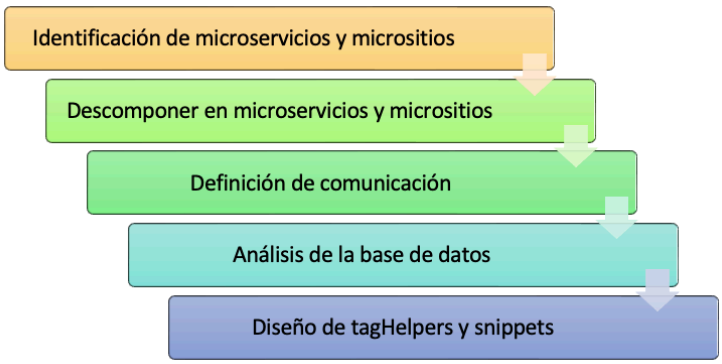

Figura 1. Proceso de migración de *ServerBox*.

Estas etapas contemplan las siguientes actividades:

1. **Identificación de microservicios y micrositios.** Analizar y descomponer las funcionalidades existentes para definir unidades de negocio independientes y conjuntos relacionados que puedan ser encapsulados en microservicios, permitiendo así una arquitectura distribuida más modular y escalable.
2. **Descomponer en microservicios y micrositios.** Descomponer el sistema monolítico, analizando funciones y dependencias para definir límites y agrupaciones coherentes, optimizando así la modularidad y escalabilidad del sistema distribuido.
3. **Definición de comunicación**. Establecer los puntos de contacto para que los diferentes componentes del sistema puedan interactuar, como puertas de entrada. Decidir estructura de los datos que se comparten entre ellos. Además, se establecen reglas sobre quién puede acceder a qué y cómo. Por último, se determina cómo se van a detectar y registrar errores.
4. **Análisis de la base de datos.** Se conserva la base de datos actual, ya que el alcance no implica la reestructuración de la misma. Pero aun así la migración se beneficia de la separación de un modelo que

contiene todas las tablas y procedimientos a 13 APIs, por lo que en la actualización del modelo se realiza en menor tiempo.

5. **Migración del sistema monolítico a distribuido.** Diseñar plantillas para microservicios y micrositios, junto con un conjunto de controles de diseño unificado. Definir fragmentos de código específicos para agilizar así el desarrollo. Por último, implementar pruebas unitarias para verificar el correcto funcionamiento de cada funcionalidad.

En Figura 2 se muestra la arquitectura propuesta, la cual consta de una librería global de funciones para los microservicios y los micrositios, un Gateway como punto de enlace entre los microservicios y los micrositios, 13 microservicios que interactúan con la base de datos tabular y cartográfica, 8 micrositios que agrupan el conjunto de herramientas para cada área de catastro, y un portal que funciona con punto de entrada del usuario al sistema.

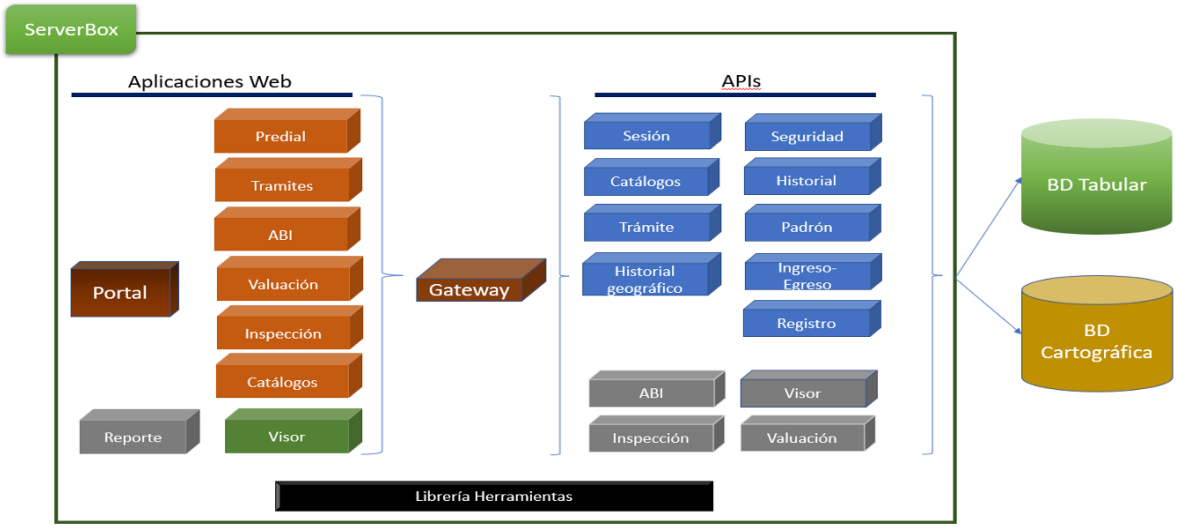

Figura 2. Arquitectura distribuida de *ServerBox*.

De manera detallada la estructura de la arquitectura distribuida es:

1. Librería de funciones globales para los microservicios y micrositios.
2. Gateway de APIs: es un componente de software que actúa como punto de entrada para gestionar y controlar el acceso a una API (Interfaz de Programación de Aplicaciones).
3. APIs (microservicios).
   a. Sesión: proporciona funciones específicas para gestionar la autenticación, autorización y el estado de la sesión de los usuarios.
   b. Seguridad: proporciona *EndPoint* CRUD para toda la parte de seguridad del sistema (usuarios y permisos).
   c. Catálogos: proporciona *EndPoint* CRUD para todos los catálogos del sistema.
   d. Historial: proporciona *EndPoint* CRUD para almacenar el historial de movimientos de las cuentas prediales, tales como pagos, adeudos, información del predio, información del propietario, información del copropietario, terreno y construcción.
   e. Trámites: proporciona *EndPoint* CRUD para el tratamiento de los tramites, solicitudes, etapas del trámite y actividades del usuario.
   f. Padrón: proporciona *EndPoint* CRUD para cualquier consulta relacionada con el predio, propietario, copropietario y cuentas prediales.
   g. Historial geográfico: proporciona *EndPoint* para almacenar el historial de los movimientos cartográficos.
   h. Ingreso – egresos: proporciona *EndPoint* CRUD para cualquier operación relacionada con adeudos, pagos, pagos cancelados, parcialidades.
   i. Registro: proporciona funciones específicas para supervisar el comportamiento de la aplicación, diagnosticar problemas y mejorar continuamente la calidad del software.

j. ABI: proporciona *EndPoint* CRUD para las operaciones exclusivas de ABI.
k. Visor: proporciona *EndPoint* CRUD para las operaciones exclusivas de visor municipal.
l. Inspección: proporciona *EndPoint* CRUD para las operaciones exclusivas de inspección.
m. Valuación: proporciona *EndPoint* CRUD para las operaciones exclusivas de valuación.

4. Web (micrositios).
   a. Portal. Aplicación web que será el núcleo de acceso a *ServerBox*.
   b. Predial. Aplicación web que contendrá todas las funcionalidades relacionadas a predial.
   c. Tramite. Aplicación web que contendrá todas las funcionalidades relacionadas a tramites.
   d. ABI. Aplicación web que contendrá todas las funcionalidades relacionadas a ABI.
   e. Valuación. Aplicación web que contendrá todas las funcionalidades relacionadas a valuación.
   f. Inspección. Aplicación web que contendrá todas las funcionalidades relacionadas a Inspección.
   g. Catálogos. Aplicación web que contendrá todas las funcionalidades relacionadas a catálogos, así como la administración de usuarios, perfiles.
   h. Reporte. Contenedor de los reportes de las aplicaciones web de los puntos ii al vii.
   i. Visor. Aplicación web que utiliza ArcGIS para realizar consultas espaciales vs datos tabulares.

La división de un microservicio aplicando la arquitectura vertical implica organizar cada funcionalidad del sistema en componentes independientes, en la Figura 3 se muestran estos componentes, *request*, *response* y *handler*, cada uno encargado de manejar un aspecto específico de la lógica de negocio. Esta organización se basa en el principio de responsabilidad única, donde cada componente se enfoca en una tarea claramente definida.

Figura 3. Arquitectura vertical.

Esta estructura de división vertical permite una organización clara y modular del código, lo que facilita el mantenimiento, la escalabilidad y la evolución del sistema a lo largo del tiempo. Cada componente se enfoca en una tarea específica y puede ser desarrollado, probado y desplegado de manera independiente, lo que favorece la agilidad y la eficiencia en el desarrollo de software.

A través de este diseño, se han identificado varias ventajas significativas que ofrecen tanto la arquitectura distribuida como la vertical en este contexto específico.

Una de las principales ventajas de la arquitectura distribuida es su capacidad para mejorar la escalabilidad del sistema. Al distribuir la carga de trabajo entre múltiples servidores, se puede aumentar la capacidad de respuesta del sistema y manejar un mayor volumen de usuarios y transacciones sin comprometer el rendimiento. Esto es especialmente relevante para *ServerBox*, donde la demanda de servicios puede variar considerablemente y se requiere una infraestructura flexible y adaptable.

Por otro lado, la adopción de la arquitectura vertical permite una mejor organización y modularidad del código, lo que facilita el mantenimiento y la evolución del sistema a lo largo del tiempo. Al dividir la funcionalidad en componentes independientes, cada uno es responsable de una tarea específica, se reduce la complejidad del código y se mejora la legibilidad y la mantenibilidad.

5. Implementación

Con la finalidad de cumplir con los objetivos de este proyecto se desarrollaron un conjunto de *TagHelpers* que son una forma de simplificar la creación de elementos HTML en ASP.NET Core, que permiten en primer lugar normalizar los controles de uso general, y en segundo disminuir las líneas de código necesarias, dejando la tarea al compilador para aplicar el cambio en tiempo de ejecución. En la Tabla 1 se muestran los 16 *TagHelpers* desarrollados con una breve descripción.

.

Tabla 1. Conjunto de *TagHelpers*.

TagHelper	Definición
XButton	*Helper* que define el diseño de un botón. Definiendo 4 variaciones solo icono, solo texto, icono y texto, icono para barra de herramientas.
XCinta	*Helper* que define el diseño de una cinta con las opciones información, precaución y error
XComboBox	*Helper* que define el diseño de un *comboBox* (*Select*) con su etiqueta. Requiere complemento de JavaScript para agregar la funcionalidad completa.
XDateRange	*Helper* que define el diseño de rango de fechas con su etiqueta.
XDateTime	*Helper* que define el diseño de un control de fecha y hora con su etiqueta.
XFlujo	*Helper* que define el diseño de la barra de herramientas con los botones necesarios para el manejo del flujo de un trámite, cuentas con los botones para lista de trámites, pausar, soltar, regresar, flujo alterno y finalizar.
XHidden	*Helper* que define el diseño de un input tipo *hidden* aplicando un formato más simple para su uso.
XIndicador	*Helper* que define el diseño de un indicador con su etiqueta, útil para mostrar totales en gráficas y reportes.
XLabel	*Helper* que define el diseño de una etiqueta donde se puede agregar opcionalmente un icono al inicio y/o al final.
XPassword	*Helper* que define el diseño de un control para la captura de la contraseña, agregando su etiqueta y un botón para ocultar/mostrar los caracteres.
XRadioButton	*Helper* que define el diseño de *radioButton* ajustando el estilo acorde al diseño general con su etiqueta.
XSwitch	*Helper* que define el diseño de una variante de *checkBox* ajustando el estilo acorde al diseño general con su etiqueta.
XTextArea	*Helper* que define el diseño de un *textArea* con su etiqueta e icono de ser necesario.
XTextBox	*Helper* que define el diseño de *textBox* editable con su etiqueta e icono de ser necesario.
ZInfoPredio	*Helper* que define el diseño de la barra de información básica de un predio, por ejemplo: propietario, cuenta predial, fecha de última modificación e indicadores (tipo persona vulnerable, tipo cuenta gobierno, uso del predio, adeudos, parcialidades, etc.).
ZInfoTramite	*Helper* que define el diseño de la barra de información del trámite contiene el número de trámite, nombre del trámite y cuenta predial.

Cada *TagHelper* tiene su propio conjunto de propiedades. Para ilustrar sus beneficios, consideremos el uso del *TagHelper X-TextoBox* en el archivo *inicio.cshtml*, sólo es necesario incluir la siguiente línea:

```
<X-TextBox X-Control="nombreControl" X-Etiqueta="Observaciones" X-Valor="uso de
tagHelper">
</X-TextBox>
```

Este elemento es fácilmente reutilizable en cualquier parte del proyecto, lo que simplifica la redundancia de código y fomenta la limpieza de los archivos. Esto permite que el enfoque del desarrollo se centre en la funcionalidad, en lugar de la implementación repetitiva en cada instancia. Además, el mantenimiento se vuelve significativamente más sencillo, ya que cualquier ajuste o modificación sólo requiere modificaciones en un único lugar. Sin el uso del *TagHelper*, el código requerido es:

```
<div class="form-group focused">
    <input type="text" class="form-control XNoSeleccionar" name="nombreControl"
value="uso de tagHelper"/>
      <span class="bar"></span>
      <label for="nombreControl"> Observaciones
      </label>
```

```
</div>
```

En la Figura 4 se ilustra el resultado obtenido al emplear este *TagHelper*.

Figura 4. Vista generada con el *tagHelper X-Texto*.

Para agilizar la codificación se crearon un conjunto de *snippets* (fragmentos de código) definidos en la Tabla 2 que permiten al desarrollador centrarse en la funcionalidad. Para utilizar los *snippet* en Visual Studio 2022 es necesario realizar lo siguiente:

```
Atajo + tab
```

Tabla 2. Conjunto de *snippets*.

Nombre	Atajo	Descripción
Método get api sin parámetros	jcActionGet	Permite agregar el fragmento de código que es utilizado en los controladores para solventar un método *get* sin parámetros.
Método get api con parámetros	jcActionGet Param	Permite agregar el fragmento de código que es utilizado en los controladores para solventar un método *get* con 1 parámetro.
Método post api	jcActionPost	Permite agregar el fragmento de código que es utilizado en los controladores para solventar un método *post*.
Notificación básica	jcNot1	Permite agregar el fragmento de código para la generación de una notificación básica la cual contiene clave de operación, detalle y observaciones.
Notificación anónima	jcNot2	Permite agregar el fragmento de código para la generación de una notificación anónima, por lo que se requiere otros cambios para identificar quién originó la notificación.
Notificación cuenta	jcNot3	Permite agregar el fragmento de código para la generación de una notificación de cuenta predial, por lo que se requiere otros cambios relaciones con la cuenta como el interlocutor, clave Catastral, número de trámite.
Excepción	jcNotX	Permite agregar el fragmento de código para la generación de una notificación de error, que incluye la excepción generada, el método que la origino, los parámetros de la solicitud y el usuario.
Request handler simple	jcRH1	Permite agregar el fragmento de código que define el *request handler* que devuelve un objeto DTO generado de la base de datos u objeto base.
Request handler lista	jcRH2	Permite agregar el fragmento de código que define el *request handler* que devuelve una lista de objetos DTO generados de la base de datos.
Request handler complejo	jcRH3	Permite agregar el fragmento de código que define el *request h*andler que devuelve un objeto que incluye un conjunto de objetos DTO generados de la base de datos u otro tipo de objetos.
MVC Controller Method	jcAction	Permite agregar el fragmento de código que es utilizado en los controladores, donde incluye una validación para determinar si se devuelve la vista, o el resultado de alguna operación CRUD.

Por ejemplo, para usar el *snippet jcActionPost*, en el editor se empieza a escribir el atajo a utilizar, como se puede ver en la Figura 5, al remarcarse se presiona doble *tab*.

Figura 5. Uso de un *snippet* I.

Al realizar esta acción el editor agrega el fragmento de código que se muestra en la Figura 6, donde **MyMethod** y **MyClave** son configurables.

```
[HttpPost("MyMethod")]
0 references | 0 changes | 0 authors, 0 changes
public async Task<IActionResult> MyMethod(MyMethodRequest request)
{
    var respuesta = await mediator.Send(request);
    await mediator.Publish(new Notificacion { claveOperacion = "MyClave", observaciones = respuesta.mensaje });
    return Ok(respuesta);
}
```

Figura 6. Uso de un *snippet* II.

La Figura 7 muestra el resultado final, en este caso al reemplazar *MyMethod* por ListaPermisos y presionar *tab* se rellena los demás *MyMethod* que se están en el fragmento recién agregado.

```
[HttpPost("Permisos")]
0 references | Jose Cruz Montes Baez, 86 days ago | 1 author, 2 changes
public async Task<IActionResult> Permisos(PermisosRequest request)
{
    var respuesta = await mediator.Send(request);
    await mediator.Publish(new Notificacion { claveOperacion = "c01f02", observaciones = respuesta.mensaje });
    return Ok(respuesta);
}
```

Figura 7. Uso de un *snippet* III.

Dependiendo de la cantidad de ambientes en los cuales se requiere desplegar los microservicios, es necesario definir un archivo individual para cada uno de acuerdo con la documentación de *ocelot*, incluyendo las correspondientes especificaciones de los *endpoints* del *gateway*.

Esta situación implica que los desarrolladores deben escribir código similar en varios archivos. Para abordar este desafío y facilitar la publicación teniendo en cuenta el número de despliegues y los entornos, se ha diseñado un archivo llamado "*config.iam*" en formato JSON, en la Figura 8 se muestra la estructura del archivo.

```
{
  "origen": {
    "ruta": "configuracion/Development",
    "host": "localhost",
    "GlobalConfiguration":"http://localhost:xxxx/",
    "puertos": [7000,7001,7002,XXXX]
  },
  "destino": [
    {
      "ruta": "configuracion/XCalidad",
      "host": "servidor",
      "puerto": 80,
      "GlobalConfiguration":"http://sDes/SB",
       "publicacion": {
         "_7000": "SBDes.Api.Sesion",
         "_7001": "SBDes.Api.Seguridad",
         "_7002": "SBDes.Api.Catalogo",
         "_7003": "SBDes.Api.IngresoEgreso",
         "_7004": "SBDes.Api.Historial",
         "_7005": "SBDes.Api.Tramite",
         "_7006": "SBDes.Api.Padron",
         "_7007": "SBDes.Api.ABI",
         "_7008": "SBDes.Api.HistorialGeografico",
         "_7009": "SBDes.Api.Inspeccion",
         "_7010": "SBDes.Api.Valuacion",
         "_7011": "SBDes.Api.Visor",
         "_7099": "SBDes.Api.Registro"
       }
  ]
}
```

Figura 8. Contenido archivo *config.iam*.

Este archivo se utiliza para implementar un algoritmo que lee su contenido y crea los directorios de publicación en base al arreglo del nodo "destino" que permite crear las *N* configuraciones, utilizando como base la configuración del ambiente de depuración. De esta manera, se asegura que todos los archivos tengan las mismas definiciones de métodos.

En el desarrollo del proyecto de transición de un sistema monolítico a una arquitectura de microservicios, se utilizó el marco de trabajo ágil Scrum para gestionar la implementación de manera eficiente. Este enfoque permitió mantener un ritmo constante de desarrollo, adaptarse rápidamente a los cambios y entregar valor de manera continua. La Figura 9 muestra la estructura base de los microservicios API el cual se desglosa de la siguiente manera:

- *Common*. Carpeta con clases generales.
    - *Behaviours*. Carpeta con clases que canalizan las peticiones.
    - *Extensions*. Carpeta con clases que extienden la funcionalidad de las clases.
    - *Handler*. Carpeta con clases que implementan manejadores globales.
- *Controllers*. Carpeta de controladores.
- *Features*. Carpeta con todas las implementaciones de funcionalidades con arquitectura vertical.
- *Infrastructure*. Carpeta que contiene los objetos de la base de datos.

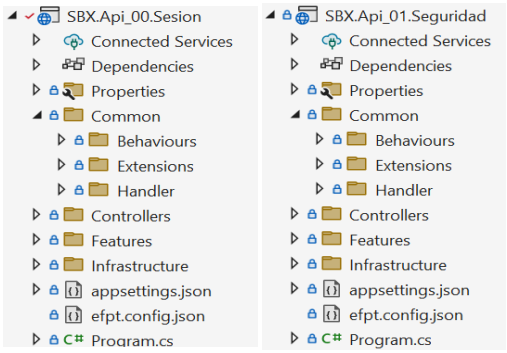

Figura 9. Estructura microservicios API.

En la Figura 10 se muestra la estructura base de los micrositios web el cual se desglosa de la siguiente manera:

- *Wwwroot*
    - CSS. Carpeta con las hojas de estilo.
    - Dist. Carpeta con archivos minificados para la publicación.
        - JS. Carpeta con archivos JavaScript minificados.
        - CSS. Carpeta con archivos de hojas de estilo minificados.
    - JS. Carpeta con archivos JavaScript sin minificar.
    - Lib. Carpeta con archivos de terceros JavaScript y hojas de estilo.
    - Recursos. Carpeta con archivos, imágenes, archivos de texto, etc.
- *Common*. Carpeta con clases generales.
- *Controllers*. Carpeta de controladores.
- *Domain*. Carpeta de clases globales.
- *View*. Carpeta con las vistas.

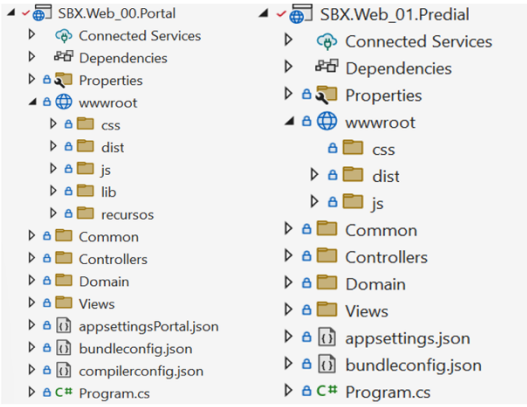

Figura 10. Estructura micrositio web.

Reuniendo todo lo anterior se definió la estructura general de la solución *ServerBox* que podemos ver en la Figura 11 como sistema distribuido aplicando una arquitectura vertical en el desarrollo de las APIs.

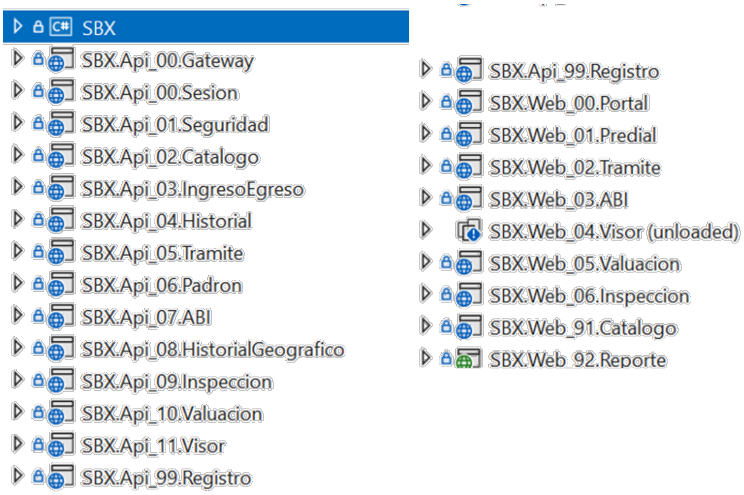

Figura 11. Estructura general solución *ServerBox*.

6. Pruebas y resultados

Para realizar las pruebas se usó la herramienta *Postman* que permite realizar solicitudes HTTP de manera eficiente y verificar las respuestas obtenidas. Usando *Postman*, podremos detectar errores en las APIs, validar los datos devueltos y asegurar la correcta integración de los microservicios.

Además, esta herramienta facilita la documentación y la repetición de pruebas, lo que es esencial para mantener la calidad y la fiabilidad del sistema a lo largo del tiempo. Los pasos para realizar las pruebas se indican en la Figura 12.

Figura 12. Pasos para prueba en *Postman*.

En la Figura 13 se muestra el resultado de ejecutar el *endPoint* que permite iniciar sesión, el cual está en la *url* https://servidor:444/SBX.Api.Gateway/Sesion/IniciarSesion todo esto sin necesidad de ejecutar toda la solución, en este caso sólo fue necesario el microservicio de sesión y el del *gateway*.

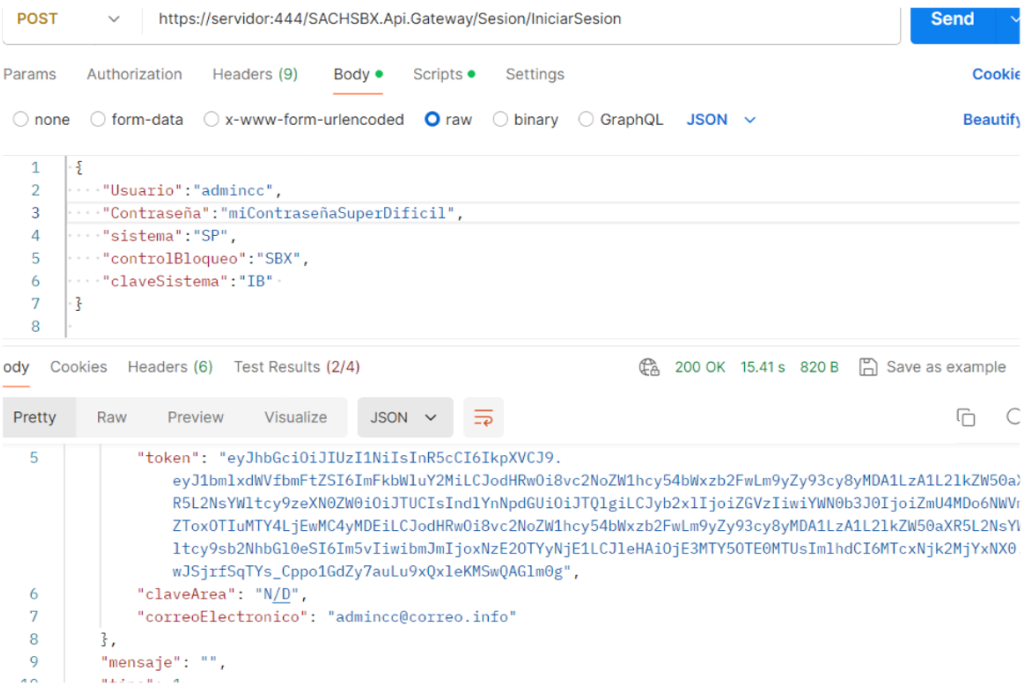

Figura 13. Prueba de iniciar sesión.

En la Tabla 3 se muestra la eficiencia de codificación que representan los *TagHelper* con una reducción en promedio del 89% de líneas de código, lo que implica código más limpio como se puede ver en la Figura 14.

Tabla 3. Eficiencia de codificación.

TagHelper	# líneas	# líneas originales	# eficiencia
XButton	1	3	67%
XCinta	1	10	90%
XComboBox	1	4	75%
XDateRange	1	6	83%
XDateTime	1	14	93%
XFlujo	1	78	99%
XIndicador	1	17	94%
XLabel	1	15	93%
XPassword	1	20	95%
XRadioButton	1	4	75%
XSwitch	1	9	89%
XTextArea	1	14	93%
XTextBox	1	15	93%
ZInformacionPredio	1	33	97%
ZInformacionTramite	1	39	97%
Promedio de eficiencia			89%

```
<X-InformacionPredio X-Control="barraPredio"></X-InformacionPredio>
<div class="row XContenido" style="max-height: 69vh;overflow: auto;">
    <div class="col-12 XFilaSinMargen">
        <kendo-tabstrip name="tabstrip">
            <popup-animation>...
            <items>
                <tabstrip-item text="General" selected="true">
                    <content>
                        <div class="card">
                            <div class="card-header">
                                Predio
                            </div>
                            <div class="card-body">
                                <div class="row">
                                    <div class="col-4">
                                        <X-Label X-Bind="info.direccionPredio" X-Etiqueta="Dirección"></X-Label>
                                    </div>
                                    <div class="col-4">
                                        <X-Label X-Bind="info.colonia" X-Etiqueta="Colonia"></X-Label>
                                    </div>
                                    <div class="col-2">
                                        <X-Label X-Bind="info.clasificacion" X-Etiqueta="Clasificación"></X-Label>
                                    </div>
                                    <div class="col-2">
                                        <X-Label X-Bind="info.cpPrppietario" X-Etiqueta="Codigo Postal"></X-Label>
                                    </div>
                                </div>
```

Figura 14. Código simplificado usando *TagHelpers*.

Para verificar la eficiencia de la generación de las publicaciones se registraron los tiempos de 100 ejercicios publicando el sistema monolítico y 100 ejercicios publicando el sistema distribuido con la variante que el sistema distribuido se genera diferente cantidad de microservicios y micrositios, con lo cual se obtienen tiempos menores al publicar sólo una parte, en comparación de siempre estar publicando todo el sistema, tal como muestra la Figura 15.

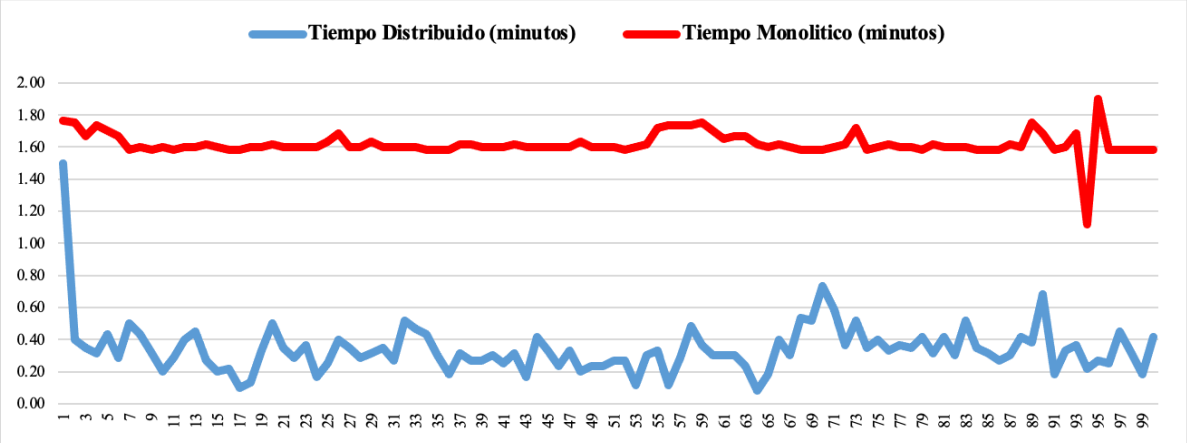

Figura 15. Tiempo de generación de publicaciones.

En la Figura 16 se muestra el tiempo acumulativo, donde la pendiente para el sistema monolítico es mayor que la del sistema distribuido. Esto implica que se ha invertido más tiempo para publicar cambios en el sistema monolítico en comparación con el sistema distribuido.

En una arquitectura distribuida, no es necesario publicar todo el sistema cada vez que se realizan cambios; sólo se despliegan las partes que han sido editadas. Esta diferencia en las pendientes refleja la mayor eficiencia y menor tiempo de actualización del sistema distribuido, demostrando su capacidad para manejar cambios de manera más ágil y escalable.

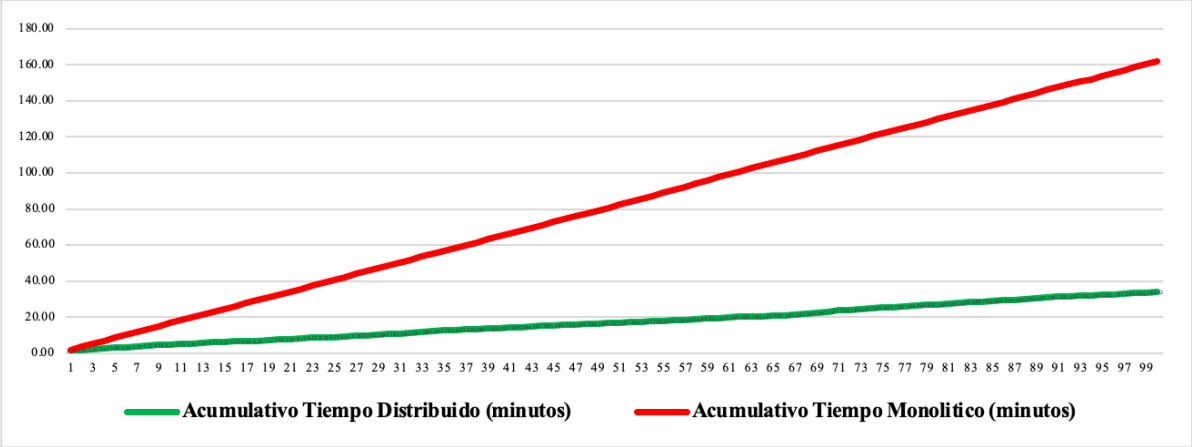

Figura 16. Tiempo acumulativo de publicación.

7. Conclusiones

Al concluir este proyecto de migración, se puede afirmar que presenta múltiples beneficios que respaldan la hipótesis planteada. Aunque inicialmente se presentaron desafíos debido al método de desarrollo anterior (monolítico), con el tiempo se hicieron evidentes las ventajas de la nueva estructura. Al principio, la arquitectura vertical parecía extraña, pero sus beneficios se volvieron claros gradualmente.

La implementación del sistema distribuido con arquitectura vertical ha sido un éxito, ya que se alcanzaron los objetivos propuestos. Aunque el primer despliegue en el servidor fue desafiante, las actualizaciones posteriores fueron más fluidas y rápidas, al no tener que publicar todo el sistema, sino solo los microservicios que se modificaron.

La arquitectura vertical aisló las funcionalidades, facilitando la comprensión del código y reduciendo la curva de aprendizaje para nuevos integrantes del grupo de desarrollo. Además, las pruebas con *Postman* fueron vitales para detectar errores en las APIs y descubrir los datos devueltos sin necesidad de ejecutar toda la solución.

Los *snippets* y *TagHelpers* han demostrado ser útiles para avanzar en el trabajo, facilitando el mantenimiento y la adición de nuevas funciones, y permitiendo centrarse en el desarrollo y/o actualización de cada funcionalidad.

Por lo tanto, migrar un sistema monolítico a uno distribuido requiere un análisis cuidadoso y debe realizarse en proyectos grandes y complejos, ya que, en proyectos pequeños, los beneficios se ven limitados.

8. Discusión

La investigación sobre la migración de un sistema monolítico a uno distribuido ha demostrado un gran aprendizaje y mejoras significativas en la eficiencia del proceso. Se logró que la separación del sistema fuera imperceptible para el usuario final, manteniendo la experiencia de uso intacta.

La división en módulos más pequeños y funciones aisladas simplificó el desarrollo, haciendo que se trabaje como si fueran pequeños proyectos individuales. Esto ha resultado especialmente útil en sistemas que requieren una evolución constante, permitiendo agregar nuevas funcionalidades sin recompilar todo el proyecto, lo que reduce tiempos y minimiza errores.

El mantenimiento también ha superado las expectativas. La posibilidad de ajustar una función sin afectar al resto del sistema mejora significativamente la flexibilidad y estabilidad del software.

No obstante, la transición presentó desafíos, principalmente debido a la resistencia al cambio de paradigmas de trabajo. Sin embargo, al superar esta barrera, se evidencian claramente los beneficios de un sistema distribuido.

Se espera que estos hallazgos contribuyan al debate sobre las mejores prácticas en la industria del software y ayuden a otros profesionales a enfrentar los desafíos asociados con la migración de sistemas monolíticos.

9. Trabajo futuro

Para futuras investigaciones, se recomienda explorar la migración de la base de datos a una arquitectura distribuida como una forma de mejorar aún más el rendimiento y la escalabilidad del sistema. Este enfoque podría permitir que cada microservicio gestione su propio conjunto de datos, superando las limitaciones de las arquitecturas monolíticas y proporcionando una solución más flexible y escalable. Investigar cómo la implementación de una base de datos distribuida puede optimizar la interacción entre servicios y facilitar la escalabilidad independiente de cada componente sería un área de gran interés.

Otro aspecto para futuras investigaciones es el desarrollo de métodos más eficientes para la publicación de microservicios y micrositios. Se podría investigar la automatización del proceso de publicación, la integración de herramientas avanzadas de integración continua (CI), y la implementación de prácticas que minimicen la intervención manual. Estos esfuerzos podrían resultar en una mayor eficiencia en el despliegue, reducción de errores y adaptación más rápida a nuevos requisitos, ofreciendo una base sólida para la evolución y el mantenimiento continuo de sistemas distribuidos en contextos similares.

10. Referencias

[1] SIGSA. (2024). *Serverbox | SIGSA.* https://www.sigsa.info/es-mx/productos/server/serverbox

[2] Montiel Luna, F. (2023). *Arquitectura de software para aplicaciones empresariales multiplataforma*. Congreso Internacional de Investigación, Puebla.

[3] García Gonzálvez, F. (2023). *Migración de un módulo software a un microservicio en un contexto industria* [Tesis de Grado]. Universitat Politécnica de Valencia, España. http://hdl.handle.net/10251/192801

[4] Iranzo Jiménez, V. A. (2018). *Desarrollo de software basado en microservicios: un caso de estudio para evaluar sus ventajas e inconvenientes* [Tesis de Grado]. Universitat Politécnica de Valencia, España. http://hdl.handle.net/10251/111173

[5] Arcidiacono, J. (2020). *Arquitectura de microservicios distribuidos para una plataforma que orquesta actividades orientadas a la recolección de datos con intervención humana.* XXIII Concurso de Trabajos Estudiantiles (EST), Argentina. http://sedici.unlp.edu.ar/handle/10915/115887

[6] Islas Vera, M. P. (2004). *Libreria SCII para la creacion de imagenes irreales en un ambiente distribuido* [Tesis de Licenciatura]. Universidad de las Américas Puebla, México. http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/islas_v_mp/

[7] Fowler, M., Lewis, J. (2014). *Microservices*. https://martinfowler.com/articles/microservices.html

[8] Santamaria, A. (2022). *Vertical Slice Architecture (+DDD).* https://www.plainconcepts.com/es/recursos/vertical-slice-architecture/

