https://doi.org/10.22201/dgtic.ctud.2025.3.1.85

Vol. 3, Núm. 1. enero-marzo 2025, págs. 8 - 17

Implementación de una arquitectura modular en Laravel para sistemas web mantenibles

Información del reporte:

Para citar este reporte técnico:

Fonseca Márquez, K. A. (2025). Implementación de una arquitectura modular en Laravel para sistemas web mantenibles. *Cuadernos Técnicos Universitarios de la DGTIC*, 3 (1) páginas(8 - 17).

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85

Karla Alejandra Fonseca Márquez

Dirección General de Cómputo y de Tecnologías de Información y Comunicación

Universidad Nacional Autónoma de México

karlafm@unam.mx

ORCID: 0009-0002-6382-8182

Resumen

Para mejorar la cohesión y reducir el acoplamiento de los componentes en sistemas web complejos, se diseñó y desarrolló una arquitectura modular en Laravel que busca superar las limitaciones del enfoque tradicional basado en el patrón MVC. Para ello, se adoptaron principios de diseño que incluían la creación de componentes especializados, como acciones para la lógica de negocio, consultas separadas en clases específicas, y la utilización de objetos para la transferencia y validación de datos. Se implementó una estructura de directorios modular que facilitó el mantenimiento y escalabilidad del código, apoyándose en el estándar PSR-4 para la carga automática de clases. La metodología incluyó la adopción de estándares de codificación, como PSR-12, y el uso de herramientas de análisis estático y control de versiones para garantizar la calidad y coherencia del proyecto. En términos de resultados, se logró una arquitectura más flexible y comprensible, con un código más limpio y fácil de modificar. Como conclusión, la adopción de una arquitectura modular en Laravel contribuyó significativamente a la mejora de la calidad del código y la escalabilidad de los sistemas desarrollados. Se recomienda continuar con la implementación de buenas prácticas de codificación, así como realizar capacitaciones periódicas para mantener la coherencia en el desarrollo y facilitar la integración de nuevas funcionalidades.

Palabras clave:

Mantenibilidad, arquitectura modular, Laravel, patrón MVC.

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 9 - 17

1. INTRODUCCIÓN

El patrón Modelo-Vista-Controlador (MVC) ha sido una de las arquitecturas más adoptadas en el desarrollo de aplicaciones web debido a su capacidad para dividir la funcionalidad del software en tres capas principales: Modelo, que gestiona la lógica de datos y las interacciones con la base de datos; Vista, que presenta la interfaz de usuario y los datos al cliente; y Controlador, que actúa como intermediario, coordinando las solicitudes del usuario y facilitando la comunicación entre el modelo y la vista (Velasco-Elizondo et al., 2018). Esta separación busca simplificar el desarrollo y mantenimiento del software, promoviendo una estructura clara y organizada.

Sin embargo, la implementación extensiva del patrón MVC ha expuesto varios desafíos cuando no se aplica correctamente ni se adapta adecuadamente a las necesidades específicas de cada proyecto. Entre los desafíos, destacan controladores sobrecargados con múltiples responsabilidades, lógica compleja en modelos, así como consultas directas a la base de datos desde capas que deberían limitarse a coordinar o presentar información (Aniche, 2016; Velasco-Elizondo et al., 2018). Además, su estructura predeterminada, basada en una separación genérica de responsabilidades para aplicaciones web, carece de un enfoque orientado al dominio particular de cada sistema, lo que puede dificultar su mantenibilidad y escalabilidad (Griffin, 2021, p. 21).

Aunque el patrón MVC proporciona una base adecuada para estructurar sistemas web, su correcta implementación requiere un enfoque riguroso para evitar malas prácticas que puedan afectar la calidad del software, dificultar su mantenimiento o limitar la capacidad de los equipos para escalar y añadir nuevas funcionalidades de forma eficiente.

El presente reporte técnico tiene como objetivo describir y analizar el diseño e implementación de una arquitectura modular en Laravel, un marco de trabajo PHP ampliamente utilizado en la industria por su enfoque orientado a objetos y su versatilidad. Diseñada en 2023 para el desarrollo de aplicaciones web, esta estructura busca superar las limitaciones del patrón MVC mediante la fragmentación de funcionalidades en módulos y componentes con responsabilidades claramente definidas, optimizando así el desarrollo, la prueba y la actualización de sistemas web complejos para mejorar su mantenibilidad, escalabilidad y calidad.

2. METODOLOGÍA

La metodología seguida para el diseño e implementación de una arquitectura modular en Laravel se basó en un enfoque iterativo y sistemático que permitió adaptar, optimizar y superar las limitaciones identificadas en estructuras tradicionales basadas en el patrón Modelo-Vista-Controlador (MVC). El objetivo central fue mejorar la escalabilidad, la cohesión, y la mantenibilidad del código, garantizando al mismo tiempo la calidad del desarrollo mediante la aplicación de buenas prácticas, ampliamente documentadas en la literatura académica y técnica, de arquitectura de software y diseño basado en dominio (Fowler, 2002, Griffin, 2021; Martin, 2018; Visser et al., 2016). Además, se priorizó la mitigación de problemas típicos de las arquitecturas monolíticas asociadas al MVC (Aniche, 2016; Velasco-Elizondo et al., 2018).

El diseño de la nueva estructura modular incluyó un esquema jerárquico que organiza componentes por dominios de negocio, lo que redujo el acoplamiento y aseguró una clara separación de responsabilidades.

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 Vol. 3, Núm. 1. enero-marzo 2025, págs. 10 - 17

Se aplicaron principios de diseño limpio como el de Responsabilidad Única (SRP), que establece que cada clase o módulo debe tener una única razón para cambiar, lo cual mejora la cohesión y la simplicidad, y el principio de Cierre Común (CCP), que propone agrupar en módulos aquellas clases que cambian por las mismas razones, lo que minimiza el impacto de los cambios y facilita el mantenimiento (Martin, 2018, p. 13).

Las decisiones metodológicas estuvieron respaldadas por estudios y bibliografía académica que destacan la necesidad de modularidad y la segmentación de responsabilidades en sistemas complejos (Barde, 2023), validando además el uso de estándares de codificación y herramientas que impactan positivamente en la calidad y escalabilidad del software (Fowler, 2018). Esta fundamentación asegura que cada elección técnica realizada respondió a criterios objetivos y no al azar, maximizando el potencial de desarrollo y mitigando riesgos asociados al acoplamiento y la complejidad.

2.1 DESARROLLO

El desarrollo de la arquitectura modular en Laravel se llevó a cabo mediante la implementación de una serie de componentes y directrices que aseguran la separación de responsabilidades y la modularización del sistema. A continuación, se describen los elementos clave del proceso de desarrollo:

Componentes y su rol en la arquitectura modular

En el diseño de la arquitectura modular, se definieron varios tipos de elementos que cumplen funciones específicas para garantizar una clara separación de responsabilidades, los cuales se presentan en la Tabla 1.

Tabla 1

Tipos de elementos en la arquitectura modular y sus responsabilidades

Tipo de elemento	Responsabilidad
Modelos	Representan las tablas y relaciones de la base de datos, limitándose a operaciones básicas de manipulación de datos
QueryBuilders	Encapsulan consultas avanzadas con la base de datos, manteniendo los modelos simples.
Actions	Contienen la lógica de negocio y representan casos de uso específicos del sistema
Enums	Definen conjuntos de constantes significativas para mejorar la comprensión y mantenibilidad del código
Excepciones	Gestionan los errores de forma independiente del flujo principal
Filters	Implementan la lógica de búsqueda en los listados de información
Componentes Livewire[1]	Permite interacciones reactivas entre la vista y la lógica del sistema, reemplazando los controladores tradicionales y mejorando la experiencia del usuario.
DTOs/FormObjects	Transportan datos entre capas de la aplicación y validan los datos, utilizando spatie/laravel-data[2] en Livewire 2 y clases FormObject nativas en Livewire 3.

1 https://livewire.laravel.com/

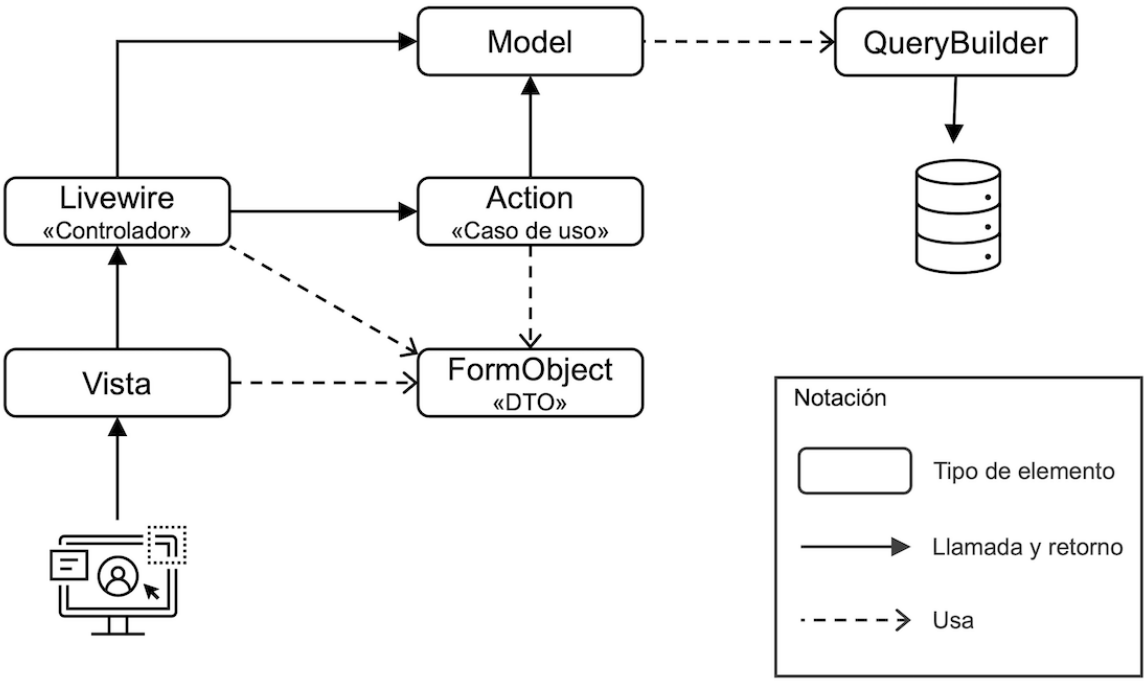
2 https://spatie.be/docs/laravel-data/v4/introduction

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 11 - 17

Los diferentes tipos de elementos del sistema se interrelacionan de acuerdo con los flujos de datos definidos en el proceso, tal como se observa en la Figura 1.

Figura 1

Representación conceptual de la arquitectura modular



Nota: La interfaz de usuario recibe las solicitudes del usuario y las envía como eventos al controlador. La transferencia de datos entre la vista, el controlador y la acción se realiza mediante DTOs (Data Transfer Objects), que tienen la responsabilidad de validar que los datos sean correctos. El controlador ejecuta el caso de uso correspondiente, el cual interactúa con otros componentes como modelos, filtros, y query builders, entre otros. Finalmente, el controlador actualiza la vista con el resultado de la acción.

En la Tabla 2, se presentan las responsabilidades típicas en una aplicación web comparando su implementación en un enfoque MVC tradicional y en la arquitectura modular propuesta.

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 12 - 17

Tabla 2

Comparación de responsabilidades entre MVC tradicional y la arquitectura modular

Responsabilidad	MVC tradicional	Arquitectura modular
Validación de datos del usuario	Realizada directamente en el controlador	Delegada a DTOs/FormObjects, que también transportan datos
Transformación o limpieza de datos	Realizada directamente en el controlador	Gestionada por DTOs/FormObjects
Lógica de negocio	Mezclada en el controlador y el modelo	Encapsulada en Actions, representando casos de uso específicos
Interacción con la base de datos	Realizada por el modelo y directo desde un controlador promiscuo	Gestionada por Modelos y consultas avanzadas a través de *QueryBuilders*
Lógica de búsqueda en listados	Agregada manualmente en el controlador o modelo	Encapsulada en *Filters*
Definición de constantes significativas	Definidas como valores literales en el código	Centralizadas en *Enums* para mejorar la mantenibilidad
Manejo de errores	Captura y manejo directamente en el controlador	Delegada a Excepciones, separando el flujo principal
Generación de respuesta a la vista	Realizada directamente en el controlador	Delegada a Componentes Livewire para mayor interactividad
Actualización reactiva de la interfaz	Requiere lógica explícita en el controlador	Realizada automáticamente mediante Componentes Livewire

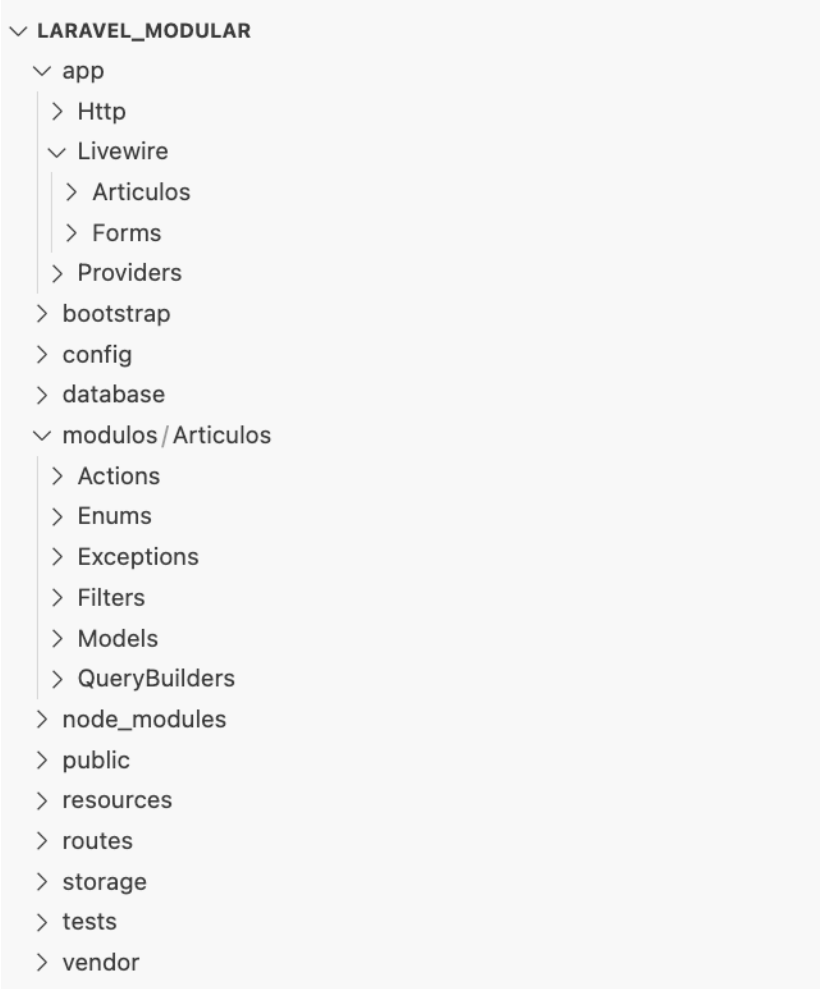
Estructuración de la aplicación por módulos

Para fomentar la modularidad, se creó una nueva estructura de directorios que agrupa componentes y funcionalidades en módulos independientes (ver Figura 2). Esta organización permite que cada módulo represente un dominio o conjunto de funcionalidades específicas, mejorando la analizabilidad y mantenibilidad. Al dividir la funcionalidad en múltiples módulos con reglas de acceso bien definidas, se facilita la gestión y el mantenimiento de cada módulo de manera independiente, lo que promueve un desarrollo más ágil y eficiente (Barde, 2023).

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 13 - 17

Figura 2

Estructura modular de directorios de una aplicación Laravel



Nota: Se añadió un subdirectorio denominado *modulos*, donde se organizaron subdirectorios correspondientes a los módulos definidos en el sistema. En la figura, se muestra un ejemplo utilizando el módulo "Artículos". Los componentes de Livewire y sus formularios deben organizarse siguiendo la misma estructura por módulos.

La carga automática de clases mediante el estándar PSR-4[3] facilitó el registro de nuevos módulos y su vinculación mediante *namespaces*. Este estándar establece un mapeo flexible de *namespaces* a directorios, facilitando la carga automática de clases según su ubicación, lo que simplifica la organización del código y mejora su mantenimiento.

3 https://www.php-fig.org/psr/psr-4/

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 14 - 17

Cumplimiento de estándares y herramientas

Se adoptó el estándar PSR-12[4] como guía para asegurar consistencia en el estilo de programación, apoyado por herramientas de análisis estático como PHP Insights[5] para evaluar el código en términos de calidad, complejidad y estilo. PSR-12 extiende las recomendaciones básicas de PSR-1 y PSR-2, proporcionando reglas detalladas sobre formato, indentación, espacios en blanco, declaraciones y uso de *namespaces*, entre otros aspectos, para fomentar un código claro y legible. Para el desarrollo, se utilizó Visual Studio Code con extensiones que automatizan la corrección de violaciones al estándar. Composer y NPM fueron utilizados para manejar dependencias, mientras que Laravel Sail[6] garantizó entornos locales consistentes con Docker.

Documentación y capacitación

Se realizó un taller de capacitación para los desarrolladores, asegurando la familiarización con la arquitectura propuesta y las tecnologías utilizadas. La documentación del proyecto se mantuvo en una página colaborativa dentro de GitLab, mientras que la documentación de requerimientos y casos de uso fue almacenada en un repositorio documental. Se realizaron reuniones de seguimiento diarias, sesiones para revisión de dudas así como sesiones periódicas de retrospectiva.

Control de versiones y flujo de trabajo

El desarrollo adoptó un flujo basado en ramas, donde cada funcionalidad o corrección se trabajó de forma aislada antes de integrarse al repositorio principal. Las solicitudes de fusión (*merge requests*) facilitaron la revisión del código, para asegurar el cumplimiento tanto de la arquitectura definida como de los estándares establecidos antes de su integración final (ver Figura 3).

4 https://www.php-fig.org/psr/psr-12/

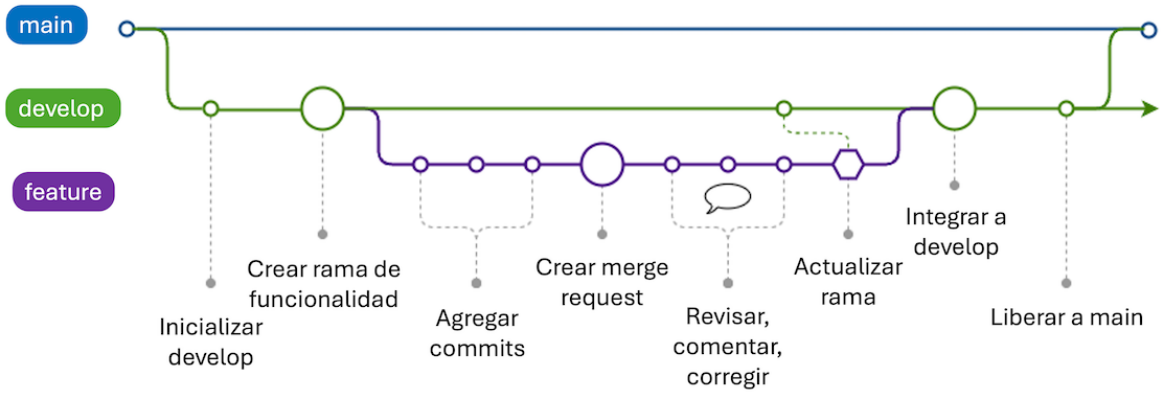
5 https://phpinsights.com/

6 https://laravel.com/docs/11.x/sail

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 15 - 17

Figura 3

Flujo de trabajo con control de versiones basado en ramas y merge requests



Nota: En el flujo de trabajo con Git, cada integrante del equipo de desarrollo crea una rama específica para cada funcionalidad que se le asigna. Una vez que completa su desarrollo, genera un merge request para que su trabajo sea revisado. El líder técnico del proyecto, o un comité de revisión designado, se encarga de evaluar los merge requests, proporcionando observaciones o retroalimentación según sea necesario. Cuando la funcionalidad cumple con los criterios establecidos y ha pasado las revisiones, se fusiona en la rama develop. Posteriormente, el líder técnico es responsable de integrar los cambios de la rama develop en la rama main como parte del proceso de liberación, asegurando que los cambios estén listos para su despliegue en el entorno de producción.

3. RESULTADOS

La implementación de la arquitectura modular en Laravel permitió resolver de manera significativa los problemas de escalabilidad y mantenibilidad identificados en proyectos previos basados en el patrón MVC tradicional. Como resultado, se obtuvo una estructura más organizada, con una clara separación de responsabilidades entre los módulos, lo que facilitó la gestión del código y la adición de nuevas funcionalidades sin afectar al sistema en su conjunto. Esta mejora también permitió un proceso de desarrollo más ágil y ordenado, reduciendo la complejidad de mantenimiento y aumentando la capacidad del equipo para implementar cambios de manera eficiente, con una notable disminución de los errores relacionados con dependencias entre módulos.

El enfoque modular evitó problemas frecuentes como controladores excesivamente grandes con múltiples responsabilidades, lo que anteriormente conducía a dificultades de mantenimiento y a código fuente más propenso a errores. Al modularizar los componentes, cada módulo pudo concentrarse en tareas específicas, mejorando la cohesión y reduciendo el acoplamiento. Esto, a su vez, simplificó la revisión y el mantenimiento del código al limitar los puntos de cambio.

Se destacaron, además, algunos hallazgos relevantes que mejoraron la experiencia de los equipos de desarrollo. La modularización del código permitió una mayor facilidad en las pruebas unitarias y en la integración de nuevas tecnologías sin comprometer la estabilidad del sistema. Sin embargo, se identificaron ciertos desafíos relacionados con la curva de aprendizaje de los desarrolladores, quienes

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 16 - 17

necesitaron tiempo para familiarizarse con la nueva estructura modular y adaptarse a los nuevos flujos de trabajo. Este aspecto incrementó ligeramente el tiempo de integración de la nueva arquitectura en los primeros proyectos implementados. Se llevaron a cabo sesiones de retrospectiva en las que los desarrolladores destacaron que la nueva arquitectura modular no sólo simplificaba el desarrollo de nuevas funcionalidades, sino que también agilizaba la integración de requerimientos adicionales y la implementación de ajustes, reduciendo el esfuerzo necesario y mejorando la productividad general del equipo.

En comparación con enfoques anteriores, como el uso del patrón MVC clásico, los resultados obtenidos confirman que la arquitectura modular no sólo resuelve los problemas iniciales de escalabilidad y mantenimiento, sino que también mejora la flexibilidad y la capacidad de adaptación de los sistemas a nuevos requerimientos y cambios tecnológicos. En este sentido, los resultados obtenidos se alinean con las expectativas iniciales y contribuyen a la mejora continua de los procesos de desarrollo.

Los resultados obtenidos reflejan un avance significativo en la calidad de los sistemas desarrollados, así como en las capacidades del equipo para abordar nuevos desafíos de manera modular, escalable y efectiva. La mejora en la calidad fue validada a través de diversas métricas e indicadores, incluyendo una reducción en el tiempo promedio de desarrollo de nuevas funcionalidades, una disminución en la cantidad y severidad de errores reportados durante la etapa de pruebas, y una mejora en la puntuación obtenida mediante herramientas de análisis estático como PHP Insights. El enfoque adoptado ha generado una base sólida para futuras mejoras, asegurando que el sistema pueda evolucionar en respuesta a las necesidades del negocio y del entorno tecnológico.

4. CONCLUSIONES

El diseño de una arquitectura modular en Laravel adaptada al patrón MVC surgió como una respuesta al problema de la complejidad y baja mantenibilidad que puede presentar una estructura convencional de modelos, vistas y controladores. Los principales retos identificados incluían controladores y modelos con responsabilidades excesivas, lo que complicaba su mantenimiento y actualización, así como la presencia de reglas de negocio dispersas que afectaban la cohesión y escalabilidad del sistema.

La solución propuesta se centró en redefinir las responsabilidades de los componentes clave mediante la creación de módulos independientes y el uso de clases con responsabilidades específicas. Este enfoque resultó en una estructura de código modular, limpia y mantenible, que aumentó la cohesión y redujo el acoplamiento. Adicionalmente, se observó una mejora significativa en la capacidad del sistema para adaptarse a nuevas funcionalidades, gracias a la organización y separación de las responsabilidades. La adopción de estándares de codificación y herramientas de análisis automatizado contribuyó a elevar la calidad del código, mientras que el uso de Livewire permitió una mejor experiencia de usuario mediante interacciones reactivas en la interfaz.

Para continuar con la mejora de la arquitectura y mitigar posibles riesgos, se recomienda mantener una supervisión constante de la modularidad y simplicidad de cada componente, evitando la acumulación de responsabilidades en clases individuales. Además, se sugiere realizar revisiones periódicas del código y capacitaciones para garantizar que todo el equipo esté alineado con los principios de desarrollo adoptados. Implementar métricas de calidad y mantenimiento de código puede facilitar la detección temprana de problemas, mejorando la capacidad de respuesta ante cambios o nuevas necesidades.

https://doi.org/10.22201/dgtic.ctud.2025.3.1.85 **Vol. 3, Núm. 1.** enero-marzo 2025, págs. 17 - 17

Estas acciones, junto con el uso continuo de herramientas de análisis estático, control de versiones y pruebas automatizadas, proporcionarán un marco robusto para mantener la sostenibilidad, escalabilidad y eficiencia del sistema desarrollado.

AGRADECIMIENTOS

Quiero expresar mi sincero agradecimiento a los colaboradores de la Subdirección de Sistemas Integrados y del Departamento de Desarrollo Tecnológico para la Educación por su valiosa participación en la implementación de la nueva arquitectura modular en Laravel. Su dedicación, compromiso y trabajo en equipo fueron clave para el éxito de este proyecto. Aprecio su disposición para aprender y aplicar nuevas metodologías, así como su apoyo en la resolución de desafíos técnicos. Gracias a su esfuerzo conjunto, hemos logrado implementar esta arquitectura modular, que ya se ha utilizado en varios proyectos de desarrollo a la medida durante 2023 y 2024.

REFERENCIAS

Aniche, M., Bavota, G., Treude, C., van Deursen, A., y Gerosa, M. A. (2016). A validated set of smells in Model-View-Controller architectures. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 233-243. https://doi.org/10.1109/ICSME.2016.12

Barde, K. (2023). Modular monoliths: Revolutionizing software architecture for efficient payment systems in fintech. *International Journal of Computer Trends and Technology*, *71*(10), 20-27. https://doi.org/10.14445/22312803/IJCTT-V71I10P103

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd Edition). Addison-Wesley Professional.

Griffin, J. (2021). *Domain-Driven Laravel: Learn to implement Domain-Driven Design using Laravel*. Apress.

Martin, R. (2018). *Clean Architecture: A Craftsman's guide to Software Structure and Design*. Prentice Hall.

Velasco-Elizondo, P., Castañeda-Calvillo, L., García-Fernández, A., y Vazquez-Reyes, S. (2018). Towards detecting MVC architectural smells. In J. Mejia, M. Muñoz, Á. Rocha, Y. Quiñonez, y J. Calvo-Manzano (Eds.), *Trends and applications in software engineering. CIMPS 2017. Advances in Intelligent Systems and Computing, vol 688* (pp. 251-260). Springer. https://doi.org/10.1007/978-3-319-69341-5_23

Visser, J., Rigal, S., van der Leek, R., van Eck, P., y Wijnholds, G. (2016). *Building Maintainable Software, Java Edition*. O'Reilly Media. https://learning.oreilly.com/library/view/building-maintainable-software/9781491955987/